A

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 to~~ 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 realizes that users will not always be online or have a central server, so it allows offline ad-hoc 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. 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.

1. 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 a nonce, making it random and hopefully unique. Lines have the code shown in Figure 1:

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

Figure 1. The data structure for a Line.

Are these locally or globally stored?

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". For text & position? a.text > b.text a.posit < b.posit ?

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.

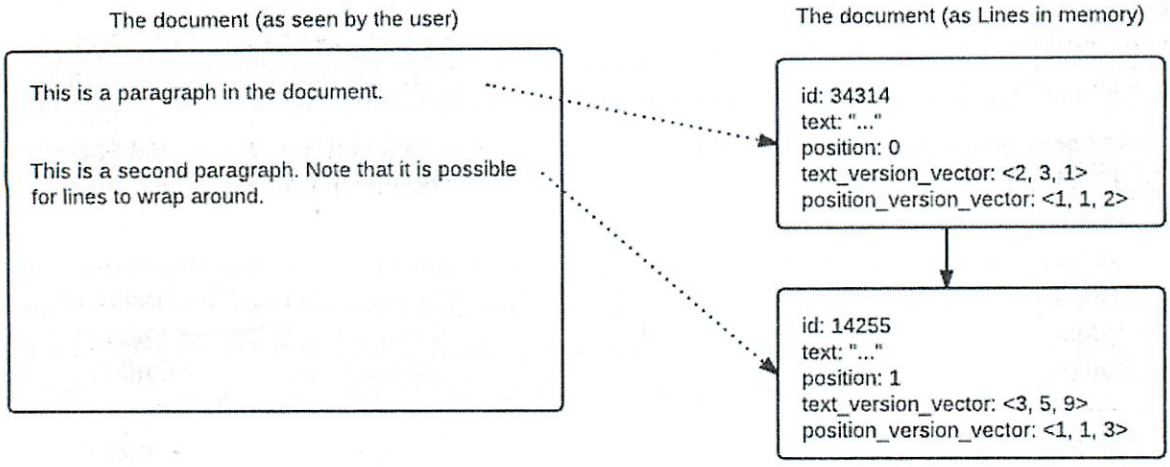

Figure 3. The data structure and its presentation. The user sees a continuous block of text, but Lines are stored in their own structures internally.

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. However, there are some additional elements because of DecentralizedDocs' unique features. When two or more users are connected over a network, a user can "sync" with a particular user. Users can also choose to propose to commit a specific version of the document.

does user know who+how many are currently online?

When the user is editing the document, he/she is actually viewing and editing a shadow copy of the document data structure that resides in memory. When the user saves, the shadow copy is serialized to disk and the pointer to the original copy of the document is moved to point to the newly serialized copy. DecentralizedDocs initializes a new shadow copy for additional edits and the next save. 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). It would be nice to periodically save.

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; correctness requirements mandate keeping the version vectors. Moving a Line updates `position`. After all updates, the corresponding variable's version vector is incremented at the index for the editing user.

→ hope you explain this

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. The interface displays the names of the users who are online. A user can click on their name to start a sync between them and that user. If there is more than one other user online, there is a special button to sync with all users.

There is also a button to start a commit operation. The UI will prompt the user to give that revision a name in a pop up window. The user will also be prompted for an expiry time for the commit.

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" involving 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 (ensuring that the merge produces correct results), and it must not leave Alice's document in a partially-merged state if either host or the network fails. what about Bob's version?

The pull operation contains 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.

Finally, Alice attempts to reconcile changes to variables that exist in both users' documents.

Alice is blocked from editing her document while the pull is in progress. If she edited the document, then she would have to simultaneously keep track of both unsaved changes (produced by herself) and saved changes (incoming from Bob); this situation could easily lead to user confusion and misunderstanding of system state. The merging algorithm is fairly simple, so the time in which Alice is blocked should not be more than a few seconds. Bob is not blocked from editing while Alice is pulling from him. DecentralizedDocs copies his Line structures before transmitting them to Alice, so his changes during this timeframe will not be visible to Alice until the next time she pulls from him.

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

what is sent? Each time, the whole file is sent?

