Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

Mechanismus zur Synchronisation verteilter Datenmodelle für kollaborative Editoren

Sebastian Runge[1]

Abstract: Innerhalb der letzten zehn Jahre haben kollaborative Editoren (Mehrbenutzer-Editoren) an Bedeutung gewonnen. Dies sind Anwendungen, die mehreren Nutzern die gemeinsame Bearbeitung von Daten erlauben, auch wenn die Nutzer zeitlich oder räumlich voneinander getrennt sind. In diesem Beitrag wird ein Mechanismus beschrieben, welcher die Datenmodelle der verschiedenen Nutzer synchronisiert und so die Zusammenarbeit ermöglicht. Am Modell eines Nutzers vorgenommene Änderungen werden über ein Netzwerk unmittelbar an alle anderen Nutzer übertragen. Weiterhin wird es den Nutzern ermöglicht, auch ohne eine Netzwerkverbindung die Daten lokal zu bearbeiten, um sich zu einem späteren Zeitpunkt wieder mit den anderen Nutzern zu synchronisieren.

Keywords: kollaborative Editoren, verteilte Datenmodelle, Modellsynchronisation

1 Einleitung

Innerhalb der letzen zehn Jahre haben kollaborative[2] Editoren, auch als Mehrbenutzer-Editoren bezeichnet, an Bedeutung gewonnen. Dies sind Computerprogramme, die zur sogenannten *Groupware* gehören und in den Bereich des interdisziplinären Forschungsgebiets CSCW (*Computer Supported Cooperative Work*) fallen. Kollaborative Editoren erlauben es ihren Nutzern, gemeinsam Dokumente wie Texte oder Bilder zu bearbeiten, auch wenn die Nutzer zeitlich oder räumlich voneinander getrennt sind. Diese Zusammenarbeit kann entweder zeitversetzt wie bei einem Versionsverwaltungssystem erfolgen oder mit einer Synchronisation in Echtzeit, bei der das Editieren eines Nutzers sofort auch auf den Bildschirmen der anderen Nutzer sichtbar ist. Ermöglicht wird diese Technik durch die starke Vernetzung moderner Computersysteme. Erforderlich wird sie durch die Tatsache, dass heutige Produktionsprozesse, nicht nur die Softwareentwicklung, häufig in einem Team stattfinden.

Ein geläufiges Anwendungsgebiet für kollaborative Editoren ist das gemeinsame Schreiben von Texten, wie es etwa in Google Docs[3] oder Etherpad[4] ermöglicht wird. Diese Anwendungen nutzen eine Synchronisationstechnik namens operationale Transformation (OT) [EG89], wobei die von Neil Fraser in [Fr09] vorgestellte differentielle Synchronisation eine Alternative dazu darstellt. Die genannten Techniken sind jedoch in erster Linie

[1] ehemals Universität Kassel, egnurs@gmail.com

[2] Im Deutschen wird unter dem Wort Kollaboration eigentlich die Zusammenarbeit mit dem Feind zu Kriegszeiten verstanden. In diesem Beitrag bezieht sich der Begriff jedoch auf die Zusammenarbeit im Allgemeinen, angelehnt an das englische und französische *collaboration*.

[3] https://www.google.de/intl/de/docs/about/

[4] http://etherpad.org/

auf die Synchronisation von Texten ausgelegt, während in diesem Beitrag die Synchronisation von allgemeinen Modelldaten behandelt werden soll.

Die Elemente, welche die Nutzer über die Benutzeroberfläche (GUI) eines Editors bearbeiten, werden durch ein Datenmodell im Arbeitsspeicher repräsentiert. Die Modelldaten bestehen, zumindest bei Anwendung des objektorientierten Programmierparadigmas, aus Objekten mit Attributen sowie aus Referenzen zwischen den Objekten. Im Falle des kollaborativen Editierens, bei dem die Nutzer in der Regel an verschiedenen Computern arbeiten und gleichzeitig ein Dokument editieren, verfügt jeder Nutzer über eine eigene Instanz beziehungsweise eine Kopie dieses Objektmodells. Damit die Zusammenarbeit der Nutzer funktioniert, müssen Änderungen, die ein Nutzer an seiner Modellinstanz vornimmt, auch an den Modellinstanzen aller anderen beteiligten Nutzer wirksam werden. Dazu ist ein Mechanismus notwending, der die Modellinstanzen mit Hilfe einer Netzwerkverbindung untereinander synchronisiert. Ein solcher Mechanismus, der im Rahmen einer Masterarbeit entstanden ist, wird in diesem Beitrag vorgestellt.

