Notes on 6.033 2012 DP2 Designs

Peter Szolovits
May 22, 2012

DP2 asks students to design a collaborative P2P editor that supports off-line editing and subsequent reconciliation of documents with either compatible or conflicting edits. The editor must then allow one of the peers to initiate a commit that makes sure that all revisions by every peer have been reconciled, and then commits to a version of the document that everyone agrees on. Although such a commit is only required one time at the end of the editing process, many groups support allowing the group to commit to a single version of the document at intermediate times as well. In that case, it is also useful to include the ability of peers to continue editing their copy of a document even while such a commit is taking place. The system must be robust to failures. An optional extension is to allow membership in the group of peers to change during the editing process.

Students in my sections created a large variety of technical approaches, but here are some of the essential components that most included.

Pairwise Reconciliation

At minimum, two people working on a document must be able to reconcile changes each has made to the document. This can happen in small increments if the two are connected as they are editing (one group did it character by character) or by larger units such as lines, paragraphs, or the entire document. Some argued that using continuous reconciliation for on-line users will reduce the amount of manual reconciliation and is thus worthwhile. With such an approach, I would worry a bit about network traffic and also the HCI aspects of seeing nearly-simultaneous changes to a collaborative document. Even with continuous reconciliation, a less immediate reconciliation mechanism is still needed for when previously off-line users come on-line or if network partitions are repaired. Students took two main approaches to reconciliation:

1. Keep logs of the actions of each user, and use a state-machine type of reconciliation where the logs of actions are sent to the other party to replay the edits of the first. It is critical that reconciliation events, including their results, are also logged.
2. Use diff-like comparisons of the resulting text to identify the (roughly) minimal sets of changes that need to be made to each document to bring it into sync with the other. This method can be applied to an entire document (though standard diff doesn't do well with moving paragraphs) or to smaller sections of the document such as paragraphs, so that paragraph re-arrangements can be handled gracefully.

Document units can be assigned unique identities as they are created or merged, or solutions can rely on character comparisons among different versions to maintain identity. In some cases, this latter requires computing an edit distance and using a

threshold of similarity to decide whether a unit that may have been edited (and possibly moved) really does correspond to its previous version. Supported changes were either (a) edits to some unit of the document or (b) rearrangements. In either case, if one user changes one such aspect of the document and the other does not, then reconciliation can be automated by simply accepting that change. Note that this allows one person to move a paragraph while another is editing it. If both parties have changed a unit or made rearrangements, then a manual reconciliation is required. Most designs maintain separate version vectors for each unit of the document and for each possible level of rearrangement. (Typically the units were paragraphs and therefore rearrangements were among whole paragraphs.)

Some designs retained previous versions of documents and could find the most recent common ancestor when reconciling newer versions edited by two users. This can simplify reconciliation by reconstructing just what changes were made by each user. Log-based reconciliation provides an alternative because the log already contains that information. One group used GIT as a basis for keeping version data and reconciliation (and also to manage group membership). That seems like a "heavy" solution, but has the advantage that it is probably quite robust and well debugged because it is widely used.

Non-redundant reconciliation

The assignment requires (scenario 2 of conflict resolution) support for the following:

> 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.

Being able to handle this scenario essentially requires maintaining and communicating version vectors so that each user can keep track of the state of edits by others. It also requires that if Alice and Bob sync, their version in the version vector must be incremented, to reflect changes caused by that reconciliation. That discipline will solve the above scenario. A number of the proposed designs either did not address this scenario or treated it incompletely.

Commit

Commit requires some implementation of a two-phase commit protocol, probably with the coordinator role played by the initiator of the commit. The only interesting variability here comes from the following considerations:

1. Does the coordinator first try to sync with every other peer before initiating the two-phase commit, or do peers simply abort the commit if they detect that a further sync is necessary.

2. For systems that support a commit while continuing to edit, each peer needs to retain or log a frozen state via which they participate in the commit, but at the same time allow further edits to be made, which will subsequently be reconciled with others. In this approach, there have to be two kinds of commits: (a) "incremental commits" that just get all the peers to agree on some nearly-current version of the document—this can help flush long edit logs, for example—and (b) "final commits" beyond which no more editing is allowed by any peer. Any peer that has a newer version of some part of the document than is attempting to get finally committed must abort that commit.
3. Some designs require that all the peers be on-line during the commit. This will tend to produce quick commits, but may be unrealistic with a large group. A two-phase commit can be accomplished without that requirement, so long as each peer can at some time communicate with the coordinator. However, in that scenario, commits may take arbitrarily long.

Peer Membership

Some solutions required the creator of a new shared file to specify the unchangeable list of peers who could edit it. Others provided some mechanism to allow new peers to join or to leave the group. In some designs, any existing peer could "invite" a new peer and add it, whereas in others, each existing peer needed to approve adding a new one; in that case, another two-phase commit had to take place to get that agreement. There were various protocols designed to maintain the group. Some created completely new "group management" protocols that ran in addition to the reconciliation and commit protocols, whereas others layered group management information onto the protocols that managed reconciliation. Generally, a peer leaving a group has to inform someone still in it in order to distinguish between leaving and just being offline for an extended time. The peer leaving must also sync with a remaining member to make sure that edits the leaving peer made are not lost.

Peers need some ID independent of their IP address so they can be recognized as they may connect from different network accesses or by direct connection to another peer. Some designs proposed using a permanent ID such as the MAC address of the machine, others simply had each peer create a large random number as an ID, with very low probability of collision, and others had more complex identification methods. If version vectors are used, then each vector must have an entry for every peer, so that if peers are added to or leave the group, all version vectors must grow or shrink. A nice approach sets the version vector entries for newly joined peers to zero, so the next reconciliation with any other peer will update the new peer to a near-current version.