Universität Ulm
Fakultät für Informatik

Konfliktresolution bei der Datensynchronisation

Diplomarbeit

in der Medieninformatik

vorgelegt von
Georg Günther Alexander Traud
am 31. März 2005

Gutachter

Prof. Dr. M. Weber
Dr. F. Kargl

Inhaltsverzeichnis

1. Zusammenfassung

Diese Diplomarbeit untersucht Abgleichverfahren für Datenbestände (Dateisysteme, Kalender, Adressbücher usw.) auf ihre Schwächen. Dazu werden Kriterien entwickelt, um einzelne Verfahren einteilen und beurteilen zu können. Ausgehend von den vorgefundenen Schwächen wird das Problem angegangen, Datenbestände in beliebigen Netzwerktopologien abzugleichen, auch dann wenn einzelne Kopien nicht dauerhaft verfügbar sind. Hierzu wird der *SyncML* Industriestandard für Abgleichverfahren um die Unterstützung für beliebige Graphen in der Netzwerktopologie erweitert.

Teil I.

Grundlagen

2. Einleitung

Der moderne Mensch muss eine Vielzahl von Aufgaben bewältigen und darf dabei nicht den Überblick verlieren. Um den Tagesablauf planen zu können, trägt fast jeder eine Uhr mit sich. Viele führen einen Kalender, um Termine zu koordinieren. Auch besitzen viele ein Adressbuch, in dem Verwandte, Freunde und Bekannte aber auch Arbeitskollegen festgehalten sind.

In Zeiten von Computern, Personal Digital Assistants (PDA) und Mobiltelefonen müssen diese Kalender und Adressbücher nicht mehr mit einem Stift auf Papier geschrieben werden, sondern können elektronisch erfasst werden. Aber welche Vorteile entstehen dadurch?

In dieser Ausarbeitung wird durchgängig das Beispiel einer kleinen mittelständische Firma namens *Schraubendreher* genutzt. Die Firma vertreibt verschiedene Arten von Schrauben an andere Firmen. Wir lernen *Herrn Maier* kennen, der Aufträge von Kunden entgegen nimmt, den Preis aushandelt und dann im Lager die entsprechenden Schrauben und deren Stückzahl bestellt. Die Firma Schraubendreher hat eine große Anzahl Stammkunden.

Beispiel: *Heute ruft die Firma Autobauer an und benötigt wie beim letzten Mal die gleiche Anzahl Schrauben. Herr Maier nutzt zur Bestellungsaufnahme seinen Computer. Da die Firma Autobauer bereits einmal bestellt hat, kann er die Adresse der Firma und die letzte Lieferung schnell auffinden und er fügt zu diesem Datenbestand die neue Lieferung hinzu. Da das Lager direkt an den Computer von Herrn Maier angebunden ist, kann er sofort Aussagen über die Lieferbarkeit der gewünschten Schrauben machen. Der Mitarbeiter im Lager sieht an seinem Computer die neue Bestellung und macht sie für den Versand fertig.*

Es lohnt sich also die Adressdaten elektronisch zu verwalten.

Heute besitzt fast jeder ein Mobiltelefon, welches nahe liegend auch über ein Adressbuch verfügt. Man trägt schließlich nicht das gesamte, gedruckte Telefonbuch mit sich, man will nicht vor jedem Anruf eine teure und Zeit raubende Telefonauskunft anrufen und nicht jeder kann die langen Telefonnummern aller seiner Kontakte auswendig lernen.

Mit einem Adressbuch direkt im Mobiltelefon sucht man den Namen heraus und lässt sich dann verbinden.

Aber auch Kalenderdaten lassen sich sehr bequem elektronisch verwalten.

Beispiel: *Herr Maier will nächste Woche ein Verkaufsgespräch führen und braucht dafür den Konferenzraum der Firma. Er macht gerade den Termin mit einem Kunden und stellt dabei an seinem Computer fest, dass der Raum am Donnerstag durch eine Fortbildung belegt ist. Entsprechend vereinbart Herr Maier als Termin den nächsten Freitag und belegt gleich daraufhin den Raum, so dass es zu keinen Überschneidungen kommen kann.*

Auch hier lohnt die elektronische Erfassung der Daten. Man kann durch eine geschickte Terminverwaltung Zeit sparen und Ressourcen besser einplanen.

Wie wir aber an diesen Beispielen bereits gesehen haben, will man seine Adress- und Termindaten nicht nur auf einem Computer in der Arbeit speichern, sondern vielleicht auch im Computer zu Hause, auf dem Mobiltelefon, im PDA oder gar in einer Armbanduhr. Die Datensynchronisation beschränkt sich aber nicht nur auf Kontakte und Termine. Weitere Anwendungsfelder sind zum Beispiel die E-Mail Verwaltung. Es werden immer mehr Haushaltsgegenstände untereinander vernetzt. So könnte man sich auch vorstellen, die Einkaufsliste am Kühlschrank einzusehen. Die Einkaufsliste wird dann mit dem Mobiltelefon abgeglichen, welches man beim Einkauf dabei haben wird.

Die Daten auf allen Geräten einzeln einzutragen, ist mühselig und fehleranfällig, da auf den meist kleinen Tastaturen schell Tippfehler entstehen. Es wäre daher wünschenswert, einen Termin nur an einem Gerät einzutagen, der dann allen anderen Geräten bekannt gemacht wird, die davon wissen sollten. Der Benutzer soll an dem Gerät, welches er gerade nutzt den aktuellen Datenbestand aller Geräte vorfinden können.

Aufbau dieser Ausarbeitung

Kapitel 3 zeigt die zwei Ansätze für Abgleichverfahren auf. In Kapitel 4 werden Kriterien aufgezeigt, die dazu dienen einen Kompromiss zwischen verschiedenen Komplexitätsstufen eines Abgleichverfahrens zu finden. In Teil II werden einzelne Verfahren vorgestellt und anhand von Kriterien untersucht. Am Ende werden noch offene Problemstellungen aufzeigt. Teil III beschäftigt sich mit der Umsetzung eines Abgleichverfahrens, um ein Verfahren aus der Industrie zu erweitern. Dieser Teil ist untergliedert in Kapitel 11, welches das schrittweise Herangehen an diese Erweiterung aufzeigt, Kapitel 12 erläutert die Umsetzung im Detail und in Kapitel 13 werden die Tests erläutert, die

zur Überprüfung der Erweiterung herangezogen worden sind. In den letzten Kapiteln 14 und 15 werden weitere mögliche Erweiterungen aufgezeigt und ein Fazit gezogen.

3. Grundlagen

3.1. Definitionen

Diese Diplomarbeit beschäftigt sich mit dem Thema *Datensynchronisation* und dieser Abschnitt definiert, was unter diesem Begriff im Folgenden zu verstehen ist.

3.1.1. Daten

Zusammengehörende Daten bilden einen Datensatz. Datensätze gleicher Art bilden eine Datenbank. Wann Daten zusammengehören und wann Datensätze gleicher Art sind, ist abhängig von dem jeweiligen Nutzen.

Beispiel: *Daten wie* Herr, Maier *und* Firma Schraubendreher *ergeben zusammen den Datensatz von Herrn Maier in einer Kontaktdatenbank.*

Daten selbst können auf verschiedene Arten vorliegen. In dieser Ausarbeitung sind es elektronische Daten unterschiedlichsten Inhalts. Daten werden immer zu Datensätzen zusammengefasst. Datensätze können andere Datensätze enthalten, liegen aber immer in einer Datenbank vor.

3.1.1.1. Hierarchisch

Die Datensätze können hierarchisch in der Datenbank vorliegen.

Beispiel: *In einem Dateisystem ist die Datenbank das Dateisystem selbst und die einzelnen Datensätze sind die abgelegten Dateien in Ordnern und Unterordnern. Ordner und Unterordner sind wiederrum Datensätze, die andere Dateien enthalten können.*

Beispiel: *In einem XML [W3C04b] Dokument liegen einzelne Elemente hierarchisch zueinander. Das XML Dokument repräsentiert eine Datenbank une die jeweiligen elemente sind Datensätze.*

3.1.1.2. Flach

Bei einer flachen Datenbank sind alle Datensätze auf der gleichen Ebene.

Reine Bytes Eine Datenbank enthält genau einen Datensatz und dieser besteht aus reinen Bytes ohne weitere Unterteilung (*Datenhaufen*). Erst eine spezielle Applikation kann diesen Datenhaufen wieder dekodieren.

Beispiel: *Ein Bilddatenformat wie zum Beispiel GIF oder ein Kompressionsdatenformat wie zum Beispiel ZIP stellen Datenhaufen dar.*

Tupeldaten Zum Beispiel wird in einer Kontaktdatenbank von Geschäftspartnern, jeder Geschäftspartner durch einen Datensatz repräsentiert, der wiederum aus Tupeln, d.h. einem Name:Wert Paar besteht. Diese Art von Datenbank ist bei Verwendung der Datenformate vCard [ver96b] und vCalendar [ver96a] üblich.

```
BEGIN:VCALENDAR
   VERSION:1.0
   BEGIN:VEVENT
      CATEGORIES:MEETING
      STATUS:NEEDS ACTION
      DTSTART:20050401T073000Z
      DTEND:20050401T083000Z
      SUMMARY:Enkel DS presentation
      DESCRIPTION:Giving an overview over Enkel DS 1.0
      CLASS:PUBLIC
   END:VEVENT
   BEGIN:VTODO
      SUMMARY:Prepare presentation
      DUE:20050401T073000Z
      STATUS:NEEDS ACTION
   END:VTODO
END:VCALENDAR
```

3.1.1.3. (Relationale) Datenbank

In einer klassischen Datenbank stehen Objekte mit Attributen in einer gewissen Relation zueinander. Wichtig ist hierbei, dass Informationen so wenig wie möglich mehrmals auftreten, damit bei Änderungen bestenfalls nur ein Feld mit dieser Information betroffen ist. Datenbanksysteme sind besonders für eine große Anzahl an Datensätzen geeignet, da mit diesen niedrigere Zugriffszeiten zu erwarten sind.

3.1.1.4. Zusammenfassung

Diese Auflistungen verschiedener Datenbank Formen soll zeigen, dass im Folgenden keine Beschränkung auf den Inhalt von Daten gemacht wird. Datenbanken können Dateisysteme, Listen von vCard/vCalendar Dateien, XML Dateien, relationale Datenbanken usw. darstellen. Die einzigen Einschränkungen sind, dass Daten elektronisch vorliegen, Datensätze erzeugen und in einer Datenbank vorliegen.

3.1.2. Kopie versus Gerät

Eine Datenbank liegt in einer Kopie vor. Eine Kopie (*Replik*) ist in den meisten Fällen ein Gerät, wie zum Beispiel ein Mobiltelefon, ein PDA, ein Computer usw. Im Folgenden wird daher Gerät und Kopie gleich gesetzt obwohl es einen Unterschied gibt. Ein Gerät kann auch mehrere Kopien derselben Datenbank enthalten. Dies kann vorkommen, wenn eine Sicherungskopie im Gerät selbst erstellt wird oder während Tests, wenn keine weiteren Geräte vorliegen. Diese Fälle sind so speziell, dass im Folgenden zur Vermeidung von Missverständnissen trotzdem Kopie und Gerät gleichgesetzt wird.

3.1.3. Synchronisation

Es gibt zwei Definitionen von Synchronisation, die in der Literatur vorkommen.

- Zwei-Wege Synchronisation
  Bei einer Synchronisation werden die jeweiligen Daten abgeglichen, d.h. nach der Synchronisation einer Datenbank von zwei Geräten, sind die Datenbanken auf beiden Seiten gleich. Erfolgt sofort wieder eine Synchronisation, müssen keine Änderungen vorgenommen werden.

- Ein-Weg Synchronisation
  Die Änderungen werden von Gerät A an ein anderes Gerät B weiter gegeben. Gerät B gibt aber nicht automatisch seine Änderungen an Gerät A weiter. Gerät B hat folglich nach dem Abgleich den aktuelleren Datenbestand. Dies wird in der Literatur [CJ04] *B synchronisiert von A* genannt. Eine Zwei-Wege Synchronisation wird dadurch erzeugt, dass man von A nach B und dann von B nach A synchronisiert.

Die Ein-Weg Definition ist flexibler, da weitere Datensynchronisations-Szenarien möglich sind. So sind auch reine Datensenken möglich, d.h. es gibt ein Gerät C welches Änderungen von A oder B bezieht, selbst aber keine Änderungen weiter gibt.

Beispiel: *Im Lager der Firma Schraubendreher gibt es einen Praktikanten, der sich mit dem Computersystem vertraut macht. Der Praktikant macht im Rahmen seiner Ausbildung noch Fehler und die Daten des Computersystems sind nur Übungsmaterial aber kein Datenmaterial, auf dem weiter gearbeitet wird. Trotzdem soll das System echte Daten aus dem täglichen Ablauf enthalten. Der Praktikant gleicht seinen Computer ab und neue Daten und Änderungen erreichen seinen Computer. Änderungen, die der Praktikant vorgenommen hat, werden (noch) nicht von den anderen Computern übernommen, da der Praktikant noch Fehler macht und diese den Betriebsablauf nicht stören sollen.*

Weiterführende Arbeiten sollten untersuchen, ob die Ein-Weg Synchronisation in beide Richtung ausgeführt tatsächlich immer das gleiche Endergebnis liefert wie eine Zwei-Wege Synchronisation: $(A \rightarrow B) + (B \rightarrow A) = (A \leftrightarrow B)$. Ist zum Beispiel noch ein drittes Gerät an der Synchronisation beteiligt, könnte dieses Auswirkungen auf den Ablauf des Verfahrens haben, je nachdem wann mit diesem synchronisiert wird (*Race conditions*). Die Literatur liefert einen Beweis für die Äquivalenz von Ein-Weg und Zwei-Wege Synchronisation noch nicht.

Falls die Bedeutung der Datensynchronisation in den jeweiligen Veröffentlichungen nicht definiert ist, dann scheinen sehr viele die Zwei-Wege Synchronisation implizit anzunehmen. Die Ein-Weg Synchronisation wird von [CJ04; PV04] explizit definiert.

Im Folgenden wird die Zwei-Wege Synchronisation als Normalfall angenommen. Nur wenn gesondert darauf hingewiesen wird, handelt es sich um die Ein-Weg Synchronisation. Dies deckt sich mit den Erwartungen eines Benutzers eines Abgleichverfahrens, denn der Nutzer erwartet gleiche Datenbestände auf beiden Seiten nach dem Abgleich.

3.2. Motivation

Bis jetzt ist erläutert worden, was man sich grob unter Datensynchronisation vorstellen kann und welche Ergebnisse ein Abgleich von Daten erzeugt. Dass der Benutzer an dem Gerät, an dem er gerade arbeitet, immer die aktuellsten Daten vorfindet, kann man allerdings auch durch andere Verfahren lösen. Dieser Abschnitt wird eine Abgrenzung der Datensynchronisation zu anderen Verfahren vornehmen und eine Motivation liefern, warum der Einsatz der Datensynchronisation bei dieser Art von verteilten Datenbanken sinnvoll ist.

Verteilte Datenbanken und gemeinschaftliche Arbeitswerkzeuge (*Computer-Supported Cooperative Work*) sind zwei Bereiche der Informatik, die sich sehr intensiv mit dauerhaft verbundenen Geräten beschäftigen. Ein gemeinschaftliches Werkzeug wäre zum Bei-

spiel eine Textverarbeitung innerhalb einer Programmierumgebung. Mehrere Benutzer arbeiten an einem Projekt. Es kann vorkommen, dass innerhalb eines Projekts mehrere Benutzer gleichzeitig an derselben Datei arbeiten.

3.2.1. Pessimistische Replikation

Beispiel: *Herr Maier arbeitet an einem Computer in der Firma Schraubendreher. Da alle Computer untereinander vernetzt sind und es eine übergreifende Zugriffkontrolle gibt, ist es Herrn Maier möglich, sich an jeden beliebigen Computer in der Firma zu setzen und auf seine Daten zu zugreifen. Er muss sich dazu nur am Gesamtssystem anmelden und es erscheinen alle seine Daten und Einstellungen.*

In diesem Szenario ist eine Datensynchronisation nicht nötig. Nicht jeder Computer an dem Herr Maier einmal saß, braucht eine Kopie aller seiner Daten. Da die Computer dauerhaft miteinander verbunden sind, kann auf einen gemeinsamen Datenbestand zugegriffen werden. Dieser Datenbestand kann im gesamten Netzwerk verteilt sein, liegt aber meist auf einem zentralen Rechner (Server) oder aber auf dem üblichen Arbeitsplatz von Herrn Maier. Wenn dann ein Datensatz (eine Datei) aufgerufen wird, gilt es Schreib/Schreib Konflikte zu vermeiden. D.h. es darf nicht sein, dass zwei Autoren Änderungen an einem Datensatz gleichzeitig durchführen, da es dann schwierig wird, diese Änderungen wieder in Einklang zu bringen.

In Verteilten Datenbanken und gemeinschaftlichen Arbeitswerkzeugen ist der Einsatz von Schreib- aber auch Lesesperren dafür eine Lösung. Diese Sperren werden beim Öffnen des Datensatzes gesetzt, so dass kein anderer Schreiber den Datensatz ändern kann.

In diesem Szenario gilt es, Sperren geschickt zu setzen, und die Frage zu beantworten, wann die Sperren wieder aufgehoben werden können. Es gilt aber auch Probleme zu vermeiden, die durch Sperren auftreten. Unter anderem können Verklemmungen bei zwei wechselseitigen Sperren entstehen.

Hier ist zu betonen, dass sich die Bedeutung von *Datensatz* durch das System ergibt. Es kann sein, dass die ganze Datenbank (z.B. ein Dateisystem), dass nur einzelne Dateien, nur ein kleiner Bereich innerhalb einer Datei oder gar nur eine Zeile in einer Textdatei gesperrt werden.

3.2.2. Optimistische Replikation

Beispiel: *Herr Maier hat mit seiner Firma ausgemacht, dass er gewisse Tage von zu Hause aus arbeiten darf. Um von zu Hause auf seine Daten zuzugreifen, muss er dazu während seiner Tätigkeit ständig mit dem Firmennetz verbunden sein. Damit Herr Maier für ganz wichtige Geschäftskunden auf seinem Mobiltelefon erreichbar ist, hat er von der Firma einen Personal Digital Assistant (PDA) erhalten. In diesem kann Herr Maier Bestellungen aufnehmen, die ihn unterwegs erreichen, wenn er gerade nicht im Büro oder zu Hause ist. Allerdings muss die Firma Schraubendreher auch an die Kosten denken. Der PDA wählt sich während der Bestellungsaufnahme nicht automatisch in das Netzwerk der Firma ein, sondern erst wenn Herr Maier in die Firma kommt, werden die Daten übernommen.*

Im ersten Fall war der Computer dauerhaft mit einem Netzwerk verbunden. Der zweite Fall stellt ein anderes Extrem dar, denn ein Gerät (der PDA) ist nur selten mit dem Netzwerk verbunden. Wenn die Geräte nicht dauerhaft untereinander verbunden sind, dann greift ein Konzept mit Sperren nicht mehr. In diesem Fall bietet sich das Konzept der *optimistischen Replikation* an. Auf allen Geräten befindet sich eine Kopie der Daten. Diese Kopie kann vollständig sein, es können aber auch nur Teile vorhanden sein. Wenn ein Datensatz geändert wird, dann sind die restlichen Kopien erst einmal nicht aktuell und korrekt. Wenn ein neuer Datensatz entsteht, dann fehlen den restlichen Kopien Informationen. Wenn ein Datensatz gelöscht wird, dann besitzen alle anderen Kopien zuviele Daten. Erst wenn die geänderte Datenbank wieder mit den anderen Kopien in Verbindung steht, können Ersetzungen, Hinzufügungen und Löschungen weiter gegeben werden. Die Kopien, die Änderungen erhalten haben, geben diese wiederum bei Verbindungen mit anderen Kopien weiter. Auf diese Weise werden die Änderungen an alle anderen Kopien im Gesamtsystem weiter gegeben.