Ein Nachteil der operationalen Transformation ist, dass sie als nicht einfach gilt. Joseph Gentle, ein ehemaliger Entwickler der OT-Anwendung Apache Wave schreibt etwa: „Die korrekte Implementierung der Algorithmen ist sehr schwierig und zeitaufwändig“ [Ge11]. Die differentielle Synchronisation hingegen hat den Nachteil, dass mehrere Kopien eines Datenmodells im Speicher gehalten werden müssen, damit die Differenzen zwischen diesen ermittelt werden können. Daher wurde in der Masterarbeit auf beide Ansätze verzichtet mit dem Ziel, den Synchronisationsmechanismus möglichst simpel zu gestalten. Die Synchronisation soll dabei unsichtbar für alle Anwender durchgeführt werden. Auftretende Merge-Konflikte sollen automatisiert aufgelöst werden, sodass der Editierfluss der Nutzer nicht durch Konfliktmeldungen und manuelles Merging gebremst wird. Auch die Entwickler der kollaborativen Editoren sollen sich nicht mit der Synchronisation auseinandersetzen müssen: Die Synchronisation des gesamten Modells soll ohne weiteren Programmieraufwand geschehen, sobald das Root-Objekt des Modells in den Mechanismus eingehängt wurde.

Der Mechanismus soll zwei Betriebsmodi unterscheiden können: den Online- sowie den Offline-Modus. Im Online-Modus arbeiten mehrere Nutzer zeitgleich, wobei Modelländerungen direkt über eine Netzwerkverbindung übertragen werden. Im Offline-Modus besteht keine Verbindung zu anderen Nutzern; ein Nutzer kann dann trotzdem die Daten weiter bearbeiten und sich zu einem späteren Zeitpunkt wieder mit den anderen Nutzern synchronisieren.

Im weiteren Verlauf werden Aufbau und Funktionsweise des Synchronisationsmechanismus behandelt (Abschnitt 2). Im Anschluss wird als Beispielanwendung ein kollaborativer UML-Editor vorgestellt (Abschnitt 3). Abschließend folgen das Fazit sowie der Ausblick auf mögliche Verbesserungen und Erweiterungen (Abschnitt 4).

2 Der Synchronisationsmechanismus

Die Implementierung des Synchronisationsmechanismus erfolgt in der Programmiersprache Java unter Zuhilfenahme der Bibliothek SDMLib[5]. SDMLib ist eine im Fachgebiet Software Engineering der Universität Kassel entwickelte Bibliothek zum Generieren von Klassenmodell-Code in Java. Weiterhin bietet SDMLib eine Objektverwaltung, bei der Objekten global eindeutige Identifikationsnummern (Objekt-IDs) zugewiesen werden sowie Änderungen an diesen Objekten erfasst und protokolliert werden können. Das grundlegende Verfahren bei der Objektverwaltung wird in [Li12] beschrieben.

2.1 Grundlegender Aufbau

Abb. 1 zeigt den grundlegenden Aufgbau des Mechanismus, der im Folgenden erläutert wird. Gezeigt werden die beiden Nutzer Alice und Bob, jedoch funktioniert der Mechanismus mit beliebig vielen Nutzern.

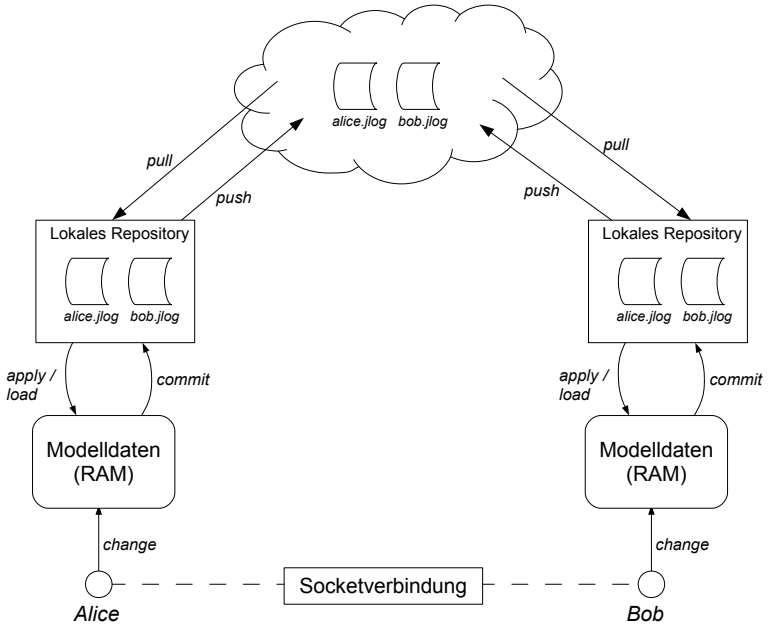

Abb. 1: Grundlegender Aufbau des Synchronisationsmechanismus

Die Synchronisation der Modelldaten geschieht ohne Zutun des Benutzers und wird in regelmäßigen Abständen automatisch wiederholt. Die Architektur ist darauf ausgelegt, dass es keinen zentralen Server gibt, der die Synchronisation vornimmt; stattdessen sollen

[5] https://github.com/fujaba/SDMLib

sich die Nutzer zu einem Peer-to-Peer-Netzwerk zusammenschließen und auch arbeiten können, wenn keine Netzwerkverbindung verfügbar ist. Die folgenden Punkte erläutern die einzelnen Komponenten aus Abb. 1 genauer:

- *Modelldaten (RAM):* Jeder Nutzer, der gerade eine Datenbasis, etwa ein Dokument, im kollaborativen Editor bearbeitet, verfügt über eine Kopie der Modelldaten in seinem Arbeitsspeicher (RAM). Änderungen, die ein Nutzer in der Editor-GUI vornimmt, werden am Modell wirksam.
- *Lokales Repository:* Jeder Nutzer verfügt über ein lokales Repository. Dabei handelt es sich letztlich um einen Ordner auf der eigenen Festplatte, in dem sich Change-Log-Dateien befinden. Diese Log-Dateien enthalten die *History* (Geschichte) der Änderungen, welche die Nutzer an den Modelldaten bislang vorgenommen haben. Die Dateien haben die Endung *.jlog*, was für „JSON-Log“ steht und ein eigenes Dateiformat ist. Der Vorteil des lokalen Repositorys ist, dass jeder Nutzer über den Datenbestand verfügt. Somit können alle Nutzer unabhängig voneinander offline arbeiten. Das Repository enthält eine indirekte Persistierung des Datenmodells, da durch das Nachvollziehen aller Änderungen das gesamte Modell rekonstruiert werden kann. Siehe dazu auch die Operation *Load* weiter unten.
- *Globales Repository:* Zusätzlich zu den lokalen Repositorys der einzelnen Nutzer gibt es ein globales (zentrales) Repository, auf das alle Nutzer Zugriff haben. In Abb. 1 ist es durch eine Wolke (*Cloud*) symbolisiert. Das globale Repository kann ein Netzwerkordner sein, auf den jeder Benutzer Zugriff hat, oder auch ein Cloud-Storage von Dienstleistern wie Dropbox[6] oder Owncloud[7]. Als effektives Mittel für die Synchronisation von Ornderinhalten hat sich auch BitTorrent Sync[8] erwiesen. Bei all diesen Alternativen ist der Administrationsaufwand verhältnismäßig gering, da kein Server eingerichtet werden muss; stattdessen muss lediglich ein Ordner auf der Festplatte eines Nutzers freigegeben oder ein Cloud-Storage-Client installiert werden. Das globale Repository enthält ebenfalls die jlog-Dateien. Über das globale Repository werden die lokalen Repositorys miteinander synchronisiert, das heißt es wird dafür gesorgt, dass alle lokalen Repositorys möglichst immer die gleichen Dateien enthalten. Um Zugriffskonflikte auf die Dateien im globalen Repository auszuschließen, wird sichergestellt, dass jeder Nutzer seine Änderungen in eine eigene jlog-Datei schreibt, die nach dem Nutzer benannt ist. Entsprechend schreibt Alice nur in die Datei *alice.jlog* und Bob nur in *bob.jlog*. Dadurch sind die einzelnen Log-Dateien disjunkt zueinander, das heißt eine konkrete Änderung am Datenmodell tritt nur in einer einzigen der Dateien auf. Eine Datei enthält somit immer nur einen Teil der History; die vollständige History beziehungsweise die vollständigen Modellinformationen ergeben sich nur, wenn die Dateien aller Nutzer vorliegen.

[6] http://www.dropbox.com/
[7] http://owncloud.org/
[8] http://www.getsync.com/

Zwischen diesen genannten Komponenten werden Operationen durchgeführt, die in Abb. 1 durch beschriftete Pfeile dargestellt werden. Die folgenden Auflistung erklärt diese Operationen genauer:

- *Commit:* Die Übertragung einer einzelnen Änderung vom Arbeitsspeicher in das lokale Repository. Ein *Commit* wird unmittelbar nach Auftreten einer Änderung durchgeführt: Unter Verwendung des Observer-Patterns ([St11], S. 169) sind an sämtlichen Attributen im Objektmodell Listener angemeldet, die jede Änderung registrieren, sobald sie am Modell vorgenommen wird. Eine Modelländerung kann dabei das Erzeugen eines Objekts darstellen oder auch die Änderung eines Attributwerts eines bereits existierenden Objekts. Aus jeder Änderung wird ein Change-Objekt erzeugt, welches alle relevanten Informationen der Änderung enthält. Bei einem *Commit* wird ein Change-Objekt in das textuelle JSON-Format persistiert und in die jlog-Datei des Nutzers im lokalen Repository des Nutzers geschrieben.
- *Apply:* Die Übertragung einer einzelnen Änderung vom lokalen Repository in den Arbeitsspeicher. Ein *Apply* ist die Anwendung einer Änderung eines anderen Nutzers am eigenen Datenmodell. Dazu werden die jlog-Dateien der anderen Nutzer im lokalen Repository regelmäßig ausgelesen. Wird dabei eine Änderung gelesen, die in der eigenen History noch nicht bekannt ist, wird diese Änderung am eigenen Datenmodell angewandt und der eigenen History hinzugefügt, nicht jedoch in die eigene jlog-Datei eingetragen. Wenn die Änderung am eigenen Datenmodell angewandt wird, würde sie jedoch wie eine eigene Änderung von einem Listener registriert werden und einen *Commit* verursachen. Da jedoch nur eigene Änderungen in die eigene jlog-Datei eingetragen werden sollen, wird vor dem *Apply* einer fremden Änderung die *Commit*-Funktionalität mit einem einfachen Boolean-Flag vorübergehend außer Kraft gesetzt.
- *Load:* Das Wiederherstellen des Datenmodells im Arbeitsspeicher anhand der Log-Dateien im lokalen Repository. Um die Modelldaten wieder in den Arbeitsspeicher zu laden, zum Beispiel nach einem Neustart des Editors, werden alle zur Datenbasis gehörenden jlog-Dateien im lokalen Repository eingelesen, wobei alle Änderungen in eine gemeinsame Liste geschrieben werden. Die Liste wird anschließend nach logischen Zeitstempeln der Änderungen sortiert und die Änderungen werden in dieser Reihenfolge am Modell angewandt, sodass das Modell Schritt für Schritt rekonstruiert wird. Das Wiederherstellen des Datenmodells, das Zusammenfügen der jlog-Dateien sowie die Behandlung von dabei auftretenden Merge-Konflikten wird in Abschnitt 2.2.1 genauer beschrieben.
- *Push:* Die Übertragung der eigenen jlog-Datei vom lokalen in das globale Repository. Durch einen *Push* werden die eigenen Änderungen den anderen Nutzern bekannt gemacht. Dazu wird die eigene jlog-Datei vom lokalen Repository in das globale Repository kopiert, insofern das globale Repository erreichbar ist. So würde Nutzerin Alice etwa die Datei *alice.jlog* vom lokalen in das globale Repository kopieren; ist im globalen Repository bereits eine Datei *alice.jlog* vorhanden, so wird diese überschrieben.

- *Pull:* Die Übertragung der fremden jlog-Dateien vom globalen Repository in das eigene lokale Repository. Durch einen *Pull* werden die Änderungen der anderen Nutzer abgeholt. Dazu werden die jlog-Dateien - mit Ausnahme der Datei, die dem Nutzer zugeordnet ist - aus dem globalen Repository in das lokale Repository kopiert, insofern das globale Repository erreichbar ist. Nutzerin Alice etwa würde die Datei *bob.jlog* vom globalen in das lokale Repository kopieren; ist im lokalen Repository von Alice bereits eine Datei *bob.jlog* vorhanden, so wird diese überschrieben. Direkt nach dem *Pull* werden die fremden jlog-Dateien im aktualisierten lokalen Repository ausgelesen, um nach noch nicht bekannten Änderungen der anderen Nutzer zu suchen. Diese werden ähnlich zu *Load* gesammelt, sortiert und in zeitlich richtiger Reihenfolge am eigenen Modell angewandt (*Apply*). Die eigene jlog-Datei im lokalen Repository muss an dieser Stelle selbstverständlich nie ausgelesen werden, da sie nur Änderungen enthält, die bereits bekannt beziehungsweise an der eigenen Modellinstanz umgesetzt sind.

Diese Operationen werden nie vom Benutzer bewusst ausgelöst, sondern finden automatisch im Hintergrund statt. Während die Operation *Commit* immer dadurch ausgelöst wird, dass ein Nutzer eine Änderung vornimmt, werden die Operationen *Push* und *Pull* in einem festen Zeitintervall wiederholt durchgeführt; vorstellbar ist eine Durchführung in jeder Minute oder in einem kürzeren Zeitraum. *Push* und *Pull* dienen zur Synchronisation des globalen Repositorys mit den lokalen Repositorys beziehungsweise letztlich zur Synchronisation der lokalen Repositorys untereinander. Da jeder Nutzer nur die eigene jlog-Datei verändert, ist zudem gesichtert, dass *Push* und *Pull* jederzeit unabhängig voneinander und in beliebiger Reihenfolge durchgeführt werden können.

Bei den angegebenen *push/pull*-Intervallen von über einer halben Minute ist eine Echtzeit-Synchronisation natürlich nicht möglich. Daher gibt es die in Abb. 1 gezeigte Komponente *Socketverbindung*. Dabei handelt es sich um eine direkte Netzwerkverbindung zwischen den Nutzern, die eingerichtet wird, sobald erkannt wird, dass die Nutzer zeitgleich mit dem Editor arbeiten. Diese Verbindung dient zur Online-Synchronisation zwischen den Nutzern, die deutlich schneller und unmittelbarer ist als die die Offline-Synchronisation über die Repositorys. Dieser Online-Modus wird in Abschnitt 2.3 genauer erläutert.