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 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. DecentralizedDocs will show Alice a conflict-resolution dialog with both versions of the variable, and ask her to provide a merged version. 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. 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>
```

<5,7> <4,9> => ?

Figure 4. The state of a variable's version vectors before and after Alice pulls from Bob.

Logging during pulls

Fault-tolerance during pulls is managed using write-ahead redo logging, making pulls all-or-nothing atomic. 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 completed. 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 - only the log has changed.

Pulls only add changes, so they never regress a document. Therefore, users have a reasonable course of action to take after any failures to verify their states; they can simply pull and see if any changes occurred.

A ⊂ ⇕ ⇒ B D

A ⇔ C ??

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.

Committing

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

Can two start at same time? Two coordinators

The commit system requires that all peers be online in order to complete a commit. They do not have to be online at the point when the coordinator initializes a commit, but the commit cannot finish until all peers come online. DecentralizedDocs's commit system will periodically attempt to resend each

message until the message can be delivered. As with reconciliation, DecentralizedDocs uses a reliable transport protocol (TCP) to deliver messages and recognize if the message has been received.

is it possible that this may never happen?

The commit process will first check to make sure that all peers are up to date (i.e. all version vectors match among all peers). If this condition is not met, the commit will be aborted. Next the coordinator will send a `prepare` message to all of the peers. Next, the users of the peers are asked if they want to commit through a user interface. All users must explicitly approve the commit before the commit occurs, 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. We want to support this use case. If any user disagrees, the process is aborted. After a user agrees to commit, their local copy is locked. The tentative commit is recorded in the peer's log, and then the `locked` response is sent back to the coordinator.

If all peers respond affirmatively, the coordinator decides to commit. A checkpoint is recorded in the coordinator's log. This is the commit point. The coordinator then tells all of the peers to actually commit, using a `commit` message. The local peer marks this in their local log, marks that version as committed, and unlocks the local copy. The peer then sends back an OK message to the coordinator.

If something goes wrong before the coordinator decides to commit, the coordinator can tell the local peer to abort and 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 that peer's log.

The coordinator has the option to cancel the commit at any time before the commit point. If he/she does so, then this is treated similar to a commit-preventing failure. The coordinator tells all peers to abort and release their locks. If the coordinator has not heard back from all of the peers by time expiry time, the coordinator will issue an abort. Peers that have agreed to a tentative commit before the expiry time, but are offline at the expiry time, will remain locked. They do not know if the server has ordered a commit without their knowledge. When these peers reconnect, they will find out if the coordinator proceeded with the commit or aborted it. This change was made to maintain the simplicity of the design; otherwise the coordinator would have to perform additional checks.

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.

more complicated than this. subsets of users already began

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. If the name is not in that particular users' log, the other systems will check their logs

to see if the name is a duplicate. Those other systems will refuse to accept the commit if the name happens to be duplicated.

Analysis

Reconciliation System

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.

Commit System

DecentralizedDocs' commit system is resilient against the failure of either the coordinator or one of the individual peers.

Users offline

If any peer does not happen to be connected during the commit process, their message will be queued until they return online. For example, imagine a scenario where A is the committer and A and B are connected, but C is not. A will ask B if he wishes to commit. Assume B indicates affirmatively. A and B's copy will be locked for commit. The tentative commit message to C will stayed queued at A until C goes back online. A will periodically attempt to send the message to C. A and B's copy will remain locked.

Coordinator crashes before commit point

If the coordinator crashes before the commit point, the coordinator will recover from its log. It will see that it has an open commit pending. The other peers will be periodically be resending their tentative commit messages. Once the coordinator receives tentative commits from everyone, the commit will proceed until planned. The other clients will remain locked until the coordinator tells those peers to either commit or abort.

Coordinator crashes after the commit point

If the coordinator crashes after the commit point, the coordinator will recover from its log. It will see that it has issued a commit. It will resend the commit message to all nodes until it has received a confirmation (ie. OK) message back from all of the other users

Peer crashes before tentative commit

If a peer (i.e. an instance other than the coordinator) crashes before the tentative commit point, the peer will process the `prepare` message as normal.

Peer crashes after tentative commit, but before actual commit

If a peer crashes after the tentative commit, the peer will notice this state in its local log. It will keep the document locked and keep sending out the `locked` response to the coordinator. It will then proceed as usual when it receives the `commit` message from the coordinator.

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: 4,342

DP 2 Notes -- Larry Rudolph (section R01, R02)

Here are some of the things I was expecting to be addressed in your projects. It did not count if they were addressed in long, complicated hand-waving descriptions. This is a quickly written list. Please also look at http://people.csail.mit.edu/psz/6.033/dp2.pdf for a more complete explanation of the subtle issues that you may have overlooked.

1. Group Membership: Comments about membership, static or dynamic, who knows, how are members identified.

2. User Interface: Explicit save and sync or implicit (edit and sync buffers). Edit disabled while sync? Failed sync get retried or require user intervention. Moving list of adjacent paragraphs and how conflict resolution works in such cases. User required to be online whole time or what happens during

3. Version Vectors. How many and what they cover and what they contain. When are they updated. The case that many projects missed is when there are two pairs concurrently synchronizing, e.g. Alice with Bob and Charlie with Debbie. Each pair resolves their conflicts independently, even if the conflicts are on the same sentence. Then Alice syncs with Debbie and Bob with Charlie. If version vectors are not updated correctly (like in most of your projects) then can end up with inconsistent results.

4. Conflict Resolution: Definition of conflict and what needs user intervention (and what if user is not on-line or not editing). Semantic resolution (words in sentence). Is whole file send every time, or just vectors and data is then requested. Alice swaps paragraphs 1 and 2, Bob deletes paragraph 2: conflict? Is this addressed at all? If GIT is used, what modifications. Correct propagation of modifications to reduce need of manual resolution.

5. Commit: If two phase commit used, how is it modified. Does it require everyone to be online at same time? Will commit succeeded if everyone eventually comes online? Can there be livelock? If leader chosen, is it chosen correctly in all cases? Use logs correctly, including write-ahead-logs. Can inconsistent version be committed?

6. Analysis: How does the design address the requirements especially the edge cases. Were all the cases considered and is there an argument that this is all the cases?

Good luck with this and other courses, good luck at MIT, in work and especially in life. Feel free to stay in touch. My MIT email should always work (rudolph@csail.mit.edu) and I use Linkedin (same mit email contact) but hardly ever use my facebook account.