3.2.2.1. Schreib/Schreib Problem

Diese Beschreibung der optimistischen Replikation löst allerdings noch nicht den Fall auf, wenn es zu einem Schreib/Schreib Problem kommt. In diesem Fall fanden mindestens zwei Änderungen am selben Datensatz auf mindestens zwei verschiedenen Kopien statt. In der optimistischen Replikation handelt es sich genauer gesagt um ein Aktualisierungsproblem. Im Folgenden wird dieses Problem aber wie in der pessimistischen Replikation *Schreib/Schreib Problem* genannt.

Beispiel: *Herr Maier ist gerade unterwegs und ändert auf seinem PDA die Bestellung der Firma Lacknit. Diese hatte ursprünglich 500 Schrauben bestellt, ändert nun aber die Bestellung und will die doppelte Menge geliefert bekommen. Als Herr Maier wieder in der Firma ankommt, konnte er noch nicht einmal seine Tasche (mit dem PDA) auspacken, da ruft die Firma Lacknit erneut an und bittet um 800 Schrauben. Herr Maier ändert die Daten entsprechend in seinem Computer. Nach dem Gespräch packt Herr Maier seinen PDA aus, um die Änderungen vom Vortag in den Computer einzuspielen.*

Die neuen und geänderten Daten der jeweiligen Systeme müssen miteinander wieder in Klang gebracht werden. Das heißt die *Daten* müssen wieder *synchronisiert* werden. Dabei überprüft ein Synchronisationsverfahren, ob in den jeweiligen Datenbanken dieselben Datensätze seit der letzten Synchronisation geändert worden sind. In unserem Fall gab es einen Schrei/Schreib Konflikt auf dem Datensatz mit der Bestellung der Firma Lacknit. Im Computer stehen 800 Schrauben und im PDA stehen 500 Schrauben. Die anderen Änderungen waren unproblematisch und können ausgetauscht werden. Es herrscht ein Konflikt auf diesem Datensatz und um diesen aufzulösen, muss ein Synchronisationsverfahren eine *Konfliktresolution* anbieten. Wie eine Konfliktresolution aussehen könnte und wie überhaupt ein Konflikt festgestellt wird, werden die nächsten Kapitel zeigen. Diese Konfliktresolution ist nötig, da nicht wie bei dauerhaft verbunden Systemen mit Sperren gearbeitet wird, denn Datensätze sind nach deren Aktualisierung nicht auf allen Geräten sofort in Einklang.

3.2.2.2. Lese/Schreib Problem

Allerdings tritt in der Datensynchronisation neben dem Schreib/Schreib Problem auch noch das Lese/Schreib Problem auf.

Beispiel: *Frau Schmidt ist in der Firma Schraubendreher für die Gehaltsabrechnungen zuständig und verwaltet unter anderem die Überstunden. Heute ist Abrechnungstag und Frau Schmidt greift auf die Arbeitszeiten der Kollegen zu. Der Computer errechnet dann die Überstunden und Frau Schmidt verbucht dies auf dem jeweiligen Gehaltskonto. Herr Maier hatte allerdings gestern Abend noch eine Überstunde und vermerkte dies auf seinem PDA. Da aber die Firma Lacknit angerufen hatte, kam Herr Maier noch nicht dazu, seine Überstunde von gestern ins System zu übertragen. Nun hat Herr Maier seine Überstunden übertragen. Frau Schmidt hat aber bereits die Aufsummierung der Stunden für den letzten Monat abgeschlossen und hat leider auf falschen Daten gearbeitet.*

Die Definition der optimistischen Replikation besagt, dass Kopien unvollständig sein können. Frau Schmidt hat auf die unvollständigen Daten im Computer von Herrn Maier zugegriffen und darauf eine Berechnung ausgeführt. Diese Berechnung lieferte aufgrund der Unvollständigkeit der Daten ein falsches Ergebnis.

Man kann sich für dieses Szenario recht schnell eine mögliche Lösung überlegen. Nur wenn alle Datenbestände vollständig sind, kann eine Berechnung durchgeführt werden. Wann weiß aber das System, dass eine Kopie vollständig ist? Hierzu wäre eine globale Sicht auf alle Kopien nötig. Sind alle Kopien synchronisiert worden? Diese globale Sicht ist sehr schwer herzustellen. War da nicht einmal ein PDA vor zwei Jahren, der dann verloren gegangen ist?

Beispiel: *Herr Maier nimmt an seinem Computer eine Bestellung über 125 Schrauben auf. Abends ruft wieder die Firma Lacknit an und bestellt nun doch 250 Schrauben. Die Bestellung eilt aber nicht sehr und sie soll erst nächste Woche ausgeführt werden. Dies trägt Herr Maier in seinen PDA ein. Am nächsten Morgen hat Herr Maier frei und der PDA wird nicht wie sonst mit dem Computer in der Firma abgeglichen. Seine Vertretung Herr Starck nimmt an diesem Tag den Anruf der Firma Lacknit an:* Wir hätten doch gerne erstmal nur 125 Schrauben. *Herr Starck schaut in Herrn Maiers Computer nach und sieht die ursprüngliche Bestellung: 125 Schrauben. Diese lässt er dann unverändert so stehen, schließlich will der Kunde ja 125 Schrauben.*

Was passiert wenn Herr Maier wieder mit seinem PDA in die Firma kommt? Wird ein Konflikt erkannt? Einem Datensynchronisationsverfahren fehlen die nötigen Informationen, einen Lese/Schreib Konflikt zu erkennen. Der Datensatz im Computer ist seit der letzten Synchronisation nicht geändert worden. In allen bekannten Synchronisationsverfahren wird auch kein Konflikt erkannt und der Datensatz aus dem PDA mit 250 Schrauben wird übernommen. Aus Sicht des Computers gibt es keinen Konflikt. Selbst aus Sicht der Firma Schraubendreher existiert kein Konflikt außer Herr Starck und Herr Maier tauschen sich untereinander aus.

Dieses Beispiel ist sehr desillusionierend, zeigt es doch, dass ein noch so gutes Synchronisationsverfahren nicht alle Konflikte und Nutzeränderungen aufspüren kann. Bei diesem Lese/Schreib Problem versagt auch ein pessimistisches Verfahren, außer es werden für Leser eines Datensatzes ebenfalls Sperren eingerichtet, es wird also ein exklusiver Zugriff auf die Datensätze realisiert. Diese Kompromisse müssen bei der Anwendung der optimistischen Replikation möglich sein. Hat ein System andere Anforderungen, müssen andere Lösungen gefunden werden.

4. Kriterien für Synchronisationsverfahren

Sinn dieses Kapitel ist es, Synchronisationsverfahren anhand von Kriterien einteilen zu können. Im Bereich der wissenschaftlichen Forschung und im Bereich von Standardisierungsgremien sind eine Vielzahl unterschiedlichster Verfahren entstanden, die oft verschiedene Ziele verfolgen. Wie im vorherigen Kapitel dargelegt, kann ein Verfahren kaum alle Ziele erreichen. Ein Verfahren muss sich auf bestimmte Rahmenparameter beschränken, damit es ordnungsgemäß arbeiten kann. Beim Entwurf eines Synchronisationsverfahrens werden viele dieser Entscheidungen nicht explizit sondern implizit gefällt.

Wenn Kompromisse explizit getroffen werden, finden sich die Begründungen nur in den seltensten Fällen in den Spezifikationen oder Dokumentationen der jeweiligen Verfahren wieder. Gerade bei Industriestandards wird es oft als unvorteilhaft angesehen, Aspekte zu nennen, die nicht erfüllt werden. Schließlich soll aufgrund überlegender Fähigkeiten die Spezifikationen von so vielen anderen Herstellern übernommen werden wie möglich. Wenn Entscheidungen implizit gefällt werden, dann kann es sein, dass diese aufgrund von Unwissenheit oder durch Annahmen aus einer Erfahrung heraus entstehen.

Die nachfolgenden Kriterien sollen einen Fragenkatalog ermöglichen. Welche Verfahren kann man in welchem Punkt vergleichen oder welche Verfahren sind so Grund verschieden, so dass ein Vergleich schwer fällt? Ein solcher Fragekatalog beantwortet, was ein bestimmtes Verfahren überhaupt leisten kann und ob etwaige Ansprüche an ein Verfahren nicht überzogen sind. Nicht zuletzt dient der Katalog zur Überprüfung, ob ein entworfenes Synchronisationsverfahren überhaupt die angestrebten Ziele erreicht.

4.1. Netzwerktopologie

Die folgenden Betrachtungen finden im ISO/OSI Referenzmodell [Tan03] auf der Anwendungsschicht statt. D.h. es geht darum wie die Kopien logisch in Beziehung zueinander stehen und nicht wie sie tatsächlich physikalisch untereinander verbunden sind. Ob die Kopien über Kabel, Funk oder Licht verbunden sind und ob sie über die ganze Welt verteilt oder in einem kleinen Raum stehen, spielt hier keine Rolle. Nur logische Verbindungen zwischen Kopie A und B sind als Verbindungen in den folgenden Netz-

werktopologien aufgezeigt. Dies bedeutet, dass nicht unbedingt eine direkte physikalische Verbindung zwischen den Kopien geben muss. Andere Geräte können als Zwischenstation dienen, um die logische Verbindung herzustellen. Es muss nur sichergestellt sein, dass irgendwann die Nachrichten Pakete eines Abgleichverfahrens gesichert von A nach B gekommen sind.

Anders formuliert, Kopie A und B müssen nur einmal in ihrer Existenz eine physikalische Verbindung (möglicherweise auch über andere Geräte) aufgebaut und ein Synchronisationsverfahren durchlaufen haben. Bereits dann existiert für immer eine logische Verbindung zwischen diesen beiden Kopien.

Im Folgenden werden typische Topologien wie die 1:1, Stern, Hierarchie und beliebiger Graph mit Zyklen angeführt und mit den jeweiligen Vor- und Nachteilen diskutiert [DBMP01; CP00].

4.1.1. 1:1

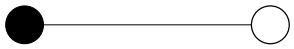

Abbildung 4.1.: 1:1 Topologie – ein Server und ein Klient

Es gibt im Gesamtsystem nur zwei Kopien der Daten. Die beiden Kopien sind untereinander verbunden. Die Daten werden mit keinen weiteren Kopien abgeglichen. Dies ist die einfachste Topologie und bietet sich an, wenn man einen Rechner auf der Arbeit mit einem Rechner daheim abgleichen will. Es gibt durchaus einige Abgleichverfahren [PV04; rsy93], die nur diesen Fall abdecken.

4.1.2. Stern (ein Hauptserver)

Alle Kopien sind genau einmal verbunden und zwar alle mit der gleichen Kopie. Dies entspricht dem Klient/Server Modell. Es gibt einen Server an dem sich mehrere Klienten anmelden. Der zentrale Rechner ist als Anlaufstelle dafür zuständig, dass eine Änderung, die an ihm oder einem Klienten erfolgt ist, an alle anderen Klienten weitergegeben wird. Dies ist zurzeit die üblichste Variante unter den Abgleichverfahren. Nur wenige Verfahren nehmen sich komplexere Szenarien als Ziel vor.

4.1.2.1. Vorteile

Die Klienten können einfachere Geräte sein, denn aus deren Sicht ist es eine reine 1:1 Kommunikation. Nur der zentrale Rechner muss die 1:n Situation verwalten können.

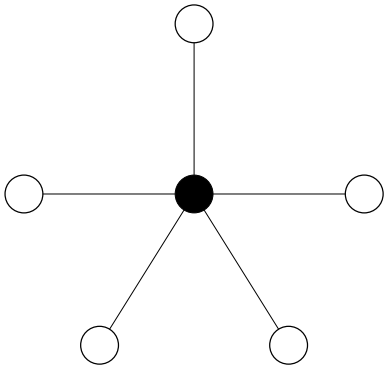

Abbildung 4.2.: Stern Topologie – ein Server, mehrere Klienten

Die Klienten Rolle ist ideal für mobile Endgeräte, die über weniger Speicher- und Rechenkapazitäten verfügen als der stationäre Rechner. Der zentrale Rechner ist auch eine eindeutige Anlaufstelle, um die neusten und vollständigsten Datenbestände zu erhalten. Er kann aufwendige Konfliktresolutionen alleine durchführen und ist auf keine anderen Klienten angewiesen.

4.1.2.2. Nachteile

Die gesamte Kommunikation muss immer über den zentralen Rechner erfolgen. Es findet keine direkte Kommunikation zwischen den Klienten statt. Dieses System hat folglich einen *single point of failure*, d.h. wenn der zentrale Rechner ausfällt, kann die Synchronisation nicht mehr fortgeführt werden. Der zentrale Rechner kann durch zu viele Anfragen von Klienten in seiner Reaktionszeit langsamer oder ganz lahm gelegt werden.

4.1.3. Hierarchie

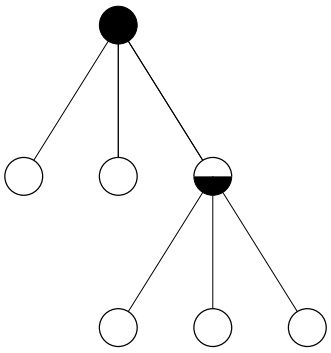

Abbildung 4.3.: Hierarchie Topologie

Jede Kopie der Daten stellt einen *Knoten* dar. Knoten sind untereinander durch *Kanten* verbunden. Diese Kanten sind die Kommunikationswege, die beim Abgleich verfolgt werden können. Die sequentielle Verknüpfung von Kanten, ergibt innerhalb der Struktur einen *Pfad*. Die Daten können entlang der Pfade in der Struktur verteilt werden. Jeder Knoten kann durch einen Pfad erreicht werden und kein Knoten kann durch mehr als einen Pfad erreicht werden. Dies ist wichtig, da mehrere Pfade zu einem Knoten bedeuten, dass Daten zirkulieren könnten, d.h. sie können erneut bei einem Knoten ankommen. In einer Hierarchie müssen Knoten diesen Fall nicht behandeln und können jede Art von Daten weiterreichen, ohne Gefahr zu laufen, dass das Netzwerk Daten unnötigerweise wiederholt transportiert. [Mus02, Abschnitt 4.7.3.1]

Einige der Abgleichverfahren die nur für eine Sterntopologie ausgelegt sind, könnten mit geringen Modifikationen auch in der Hierarchie arbeiten. Hierzu müsste jeder Knoten, der kein *Endknoten* ist, sowohl eine Server- als auch eine Klientenfunktionalität bereitstellen. Ein Knoten heißt Endknoten, wenn er maximal mit einem Knoten verbunden ist (also aller *Blätter* und die *Wurzel*). Zu allen Knoten hierarchisch unter ihm, erscheint er als Server, zu allen Knoten über ihm, verhält er sich als Klient.

4.1.3.1. Vorteile

Diese Art der Topologie entspricht Organisationsformen in Betrieben und Militär, die mit dem Einliniensystem z.B. eine reine Projektorganisation folgen. Jeder Mitarbeiter ist genau einem Vorgesetzten unterstellt und alle Informationen werden über den Dienstweg weitergegeben.

4.1.3.2. Nachteile

Fällt ein Knoten aus, dann teilt sich das Netzwerk in bis zu drei unabhängige Teile. Diese Netzwerke können meist weiter arbeiten, aber Änderungen werden nicht mehr im Gesamtnetz verteilt sondern nur noch im Teilnetz. Es ist auch anzumerken, dass sehr wenige Organisationen mit einem Einliniensystem arbeiten. Häufig sind Stabstellen und weitere Querverknüpfungen zwischen den Knoten vorhanden. Diese Szenarien besitzen eine Netzwerktopologie mit Zyklen und sind nicht mehr rein hierarchisch angeordnet.

4.1.4. Cluster

In der Literatur [HMPT03; DBMP01] erscheint oft noch der Begriff eines *Clusters*.

Es handelt sich um einen Spezialfall der Hierarchie. Das Netzwerk ist in zwei Ebenen aufgeteilt. Die obere Ebene besteht nur aus zentralen Rechnern (Servern), die jeweils

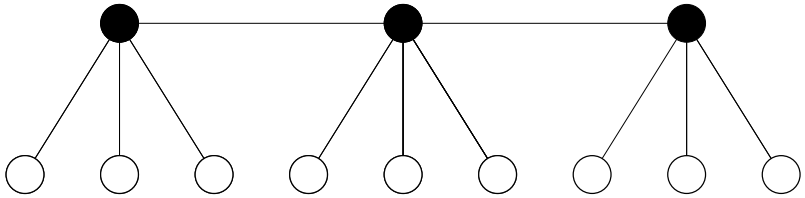

Abbildung 4.4.: Cluster Topologie

eine Kopie jedes Datensatzes enthalten. Meist hat ein Server die maßgebliche Kopie, d.h. seine Kopie ist im Zweifelsfall die vertrauenswürdige. Die darunter liegende Ebene besteht aus Klienten, die einfacher aufgebaut sein können als der Server, da diese wieder aus ihrer Sicht nur eine 1:1 Beziehung zum Netzwerk aufbauen.

4.1.4.1. Vorteile

Nur die Server müssen eine komplexe Datensynchronisation verwalten können. Die Klienten können einfachere Geräte wie zum Beispiel Mobiltelefone oder PDAs sein. Genau wie in der Hierarchie können die Teilnetze weiter arbeiten, wenn einer der Server ausfällt. Obwohl nur die logischen Verbindungen zwischen Knoten betrachtet werden, kann man diese Topologie auch sehr gut für die geographische Verteilung nutzen. Server befinden sich physikalisch in relativer Nähe zu den jeweiligen Klienten und die Verbindung zwischen den Servern untereinander kann größere Strecken überwinden.

4.1.4.2. Nachteile

Es handelt sich um einen Sonderfall der Hierarchie, bietet aber kaum wesentliche Vorteile gegenüber dieser. Eine Implementierung eines Synchronisationsverfahrens ist nicht erheblicher einfacher als im Fall der Hierarchie.

4.1.5. Beliebiger Graph (mit Zyklus)

In dieser Topologie sind alle Verbindungen z.B. auch Querverbindungen möglich, die Zyklen entstehen lassen. Es muss kein vollständig vermaschtes Netz sein, d.h. nicht jeder Knoten muss mit allen anderen verbunden sein. Es reicht bereits, wenn ein Knoten aufgrund seiner Querverbindungen einen Zyklus erzeugt.

