Online synchronization of replicated models

Stefan Lindel
SE Group, Kassel University
stefan.lindel@cs.uni-kassel.de

Albert Zündorf
SE Group, Kassel University
zuendorf@uni-kassel.de

General Terms

Models, Versioning, Distribution

1. MOTIVATION

This paper reports about our use of model versioning mechanisms in the industrial project *ConfNet*. The ConfNet software is used to organize sessions and talks for medical conferences. In these medical conferences, the presenters come with PowerPoint files on USB memory sticks. The technical team checks the PowerPoint files for external references e.g. to images or videos and converts each presentation to a ppsx file. ppsx files are self contained and PowerPoint opens them in presentation mode directly. Presentations happen in parallel sessions in different rooms, cf. Fig. 1. For each session, ConfNet provides an HTML page listing the talks. This HTML page is shown in full screen mode. By clicking on a talk, the corresponding PowerPoint presentation is opened (directly in presentation mode) and after the talk we fall back to the session overview page. Each room has its own laptop and video projector. In principle, all rooms are connected to a site local network. However, as these conferences run in large hotels or conference centers, the network infrastructure is regularly subject to failures, e.g. when the hotel technician reconfigures some network switches because some guest has problems with internet access in some room. To deal with such network failures, the presentation laptops do not rely on a central server but all presentation files and all session HTML files are copied to the corresponding presentation laptops. The HTML page and the presentations are then accessed from the local disk.

Usually, the conference team provides one central media room, where the presenters deliver their PowerPoint files. In that media room some five registration desk computers are provided where technically skilled personal checks the PowerPoint files, converts videos, etc. and packs the ppsx files. Then these files are transferred via site local network

CVSM2012 Essen, Germany

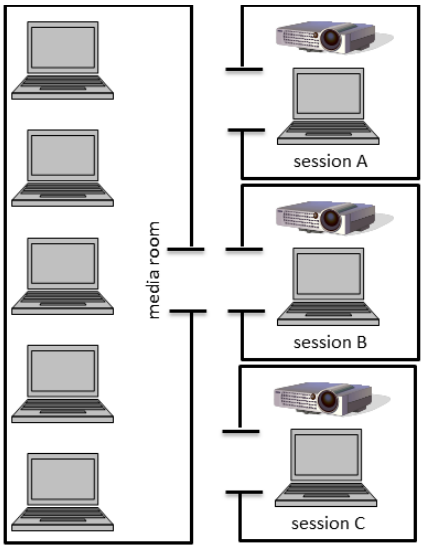

Figure 1: ConfNet Scenario

to the presentation laptops in the different rooms. To enhance fail safety, all files are additionally copied to all other registration desk computers and to some spare laptops that step in, if a presentation laptop crashes. If some computer is offline while some new files are submitted, these files and changes are transferred as soon as that computer becomes online again. In addition to the actual presentation files, ConfNet maintains an object model that reflects the submission state of each talk. This allows to check whether all presentation files have been submitted to the system and whether they have been transferred to the corresponding presentation laptops before the conference sessions start. It is also possible to fix some typos in talk titles or presenter names and to regenerate the overview HTML pages for the corresponding sessions. In addition, the personal may add some comments to the talk as e.g. “has video with sound” in order to tell technicians in the session room to switch the sound on for that talk. A typical medical conference has some 400 presentations on three days in about 5 or 6 rooms.

2. DESIGN

As discussed, we want to be able to work on multiple computers concurrently on the same data while the network is unreliable and we want to be as fail save as possible. Usually, if there are multiple computers working on the same

data concurrently, we use a (relational) database and its transaction concept. In our case, a central database would introduce a single point of failure and the unreliable network might interrupt access to the database during the presentations. In addition, ConfNet deals with a relatively small amount of data that easily fits into main memory even for small computers. Thus, a database designed to deal with mass data seems inappropriate. Instead, we have designed a peer to peer system where each peer has its own copy of the whole data and each peer works on its copy, independently. To coordinate the peers, each peer records all changes to the data model and stores a local version history. When a network connection is available, we merge the local version history of each peer with the histories of all other peers in a GIT [1] like fashion. As each peer maintains the whole data as well as the whole change history, a crash of some computer does not harm the overall system and the crashed computer is easily replaced by one of its peers. In case of network problems, each computer may still work independently of the others and record changes in its local history. When the network connection is reestablished, the different history chunks are mutually exchanged and merged and all peers are synchronized, again. While this works great in general, special care needs to be taken to deal with merge conflicts. This will be discussed in the next sections. And, to be honest, this is not a system for flight booking or managing bank accounts. Lost updates might occur and we need to deal with them in an application specific way.