2.2 Die Change-Log-Dateien

Listing 1 zeigt beispielhaft eine einzelne Änderung im JSON-Format, wie sie in der Praxis in einer jlog-Datei zu finden wäre. Der JSON-Code wird ohne die Zeilenumbrüche in die Datei geschrieben, sodass jedes Change-Objekt genau eine Zeile in der Datei einnimmt.

```
{
  "class" : "org.sdmlib.replication.ReplicationChange",
  "id"    : "alice.R51",
  "prop"  :
  {
    "historyIdPrefix" : "alice",
```

```
      "historyNumber"   : 23,
      "targetObjectId"  : "bob.S27",
      "targetProperty"  : "name",
      "changeMsg"       :
      {
        "id"  : "bob.S27",
        "upd" :
        {
          "name" : "Otto"
        }
      }
    }
}
```

List. 1: jlog-Eintrag einer einzelnen Änderung am Modell

In der gezeigten Änderung hat Nutzerin Alice den Wert eines Namensattributs mit „Otto“ überschrieben. Die ID der Änderung, *alice.R51*, ist zusammengesetzt aus dem *historyId-Prefix* (hier *alice*) und einer fortlaufenden Nummer (hier *51*). Die fortlaufende Nummer gibt die zeitliche Abfolge der Änderungen vor. Der Präfix gibt nicht nur den Urheber der Änderung an, sondern stellt sicher, dass die Änderungs-IDs systemübergreifend einmalig sind: Wenn die Nutzer Alice und Bob ohne Verbindung zueinander arbeiten, können sie Änderungen mit gleicher fortlaufender Nummer erzeugen - dies kann in selteneren Fällen auch im Online-Modus passieren. Durch das Hinzufügen des Nutzernamens zur ID können jedoch die Änderungen *alice.R51* und *bob.R51* voneinander unterschieden werden.

Die *targetObjectId* (hier *bob.S27*) entspricht der ID des Objektes, auf welches sich die Änderung bezieht. Wie zu sehen ist, sind die IDs von Objekten im Modell genau wie die Änderungs-IDs aufgebaut; der Präfix gibt an, dass das Objekt ursprünglich von Nutzer Bob erzeugt worden ist. Die *targetProperty* schließlich gibt den Namen des Attributs an, das verändert wird.

2.2.1 Wiederherstellung des Datenmodells

Zur Rekonstruktion des Objektmodells müssen die Änderungslisten der Nutzer gemischt (*merge*) werden. Dazu werden die Änderungen aus den einzelnen jlog-Dateien ausgelesen und in einer gemeinsamen Liste gespeichert. Die Änderungen in der Liste werden nun nach ihrer zeitlichen Abfolge, also nach der fortlaufenden Nummer in der Objekt-ID, sortiert. Sollten zwei Nummern gleich sein, so wird nach den Nutzernamen alphabetisch sortiert. Abb. 2 zeigt ein Merge-Beispiel mit den Änderungen der Nutzer Alice und Bob, wo die Änderung *alice.R24* zeitlich vor der Änderung *bob.R24* sortiert wird, da Alice weiter vorne im Alphabet als Bob liegt.

Sobald die Liste sortiert ist, können die Änderungen in dieser Reihenfolge angewandt werden. Haben mehrere Nutzer das gleiche Attribut eines Objekts verändert, liegt ein Merge-

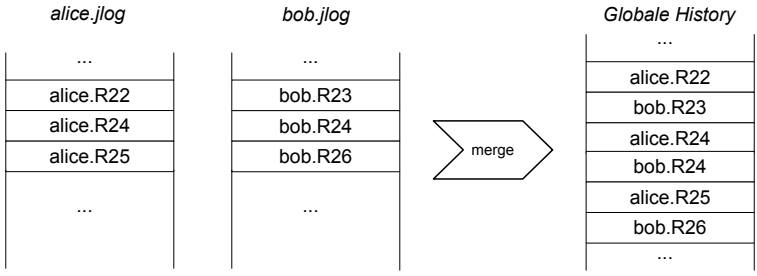

Abb. 2: Einfaches Merge-Beispiel mit ID-Nummern-Überschneidung

Konflikt vor. Ein solcher wird durch die Sortierung jedoch automatisch aufgelöst: Enthält die Änderungsliste mehrere Änderungen am gleichen Objekt und Attribut, so führt die konsekutive Anwendung der zeitlich sortierten Änderungen dazu, dass der Wert jenes Attributs mehrfach überschrieben wird und die zeitlich letzte Änderung an diesem Attribut schließlich den finalen Wert bestimmt.

2.3 Der Online-Modus

Sobald mehrere Nutzer zeitgleich eine Datenbasis bearbeiten, wird eine Socket-Verbindung zwischen ihnen aufgebaut und der Online-Modus gestartet. Dies ermöglicht eine unmittelbare Zusammenarbeit zwischen den Nutzern durch eine direkte Synchronisation der Modelldaten sowie der Übertragung zusätzlicher Nutzerdaten wie die Mauscursor-Positionen. Die Übertragung zusätzlicher Informationen ist dabei vom Synchronisationsmechanismus entkoppelt, sodass etwa die Mauscursor-Positionen nicht in die History aufgenommen werden und so mit höheren Update-Raten als andere Änderungen übertragen werden können. Damit alle Teilnehmer des Online-Modus von Beginn an synchron sind, tauschen die Editoranwendungen direkt nach dem Einrichten der Verbindung ihre Änderungshistorien untereinander aus.