Beispiel: *Herr Maier hat einen Computer in der Arbeit, einen Computer zu Hause und einen PDA. Die beiden Computer sind miteinander verbunden, damit Herr Maier seine*

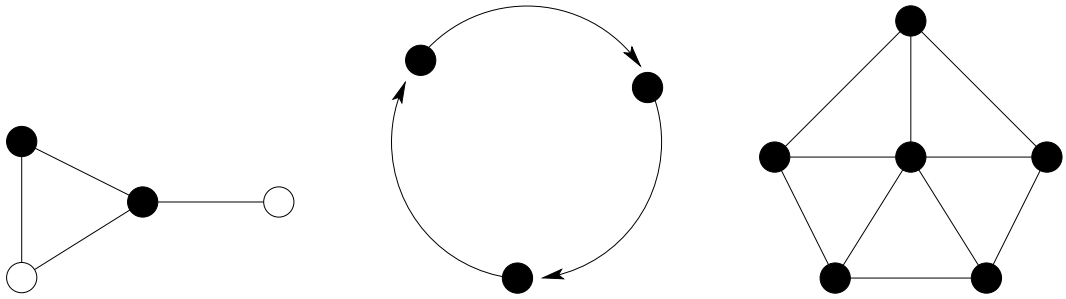

Abbildung 4.5.: Beliebige Graphen

Heimarbeit verrichten kann. Nun möchte Herr Maier seinen PDA mit dem Heimcomputer synchronisieren, da er weiß, dass er am nächsten Tag nicht in die Firma kommen wird, um den PDA dort abzugleichen. Nach dieser Datensynchronisierung kontaktiert der Heimcomputer den auf der Arbeitstelle und diese tauschen sich aus.

Obwohl der PDA in diesem Szenario nicht gleichzeitig eine physikalische Verbindung zu beiden Computern besitzt, hat er doch eine logische Beziehung zwischen den beiden. Der Datenbestand des PDA ist eine Kopie vom Computer bei der Arbeit. Der Heimcomputer holt sich die Daten des PDAs, um seine Kopie zu ergänzen.

In diesem Netzwerk von drei Kopien existiert ein Zyklus. Wenn Herr Maier wieder mit dem PDA in die Arbeit fährt und dort mit seinem Computer abgleicht, dann sollen die Änderungen, die der Heimcomputer vom PDA überspielt hat, nicht erneut wieder in den PDA als Änderung gelangen. Dies könnte zu Duplikaten oder Konflikten führen. Schließlich sind alle Änderungen, die seit dem letzten direkten Abgleich zwischen PDA und Computer stattgefunden haben, über den Heimcomputer im Computer in der Arbeit eingetroffen.

4.1.5.1. Vorteile

Fällt ein Knoten aus, dann ist bei geschickter Wahl der logischen Verbindungen, der Abgleich unter den restlichen Knoten nicht unterbrochen. Falls es sich um ein vollständig vermaschtes Netz handelt, dann fällt ein ausgefallener Knoten nicht ins Gewicht. Ein Abgleichverfahren mit Zyklenunterstützung ist ideal für mobile Ad-hoc Netzwerke. Dynamische Netzwerke können sich bilden, da deren Teilnehmer beliebig kommen und gehen können.

4.1.5.2. Nachteile

Zyklen in einem Abgleichverfahren zu unterstützen ist vergleichsweise schwierig, da erhebliche Verwaltungsaufgaben entstehen. Diese Aufgaben benötigen womöglich zusätzlichen Speicherplatz und Rechenleistung, über die mobile Endgeräte nicht in dem Ausmaß wie stationäre Personal Computer verfügen.

Die Beispiele der bis jetzt behandelten Topologien handeln von einem oder zwei Benutzer und einiger wenigen Kopien. Die Datensynchronisation im Allgemeinen kann tausende Benutzer und viele tausend Kopien umfassen. Viele Verfahren mit Zyklenunterstützung skalieren sehr schlecht, d.h. wenn die Anzahl der Knoten und/oder der Datensätze steigt, dann steigt der Verwaltungsaufwand überproportional. Selbst eine proportionale Steigerung ist bei einem sehr großen Netz nicht vorteilhaft, denn bereits diese erhöht den Speicherbedarf der Verwaltungsdaten enorm. Ein weiterer Punkt ist die zunehmend längere Dauer eines Abgleich, da mehr Metadaten übertragen werden müssen.

Eine Zyklenunterstützung erschwert die Konfliktresolution bzw. wie und wem Konflikte überhaupt gemeldet werden. Bei einem Netz aus drei Kopien mit nur einem Nutzer, ist dies weniger problematisch, als wenn es 1000 Kopien und 500 Nutzer sind. Welche Nutzer waren an dem Konflikt beteiligt und an welchen Knoten muss ein Konflikt gemeldet werden?

Zusätzlich ist es schwerer Aussagen darüber zu treffen, ob und welche Kopien auf dem neusten Stand sind.

4.1.6. Zusammenfassung der Netzwerktopologien

Das Beispiel von Herrn Maiers Situation zeigt aber, dass es durchaus erstrebenswert ist Zyklen zu unterstützen. Bereits drei Kopien können einen Zyklus bilden. Das Beispiel schildert keinen seltenen Fall. Herr Maier muss sich sicher sein können, dass keine Daten oder Änderungen verloren gehen und er will so selten wie nur möglich Konflikte auflösen oder Duplikate löschen.

Die Unterscheidung nach der angestrebten Netzwerktopologie des jeweiligen Abgleichverfahrens ist die wohl bedeutendste Unterscheidung, die angestellt werden kann. Die Topologie führt unweigerlich zu bestimmten Eigenschaften des Systems und trägt zu einer Vielzahl von Implementierungsentscheidungen bei.

Eine Abstufung der einzelnen Komplexitäten unter den Topologien ergibt sich wie folgt:

$$1{:}1 < \text{Stern} < \text{Hierarchie/Cluster} < \text{beliebiger Graph}.$$

1:1 Verbindungen sind der einfachste Fall, die Unterstützung eines beliebigen Graphen der Komplizierteste.

4.2. Anzahl der Kopien

Ein weiteres wichtiges Kriterium bei der Einteilung eines Abgleichverfahrens, ist die Anzahl der Kopien. Auf der einen Seite kann die Anzahl der Kopien von Anfang an fest stehen oder sogar die Kopien selbst sind für immer fest vorgegeben, auf der anderen Seite sind die Kopien und deren Anzahl beliebig und können sich jederzeit ändern.

4.2.1. Fest

Die Anzahl der Kopien ist von Anfang an bekannt und ändert sich nie. Zusätzliche Kopien können nicht in den Datenabgleich aufgenommen werden.

4.2.1.1. Feste Kopien

Wenn nicht nur die Anzahl der Kopien festgelegt ist, sondern die Kopien selbst festgelegt sind, dann ist das System rein statisch und es gibt keine Änderung innerhalb der logischen Verbindungen. Dies ist für ein Abgleichverfahren der einfachste Fall, da bereits sehr viel Kenntnis über das Gesamtsystem besteht. Soll allerdings eine neue Kopie dazukommen oder eine Kopie ausgetauscht werden, dann muss der Datenabgleich komplett neu aufgesetzt werden, d.h. eine Kopie wird als aktuell erklärt und ausgehend von dieser Kopie startet der Datenabgleich zu allen anderen Kopien, als wären diese vorher nie abgeglichen worden.

4.2.2. Beliebig

Die Anzahl der Kopien kann sich dynamisch über die Zeit ändern. In vielen Netzwerken benötigen Kopien eindeutige Bezeichnungen. Wenn die Kopien beliebig sind, dann muss ein eindeutiges Bezeichnungsschema gefunden werden. Einige Abgleichverfahren können aufgrund ihrer Technik zur Änderungserkennung nicht mit dynamisch auftretenden Kopien umgehen. Andere Verfahren geben statische Speichergrößen für Verwaltungsaufgaben vor und können nicht dynamisch erweitert werden. Wenn Kopien das Netzwerk verlassen, stellt sich auch die Frage, wann Metainformationen gelöscht werden können, die im restlichen System über diese Kopien geführt wurden.

4.3. Datentypen

Dieses Thema ist bereits im Abschnitt 3.1.1 der Definitionen angesprochen worden. Es stellt sich die Frage, welcher Typ an Datenbank und Datensatz abgeglichen wird:

- Flach
  - Binäre Daten
  - Tupeldaten
- Hierarchisch
- (Relationale) Datenbanken.

Die Datentypen beeinflussen die Implementierungen. In (relationalen) Datenbanken müssen bei Änderungsoperationen umfangreiche Abhängigkeitsbeziehungen berücksichtigt werden. Um einzelne Änderungen einer relationalen Datenbank an eine Kopie dieser Datenbank zu übermitteln, kann ein Abgleichverfahren nicht einfach ein paar Bytes von der Festplatte laden und diese übermitteln, da eine Datenbank von der physischen Speicherung der Daten abstrahiert und ein logisches Abbild bereitstellt, damit Anwender und Applikationen die Daten unabhängig von dem darunter liegenden Speichermedium verwalten können.

Liegen die Daten hierarchisch vor (z.B. in einem Dateisystem) bestehen ebenfalls Abhängigkeiten zwischen den einzelnen Datensätzen. Diese sind weniger komplex als in einer relationalen Datenbank. Jeder Datensatz hat nur eine Beziehung zu seinem übergeordneten Verzeichnis. Wenn in einem Dateisystem ein Verzeichnis gelöscht wird, dann werden alle Verzeichnisse und Dateien innerhalb dieses gelöscht. Ein Problem sind Plattform übergreifende Abgleiche, bei denen die Dateisysteme andere Dateiattribute besitzen oder bei der Namensgebung nicht zwischen Groß- und Kleinschreibung unterscheiden. Ein Spezialfall sind Querverknüpfungen (z.B. Link in UNIX, Alias im Mac OS oder Verknüpfung in Windows), die extra behandelt werden müssen und daher in Abgleichverfahren oft nicht unterstützt werden. Ein weiteres Problem besteht darin, Änderungen zu erkennen und weiter zu geben.

Oft ist die Reihenfolge der Datensätze auf einer Ebene nicht relevant.

Beispiel: *Ob die Datensätze in Kopie A nach Alphabet und in Kopie B nach Änderungsdatum sortiert sind, soll keine Auswirkung auf das Endergebnis der Synchronisation in beiden Kopien haben.*

Flache Daten sind vergleichsweise einfach abzugleichen.

Binären Daten können Bitweise in den Kopien verglichen werden. Ist das Bit in einer Kopie anders gesetzt, dann liegt eine Änderung vor. Wenn Änderungen protokolliert werden sollen, dann muss nur der Datensatz als geändert markiert werden.

Bei Tupeldaten innerhalb eines Datensatzes, kann die Datensynchronisation intelligenter verlaufen

Beispiel: *In Kopie A ist ein neues Feld mit der Bezeichnung* Name *eingeführt worden, in Kopie B ein Feld* Anschrift*. Diese beiden Felder können ohne einen Konflikt abgegelichen werden, da im Endergebnis eines Abgleichs das Feld* Name *als auch* Anschrift *erscheint, obwohl Änderungen am gleichen Datensatz vorgenommen worden sind.*

Bei Konflikten von Tupeldaten kann das Abgleichverfahren erkennen welche Felder betroffen sind. Einem Benutzer werden die Versionen des Datensatzes von beiden Kopien zur Wahl gestellt. Nur geänderte Felder werden hervorgehoben und der Benutzer muss nicht den ganzen Datensatz vergleichen, sondern erkennt schnell, welche Felder in Konflikt stehen.

Wie bei den Netzwerktopologien kann eine Ordnung bezüglich der Implementierungs-Komplexität aufgestellt werden:

$$\text{Binäre Daten} \leftrightarrow \text{Tupeldaten} \leftrightarrow \text{Hierarchien} \leftrightarrow \text{(Relationale) Datenbanken.}$$

Binäre Daten ohne Abhängigkeiten können einfacher abgeglichen werden, als eine komplexe relationale Datenbank. In relationalen Datenbanken können Änderungen genauer identifiziert werden, sind dort aber schwieriger zu erkennen und abzugleichen. Bei einer binären Datei ist eine Änderung sehr einfach zu ermitteln, aber die Analyse um was für eine Änderung es sich gehandelt hat, ist nicht weiter möglich. Der Abgleich gestaltet sich sehr einfach, allerdings können die Datenmengen, die zwischen den Kopien ausgetauscht werden, erheblich größer sein, als wenn nur einzelne Änderungen zwischen Datenbanken abgeglichen werden.

4.4. Granularität der Änderungserkennung

Bei diesem Kriterium geht es darum, wie fein Änderungen innerhalb der Daten durch das Abgleichverfahren erkannt werden können.

Je genauer Änderungen erkannt werden, umso weniger Daten müssen an andere Kopien übertragen werden. Gerade im Bereich des *Mobile Computing* sind Datenübertra-

gungsverfahren vergleichsweise langsam. Dies liegt daran, dass Strom sparende Geräte zum Einsatz kommen aber schnelle drahtlose Datenübertragungsverfahren in der Regel mehr Strom verbrauchen. Um die Dauer des Datenabgleichs zu verringern, sind geringe Datenmenge vorteilhaft. Eine feinere Erkennung von Änderungen kann zu kleineren Datenmengen beim Datenabgleich führen. Zum Beispiel muss nicht das ganze Dateisystem übertragen werden, wenn nur eine Datei gelöscht wurde.

Die Datensynchronisation kann so auch viel schneller abgeschlossen werden. Bei sehr großen Datenmengen sind selbst bei ausgeklügelten Abgleichverfahren Zeitspannen bis zu mehreren Stunden möglich. Je nach Verfahren kann in dieser Zeitspanne eine Benutzerinteraktion mit der jeweiligen Datenbankkopie unmöglich sein. Bei einem Mobiltelefon könnte dann das ganze Gerät für einige Stunden seiner eigentlichen Aufgabe nicht gerecht werden.

Eine feinere Änderungserkennung hat allerdings auch erhebliche Nachteile. Um die Änderungen zu erkennen, müssen in den Kopien mehr Berechnungen durchgeführt werden. Die Daten müssen sehr viel genauer untersucht und damit auf mehr Daten zugegriffen werden. Die Protokollierung der jeweiligen Änderungen kann sehr viel Speicherplatz kosten. Dabei ist zu vermeiden, dass der Speicherplatz für die Hintergrundinformationen des Verfahrens, den Speicherplatz der eigentlichen Datenbank übertrifft, da Speicherplatz in mobilen Geräten sehr teuer und beschränkt ist.

4.4.1. Datenfeldweit

Jede Änderung in einem Datenfeld wird erkannt und beim Abgleich übermittelt. Im Idealfall muss nur diese eine Änderung übermittelt werden.

4.4.2. Datensatzweit

Es wird der gesamte Datensatz als geändert markiert und übertragen. Der Vorteil liegt darin, dass ein Datensatz zu einer logischen Einheit wird, der unabhängig von seinem eigentlichen Inhalt behandelt werden kann. So muss ein Abgleichverfahren nicht den Datentyp kennen oder verstehen, sondern synchronisiert abstrakte Dateien von einem Dateisystem zum Nächsten. Dadurch ist das Abgleichverfahren vielseitiger für unterschiedlichste Datenbestände ohne weitere Modifikationen einsetzbar.

4.4.3. Datenbankweit

Wenn eine Änderung in einer Datenbank geschieht, wird immer die gesamte Datenbank übermittelt.

4.4.4. Zusammenfassung der Änderungserkennungen

Eine rein Datenbankweite Änderungserkennung wird in der Regel nicht praktiziert. Die Datenfeldweite Änderungserkennung geschieht aus Speicherplatz- und Berechnungsgründen selten. Es ist wichtig anzumerken, dass es zwei Ebenen der Änderungserkennung gibt. Zum einen stellt sich die Frage, welche Änderungen in der Regel markiert und übermittelt und welche Änderungen in einer Konfliktsituation erkannt werden. In der Regel wird nur eine Datensatzweite Änderungserkennung betrieben. Im Konfliktfall wird eine Auflösung auf der Ebene von Datenfelder versucht, um dem Benutzer entsprechende Vorschläge zu unterbreiten oder beim vollautomatischen Abgleich so wenig Daten wie möglich zu verändern.

Ähnlich wie bei den vorherigen Kriterien kann eine Ordnung bezüglich der Optimierung des Speicherplatzverbrauchs und der Berechnungen während eines Synchronisationsverfahrens abgleitet werden (rechts schlechter):

$$\text{Datenbank} > \text{Datensatz} > \text{Datenfeld}.$$

besser schlechter

Für die Optimierung des Zeitbedarfs bei der Übertragung ergibt sich das umgekehrte Bild:

$$\text{Datenfeld} > \text{Datensatz} > \text{Datenbank}.$$

besser schlechter

4.5. Architektur

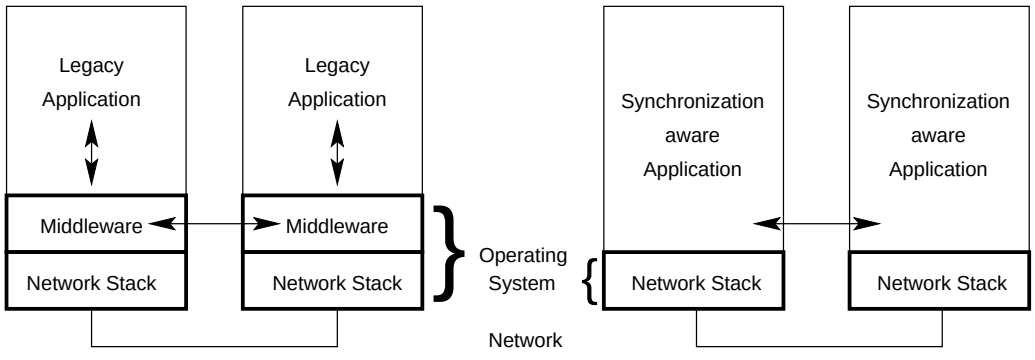

Abbildung 4.6.: Middleware vs. Benutzerapplikation

4.5.1. Middleware / Betriebssystem

Eine Middleware bezeichnet die Schnittstelle zwischen einer bestimmten Technologie und einem Anwendungsprogramm. Das Programm muss sich nicht weiter um die Technologie kümmern, sondern die Middleware erledigt das *Wie*. Besonders wenn die Technologie komplex ist, bietet es sich an, dass eine Middleware Probleme und Vorgehensweisen gegenüber einem Anwendungsprogramm abstrahiert. Im Fall der Datensynchronisation wären das *Wie* der Zugriff auf die Verbindung (das Netzwerk) zwischen zwei Knoten, die Übertragung der einzelnen Datenpakete und deren Aufbau. Das Programm fordert die Middleware auf, bestimmte Daten *abzugleichen*. Nicht alle Anwendungsprogramme können den gleichen Abstraktionsgrad nutzen. Im Konfliktfall soll das eine Programm Kontrolle über die abzugleichenden Daten besitzen, das andere Programm möchte Parameter der Transportmechanismen ändern, da es Daten mit bestimmter Güte abgleicht. Solche Programme nennt man im Englischen *synchronization aware*. Es kommt aber häufiger vor, dass eine Middleware genau dazu eingesetzt wird, dass Anwendungsprogramme überhaupt kein Wissen über Datensynchronisation besitzen müssen. Dies nennt man *synchronization transparent*. Im Falle einer Middleware kann sich ein Anwendungsprogrammierer aussuchen, ob er sich mit Belangen der Datensynchronisation auseinander setzen will oder nicht.