3. IMPLEMENTATION

Figure 2 shows the internal structures of our ConfNet peers. Each peer holds a copy of the current *data*. The data model is fairly simple. It deploys classes like `Conference`, `Room`, `Session`, `Talk`, `File`, etc. with the attributes one expects. Next we use a *JSON* [2] based serialization mechanism. This JSON serialization is able to serialize whole object structures and also small property changes. The JSON serialization is based on a *reflection* layer. The reflection layer is generated using our new modeling tool *SDMLib [3]* that has been developed along with the ConfNet system. Basically, SDMLib provides means for the editing of class diagrams and a very flexible code generation approach. For each model class, SDMLib generates a reflection class that allows to create model objects by providing a string with the desired class name and that allows to read and write attributes of a given model object by providing a string with the attribute name. The reflection class also provides a list of attributes for each model class. Thus, to serialize an object, one asks the object for its class and that class for its name, then we ask the reflection layer for the list of attribute names of such an object and for each attribute name we ask the reflection layer to retrieve the attribute value. This is then stored in JSON format.

To handle attributes that contain pointers to other objects, we deploy unique object ids. These unique object ids are maintained by the *idMap* component. The idMap component provides object ids that are unique across the whole peer network. This is achieved by a running object number plus a unique peer id. The unique peer id is constructed from the IP address of the peer node plus the port number that this peer uses for communication with other peers. Adding the port number allows to run multiple peers on one computer. The idMap component also deploys two hash maps: one hash map that uses the unique object id as key to store model objects and a second hash map that uses the model object as key to retrieve the corresponding unique key. At program start, the root object of our data, i.e. the `Conference` object is added to the idMap. As soon as an model object is added to the idMap, the idMap uses the reflection layer to access the attributes of the model object and to retrieve its neighbor objects. These neighbor objects and their neighbors are added recursively. Note, we provide a special removeYou operation to remove model objects from the current data as well as from the idMap.

In addition, the idMap subscribes itself as listener to all property changes of all model objects. Each property change is then recorded in the *history* component. All entries to the history component again get a unique id composed of a running number and the unique peer id. Next, the idMap uses the *peer stubs* component to retrieve the IP addresses and communication ports of currently active peers. Each recorded property change is also serialized to JSON and send to the other active peers. The idMap component also receives property change messages from other peers. Such property changes are de-serialized and with the help of the reflection layer the changes are applied to the local data of the current peer.

To handle network problems and loss of messages, each property change message gets a reference to the previous change in the change history. When our idMap receives e.g. a message with number peerB:42 and with predecessor peerB:41, it checks whether change peerB:41 is already known in its change history. If not, it asks the sender of change peerB:42 to send change peerB:41, too. This is done recursively, until all missed messages have been transferred. Then the changes are applied in the order of their occurrence.

Note, as each change has a running number plus the unique id of the peer that has first recorded this change, the ids of changes are unique across the whole peer network and based on these unique change ids we are able to sort all changes within our history based on these ids. Our peers interpret this sorting as a timely order. This timely order is used to resolve merge conflicts: when two peers change e.g. the title of a certain talk “at the same time”, these two changes are recorded by the idMaps of those two peers and these two changes get different unique ids e.g. peerA:88 and peerB:88, cf. Fig 3, time t1. Once the changes have been exchanged between the two peers, each peer adds the received change to its local history, cf. Fig [fig:Merging-changes], time t2. As discussed the history is sorted based on the unique change ids. When peer A receives change peerB:88 this change is “later” then its own change peerA:88 and thus the later change overwrites the earlier change, the title becomes “T_A”. When peer B receives change peerA:88 its history already contains a “later” change peerB:88. In addition, peer B recognizes that the two changes represent a merge conflict as they both change the same attribute of the same model object in different ways. Peer B handles this conflict as if change peerA:88 had occurred first and then the corresponding attribute has been changed again by change peerB:88. This means, the “old” change peerA:88 is just ignored.