2.3.1 Verbindungsherstellung

Auch wenn die Socket-Verbindung besteht, wird weiterhin die Synchronisation über die Repositorys sowie das regelmäßige Auslesen der jlog-Dateien durchgeführt. Zwar sind die Änderungen, die auf diese Weise registriert werden, in der Regel bereits durch die schnellere Online-Synchronisation bekannt und können verworfen werden; jedoch können die Repositorys zur Verbindungsherstellung genutzt werden, wenn Nutzerinformationen Teil des synchronisierten Datenmodells sind. So kann zum Beispiel für jeden Nutzer eine Boolean-Variable im Modell angelegt werden, die angibt, ob ein Nutzer gerade online (*true*) oder

offline (*false*) ist, das heißt ob ein Nutzer gerade den Editor verwendet sowie Netzwerkzugriff auf das globale Repository hat. Dazu kommen Attribute für das Hinterlegen von IP-Adresse und Port-Nummer. Sobald ein Nutzer über die Repositorys registriert, dass jenes *Online-Flag* eines anderen Nutzers auf *true* gesprungen ist, kann mit den Angaben von IP-Adresse und Port eine Socket-Verbindung zu diesem Nutzer hergestellt werden. Der Übergang vom Offline-Modus in den Online-Modus oder umgekehrt geschieht dabei fließend und im laufenden Betrieb.

2.3.2 Behandlung von Übertragungsfehlern

Bei der Übertragung der Änderungen über das Netzwerk kann es zu Fehlern kommen, sodass zum Beispiel Nachrichten verloren gehen und nie empfangen werden. Damit die Datenmodelle der verschiedenen Nutzer jedoch synchron zueinander bleiben, ist es nötig, dass jede einzelne Änderung von jedem Nutzer erfasst wird. Für die Echtzeit-Synchronisation im Online-Modus werden verlorengegangene Änderungen daher erneut über das Netzwerk angefordert.

Dazu enthält eine Change-Nachricht, die über das Netzwerk verschickt wird, eine Referenz auf die vorhergehende Nachricht. Wird eine Change-Nachricht empfangen, wird aus ihr die ID der vorhergehenden Nachricht ausgelesen. Ist diese ID in der eigenen History nicht verzeichnet, bedeutet dies, dass mindestens eine Nachricht verpasst wurde. In diesem Fall wird das nochmalige Senden der vorhergehenden Änderung angefordert. Dieser Vorgang wird rekursiv wiederholt, bis alle verpassten Nachrichten wieder eingefangen wurden. Diese Behandlung von Übertragungsfehlern wird dabei nur bei der Übertragung von Modelländerungen, nicht aber bei der Übertragung von GUI-Informationen wie Mauscursor-Positionen angewandt.

2.3.3 Netzwerktopologie

Die in Abschnitt 2.3.1 beschriebene Methode zur Verbindungsherstellung führt dazu, dass jeder Nutzer eine Netzwerkverbindung zu jedem anderen Nutzer aufbaut. Dies entspricht der Netzwerktopologie des vollständigen Graphen (*Full Mesh*). Diese Topologie hat zunächst den Vorteil der Adaptivität: Bei den Verbindungen beim kollaborativen Editieren handelt es sich schließlich nicht um ein Netzwerk mit konstanter Teilnehmerzahl, denn es können sich jederzeit Nutzer abmelden oder anmelden. Beim Full Mesh ist es recht einfach, im laufenden Betrieb einen neuen Teilnehmer in das Netz zu integrieren oder einen bestehen Teilnehmer zu entfernen, da dazu nur Verbindungen eingerichtet oder entfernt werden müssen, die nur jenen Teilnehmer betreffen. Bei einer Ring-Topologie hingegen müsste eine zusätzliche Softwarekomponente sicherstellen, dass beim An- und Abmelden von Nutzern die Ringstruktur bestehen bleibt.

Ein weiterer Vorteil der Full-Mesh-Topologie ist die hohe Ausfallsicherheit, denn der vollständige Graph hat den höchsten Zusammenhangsgrad. Der Zusammenhangsgrad gibt an, wieviele Leitungen unterbrochen werden können, ohne dass ein Teilnehmer von der

Kommunikation im Netz ausgeschlossen wird ([Ka96], S. 263ff.). Wenn eine direkte Verbindung zu einem Nutzer ausfällt, können diesen auch weiterhin Nachrichten über Umwege erreichen. Zwar gibt es hierfür kein Routing, doch werden Änderungsnachrichten ähnlich zum sogenannten *Flooding* ([Ta03], S. 393f.) im Netzwerk verteilt, indem noch nicht bekannte empfangene Nachrichten auch an die benachbarten Knoten weiterversendet werden. Zu beachten ist hier, dass sich das Flooding negativ auf die Performance auswirken kann und bei hohen Nutzerzahlen möglicherweise eingestellt werden muss.