Die Vorteile liegen darin, dass auch ältere Programme Verfahren der Datensynchronisation nützen könnten, da diese automatisch im Hintergrund abläuft und das Programm keine Kenntnis darüber haben muss. Keine Applikation muss Datensynchronisations-Bibliotheken liefern, sondern kann direkt die Middleware nutzen. Auf diese Weise kann sich wiederholende Programmierarbeit vermieden werden.

Der Nachteil liegt darin, dass eine Middleware kaum alle Datentypen kennen kann und in einem Konfliktfall nicht unbedingt die beste Lösung findet. Die Erkennung einer Änderung aber auch von Konflikten kann der Middleware schwer fallen. Eine Middleware deckt möglicherweise nicht alle Ziele ab, zum Beispiel wenn sie keine Zyklen in der Topologie unterstützt oder sogar von falschen Annahmen ausgeht. Dies kann zu fehlerhaften Abgleichen führen. Eine Middleware sollte daher extrem flexibel sein, was gleichzeitig ihre Implementierung erschwert. [Mus02, Abschnitt 4.1.1]

4.5.2. Benutzerapplikation

In diesem Fall stellt die Applikation die gesamte Datensynchronisations-Logik. Die Applikation weiß, wie sie mit Hilfe des Betriebssystems eine Verbindung zwischen den Knoten aufbaut, wie die auszutauschenden Datenpakete aussehen, wo die Daten liegen, was

Änderung sind und wie im Konfliktfall zu verfahren ist.

4.6. Benutzeranzahl

4.6.1. Ein-Benutzer System

Das Gesamtsystem wird von nur einer Person genutzt. Nur diese einzelne Person nimmt Änderungen am Datenbestand vor. Nicht automatisch lösbare Datenkonflikte können eindeutig diesem Benutzer zugeordnet und gemeldet werden.

Beispiel: *Herr Maier besitzt einen Computer auf der Arbeit, einen Computer zu Hause, einen PDA und ein Mobiltelefon. Zwischen diesen vier Geräten tauscht Herr Maier sein privates Adressbuch aus. Dieses private Adressbuch pflegt er auf allen vier Geräten.*

Nur einen Benutzer zu erlauben, macht für ein optimistisches Abgleichverfahren am meisten Sinn. Die Konflikte können in der Regel nicht durch gleichzeitiges Ändern geschehen. Ein Benutzer kann durch sorgfältige regelmäßige Abgleiche Konflikte vermeiden und sollte doch einmal ein Konflikt auftreten, können diese genau einem Benutzer (dem Verursacher) gemeldet werden. [CJ02] bezeichnet den Benutzer sogar als *Schreib-Token*, was sicherstellt, dass Konflikte nur im begrenzten Rahmen auftreten. Die Erfahrung zeigt aber, dass Benutzer schnell dazu aufgefordert werden, gewisse Regeln einzuhalten, so zum Beispiel in kurzen regelmäßigen Abständen Datenabgleiche durchzuführen oder gar nur eine Kopie als Schreibkopie zu verwenden und alle anderen Kopien als reine Lesekopien anzusehen. Einem Benutzer solche Regeln aufzuerlegen, gilt es zu vermeiden.

4.6.2. Mehr-Benutzer System

Bei einem Mehr-Benutzer System arbeiten verschiedene Personen an einem Datenbestand.

Beispiel: *Herr Maier führt einen Kalender mit seinen beruflichen Terminen. Damit Termine nicht kollidieren und Sitzungsräume nicht doppelt belegt werden, arbeiten alle Mitarbeiter in der Firma mit dem gleichen zentral verwalteten Terminkalender.*

Wenn ein Konflikt auftritt, betrifft dies meist mehr als eine Person. Es sind Regeln nötig, wie in einem solchen Fall vorgegangen wird:

- Werden alle diese Personen zur Entscheidung im Konfliktfall herangezogen?

- Entscheidet eine bestimmte Anzahl unter diesen Personen?
- usw.

4.6.2.1. Benutzer festgelegt

Für die Auflösung von Konflikten können viel bessere Mechanismen entworfen werden, wenn alle Benutzer des Gesamtsystems bekannt sind und deren Anzahl nicht variabel ist.

Beispiel: *In der Firma Schraubendreher hat im Konfliktfall immer der Abteilungsleiter zu entscheiden.*

4.6.2.2. Benutzer nicht festgelegt – freie Anzahl

Ist die Benutzergruppe nicht festgelegt, dann ist das Gesamtsystem eine Art Kiosk, wo jeder vorbei kommen und eine Änderung im Datenbestand vornehmen kann. In diesem Szenario ist es sinnvoll, bestimmte Zugriffskontrollen einzuführen, um private Daten zu schützen. Bei Daten auf die gemeinsam zugegriffen werden kann, ist es im Konfliktfall aufwendig alle Beteiligten zum Konflikt zu befragen. Einige Benutzer werden nicht mehr am System mitwirken und gar nicht mehr auffindbar sein. Oder die Anzahl der Beteiligten ist so groß, dass eine Konsensfindung durch alle keinen Sinn macht.

4.7. Gründe für Abgleiche

Dieses Kriterium beschäftigt sich mit der Frage, wann oder weswegen die einzelnen Kopien abgeglichen werden. Dieses Kriterium ist abhängig von der Netzwerktopologie und der Anzahl der Benutzer im Gesamtsystem. [DBMP01]

4.7.1. Automatisch

4.7.1.1. Zeit basiert

Sofort Wenn eine Änderung immer sofort weiter gegeben wird, dann müssen die einzelnen Knoten auch jederzeit eine Netzwerkverbindung aufbauen können. Dies widerspricht der Idee, dass Knoten zeitweise oder längere Zeit voneinander getrennt sind. In einem solchen Fall können die Knoten ebenso gut dauerhaft miteinander verbunden sein und die klassische pessimistische Replikation aus dem Bereich der verteilten Datenbanksysteme angewendet werden. Der Zugriff auf einen Datensatz wird dann in allen Kopien gesperrt und es treten keine Schreib/Schreib Konflikte auf.

Beispiel: *Herr Maier arbeitet zu Hause an seinem Computer. Dieser ist mit dem Netzwerk der Firma verbunden und alle Zugriffe werden so ausgeführt, als wäre Herr Maier in der Firma.*

Datum In diesem Fall müssen alle Kopien zu einem vordefinierten Zeitpunkt verbunden sein. Zu diesem Zeitpunkt werden die Kopien abgeglichen.

Beispiel: *Alle Außendienstmitarbeiter der Firma Schraubendreher treffen sich immer Montags zu einem Gespräch im Konferenzraum. Bei dieser Gelegenheit werden die aktuellen Strategien und Weisungen der Firma auf alle PDAs aufgespielt und Kommentare und Anregungen der Außendienstmitarbeiter aufgesammelt.*

Zeitintervall Der Benutzer oder das System gibt einen bestimmten Zeitraum vor, nach dem synchronisiert werden soll. Dies ist eine der häufigsten Formen der Durchführung von Datensynchronisationen. Wenn oft Änderungen entstehen, sollte das Intervall klein gewählt sein, damit weniger Konflikte auftreten. Wenn wenig geändert wird, kann das Zeitintervall größer sein, um unnötige Datenmengen im Netzwerk zu vermeiden.

Beispiel: *Herr Maier hat sein E-Mail Programm so eingestellt, dass es alle Stunde den IMAP4 [Cri03] Server kontaktiert, um anzufragen, ob neue Nachrichten eingetroffen sind. Bei IMAP4 liegen alle Nachrichten zentral auf einem Server. Die Nachrichten werden vom IMAP Klienten beim Laden nicht vom Server gelöscht, sondern nur gelesen. Auch IMAP wird als Datensynchronisationsverfahren angesehen [IMA99].*

4.7.1.2. Datenbasiert

Bei Daten orientierten Regeln, muss die Datensynchronisations-Software die Daten inhaltlich zu einem gewissen Grad kennen, oder den Zugriff auf die Daten mitverfolgen können. Dies ist nicht unbedingt in allen Systemen möglich, was diese Regeln für solche Systeme ausschließt.

Sofort Dieser Fall ist identisch mit der sofortigen Zeit basierten Synchronisation. Wenn Daten geändert werden, werden diese sofort abgeglichen. Auch hier ist es sinnvoller auf pessimistische als auf optimistische Replikaktion zu setzen.

Priorität Jeder Datensatz bekommt vom Nutzer oder durch das System eine feste Priorität zugeteilt. Die möglichen Prioritätsstufen und ihre Folgen können wie folgt aussehen:

- Sofort aktualisieren
- Als Erstes synchronisieren
  Dies ist sinnvoll, wenn der Datensatz wichtiger als andere ist. Bricht die Synchronisation ab, dann sind Datensätze höherer Priorität zu einer gewissen Wahrscheinlichkeit bereits übertragen. Je häufiger eine nicht abgeglichene Änderung zu einem Konflikt führt, umso höher ist die Prioritätsstufe zu setzen.
- usw.

Relevanz Die Änderung sollte so zeitnah wie möglich übertragen werden, wenn es wahrscheinlich ist, dass andere Kopien diesen Datensatz benötigen, damit spätere Lese/Schreib oder Schreib/Schreib Konflikte vermieden werden. Im Gegensatz zu Prioritäten wird hier keine feste Prioritätsstufe vergeben, sondern dynamisch die Relevanz der Änderung ermittelt. Einer Kopie ist normalerweise unmöglich diese Regel zu erfüllen, da sie nur schwer die Zukunft voraussagen kann.

Beispiel: *Der Chef der Firma Schraubendreher hat alle Termine seiner Mitarbeiter in Kopie vorliegen und kann jederzeit neue Termine eintragen. Herr Maier arbeitet gerade an seiner Terminplanung für die nächste Woche. Der Chef ergänzt zur gleichen Zeit Herrn Maiers Kalender, um ihn nächste Woche zu treffen. Es wäre natürlich vorteilhaft, um Terminkonflikte zu vermieden, wenn jede Änderung, die Herr Maier und sein Chef gerade durchführen, sofort beim jeweils anderen sichtbar wird. Woher soll aber das System wissen, dass der Terminkalender von Herrn Maier gerade eine hohe Relevanz besitzt und abgeglichen werden sollte?*

Verlauf orientiert Mit der Anzahl der Änderungen steigt auch die Anzahl der Abgleiche. Wenn wieder weniger Änderungen auftreten, dann können die Abgleich-Intervalle wieder größer werden. So kann zum Beispiel eine Regel lauten, dass nach 10 Änderungen eines Datensatzes ein Abgleich durchgeführt wird.

Zugriffs orientiert Ein Datensatz der viele Schreiber auf verschiedenen Kopien besitzt, wird in kurzen Abständen synchronisiert, um Konflikte unter den Schreibern zu vermeiden.

4.7.1.3. Netzwerksituation

Um die folgenden Regeln nutzen zu können, muss die Datensynchronisations-Software eine gewisse Kenntnis über die möglichen Netzwerktechnologien besitzen (Verfügbarkeit, Geschwindigkeit, Kosten). Je nach eingesetzter Netzwerktechnologie kann es für eine Software kompliziert sein, zu erkennen, ob eine Verbindung möglich ist oder bereits besteht. Eine schnelle Verbindung kann dagegen relativ leicht über den Durchsatz an Daten pro Zeiteinheit gemessen werden. Dafür muss eine Prüfung mit Testdaten oder eine Testsynchronisation stattfinden. Ob eine Verbindung kostengünstig ist, kann eine Software nur schwer pauschal wissen. Hier ist die Software auf weitere Eingaben angewiesen, sei es durch eine andere Software oder durch den Benutzer.

Verbunden Automatisch wenn eine Verbindung möglich ist oder hergestellt wird, soll ein Datenabgleich erfolgen.

Schnelle Verbindung Ein Abgleich kann bei sehr großen Datenmengen sehr lange und bis zu mehreren Stunden dauern. Wenn die Verbindung langsam ist, verzögert dies einen Abgleich darüber hinaus und er sollte daher nicht oder nur selten stattfinden. Wenn eine schnelle Verbindung besteht, können Abgleiche öfters geschehen.

Kostengünstige Verbindung Genau wie bei einer schnellen Verbindung, darf es nicht sein, dass ein System jede mögliche Verbindung aufbaut. Wenn die Verbindung vergleichsweise teuer ist, dann sollte eine Synchronisation eher selten stattfinden (zum Beispiel in einem Mobilfunknetz).

4.7.1.4. Weitere Gründe

Je nach System können eine Reihe weiterer Gründe Ausschlag gebend für einen Abgleich und dessen Häufigkeit sein: Wenn ein Gerät zum Beispiel Batterie betrieben arbeitet, sollten die Abgleiche seltener werden. Wenn das Gerät dagegen dauerhaft Strom beziehen kann, können Aktualisierungen öfters eintreten. Aber auch im militärischen Einsatz der Datensynchronisation sollte seltener synchronisiert werden, wenn man sich in Feindesgebiet befindet, um nicht den Standort durch Funkwellen zu verraten.

4.7.2. Manuell durch einen Benutzer

Ein Zeit basierter Abgleich benötigt eine Netzwerkverbindung zu einer bestimmten Zeit. Ein Daten basierter Abgleich benötigt Hintergrundwissen über die Daten. Ein Topologie

basierter Abgleich benötigt eine weitergehende Verknüpfung mit den Netzwerkschichten und der jeweiligen Netzwerktechnologie.

Da eine Datensynchronisations-Software dies nicht immer leisten kann oder will, hat sich die manuelle Synchronisation durch den Benutzer etabliert. Der Benutzer öffnet die Software, woraufhin versucht wird, die Datensynchronisation zu starten. Der Benutzer weiß oft viel besser, wann ihm eine Synchronisation wichtig ist, um etwaige Schreib/ Schreib oder Lese/Schreib Konflikte zu vermeiden. Es liegt in der Entscheidung des Benutzers, wann er eine Verbindung als kostengünstig oder schnell ansieht.

Beispiel: *Herr Maier besitzt ein Mobiltelefon, welches das Adressbuch und den Kalender mit den privaten Daten auf dem Rechner daheim abgleicht. Das Mobiltelefon wird auch für die beruflichen Daten mit dem Rechner in der Firma abgeglichen. Da Herr Maier fast immer seinen PDA dabei hat, muss das Telefon nicht immer auf dem neusten Stand sein. Jetzt hat aber Herr Maier gestern mit einem seiner Auftraggeber einen Termin zum Golfen ausgemacht und möchte diesen nun gerne absagen. Herr Maier ist gerade im Freizeitpark und hat seinen PDA daheim vergessen. Aber sein Mobiltelefon ist da. Da sein Geschäftspartner mit seiner Telefonnummer nicht bei der Auskunft geführt wird, initiiert Herr Maier über das Mobilfunknetz eine Synchronisation mit dem Rechner auf der Arbeit. Nun erhält Her Maier die Telefonnummern und kann den Termin absagen. Das System (das Mobiltelefon) hätte ohne Herrn Maier kaum wissen können, dass nun eine Datensynchronisation nötig gewesen wäre.*

4.8. Übertragung der Änderung

4.8.1. Datensatz komplett

Wenn in einem Datensatz eine Änderung vorliegt, dann wird der gesamte Datensatz zur anderen Kopie übertragen. Dies ist bei neuen Datensätzen unumgänglich. Bei gelöschten Datensätzen ist dies meist nicht nötig.

4.8.2. Delta

Alternativ kann beim Abgleich nur die Änderung übertragen werden. Ein sehr großer Datensatz wie z.B. eine Programmdatei müsste so nicht komplett übertragen werden, sondern im Idealfall nur die Bits, die geändert worden sind. Gerade bei der Synchronisation von Dateisystemen spielen Verfahren zur Erkennung der Unterschiede (*Delta*) zwischen zwei Kopien eine große Rolle, da nur Teile eines Datensatzes übertragen werden

müssen [rsy93; KWK03; PV04]. Beim Abgleich von Daten aus einem Adressbuch oder Kalender kommt dies in der Regel nicht zum Einsatz, da die Verwaltungsdaten größer werden könnten, als der eigentliche Datensatz selbst.

4.9. Technik der Änderungserkennung

Ein weiteres Kriterium für ein Abgleichverfahren ist die Art und Weise, wie Datenänderungen erkannt werden. Oft ergibt sich die Art der Änderungserkennung aus den vorausgegangenen Kriterien. Sie ist abhängig von der Art der Daten und der gewünschten Granularität der Änderungserkennung. Es gibt Techniken die nicht für alle Arten der Netzwerktopologie sinnvoll sind. Besonders Zyklen im Netzwerk erfordern effiziente Techniken, damit Änderungen so präzise wie möglich erkannt und so effizient wie möglich übertragen werden, damit die Datenabgleiche nicht zu lange dauern. Auch die Architektur beschränkt die Änderungserkennung. Die Technik der Änderungserkennung bedingt das Verfahren zur Konfliktresolution, auf das im nachfolgenden Abschnitt 4.10 eingegangen wird.

4.9.1. Zustand basiert

Bei einem Zustand basierten System wird ein reiner Datenvergleich vorgenommen. [Mus02, Abschnitt 4.7.4.1][PV04, Kapitel 5] Dies bedeutet die beiden Kopien werden Datensatz für Datensatz miteinander verglichen. Ein Datensatz ist die kleinste Einheit bei der Änderungen erkannt werden können. Wenn der Datensatz in einer Kopie nicht vorhanden ist oder wenn die beiden Datensätze nicht gleich sind, dann wird eine Änderung erkannt. D.h. es wird nur der Zustand beider Kopien betrachtet, ein Vergleich zwischen beiden gezogen und daraus werden die Änderungen erkannt.

Ein solches System hätte allerdings eine Hinzufügen/Löschen Zweideutigkeit. D.h. es kann nicht zwischen Hinzufügungen und Löschungen unterscheiden.

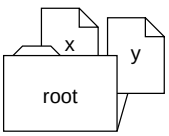

Abbildung 4.7.: Zustand Knoten A

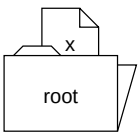

Abbildung 4.8.: Zustand Knoten B

Beispiel: *In Abbildung 4.7 ist der aktuelle Stand von Knoten A zu sehen. Er verfügt über zwei Datensätze* x *und* y *in der Datenbank* root*. In Abbildung 4.8 ist der Zustand des Knotens B abgebildet. Er hat nur einen Datensatz* x *in* root*. Aufgrund der Zustände von A und B kann nicht bestimmt werden, ob* y *in B gelöscht oder in A hinzugefügt worden ist.*

Um dieses Problem zu lösen, führen alle Zustand basierten Systeme einen dritten Zustand ein, und zwar jenen nach dem letzten Abgleich. Dieser Zustand wird gespeichert und beim darauf folgenden Abgleich abgerufen.

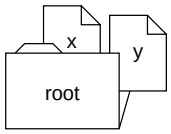

Abbildung 4.9.: Dritter Zustand

Beispiel: *Abbildung 4.9 macht deutlich, dass in diesem Fall der Datensatz* y *gelöscht wurde. Um beide Datenbanken wieder in Einklang zubringen wird* y *in Knoten A gelöscht.*

Über den archivierten Zustand definiert sich in einem Zustand basierten System eine *Änderung*. D.h. jeder Unterschied zwischen der archivierten Kopie und dem aktuellen Zustand (vor dem Abgleich) ist eine Änderung. Tritt in beiden Kopien im gleichen Datensatz eine Änderung auf, entsteht ein Konflikt.