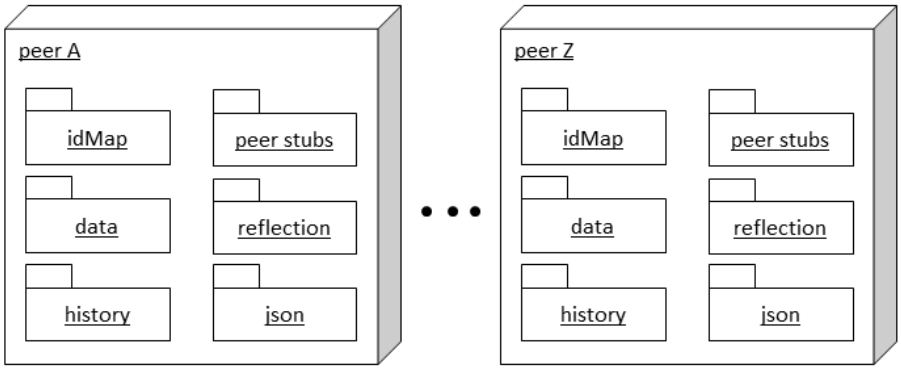

Figure 2: Peer Structure

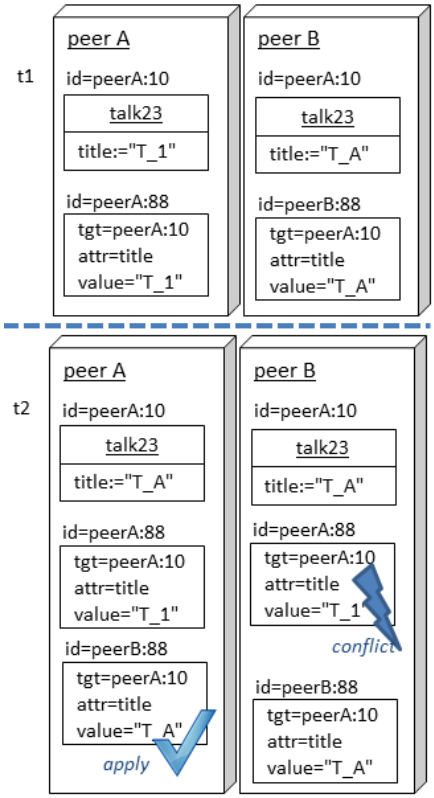

Figure 3: Merging changes

To some extend, this results in a lost update: the change on peer A is overwritten by the "later" change on peer B although peer B did not see that the same attribute has been changed at peer A "at the same time". Well, if peer A would have been some milliseconds faster, peer B would have received change peerA:88 and the graphical user interface of peer B would have shown the new title just before the user at peer B would have clicked into that text field and would have changed it. The user might have ignored the sort flicker of his or her user interface and thus just have overwritten the change done at peer A. This would have resulted in the same "lost update" as our handling of merge conflicts.

In our ConfNet tool such conflicts never occurred as no two people tried to change the same talk at the same time at two different peers. At each peer each presenter changed only the status of his or her own talk.

4. CONCLUSIONS

Within the ConfNet project we have developed a quite general model data replication mechanism that enables concurrent model changes on multiple computers. The recording of changes and the mutual exchange of history changes result in a very stable peer to peer network that easily deals with crashes of some peers and with temporal network problems. The ConfNet system has been used with great success in 3 big medical conferences now. In this practical system usages, especially the flexible handling of system crashes without any loss of data proved to be a major strength of the system. So far, our model replication approach is limited to very small amounts of data (some giga bytes). Actually, the data must fit in main memory. Actually also the change history needs to fit in main memory and the size of the change history grows with the actual number of changes i.e. proportional to the up time. Strategies to reduce the size of the history (i.e. to abandon old changes) are current work. Due to the success of ConfNet, we currently use its model replication mechanism for a new software for rock festivals. Rock festival management software face a situation quite similar to medical conferences: The people need to work on the same data on multiple computers concurrently but they do not have a reliable network infrastructure. Actually, they do not even have a reliable power grid.

5. REFERENCES

[1] git-scm.com. Website, Sept. 2012. Online at http://git-scm.com/ visited on 02.09.2012 00:42.

[2] json.org. Website, Mar. 2012. Online at https://github.com/douglascrockford/JSON-java visited on 08.03.2012 00:42.

[3] sdmlib.org. Website, Sept. 2012. Online at http://sdmlib.org/ visited on 02.09.2012 00:42.