3 Anwendungsbeispiel: UML-Editor

Um die praktische Anwendbarkeit des Synchronisationsmechanismus zu überprüfen, wurde eine Reihe von Beispielanwendungen in JavaFX entwickelt. Eine davon ist ein kollaborativer UML-Editor, der hier kurz vorgestellt werden soll. Abb. 3 zeigt einen Screenshot des Editors.

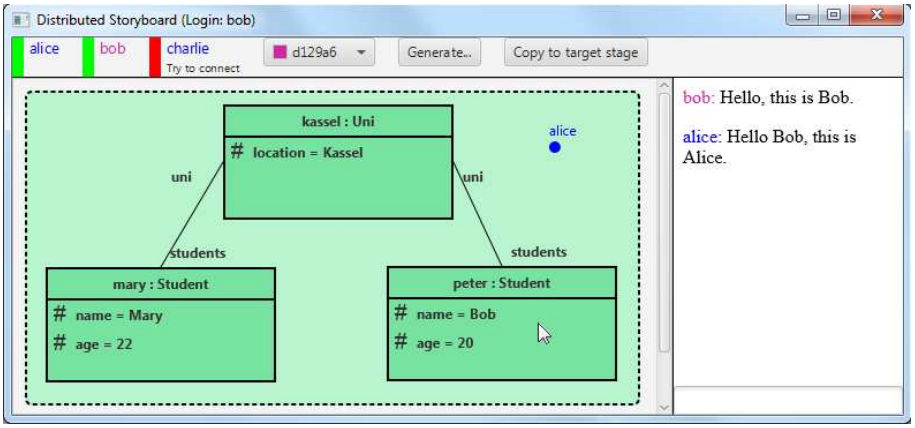

Abb. 3: Screenshot des UML-Editors

Mit dem UML-Editor kann ein Objektdiagramm gezeichnet werden, indem Objektkästen erzeugt werden, die mit Attributen sowie Attributwerten versehen werden können und mit Referenzen untereinander verknüpft werden können. Aus diesem Objektdiagramm wird dann ein Klassenmodell abgeleitet, für welches auch Java-Code generiert werden kann. Im Rahmen dieses Beitrags interessant sind jedoch hauptsächlich die kollaborativen Elemente des Editors.

In einer Leiste am oberen Fensterrand sind alle Nutzer aufgelistet, die an der Bearbeitung des UML-Diagramms beteiligt sind. In Abb. 3 sind dies die Nutzer Alice, Bob und Charlie. Ein farbiger Balken neben dem Nutzernamen gibt an, ob der Nutzer derzeit online (grün) oder offline (rot) ist. Hier sind Alice und Bob jeweils online und arbeiten daher im Online-Modus zusammen, während Charlie gerade offline ist. Gehen Nutzer online oder offline, ändert sich die Farbe der Balken zur Laufzeit entsprechend, sodass allen Nutzern stets bewusst ist, mit welchen anderen Nutzern sie derzeit zusammenarbeiten.

Abb. 3 zeigt den UML-Editor aus der Sicht von Nutzer Bob. Sein Mauszeiger ist der des jeweiligen Betriebssystems (hier Windows 7). Zusätzlich werden auch die Mauszeiger der anderen Nutzer in der GUI als mit dem Nutzernamen beschriftete farbige Punkte dargestellt: In Abb. 3 ist als Beispiel Alices Mauszeiger zu sehen. Durch die Darstellung der anderen Mauszeiger ist es einem Nutzer ersichtlich, welche Elemente derzeit von den anderen Nutzern bearbeitet werden. Auch können Nutzer so auf bestimmte Elemente zeigen, während sie zum Beispiel über IP-Telefonie miteinander kommunizieren.

Als weiteres Kommunikationsmittel zwischen den Nutzern dient eine einfache Chatleiste auf der rechten Seite des Fensters. Über ein Color-Picker-Element an der oberen Fensterleiste kann jeder Nutzer eine Farbe für sich wählen und jederzeit ändern. In dieser Farbe werden der Nutzername in der Nutzerleiste, der Nutzername in den Chatnachrichten sowie der Mauszeiger des Nutzers dargestellt. Die Idee hinter dem Farbschema ist, dass jeder Nutzer so Handlungen anderer Nutzer schneller ihrem Urheber zuordnen kann. So sind durch die unterschiedlichen Farben etwa die Mauszeiger schneller voneinander unterscheidbar.

Weiterhin wird das UML-Diagramm zwischen den Nutzern synchronisiert, bei den Nutzern im Online-Modus auch in Echtzeit: Wenn Alice etwa einen neuen Objektkasten in der Editoroberfläche platziert, ein neues Attribut hinzufügt oder eine neue Kante zieht, sind diese Änderungen auch sofort in Bobs GUI sichtbar. Weiterhin werden bidirektionale JavaFX-Bindings [Ja15] genutzt, um die Textfelder im Editor an die entsprechenden String-Attribute im Datenmodell zu binden, sodass der Inhalt dieser Felder buchstabenweise synchronisiert wird. Wenn also zum Beispiel Bob einen Rollennamen einer Referenz ändert, ist der Tippvorgang an dieser Stelle auch in Alices GUI sichtbar.