Dies ist allerdings nicht die Definition von *Änderung* innerhalb des Schreib/Schreib Problems, welches bis jetzt als Basis genommen worden ist. Beim Schreib/Schreib Problem wird jedes Schreiben auf einem Datensatz als vermeintliche Änderung angesehen. Zur Erinnerung beim Lese/Schreib Problem müsste jedes Lesen eines Datensatzes als vermeintliche Änderung angesehen werden.

Beispiel: *Herr Maier erhält den Auftrag der Firma Lacknit 705 Schrauben bis nächste Woche zu liefern. Herr Maier trägt dies in seinen PC ein. Abends synchronisiert Herr Maier noch schnell seinen PDA und verlässt das Büro. Da ruft die Firma Lacknit auf dem Mobiltelefon von Herrn Maier an und meint es wären zwei Schrauben vergessen worden. Es werden 707 Schrauben bestellt. Dies trägt Herr Maier in seinen PDA ein. Am nächsten Tag – Herr Maier konnte sich noch nicht einmal in seinem Büro hinsetzen – ruft erneut die Firma Lacknit an. 706 Schrauben werden nun benötigt. Herr Maier trägt dies in seinen PC ein. Keine Minute später ruft die Firma Lacknit wieder an. Herr Maier hat*

seine Tasche (mit seinem PDA) immer noch nicht auspacken können. Derjenige der die Schrauben zählen sollte, war wohl noch ein junger unerfahrener Praktikant. Die Firma Lacknit entschuldigt sich viele Male und will nun doch wieder die anfänglich bestellten 705 Schrauben. Herr Maier trägt die Anzahl der Schrauben in seinen PC ein und hat mit den ständigen Änderungen kein großes Problem, denn der PC behält den Überblick.

Allerdings hat der PC bei einem Zustand basierten System nur bedingt einen Überblick über die letzten Änderungen. Wenn nun Herr Maier seinen PDA auspackt und mit dem PC abgleicht, dann werden die 707 Schrauben vom Vorabend ausgeliefert. Begründung: Ein Zustand basiertes System erkennt auf dem PC von Herrn Maier keinen Unterschied. Der archivierte Zustand enthält 705 Schrauben und der aktuelle Zustand sind 705 Schrauben. Danach liegt *keine Änderung* vor. Allerdings besteht eine Änderung im PDA von 705 auf 707 Schrauben. Da in beiden Knoten im selben Datensatz keine Änderung stattgefunden hat, liegt auch kein Konflikt vor. Die Änderung vom PDA kann ganz einfach ohne einen Konflikt zu erzeugen in den PC übertragen werden. Folglich hat der PC nun 707 Schrauben vermerkt und diese Anzahl wird ausgeliefert. Ein Zustand basiertes System kann folglich nicht alle Schreib/Schreib Konflikte erkennen.

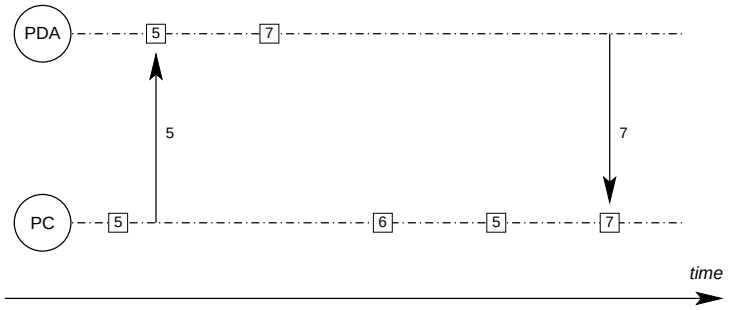

Abbildung 4.10.: In diesem Beispiel versagt ein Zustand basiertes Verfahren.

Ein weiteres Problem sind die entstehenden Speichermengen. Es muss mindestens ein archivierter Zustand pro Kopie Paarung vorhanden sein. Die meisten Zustand basierten Systeme arbeiten daher mit sehr aufwendigen Algorithmen, um diese Speichermengen klein zu halten.

Ein großer Vorteil dieser Systeme ist, dass die Technik der Änderungserkennung formal bewiesen worden ist [PV04] und Korrektheit unter den Annahmen und Definitionen besteht. Es ist allerdings zu wiederholen, dass die Definition einer Änderung nicht mit der beim Schreib/Schreib Konflikt übereinstimmt.

Ein Grund warum Zustand basierte Systeme entworfen werden, liegt in der Entschei-

dung für eine Architektur als Benutzerapplikation. Die Applikation kann jederzeit gestartet werden und auch Daten von anderen Applikationen verarbeiten, die selbst keine Datensynchronisation vorsehen. Außerdem kann eine solche Applikation schneller und Plattform übergreifender programmiert werden, als eine Middleware, die Anfragen gewöhnlicher Applikation abfängt und diese dann an das Betriebssystem weiter gibt.

Oft nutzen solche Systeme auch die Deltaübertragung von Änderungen, da dies aus dem vorliegenden dritten Zustand leicht zu errechnen ist.

4.9.2. Verlauf orientiert

Im Gegensatz zum Zustand basierten System, werden bei einem Verlauf orientierten System, Änderungen an einem Datensatz registriert und in einer bestimmten Form notiert [CJ04]. Diese Systeme sind entweder Middleware Systeme oder die Synchronisations-Software ist gleichzeitig die Verwaltungssoftware für die Daten. Je Abgleichverfahren wird der gesamte Verlauf oder nur ein Teil davon gespeichert. Für die meisten Systeme reicht es bereits aus, wenn die letzte Änderung an einem Datensatz gespeichert wird. Nur ganz wenige Systeme speichern die komplette Geschichte eines Datensatzes. Diese Systeme profitieren von diesen zusätzlichen Verwaltungsdaten, so dass bei Änderungen nur ein Delta übertragen werden muss [KWK03].

4.9.2.1. Zentrale Koordination

Bei der zentralen Koordination ist die Netzwerktopologie mit einem Server vorgegebenen. Es gibt eine zentrale Kopie (Server), die einen Versionszähler für jeden Datensatz pflegt. Wenn eine Kopie (Klient) diesen Datensatz auslesen will, wird die lokale Version mit der im Server verglichen. Ist die Version dort höher, wird der Datensatz vom Server geladen. Ist der lokale Datensatz verändert worden, wird er an den Server übermittelt. Ein Konflikt tritt auf, wenn der lokale Datensatz geändert worden ist, aber im Server eine höhere Version vorhanden ist. In diesem Fall kann davon ausgegangen werden, dass der Datensatz sowohl lokal als auch im Server oder in einem der ihm angeschlossenen Klienten seit der letzten Synchronisation geändert wurde. CVS nutzt diese Technik zur Verwaltung von Änderungen. [CVS85]

Beispiel: *Herr Maier legt einen neuen Datensatz im Firmenserver. Der Datensatz erhält die Versionsnummer 1.0. Im Lager wird der Datensatz herunter geladen und geändert. Daraufhin wird der Datensatz wieder auf den Server übertragen und erhält damit die Version 1.1. Herr Maier hat ebenfalls noch einen Fehler in dem Datensatz gefunden und*

hat ihn geändert. Nun will er diesen Datensatz zum Server schicken. Hier erkennt das Abgleichverfahren, dass bereits eine Änderung statt gefunden hat, da die Version auf dem Server eine andere Nummer trägt als die ursprüngliche, lokale Datei. Es handelt sich um einen Schreib/Schreib Konflikt. Herr Maier löst diesen mit Hilfe der Synchronisations-Software auf.

4.9.2.2. Lokaler Verlauf mit Log Datei

Bei einem Log Datei basierten Abgleichverfahren führt jede Kopie lokal eine Geschichte der Änderungen. Jede Änderung an einem Datensatz wird Datenbankweit protokolliert. Je nach System wird nur die letzte Änderung (Hinzufügung, Ersetzung, Löschung) oder die komplette Geschichte des Datensatzes gespeichert. Darüber hinaus erhält jede Änderung einen Zeit- oder Zählerstempel. Die Zeit-/Zählerstempel werden Datenbank weit geführt, d.h. jede Datenbank besitzt einen Zähler. Jede Änderung eines Datensatzes in der Datenbank führt zu einer Erhöhung des Zählers.

Beispiel: *Die Änderung an Datensatz A und die Änderung an Datensatz B hat den Zähler um zwei Einheiten erhöht.*

Bei einem Abgleich der Daten wird diese Log Datei ausgetauscht. Danach haben beiden Kopien wieder den gleichen Stand. Um nicht immer die komplette Log Datei zu übertragen, werden die Zeit-/Zählerstempel genutzt. Die Kopien merken sich nach dem Abgleich den Stempel der anderen Kopie. Auf die Weise können beim nächsten Abgleich nur die Änderungen seit diesem Stempel angefordert werden. Wenn in der Log Datei eine Hinzufügung, Ersetzung oder Löschung für einen Datensatz auftaucht, wird dies auf den lokalen Datensatz angewandt. Enthält allerdings die eigene Log Datei in diesem Zeitraum ebenfalls einen Eintrag für diesen Datensatz, dann liegt ein Konflikt vor. Wie die Log Dateien aufgebaut sind und wie die Änderungen übertragen werden, ist abhängig von dem jeweiligen Abgleichverfahren. *SyncML* ist ein solches Log Datei basiertes Verfahren. Dieses und weitere Log Datei basierte Verfahren werden im nächsten Kapitel II näher beschrieben.

Wichtig ist die Unterscheidung zwischen zentraler Koordination und Log Datei. Bei der zentralen Koordination führt nur der Server für jeden Datensatz eine Versionsnummer. Die einzelnen Kopien tauschen sich nur mit dem Server aus. Beim Log Datei Verfahren führt jede Kopie einen Verlauf und nur die Kopie selbst schreibt in ihre Log Datei. Andere Kopien lesen diesen Verlauf lediglich aus und bauen daraus lokal den aktuellen Zustand des Datensatzes auf.

Ist die Netzwerktopologie komplexer als die Sterntopologie, dann ist eine zentrale Koordination ungeeignet. Angenommen es gäbe ein Netz mit zwei Servern, die jeweils einen Klienten besitzen. In allen vier Kopien ist Version 1.0 eines Datensatzes gespeichert (siehe Abbildung 4.11).

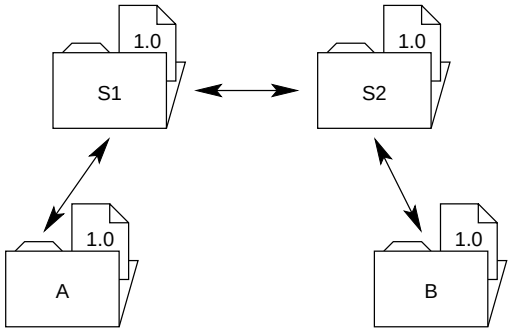

Abbildung 4.11.: Zentrale Koordination am Beispiel CVS

Klient A ändert nun den Datensatz und gleicht diesen mit dem Server S1 ab. In S1 gibt es nun eine Version 1.1. Klient B ändert ebenfalls den Datensatz und gleicht diesen mit dem Server S2 ab. Auch hier entsteht eine Version 1.1. Wenn man nun davon ausgeht, dass es ein reines zentral koordiniertes Abgleichverfahren ist, dann müsste sich einer der beiden Server als Klient beim anderen Server anmelden und dasselbe Klient-Verfahren wählen.[1] Wenn sich nun S2 als Klient bei S1 anmeldet, dann stellt S2 fest, dass bereits die aktuellste Version vorhanden sei, da S2 die gleiche Versionsnummer 1.1 für den Datensatz besitzt. Der Datensatz wird nicht übertragen. Nun hatte allerdings Klient A den Datensatz auf den Wert *705* geändert. Klient B hatte seinen Datensatz auf *707* geändert. Die Version liegt zwar nun auf beiden Servern in Version 1.1 vor, allerdings ist der Inhalt nicht mehr derselbe und die Daten sind inkonsistent geworden.

4.9.2.3. Globaler Verlauf mit Vektorzeit

Wenn ein Abgleichverfahren Zyklen in der Netzwerktopologien unterstützen will, dann müssen Knoten selbstständig entscheiden können, welcher Datensatz neuer ist und ob ein Konflikt vorliegt.

Bei einem beliebigen Graphen kann potentiell jeder Knoten mit jedem eine Verbindung besitzen. Es handelt sich dabei um eine logische Verbindung, d.h. die beiden Knoten müssen nicht dauerhaft miteinander verbunden sein, sondern waren irgendwann in ih-

[1] Wenn wir diese Annahme nicht treffen, arbeitet der Server womöglich mit einem geänderten Verfahren und dann sind Wirklichkeit zwei verschiedene Abgleichverfahren im Netzwerk vorhanden. Es geht hier aber um die Betrachtung eines Verfahrens und dessen Tauglichkeit für verschiedene Topologien.

rer Existenz einmal miteinander verbunden und haben einen Datenabgleich durchlaufe. Wenn diese beiden Vorraussetzungen erfüllt sind, dann besteht eine logische Verbindung zwischen zwei Knoten. Wenn zwei Knoten eine physikalische Verbindung aufbauen, dann besitzen die Knoten nicht unbedingt gleichzeitig zu allen anderen Knoten zu denen eine logische Verbindung existiert auch eine physikalische Verbindung. Im Fall von mobilen Geräten ist es sogar sehr wahrscheinlich, dass zwischen den beiden Knoten während eines Abgleichs die einzige physikalische Verbindung besteht. Dies hat zur Folge dass bei einem Datenabgleich andere Knoten nicht befragt werden können. Nur die beiden Knoten können miteinander kommunizieren.

Um eine erfolgreiche Synchronisation zwischen den Knoten zu ermöglichen, müssen die beiden aus ihrem Wissen (ihren Zuständen) entscheiden können, welche Datensatz Version aktueller ist oder ob ein Konflikt vorliegt.

Kausale Historien Dies ist ein wohl studiertes Problem aus dem Bereich der Verteilten Systeme. Beide Knoten tauschen die Geschichten (Historien) aller Änderungen (Hinzufügungen, Ersetzungen, Löschungen) aus. Dadurch können sie feststellen, ob der andere Knoten eine Änderung auf einem Datensatz besitzt, welche noch nicht bekannt war.

Beispiel: *Herr Maier hat in seinem Mobiltelefon den Namen von Herrn Schmidt korrigiert. Am nächsten Tag fällt Herr Maier das Gleiche am Computer auf, Herr Maier vertippt sich allerdings und schreibt* Schmiedt.

Beide Datensätze besitzen eine Änderung in ihrer Historie, die der andere Knoten nicht kennt. Es besteht ein Konflikt. Sind diese beiden Historien gleich, dann muss der Datensatz nicht ausgetauscht werden. Kennt Knoten A eine Änderung am Datensatz von der Knoten B nichts weiß, Knoten B kennt alle vorherigen Änderungen die neue Änderung aber nicht, dann hat B einen veralteten Datensatz und erhält die aktuelle Version von A.

Dieses Konzept wird *Kausale Historie* genannt [KWK03]. Die Historie ist geordnet, d.h. die Änderungen sind bezüglich ihrer Relation zueinander angeordnet.

Beispiel: *Ein Datensatz wird erstellt und die Änderung e_0 wird protokolliert. Die Hinzufügung eines Datensatz ist die erste Änderung. Dann folgt eine Ersetzung e_1, dann eine weitere Ersetzung e_2. Die geordnete Historie sieht wie folgt aus: $e_0 \rightarrow e_1 \rightarrow e_2$. Die Änderungen hängen kausal voneinander ab. e_1 hängt kausal von e_0 ab, d.h. erst wenn der Datensatz hinzugefügt wurde, kann auch eine Ersetzung stattfinden.*

Kausal abhängig bedeutet, dass eine Änderung e_j nicht ohne die Änderung e_i hätte auftreten können. Eine solche Beziehung zwischen zwei Ereignissen wird auch *happens before* genannt. Ereignisse die nicht kausal abhängig sind, nennt man nebenläufig. [Hof94]

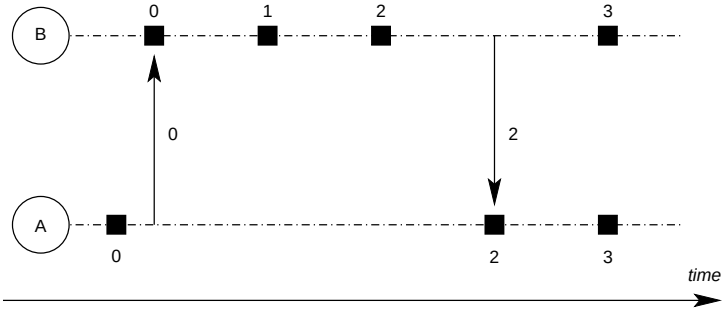

Abbildung 4.12.: Kausale Historie

Zur Erkennung von Konflikten oder der neueren Version ist eine Ordnung auf der Historie in der Regel nicht nötig. Nur wenn Änderungen als Deltas übertragen werden sollen, ist eine Ordnung auf den Änderungen nötig.

Beispiel: *In Abbildung 4.12 kennt Knoten A* e_0 *und Knoten B kennt* e_0 *bis* e_2*. Damit Knoten A den gleichen Zustand des Datensatzes erreicht, muss erst* e_1 *und dann* e_2 *auf* e_0 *angewendet werden, genau wie es in B geschah.*

Nur so ist es möglich, dass beide Knoten nach dem Abgleich den gleichen Zustand besitzen. Dies nennt man die Herstellung einer *konsistenten Sicht.*

Der Speicherbedarf für Kausale Historien wächst sehr schnell. Diese Datenmengen können reduziert werden, wenn keine Deltas sondern vollständige Datensätze übertragen werden. Dadurch ist die Speicherung von Deltas nicht nötig und nur die aktuelle Version des Datensatzes ist relevant.

Vektorzeiten Es stellt sich die Frage, ob es eine Möglichkeit gibt, Kausalaussagen mit weniger Verwaltungsdaten zu treffen. Eine Lösung aus dem Bereich der Verteilten Systeme sind Vektorzeiten [Hof94; Fid91; Mat89]. Jeder Knoten besitzt einen Zähler der monoton bei jeder Änderung wächst. Es handelt sich dabei um eine logische Uhr. Eine physikalische Uhr wäre die tatsächliche Uhrzeit. Jeder Knoten erhält eine eindeutige Bezeichnung. In den folgenden Beispielen sind dies A, B, C und D, die im Gesamtsystem der logischen Netzwerkverbindungen eindeutig sind. Bezeichnungen könnten aber auch global eindeutig sein. Jede Änderung erhält einen Zeitstempel, der eine Kombination aus dem aktuellen Knoten und der logischen Uhr darstellt, z.B. A1 für die erste Änderung

im Knoten A. Jeder Datensatz besitzt einen Vektor, der für jeden Knoten im System einen Wert enthält. Entsprechend könnte ein Vektor eines neu angelegten Datensatzes wie folgt aussehen: $(1, 0, 0)^T$, wobei die erste Zahl für Kopie A, die zweite für Kopie B und die dritte für Kopie C steht. Zur besseren Anschauung wird im Folgenden immer die Kopie vor die Zeit geschrieben: (A1, B0, C0). Beim Abgleich der Daten werden diese Vektoren untereinander Komponentenweise verglichen:

- $\vec{u} = \vec{v}$
  Zwei Vektoren sind gleich, wenn sie in jedem Element übereinstimmen $\forall i, \vec{u}_i = \vec{v}_i$, z.B. (A1, B6, C4) = (A1, B6, C4).

- $\vec{u} < \vec{v}$
  Der Vektor $\vec{u}$ ist gegenüber $\vec{v}$ kleiner, wenn gilt $\forall i, \vec{u}_i \leq \vec{v}_i$ und $\vec{u} \neq \vec{v}$, z.B. (A1, B6, C4) < (A1, B7, C4), da $\vec{u}_2 < \vec{v}_2$ ist. Wenn die Zusatzbedingung $\vec{u} \neq \vec{v}$ nicht gelte, dann können $\vec{u}$ und $\vec{v}$ nach $\forall i, \vec{u}_i \leq \vec{v}_i$ auch gleich sein. Es soll aber strikt die Kleiner-Relation gelten.

- $\vec{u} > \vec{v}$
  Der Vektor $\vec{u}$ ist größer als $\vec{v}$, wenn gilt $\forall i, \vec{u}_i \geq \vec{v}_i, \vec{u} \neq \vec{v}$, z.B. (A9, B6, C4) > (A1, B6, C4), da $\vec{u}_1 > \vec{v}_1$ ist.

- $\vec{u} \parallel \vec{v}$
  In diesem Fall sind $\vec{u}$ und $\vec{v}$ inkompatibel, d.h. es gilt weder $\vec{u} = \vec{v}$, $\vec{u} < \vec{v}$ noch $\vec{u} > \vec{v}$. In (A9, B6, C4) $\parallel$ (A1, B7, C4) ist $\vec{u}_1 > \vec{v}_1$ und $\vec{u}_2 < \vec{v}_2$.

Diese Vergleiche der Vektoren werden beim Datenabgleich erstellt. Ist der Vektor eines Datensatzes in beiden Kopien gleich, dann liegt beiden die gleiche Version vor und der Datensatz muss nicht übertragen werden.

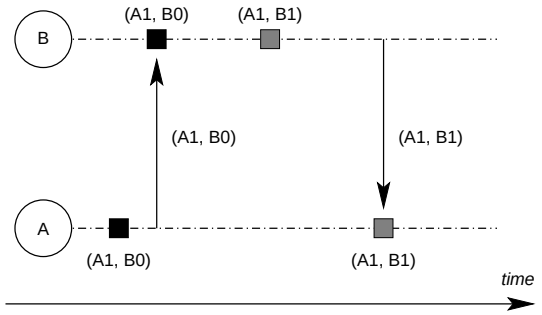

Abbildung 4.13.: Vektorzeit – A übernimmt die Änderung von B

Ist der Vektor in Kopie A größer ($\vec{u} > \vec{v}$), dann besitzt B mindestens eine Änderung, die A noch nicht bekannt war. A übernimmt den Datensatz von B wie in Abbildung 4.13 gezeigt.

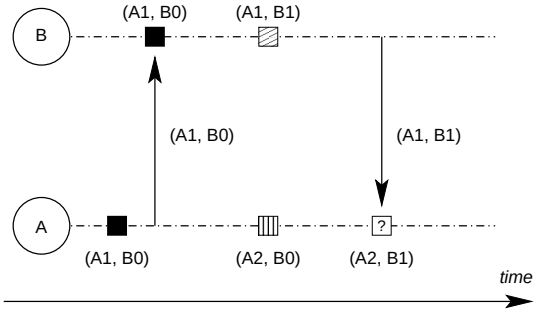

Abbildung 4.14.: Vektorzeit – zwei konkurrierende Änderungen

Sind die beiden Vektoren inkompatibel ($\vec{u} \parallel \vec{v}$), dann besitzt sowohl A als auch B eine Änderung, welche die andere Kopie noch nicht kannte. Es besteht ein Konflikt. Der Konflikt wird aufgelöst und der Vektor wird aus dem komponentenweisen Maximum gebildet, z.B. wird in Abbildung 4.14 aus (A1, B1) $\parallel$ (A2, B0) nach der Konfliktresolution (A2, B1):

$$\max(\vec{u}, \vec{v}) = \forall i, 0 \leq i \leq |\vec{u}|, \max(\vec{u}_i, \vec{v}_i).$$

Es wird die Größe n der Vektoren bestimmt und für jedes Komponentenpaar das Maximum übernommen.

Dynamische Vektoren Ein System welches Vektorzeiten nutzt, kann Zyklen in der Topologie unterstützen, aber die Anzahl der Kopien im System ist fest vorgegeben. Um dies zu lösen, wandelt man die Vektoren in Mengen um, in denen jede Kopie maximal einmal vertreten ist. Jeder Zeitstempel muss dann mit dem jeweiligen Knoten in Verbindung gebracht werden können, d.h. {1, 0, 0} kann nicht mehr geschrieben werden. Wie es bereits zur besseren Anschauung geschah, muss {A1, B0, C0} verwendet werden. Die geschweiften Klammern deuten eine Menge an, runde Klammern repräsentierten einen Vektor. Diese Menge kann dynamisch um neue Knoten ergänzt werden. Dies geschieht immer dann, wenn das Maximum über zwei Mengen gebildet wird oder eine Änderung in der lokalen Kopie auftritt. Kopie D besitzt eine Menge mit {A1, B6, C4}. Erst wenn in Kopie D eine Änderung eintritt, wird die Menge z.B. auf {A1, B6, C4, D12} erweitert. Die Regeln der Vergleiche ändern sich nur minimal. Besitzen zwei Mengen nicht die gleichen Kopien, sind diese automatisch inkompatibel.

Speicherplatzproblem Vektorzeiten benötigen immer noch sehr viel Speicherplatz. Pro Datensatz muss ein Vektor über alle Kopien mitgeführt werden. Die Vektordaten von gelöschten Datensätzen, müssen bei dem vorgestellten Verfahren erhalten bleiben und können nicht gelöscht werden. Kopien die nicht mehr an der Datensynchronisation beteiligt sind, bleiben für immer in den übrigen Kopien gespeichert. Dies ist das Speicherplatzproblem von Vektorzeiten.

Zeitdauerproblem Bei einem Abgleich müssen die Metadaten von allen Datensätze ausgetauscht werden. Da es sich um gewisse Datenmengen handelt, führt dies zu länger dauernden Synchronisationen. Bei langsamen Verbindungen und vielen Datensätzen führt dies zu Abgleichen, die Stunden dauern können. Dies ist das Zeitdauerproblem von Vektorzeiten.

Mangelnde Konfliktresolution Bei einer Konfliktresolution kann die Version aus Knoten A oder aus Knoten B übernommen werden. Es kann auch eine neue Version aus Teilen von A und B entstehen. Diese drei Fälle müssen korrekt an andere Knoten weiter gegeben werden bzw. wenn es sich um eine Ein-Weg Synchronisation handelt, dann muss diese Entscheidung auch bei einer späteren Synchronisation in die andere Richtung vermittelt werden.

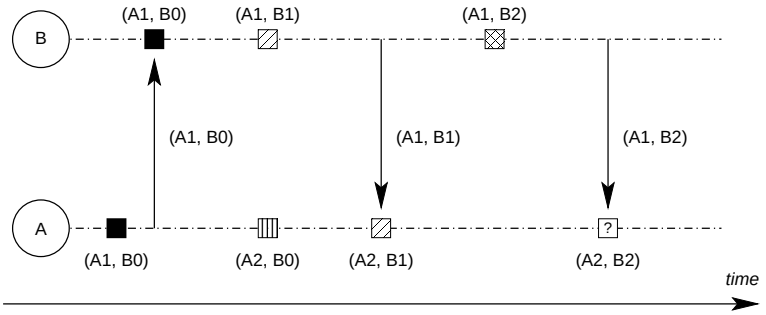

Abbildung 4.15.: Konfliktlösungen werden nicht gespeichert

Vektorzeiten können dies nicht leisten. Beim letzten Abgleich wird erneut ein Konflikt gemeldet, obwohl keiner vorliegt. A hat die Version von B übernommen. B ändert seine Version erneut, d.h. dessen Version ist die aktuellste im Gesamtsystem. Obwohl die Version auf A nicht geändert worden ist und obwohl diese Version der früheren von B entspricht, wird erneut ein Konflikt gemeldet. Die Vektoren in Abbildung 4.15 sind inkompatibel, geben aber in Wirklichkeit keinen Konflikt an:

- Kopie A hat den Vektor {A2, B1} und

- Kopie B hat den Vektor {A1, B2}.

Beim vorhergehenden Abgleich wird korrekterweise kein Konflikt gemeldet. [CJ05, Abschnitt 3.3.1][CJ05, Abschnitt 5.1]

Exkurs Beginn

Angenommen nach einer Konfliktresolution würde nicht immer das Maximum gebildet, sondern nur wenn die lokale Version oder eine Mischung aus beiden Versionen gewinnt. Gewinnt die entfernte Version, wird deren Vektor übernommen. In diesem Fall kann die Vektorzeit im Vergleich zu vorher kleiner werden.

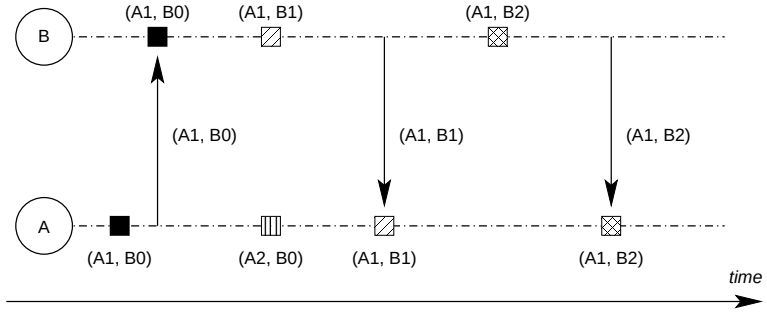

Abbildung 4.16.: Kein Maximum nach Konfliklösung

Durch diese Regelanpassung wird in Abbildung 4.16 kein unnötiger Konflikt gemeldet. Allerdings werden dann in anderen Situationen Konflikte gemeldet, die nicht existieren.

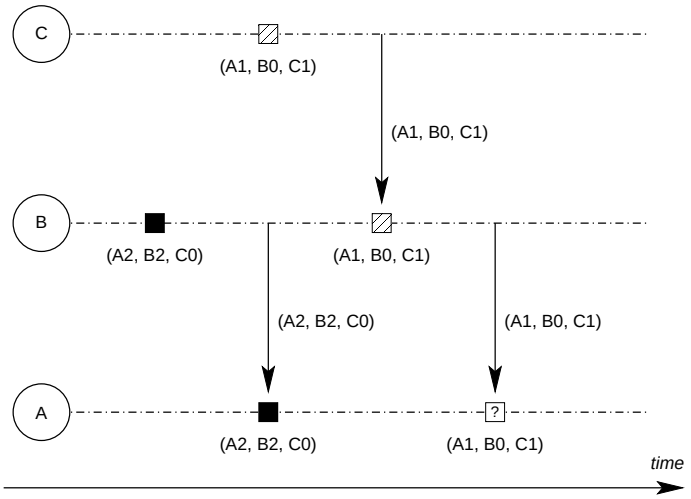

Abbildung 4.17.: Trotzdem erneuter Konflikt

In Abbildung 4.17 hat B bereits die Version von C als aktuellste Version anerkannt. Würde immer das Maximum über die Vektoren nach einem Konflikt gebildet werden, dann würde C korrekterweise keinen Konflikt melden.

Exkurs Ende

Zusammenfassung Vektorzeiten Das Speicherplatzproblem, das Zeitdauerproblem und die mangelnde Konfliktresolution stellen Probleme für den Einsatz von reinen Vektorzeiten in der mobilen Datensynchronisation dar. Eine Reihe von Projekten versucht, diese Probleme zu lösen und das nächste Kapitel II wird dazu einige Verfahren vorstellen.

Eine konsistente Sicht auf das Gesamtsystem ist nötig, wenn eine Netzwerktopologie mit beliebigen Graphen unterstützt werden soll und darin Zyklen vorkommen können. Kausale Historien sind nur bei Delta Verfahren nötig. Vektorzeiten eignen sich, wenn bei einer Änderung immer der gesamte Datensatz übertragen wird.

Abgrenzung zu Log Dateien Log Dateien liefern Unterstützungen für Sterntopologien aber auch für Cluster und Hierarchien. Log Dateien (mit eindeutigen Bezeichnungen für Datensätze) eignen sich allerdings nicht für Graphen, die Zyklen enthalten können.

Beispiel: *In Abbildung 4.18 ändert Knoten A den Datensatz 67 und trägt dies in sein Log mit dem Zeitstempel 3 ein. Knoten B kennt alle Änderungen bis zum Zeitpunkt 2 und erhält daher den gerade geänderten Datensatz. Kopie C kennt alle Änderungen von A bis zum Zeitpunkt 2 und von Kopie B bis Zeitpunkt 4. Wenn Kopie C erst mit A abgleicht, erhält C den neuen Datensatz. Wenn dann Kopie C mit B abgleicht erhält C den neuen Datensatz, der ursprünglich von A kam. Allerdings liegt lokal eine Änderung auf diesem Datensatz vor. Ein Konflikt wird gemeldet, obwohl keiner vorliegt.*

Das Problem scheint in Abbildung 4.18 die nicht-Weitergabe des ursprünglichen Zeitstempels zu sein. Selbst wenn mit physikalischen Uhren gearbeitet wird, gibt es Probleme.

Beispiel: *In Abbildung 4.19 ändert Kopie A den Datensatz um 16 Uhr. Um 18 Uhr erhält Kopie B diesen Datensatz und sortiert die Änderung chronologisch in seine Log Datei ein. Kopie C kennt Kopie B aber bis 17 Uhr. Kopie C fragt Kopie B an und erhält den neuen Datensatz nicht.*

Log Dateien sind nicht für Zyklen entworfen, außer die gesamte Log Datei würde immer übertragen.

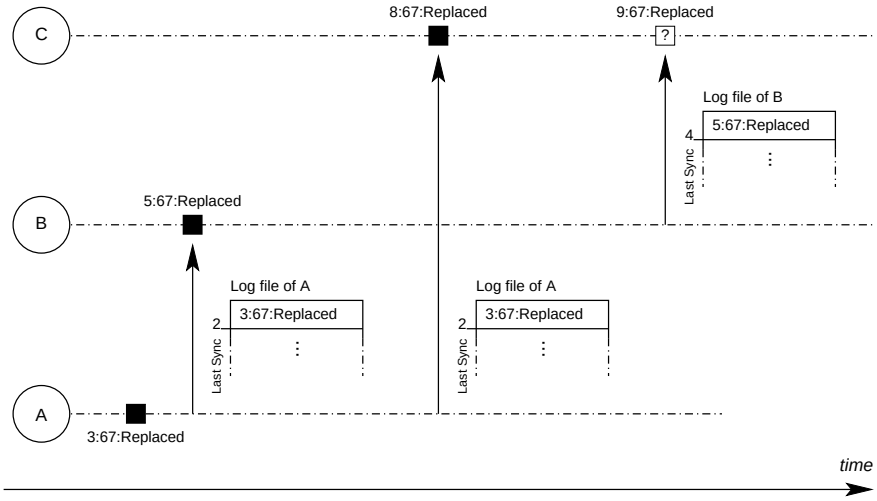

Abbildung 4.18.: Log Datei und Zyklen – normales Verhalten

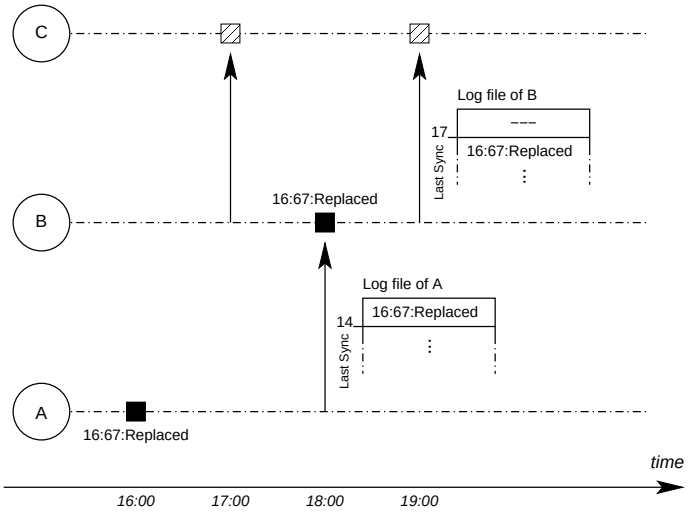

Abbildung 4.19.: Log Datei und Zyklen – Speicherung der ursprünglichen Zeitstempel

4.10. Konfliktresolution: Verfahren

Konflikte ergeben sich aus den vorgestellten Schreib/Schreib und Lese/Schreib Problemen. Wie im vorherigen Abschnitt 4.9 beschrieben, wird je nach Technik der Änderungserkennung nur ein Teil dieser Konflikte erkannt. Dieses letzte Kriterium nennt unterschiedliche Verfahren zur Konfliktauflösung, die zur Wahrung der Konsistenz dient und weitere Konflikte im Gesamtsystem vermeidet.

4.10.1. Keine Konfliktresolution

4.10.1.1. Duplikate

Dieser Fall tritt normalerweise nur ein, wenn einzelne Datensätze nicht mehr eindeutig in Beziehung stehen. D.h. Datensatz 4 in Knoten A und Datensatz 4 in Knoten B stehen nicht mehr in ihrer Äquivalenz Beziehung. Das Abgleichverfahren weiß nicht mehr, dass die beiden Datensatz Kopien ursprünglich denselben Datensatz repräsentierten.

Es kann aber auch sein, dass ein Abgleichverfahren mit Duplikaten Informationsverlust vermeiden will. Das Verfahren kann die Datensätze nicht anders in Einklang bringen, will aber die Synchronisation nicht abbrechen und auch keine Änderungen verlieren. Ein intelligenteres System oder meist eher ein menschlicher Benutzer soll dann später die Duplikate entfernen bzw. zusammenführen.

4.10.1.2. Abbruch

In diesem Fall wird bei der Erkennung eines Konflikts die Synchronisation für diesen Datensatz oder die gesamte Datenbank abgebrochen. Dieser Fall ist eher unüblich aber eine berechtigte Möglichkeit, wenn die Annahme besteht, dass Konflikte in einem gegebenen System überhaupt nicht hätten auftreten dürfen. Auch hier ist wieder ein intelligenteres System oder der menschliche Benutzer gefragt, um die Konflikte aufzulösen.

4.10.1.3. Vermeidung

Auf jeder Kopie dürfen keine Änderungen durchgeführt werden. Der Datensatz darf nur auf der Kopie bearbeitet werden, auf dem der Datensatz entstand. Es ergeben sich zwei Stellungen von Kopien: Die original Kopie besitzt alle Lese- und Schreibrechte. Alle anderen Kopien besitzen (maximal) Leserechte. [RU01, Abschnitt 4.4]

Auf der ersten Blick mag diese Regel wenig Sinn machen, da sich bis jetzt alle Beispiele um Herrn Maier drehten, der mehrere Geräte sein Eigen nannte und auf allen

Geräten die gleichen Daten haben wollte. Diese Geräte verfügten auch immer über Eingabemöglichkeiten die Daten zu manipulieren, um neue Kontakte im Adressbuch hinzuzufügen, egal wo Herr Maier sich gerade befindet, egal welches Gerät zur Hand ist.