Für eine effektive Zusammenarbeit ist es wichtig, dass das Layout des UML-Diagramms in den GUIs aller Nutzer gleich ist. Daher wurden die x- und y-Position der Objektkästen ebenfalls mit in das Datenmodell aufgenommen, sodass auch Änderungen an diesen Positionsdaten durch den Synchronisationsmechanismus an alle Nutzer geschickt werden. Dies bedeutet, dass wenn etwa Alice einen Objektkasten des Diagramms auf ihrem Bildschirm mit der Maus verschiebt, diese Verschiebungsbewegung auch in der GUI von Bob zu sehen ist. Die Linien, welche die Referenzen darstellen, sind an den Kästen verankert und verschieben sich dadurch in allen GUI-Instanzen automatisch mit.

4 Fazit und Ausblick

Zwar wurden keine systematischen Peformancetests bezüglich Laufzeitverhalten und Objektmenge durchgeführt, jedoch deutet die testweise Nutzung des kollaborativen UML-Editors darauf hin, dass der Synchronisationsmechanismus zumindest für diesen Anwendungsfall gut funktioniert. Das kollaborative Editieren wurde mit drei Personen gleichzeitig getestet, wobei sich die Anwendung nicht langsamer anfühlte als ein Einzelnutzer-Editor. Auch der fliegende Wechsel zwischen Online- und Offline-Modus verhielt sich wie gewünscht.

Für die Zukunft ist eine Reihe von Verbesserungen am Synchronisationsmechanismus denkbar. Eine mögliche Problematik ist, dass die jlog-Dateien immer weiter anwachsen, sodass bei langfristiger Nutzung eines Editors das Auslesen der Dateien zu viel Zeit in Anspruch nehmen kann. Die Dateien enthalten dabei viele alte Änderungen, die für die Rekonstruktion des Datenmodells oft nicht mehr relevant sind. Dementsprechend wäre eine Kompression der Dateien wünschenswert, welche obsolet gewordene Änderungen entfernt und den Umfang der jlog-Dateien entsprechend reduziert.

Eine weitere noch nötige Funktionalität ist das Löschen von Objekten, das in der momentanen Objektverwaltung von SDMLib noch nicht möglich ist. Sobald das Löschen von Objekten aus dem Objektmodell zwischen den Nutzern synchronisiert werden kann, können auch Undo- und Redo-Operationen implementiert werden, die für Editoren in der Regel von wesentlicher Bedeutung sind.

Weiterhin ist ein Problem, dass Änderungen an einem komplexen Datenmodell oft nicht atomar sind, sondern aus mehreren aufeinanderfolgenden Änderungen bestehen. Diese Änderungen sollten zu Gruppen zusammengefasst werden und mit dem Transaktionsprinzip ausgeführt werden können, um die Konsistenz des Datenmodells zu gewährleisten.

Wünschenswert wäre darüberhinaus die Integration von Ressourcen: Die Nutzer sollten in den Editoren Ressourcen wie Bilder, Musikdateien oder ähnliches einfügen können. Dazu muss der Mechanismus derart erweitert werden, dass solche Ressourcen verwaltet und unter den Nutzern verteilt werden können.

Ein wichtiger Punkt ist schließlich noch die Synchronisation sortierter Listen wie etwa Javas `ArrayList`. Die Objektverwaltung von SDMLib unterstützt bislang nur unsortierte Sets. Sortierte Listen sind jedoch wichtig, wenn Objekte zum Beispiel in einer bestimmten Reihenfolge in der GUI dargestellt werden sollen, sodass alle Nutzer eines kollaborativen Editors die Objekte in genau dieser Reihenfolge sehen können.

Literaturverzeichnis

[EG89] Ellis, C. A.; Gibbs, S. J.: Concurrency Control in Groupware Systems. International Conferfence on Management of Data (SIGMOD), 1989.

[Fr09] Fraser, N.: Differential Synchronisation. DocEng 09, 2009.

[Ge11] ShareJS. Live concurrent editing in your app, sharejs.org/, Stand: 25.02.2015.

[Ja15] JavaFX Bindings, doc.oracle.com/javase/8/javafx/api/javafx/beans/binding/Bindings.html, Stand: 25.02.2015.

[Ka96] Kauffels, F.-J.: Lokale Netze. Datacom, Bergheim, 1996.

[Li12] Lindel, S.: Online synchronization of replicated models. CVSM2012, 2012.

[St11] Starke, G.: Effektive Software-Architekturen. Carl Hanser, München, 2011.

[Ta03] Tanenbaum, A. S.: Computernetzwerke. Pearson Studium, München, 2003.