Es gibt aber durchaus Möglichkeiten wo eine Konfliktvermeidung Sinn macht:

- Gemeinschaftliches Arbeiten
  Zwei Benutzer tauschen untereinander ihre Kalenderdaten aus. Auf Jörgs PDA sind alle Termine von Stephan gespeichert. Jörg hat zur Wahrung Jörgs Privatsphäre nicht den Grund der Termine erhalten sondern nur die Zeiträume. Dies aber reicht Jörg aus, um einen gemeinsamen Termin zu definieren und an Stephan weiter zu geben. Es macht keinen Sinn, dass Jörg Termine von Stephan ändert.[2] [RU01]

- Ein Gerät welches keine Datenmanipulationen zulässt, ist ein weiteres Beispiel für die Anwendung der Konfliktvermeidung. Das Gerät hat zwar Zugriff auf eine Kopie der Daten und gleicht diese auch mit einer anderen Kopien ab. Das Gerät selbst kann aber keine Einträge ändern, sondern nur neue aufnehmen. Ein konkretes Beispiel wäre ein Mobiltelefon für den Freizeitbereich. Auf einer Party möchte man so wenig wie möglich mitnehmen, schließlich könnte etwas verloren gehen. Daher sollte das Mobiltelefon so klein wie möglich sein. Eine Möglichkeit wäre, dass es keine Eingabemöglichkeiten bietet. Neue Bekanntschaften auf der Party können diesem Mobiltelefon elektronische Visitenkarten zukommen lassen. Diese können dann später mit den anderen Datenkopien synchronisiert werden. Ein Konflikt wird vermieden, da es sich bei den Visitenkarten um neue Datensätze handelt, d.h. ein Schreib/Schreib Konflikt auf ein und demselben Datensatz ist ausgeschlossen.

4.10.2. Automatische Konfliktresolution: Regel basiert

Falls ein System vollständig automatisiert ist und kein unmittelbarer menschlicher Benutzer existiert, eignen sich feste Regeln für die Konfliktresolution. Aber auch die Wahl der Architektur kann vorgeben, dass nur automatische Resolutionen möglich sind. Gerade bei einer Middleware [Byo93] ist es der Sinn, dass ein Programm nicht unbedingt über die Existenz von Datensynchronisation im System wissen muss. Die Middleware hat dann nicht unbedingt eine Möglichkeit einen Konflikt zurück zum Programm zu melden oder gar das Programm anzuhalten, um einem Benutzer Fehlermeldungen bezüglich Konflikte während einer Synchronisation aufzuzeigen.

[2] In diesem Szenario wäre allerdings neben der Schreib/Schreib Konfliktvermeidung auch eine Lese/Schreib Konflikterkennung sinnvoll. Wenn sich gerade noch Termine bei Stephan ergeben haben, sollte Jörg dies erfahren.

Die Netzwerktopologie des Gesamtsystems kann gewisse Regeln nahe legen. Diese Regeln können auch von einem Benutzer aufgestellt werden. Andere Regeln ergeben sich aus der Technik zur Änderungserkennung.

Im Folgenden werden typische Regeln für die Konfliktresolution in aktuellen Datensynchronisationsverfahren aufgezeigt. [LKC04]

Server gewinnt

Im Konfliktfall wird der Datensatz des Servers übernommen. Diese Regel geht davon aus, dass eine Server/Klient Beziehung in der Netzwerktopologie existiert und dass ein Server eine höhere Vertrauenswürdigkeit als (mobile) Klienten im System hat. Der (stationäre) Server dient als letzte Weisheit.

Beispiel: *In der Firma Schraubendreher werden alle Dateneingänge im Server nochmals von einem Menschen kontrolliert und gegebenenfalls korrigiert. Entsprechend gelten die Daten auf dem Server vertrauenswürdiger.*

Klient gewinnt

Im Gegensatz zur vorherigen Regel wird im Konfliktfall der Datensatz des Klienten übernommen. Auch diese Regel geht davon aus, dass eine Server/Klient Beziehung besteht. Es wird weiter davon ausgegangen, dass die (mobilen) Klienten immer aktuellere Daten liefern als der (stationäre) Server. Sinnvoll ist diese Regel besonders dann, wenn sofort nach einer Änderung, eine Datensynchronisation durchgeführt wird. Dann war die Version im Klienten auch die Neuste.

Empfänger gewinnt

Diejenige Kopie, die darauf aufmerksam gemacht wird, dass eine Änderung vorliegt, akzeptiert diese nicht, sondern behält ihre eigene Änderung. Diese Regel geht davon aus, dass die lokalen Daten im Zweifel die Korrekten sind und keine lokalen Daten verloren gehen dürfen. Dies ist zum Beispiel sinnvoll, wenn verschiedene Benutzer eine Datensynchronisation durchführen und man eigene Änderungen (z.B. gemachte Anmerkungen zu einem Adressbucheintrag) behalten möchte.

Urheber gewinnt

In diesem Fall gewinnt derjenige, der die Datensynchronisation dieses Datensatzes ausgelöst hat. Es ist damit das Gegenteil zur *Empfänger gewinnt* Regel. Sinnvoll kann dies

sein, wenn ein Benutzer mehrere Geräte besitzt, die Synchronisation aber immer vom Computer aus gestartet wird und dieser die Maß gebende Instanz im Gesamtnetz darstellt.

Autor gewinnt

Der Autor hat den Datensatz erstellt. Ist der Autor an der Datensynchronisation beteiligt und es entsteht ein Konflikt in *seinem* Datensatz, dann wird seine Version genommen. Es wird angenommen, dass es im Gesamtsystem eine Autorenverwaltung gibt, d.h. jedem Datensatz können Verwalter zugeordnet werden. Ein Verwalter könnte zum Beispiel derjenige sein, der den Datensatz ursprünglich angelegt hat und diesen in der Zukunft auf seine Richtigkeit und Vollständigkeit überwacht (*Autor*).

Höchste Priorität gewinnt

Jede Kopie im Netzwerk erhält eine Priorität. Diese Priorität stellt die Vertrauenswürdigkeit oder Stellung im Gesamtsystem dar. Im Konfliktfall gewinnen Kopien mit höherer Priorität gegenüber Kopien mit geringer Priorität.

Neuste Version gewinnt

Wenn zwei Kopien Datensätze austauschen und auf beiden Seiten ist derselbe Datensatz bearbeitet worden, dann wird in diesem Konfliktfall der Datensatz übernommen der zuletzt bearbeitet worden ist. Diese Regel ist sehr viel versprechend Gerade, falls es nur einen menschlichen Benutzer gibt, der aber mehrere Geräte besitzt. Normalerweise wird ein Benutzer einen Datensatz immer soweit ändern, dass die lokale Kopie bereits alle Änderungen enthält. In diesem Fall ist der zuletzt geänderte Datensatz zu bevorzugen.

Beispiel: *In seinem Mobiltelefon fügt Herr Maier zum Eintrag der Firma Lachnit eine Fax Nummer hinzu. Diese Informationen hat er von einer Visitenkarte. Dabei bemerkt er, dass sich die Firmenadresse geändert hat. Da die Tatstatur seines Mobiltelefons nicht sehr komfortabel ist, löscht Herr Maier das Adressfeld, damit keine falschen Informationen im System verbleiben. Nun ließ Herr Maier sein Mobiltelefon auf der Arbeit liegen. Abends am Computer ändert er den Eintrag der Firma Lachnit entsprechend der aktuellen Visitenkarte (Fax Nummer und Adresse). Er gleicht seinen heimischen Computer mit dem in der Firma ab. Am nächsten morgen wieder auf der Arbeit angekommen, gleicht er dann sein Mobiltelefon mit dem Computer ab.*[3] *Wenn Herr Maiers Synchronisations-Software so eingestellt ist, dass immer die neuste Version gewinnt, dann wird nun der*

Datensatz auf seinem Mobiltelefon verworfen, da dieser vor dem Datensatz auf dem Computer geändert wurde.

Die neuste Version zu übernehmen, ist wegen ihrer Praxis-Nähe eine häufig angestrebte Regel. Allerdings stützt sie sich auf zwei Annahmen, die für ein verteiltes System gravierende Nachteile mit sich bringen:

1. Alle Geräte im System müssen sich über die Uhrzeit einig sein.
2. Der letzte Änderungszeitstempel muss beim Transfer übernommen werden und darf nicht durch einen lokalen Änderungszeitstempel überschrieben werden.

Zeit In verteilten Systemen ist es problematisch verschiedene Uhren abzugleichen. Jedes Gerät besitzt eine eigene Uhr, welche eine Drift besitzt. Eine Drift gibt an, um wie viel sich eine Uhr über eine Zeitdauer verstellt.[4] Uhren laufen entweder langsamer oder schneller, weswegen sie regelmäßig abgeglichen werden müssen. Uhren besitzen eine verschieden starke Drift. Selbst wenn man zwei Uhren aus der gleichen Produktion nimmt, kann man nach einer bestimmten Zeitdauer davon ausgehen, dass die Uhren nicht mehr dieselbe Aussage über die Uhrzeit treffen.

Bei einem Ein-Benutzer Datensynchronisations-System sind diese Driften aber nicht relevant. Ein Benutzer wird nicht innerhalb von Sekunden den gleichen Datensatz auf zwei verschiedenen Geräten ändern. Viel wichtiger ist hier, dass die Systeme fast die gleiche Uhrzeit besitzen oder wenigstens die Zeitdifferenz zwischen den Geräten ermittelt werden kann.

Wenn Gerät A 12 Uhr anzeigt und B 10 Uhr, dann stellt sich die Frage, ob sich die beiden Geräte auch in der gleichen Zeitzone befinden und damit die gleiche Uhrzeit repräsentieren oder ob sie Stunden auseinander liegen. Die nächste Frage ist dann, ob die jeweiligen Geräte überhaupt Informationen wie Zeitzonen angeben und verwalten können. Gerätetyp übergreifende Abgleichverfahren können nicht davon ausgehen, dass alle Geräte auch einheitlich Uhrzeiten austauschen können.

Nicht jedes Gerät verfügt über die Möglichkeit die Uhrzeit regelmäßig mit einem Netzwerk abzugleichen. Den Uhrenabgleich bei der Datensynchronisation durchzuführen, ist ebenfalls nur bedingt sinnvoll. Eine Synchronisation wird meist in solchen Abständen geschehen, die den Drift haben zu groß werden lassen. Wird dann die Uhrzeit verstellt,

[3] Für das Abgleichverfahren ist der Eintrag der Firma Lacknit, die kleinste Einheit für die Änderungen erkannt werden (Stichwort *Datensatz*). Das Synchronisationsverfahren erkennt, dass sich der gesamte Eintrag und nicht nur das Adressenfeld geändert hat.

[4] Dies ist eine natürliche Ungenauigkeit, die auftritt, um ein exaktes Maß zu bestimmen.

können Zeitstempel in der Zukunft liegen oder neuere Änderungen vor alten Änderungen einsortiert werden.

Übernahme des Änderungszeitstempels Speichert ein Gerät beim Abgleich einer Änderung die aktuelle Zeit, werden spätere Abgleiche diesen Zeitstempel als Entscheidungsbasis nutzen. Dann kann die Entscheidung zu Gunsten der falschen Version fallen, da der Zeitstempel nicht mehr die Änderung angibt sondern den letzten Abgleich.

Beispiel: *Wie in Abbildung 4.20 zu sehen ist, hat Herr Maier den Adressbucheintrag der Firma Lachnit auf seinem PDA um 15 Uhr geändert. Um 16 Uhr änderte er denselben Eintrag auf seinem Mobiltelefon. Um 17 Uhr gleicht Herr Maier seinen PC mit dem PDA ab. Danach gleicht er das Mobiltelefon mit dem PC ab. Bei korrekter Anwendung der* neusten Version *Regel, müsste der Eintrag des Mobiltelefons von 16 Uhr gewinnen. Der Eintrag wird vom PDA auf den PC übertragen. Dort liegt kein Konflikt vor und der Eintrag behält seinen Änderungszeitstempel von 15 Uhr. Beim Abgleich mit dem Mobiltelefon liegt ein Konflikt vor. Anhand der Zeitstempel gewinnt die Version des Mobiltelefons. PDA bzw. PC kennen eine letzte Änderung um 15 Uhr. Das Mobiltelefon kennt eine neuere Änderung von 16 Uhr.*

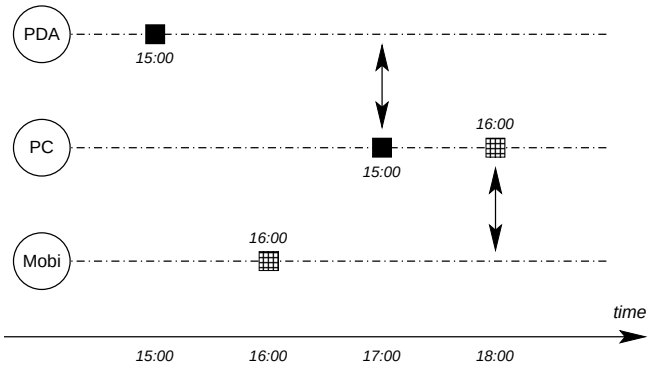

Abbildung 4.20.: Physikalische Uhrzeiten – richtig angewendet

Den Eintrag mit einem aktuellen Änderungszeitstempel (17 Uhr) beim Abgleich zwischen Computer und PDA zu versehen, wäre falsch. In Abbildung 4.21 speichert der PC einen Zeitstempel mit 17 Uhr. Beim Abgleich mit dem Mobiltelefon gewänne die Version vom PC, da dass Mobiltelefon eine letzte Änderung von 16 Uhr besitzt, der Computer aber meint die letzte Änderung sei um 17 Uhr erfolgt, obwohl keine Änderung vorlag, sondern nur der Eintrag vom PDA kopiert worden ist.

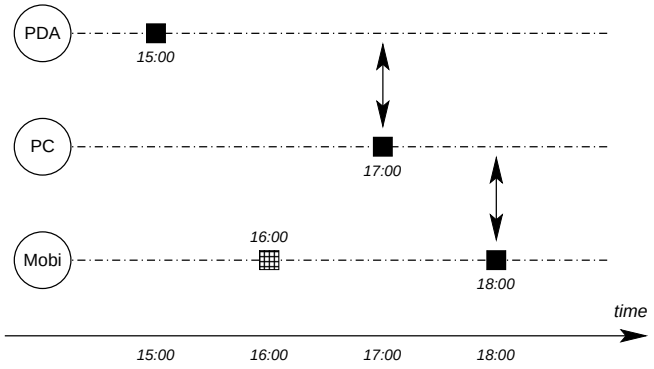

Abbildung 4.21.: Phsysikalische Uhrzeiten – ohne Übernahme der Zeitstempel

Unterschiedlichste Version gewinnt

Diese Regel besagt, dass diejenige Version übernommen wird, die sich am meisten von der Version des letzten Abgleichs unterscheidet. Erneut sei das Beispiel aus dem Abschnitt 4.9.1 über Zustand basierte Abgleichverfahren genannt.

Beispiel: *Abbildung 4.10 verdeutlichte dies. Herr Maier legt einen Datensatz mit 705 Schrauben an. Übertragt diesen Datensatz auf seinen PDA. Im PC ändert Herr Maier den Datensatz auf 706 Schrauben, im PDA auf 707 Schrauben und danach im PC wieder auf 705 Schrauben. Die 707 Schrauben auf dem PDA werden beim nächsten Abgleich gewinnen und im PC übernommen.*

Bei dieser Regel gewinnt nicht die Version, die am häufigsten geändert worden ist, sondern diejenige, die am Unterschiedlichsten zur letzten Version des gemeinsamen Abgleichs ist. Diese Regel ist ungewöhnlich und wird von keinem bekannten Abgleichverfahren explizit verwendet, muss aber der Vollständigkeit halber erwähnt werden. Sie findet implizit bei Verfahren Anwendung, die eine Zustand basierte Änderungserkennung verfolgen. Im Abschnitt 4.9.1 sind bereits die Ursachen für dieses Problem aufgeführt worden. Im nächsten Kapitel II wird genauer auf Vertreter Zustand basierter Verfahren eingegangen, aber schon hier sei zu ihrer Verteidigung aufgeführt, dass z.B. Unison auch Zeitstempel der Datensätze berücksichtigt, was dazu führt, dass diese Regel im täglichen Gebrauch nicht auftritt.

4.10.3. Programmautor liefert Regeln für die Middleware

Eine Middleware für die Datensynchronisation weiß meist nicht, was für Daten abgeglichen werden. Eine Regel für alle Daten zu wählen, ist unpassend.

Beispiel: *Familie Maier hat einen elektronischen Bestellzettel im Kühlschrank. Herr Maier erinnert sich unterwegs daran, dass er am Wochenende noch zwei Eier für einen Kuchen benötigt. Dies trägt er in seinen PDA ein. Daheim hat sein Sohn ebenfalls zwei Eier direkt am Kühlschrank eingetragen, die er für eine Party benötigt. Herr Maier kommt nun nach Hause. Sein PDA synchronisiert sich automatisch mit dem Kühlschrank und überträgt die Bestellung von zwei Eiern.*

In diesem Fall wäre es notwendig, dass der Kühlschrank die beiden Werte addiert und vier Eier bestellt. Das Verfahren muss die Daten kennen und wissen, dass es sich bei den Wertangaben um Zahlenwerte handelt, die einfach addiert werden können.

Es wurde bereits eine Reihe von Regeln aufgezeigt, aber keine der genannten deckt einen solchen Fall ab. Es ist daher vorteilhaft, wenn ein Programmierer beim Aufruf von Middleware Funktionen eigene Regeln für die Konfliktresolution liefern kann oder wenn die Middleware im Konfliktfall das Anwendungsprogramm für eine Entscheidung anruft, wie der Konflikt gelöst werden soll. Dem Programmierer sind spezielle Datentypen und deren Verwendung bekannt, was ihm ermöglicht optimale automatische Konfliktresolutionen anzubieten. [TTP+95][Mus02, Abschnitt 3.4]

4.10.4. Manuell

Manuell bedeutet, dass ein Abgleichverfahren alle Konfliktfälle signalisiert und der Benutzer, welche Kopie jeweils gewinnt.

Gerade bei einem Ein-Benutzer System ist dies eine hervorragende Lösung, da der Benutzer alle Änderungen durchgeführt hat und am Besten entscheiden kann, welche Daten korrekt oder die Aktuellsten sind. Auch bei einem Mehr-Benutzer System kann eine manuelle Resolution durchaus Sinn machen. Besonders Vorteilhaft wäre es, wenn das System alle an der Änderung beteiligten Benutzer aufzeigt, damit bei diesen im Zweifelsfall nachgefragt werden kann.

4.10.5. Zusammenfassung der Konfliktresolutionen

Eine Datensynchronisation soll so weit wie möglich automatisch ablaufen, schließlich hat ein Benutzer als sein Hilfsmittel für dieses Problem eine Software gewählt. Eine Software kann aber zurzeit noch nicht erahnen, was sich der Nutzer bei einer Änderung gedacht hat. 2 + 2 = 4, doch nur 2 Eier bestellen oder diesen Konflikt signalisieren, damit man dem Sohnemann irgendwelche Schweinereien auf einer Party austreiben kann? Daher arbeiten viele Verfahren nicht nur mit einer Regel, sondern bauen auf mehreren auf. Im

Idealfall können eigene Regeln definiert werden und der Benutzer kann jederzeit manuell in die Konfliktresolution eingreifen.

Teil II.

Synchronisationsverfahren

Zusammenfassung

Im Folgenden werden einige Projekte aus dem Bereich der optimistischen Replikationsverfahren aufgegriffen und anhand der Kriterien aus Kapitel 4 analysiert. Die Einschränkung auf rein optimistische Verfahren begründet sich darin, dass von Endgeräten ausgegangen wird, die nicht zu jeder Zeit eine Verbindung zu einem Netzwerk aufbauen können oder wollen.

5. Forschungsprojekte

5.1. Unison

Bei Unison [PV04] handelt es sich um einen Verfahren zum Abgleich eines kompletten hierarchischen Dateisystems (*Dateisynchronisation*). Zurzeit existiert vom selben Autor ein Projekt namens Harmony [GGM+03], welches die Konzepte aus Unison auf die allgemeinere Datensynchronisation anwendet.

Die Netzwerktopologie von Unison ist sehr eingeschränkt, da nur eine weitere Kopie unterstützt wird. Dies begründet sich unter anderem darin, dass eine formale Spezifikation der Korrektheit des Verfahrens geliefert wird. Unison stellt damit den ersten Schritt dar, Abgleichverfahren auch formal zu beweisen. Unison soll darüber hinaus eine Benutzerapplikation sein. Diese Anforderung führt einer Zustand basierten Änderungserkennung der Daten, die in Abschnitt 4.9.1 näher beschrieben wurde. Der kleinste erkennbare Datensatz ist eine Datei im Dateisystem. Außer den Metainformationen des Dateisystems (z.B. Zugriffrechte und Zeitstempel) wird eine Datei nicht weiter analysiert. Der Benutzer startet einen Abgleich manuell und löst etwaige Konflikte auf.

Die Implementierung [PV98] von Unsion besitzt noch weitere Eigenschaften, die das eigentliche Abgleichverfahren und dessen Einordnung nicht weiter beeinflussen, weswegen diese auch nur kurz erwähnt seien: Unsion bietet ausgeklügelte Algorithmen, um aus den Zuständen der Dateisysteme Unterschiede zu erkennen (*Hierarchie-Baum Transformation*). Da Unison den letzten gemeinsamen Zustand beider Kopien kennen muss, liegt nicht das gesamte Dateisystem lokal ein zweites Mal vor, sondern es werden nur Hashwerte über den Dateiinhalten (*Fingerprints*) gespeichert.

5.2. CIPSync

CIPSync [STA03] ist ein Projekt, welches die Dauer des Datenabgleichs optimieren will. Die Dauer ergibt sich zum einen durch die Menge an zu übertragenden Daten und den Berechnungen, die auf den jeweiligen Kopien durchgeführt werden müssen, um die Unterschiede zwischen den Kopien zu ermitteln. CIPSync konzentriert sich dabei auf

die Datenmengen, die übertragen werden sollen. Die Dauer des Datenabgleichs soll in CIPSync nicht linear mit der Anzahl der Datensätze wachsen sondern mit der Anzahl der Änderungen an einzelnen Datensätzen. Auffällig ist bei diesem Projekt, dass nicht mit einer Log Datei oder Vektorzeit gearbeitet wird, sondern ein charakteristisches Polynom übertragen.

Die Einordnung dieses Projekts in die Kriterien aus dem vorherigen Kapitel fällt schwer, da solche Betrachtungen von den Autoren am eigenen Projekt anders durchgeführt wurden. In [STA02] findet man einen Abschnitt, der sich mit der Skalierbarkeit von verschiedenen Abgleichverfahren auseinandersetzt. Es wird die zu übertragende Datenmenge, der zuleistende Rechenaufwand während dem Abgleich, die Anzahl der Knoten eines Netzwerks, der lokale Speicherbedarf für Metainformationen und eine Robustheit als Kriterium angegeben.

Verglichen werden vier Verfahren darunter auch SyncML [OMA03a] und CIPSync. Interessant sind dabei die getroffenen Einteilungen an Hand einer Skala. Hier wird SyncML als sehr robust eingestuft. Dies wird damit begründet, dass SyncML nicht nur einen zentralen Server beim Datenabgleich nutzt, sondern weitere Netzwerktopologien zulässt. Allerdings wird in der Literatur [HMPT03; LKC04] immer angefügt, dass OMA SyncML DS nur für den Datenabgleich mit einem einzigen zentralen Server optimiert sei. Das Kriterium der Menge der zu übertragenden Daten bleibt ebenfalls unklar. Es ist davon auszugehen, dass hier nicht von dem so genannten *Slow Sync* Modus ausgegangen wird. In diesem Modus müssen alle Datensätze übertragen werden, da die normale Synchronisation aufgrund eines Datenmangels nicht erfolgen kann. SyncML zeige ein gutes Verhalten, obwohl OMA SyncML DS auch im *Slow Sync* Modus arbeiten kann, welches gerade sehr viele Daten überträgt.

Nach den vorliegenden Informationen ist unklar, ob CIPSync für Zyklen innerhalb der Netzwerktopologie geeignet ist. Es wird in dem Vergleich nur eine Unterscheidung nach der Anzahl der unterstützen Knoten durchgeführt. SyncML sei in diesem Punkt schlechter als alle anderen Verfahren, da es Metainformationen proportional in Bezug auf die Anzahl der Knoten verwalten müsse. Die gleiche Bewertung erfolgt ohne Begründung beim lokalen Speicherbedarf.

Ein weiterer Punkt dieses Projekts ist der quantitativer Vergleich von CIPSync mit dem Slow Sync Modus. Bei Slow Sync werden alle Datensätze übertragen. Slow Sync ist damit die untere Meßlatte für ein Abgleichverfahren. Ein Verfahren sollte in den seltensten Fällen mehr Daten übertragen als Slow Sync. Im Mittel darf es auf keinen Fall mehr Daten übertragen, da sonst das Verfahren keinen Vorteil gegenüber dem Slow Sync aufweisen kann.

Dieser Versuch eines Vergleiches zwischen verschiedenen Verfahren war eine Inspiration für die in dieser Ausarbeitung genannten Kriterien. Es benötigt klare quantitative und qualitative Kriterien, um eine Einordnung und ein Vergleich der jeweiligen Verfahren zu erlauben.

> The management of replicated, distributed, databases requires the development of sophisticated algorithms for guaranteeing data consistency and for resolving conflicting updates. Several architectures, such as BAYOU [...] have been proposed to address these important problems. We consider CPISync to be complementary to these architectures. The CPISync methodology permits the efficient determination of the differences between databases, while the mentioned architectures can be used to resolve which data to keep or to update once the differences are known. [STA03]

CIPSync [STA03] sieht sich selbst nicht als vollständiges Abgleichverfahren inklusive von Konfliktresolutionen, sondern beschäftigt sich mit der Problematik ohne weitere Metainformationen schnell Unterschiede zwischen zwei Kopien zu ermitteln. Aufgrund dieser Sonderstellung macht eine Einteilung nach den dargelegten Kriterien nur bedingt Sinn. Einzig die unterstütze Netzwerktopologie wäre interessant. Die von CIPSync angeführte Musterimplementierung synchronisiert einen Computer mit einem PDA. Es ist aber nach dem obigen Zitat davon auszugehen, dass CIPSync nur Unterschiede meldet und eine zusätzliche nicht weiter genannte Konsistenzverwaltung entscheidet, welche Änderung (Hinzufügungen, Löschungen, Änderungen oder eine Konfliktresolution) auf den jeweiligen Datensätzen durchgeführt wird, wobei die Konsistenzverwaltung für die Unterstützung von bestimmten Netzwerktopologien zuständig wäre.

5.3. Footloose

Footloose [PSYC03] ist ein recht neues Projekt, welches auf der Vorarbeit vieler Projekte aufbauen konnte. Der Schwerpunkt dieses Projektes liegt darin, dass ein Benutzer eine Vielzahl an Geräten besitzt und diese nicht wie bei anderen Projekten über das globale Internet verbunden sind, sondern so genannte *Personal Area Networks* bilden, d.h. mehrere Geräte haben einen geringen räumlichen Abstand zueinander ($\approx$ 10 Meter) und bauen darüber eine pysikalische Verbindung auf. Besteht eine solche Verbindung werden Datenabgleiche automatisch vorgenommen. Der Benutzer wird höchstens im Konfliktfall kontaktiert.

Footloose ist für alle Netzwerktopologien also für beliebige Graphen mit enthaltenen Zyklen geeignet. Die unterstützten Datentypen werden nicht weiter bestimmt. Intern wird jedem Datensatz eine ID zugewiesen. Die eigentlichen Daten liegen entweder intern in einem Speicher von Footloose oder bei einer beliebigen Anwendung, welche durch Footloose, wenn nötig, angesprochen wird. Änderungen werden daher von Footloose immer nur Datensatzweit erkannt. Die beschriebene Footloose Implementierung arbeitet nur mit einer fest vorgegebenen Anzahl an Kopien und die Kopien selbst müssen von Anfang an festgelegt werden. Die Autoren von Footloose wollen dies aber in der Zukunft ändern. Bereits die Erwähnung von beliebigen Anwendungen und deren Datentypen macht deutlich, dass es sich um eine Middleware handelt. Andere Anwendungen rufen die Schnittstellen von Footloose auf, um sich und ihre Daten zum Datenabgleich anzumelden. Die einzig große Einschränkung dieses Systems (neben den festen Kopien) liegt darin, dass es nur für einen Benutzer geeignet ist. Dies deckt sich aber mit dem Gedanken hinter einem Personal Area Network. Ein Benutzer hat mehrere Geräte und seine Daten sollen nur zwischen diesen ausgetauscht werden.

Abgleiche erfolgen voll automatisch, wenn sich die Geräte in räumlicher Nähe finden. In der Beispiel-Implementierung erfolgten die Abgleiche allerdings auf einem Computer, der mehrere Kopien enthielt. D.h. Footloose bietet zurzeit noch keine Regeln oder Empfehlungen, wie Netzwerktechnologien automatisch gegenseitige Erkennungen bewerkstelligen könnten. Footloose beschreibt nur, dass die einzelnen Kopien über die Zeit gesehen ein Netzwerk bilden. Diese Forderungen werden dadurch bestärkt, dass angenommen wird, dass der Benutzer meistens ein Gerät mit sich führt, welches dann eine logische Verbindung zwischen Geräten zum Beispiel auf der Arbeit und Daheim herstellt. Da Footloose alle Netzwerktopologien erlaubt, muss es sich bei diesem Gerät nicht immer um dasselbe handeln. Es kann einmal ein Laptop, eine Digitalkamera, ein Musikspieler oder ein Mobiltelefon sein. Die Autoren schlagen allerdings einen Metadatenspeicher vor, den man immer mit sich führt. Eine Besonderheit an Footloose ist dabei, dass nicht jedes Gerät die Daten des Abgleich verstehen muss. Dies bedeutet, dass zum Beispiel ein Mobiltelefon als Zwischenspeicher genutzt werden kann, um Daten an andere Kopien weiter zu reichen. Das Mobiltelefon dient dabei nur als Übermittler aber es bietet keine Anzeige- oder Eingabemöglichkeiten für die Daten.

Damit ein Gerät, welches nur begrenzten Speicher besitzt, nicht für unnötige Daten als Zwischenspeicher missbraucht wird, ist eine Unterscheidung in zwei Geräteklassen entwickelt worden. Ein *schlaues* Gerät kann mit bestimmten Datentypen etwas anfangen und bekundet sein Interesse an solche Datentypen bei allen anderen Geräten. In Footloose wird dies als ein *Wunsch* nach bestimmten Datentypen bezeichnet. Ein *dummes* Gerät

kann mit dem Datentyp nichts anfangen und löscht diesen bei Speichermangel. Aber jedes Gerät sammelt die Wünsche anderer Geräte und leitet diese zusammen mit den eigenen Wünschen an andere Geräte weiter. Dadurch entstehen im Verlauf der Zeit Routingtabellen für alle Datentypen über das gesamte logische Netzwerk. Die Daten werden nicht nur an die kürzeste Route gesendet, sondern an alle, die auf einer entsprechenden Route für solche Datentypen liegen. Ein dummes Gerät kann die unverstanden Daten bei Speichermangel verwerfen, erhält diese Daten beim nächsten Abgleich aber erneut. In Footloose sind somit sehr viele Daten im gesamten Netz verteilt. Footloose versendet dabei Änderungsmitteilungen. Ob die Änderungen nur als Delta oder immer der jeweilige Datensatz mitgeteilt wird, beleibt aber unklar. Es ist davon auszugehen, dass die jeweilige Anwendung, die Footloose als Middleware benutzt, sich um den Inhalt dieser Änderungsmitteilungen kümmert. Wie Änderungen an Datensätzen erkannt werden und wie Konflikte aufgelöst werden, überlässt Footloose den jeweiligen Anwendungen, die Footloose als Middleware nutzen.

Die Frage wann Änderungsmitteilungen gelöscht werden können, ist ein wichtiger Bestandteil von Footloose. Es wird ein Zwei-Phasen-Löschprotokoll ähnlich dem Zwei-Phasen-Commit-Protokoll aus dem Bereich der Datenbanken bzw. des Transaktions-Managements vorgeschlagen. Erst wenn alle beteiligten schlauen Geräte die Löschnachricht für die Änderungsmitteilung erhalten haben und diesen Erhalt allen anderen schlauen Geräten bestätigt haben, wird die Änderungsmitteilung gelöscht.

Footloose ist ein sehr mächtiges und innovatives Projekt, welches allerdings auch viele Fragen offen lässt. Es ist zwar angestrebt, dass die Geräte sich automatisch abgleichen, als einzige Regel wird aber die räumliche Nähe aufgezeigt. Deren Erkennung wird für keine Technologie exemplarisch dargelegt. Ein weiteres Problem sind die Datenmengen (Routing und Änderungsmitteilungen) die im Gesamtsystem gehalten werden. Hier werden keine Zahlen genannt, wie hoch diese Datenmengen in der Beispiel-Implementierung sind. Sehr gut untersucht wurde allerdings die Anzahl der Synchronisationsvorgänge, die bei verschiedenen Topologien nötig sind, um die Datenkonsistenz wieder herzustellen. Aufgrund des Zwei-Phasen-Protokolls sind vergleichsweise viele Abgleiche nötig. Da Footloose Routingtabellen über das gesamte Netzwerk erstellt, sind die Speichermengen auch von der Anzahl der Geräte abhängig. Dies ist allerdings abschätzbar, da Footloose zurzeit nur feste Kopien erlaubt. Weitaus problematischer gestalten sich Ausfälle von einzelnen Knoten im Netzwerk. Wie in einem solchen Fall das Zwei-Phasen-Protokoll durchlaufen oder sonst vorgegangen wird, geht Footloose nicht an und wird auch nicht im Ausblick auf zukünftige Entwicklungen erwähnt.

5.4. OceanStore: Hash History

OceanStore [KWK03] will einen großen verteilten Datenspeicher aufbauen. Viele Server bilden ein Netzwerk und bieten Speicherplatz an. Wenn man Speicherplatz belegt, kann dieser überall verteilt sein. Ein Bezahlsystem garantiert, dass man als Nutzer nur bei einem OceanStore Anbieter zahlen muss, die Daten aber bei verschiednen Anbieter lagern können. Es werden lokale Kopien erstellt, um den Datenzugriff zu beschleunigen und um die Ausfallsicherheit des Systems sicherzustellen. Durch diese Komponenten- und Informationsredundanz können einzelne Server ausfallen, die Daten beleiben aber erhalten.

Im Rahmen dieses Projekts sind viele Publikationen erschienen, die sich aufgrund der Natur des Projekts mit einer Vielzahl an Aspekten auseinander setzen müssen. Ein Aspekt ist die Erstellung von lokalen Kopien und deren Abgleich mit anderen Geräten. In diesem Zusammenhang sind Hash Histories entstanden, die mit der kausalen Historie verwandt sind. Alle Änderungen auf einer Kopie werden mit einer ID gespeichert. Diese ID wird mit Hilfe einer Hash Funktion über den Nutzdaten und zusätzlich mit einem Zeitstempel gebildet.

Erneut ist es schwierig das Verfahren in die vorgestellten Kriterien einzuordnen. Hash History ist zwar ein komplettes Abgleichverfahren, aber die vorliegende Quelle nennt viele Aspekte nicht explizit. Eine Kopie scheint als ein großer Datensatz angesehen zu werden. Die Granularität mit der Änderungen erkannt werden, ist allerdings nur wenige Byteblöcke groß. Da Hash Histories immer nur Unterschiede (Deltas) übermitteln, erzeugen kleine Änderungen nicht gleich große Datenübertragungen zwischen den einzelnen Kopien.

Das wohl wichtigste Kriterium – die unterstütze Netzwerktopologie – wird ebenfalls nicht erwähnt. Es ist aber aufgrund des Projektziels, der angeführten Beispiele und nicht zu letzt durch die Technik der Änderungserkennung zu vermuten, dass Hash Histories Zyklen und damit beliebige Graphen unterstützen.

OceanStore nennt die Möglichkeit Milliarden von Benutzern verwalten zu können – also die Ganze Menschheit. Daher muss die Anzahl der Kopien beliebig sein. Vektorzeiten wären für dieses Projekt ungeeignet, da deren Größe mindestens linear mit der Anzahl der Kopien wächst. Dieser zusätzliche Speicherplatzverbrauch wäre bei mehreren Tausend Kopien nicht tragbar. Eine zentrale Koordination mit einem single-point-of-failure kommt wegen der angestrebten Ausfallsicherheit ebenfalls nicht in Betracht. Daher ist auf die kausale Historie zurückgegriffen worden. Diese wächst nicht mit der Anzahl der Kopien sondern mit der Anzahl der Änderungen. Mit Hilfe der IDs in der kausalen His-

torie kann entschieden werden, welche Änderungen die andere Kopie noch nicht kennt.

Wann Abgleiche stattfinden und wie Konflikte aufgelöst werden, ist nicht Bestandteil von Hash History. Diese Aufgaben übernehmen andere Programmteile. Entsprechend kann in diesen Punkten keine Einordnung in die Kriterien aus kapitel 4 stattfinden, da die Ausführungen zu diesen Punkten in der Beispiel-Implementierung wage bleiben.

Bei kausalen Historien fallen allerdings weiterhin sehr viele Daten an. Da bei vielen Kopien die Historien gleich sind, können spezielle Server im Netzwerk die gleiche Historie für eine Vielzahl an Kopien speichern. Nur lokale Änderungen verbleiben in der jeweiligen Kopie. Die Zusammenfassung der Historien auf bestimmten Servern muss in Hinblick auf die Ausfallsicherheit mit Bedacht geschehen. Darüber hinaus wird vorgeschlagen, Teile der Historie nach einem bestimmten Zeitraum (z.B. 32-128 Tage) zu verwerfen. Diese Werte sind allerdings sehr vom jeweiligen Projekt abhängig.

5.5. Roma

Roma [SKW+02] beschreibt einen komplett anderen Weg, denn es nutzt einen zentralen Metadaten Server, den ein Benutzer immer mit sich führt. Auf den anderen Geräten werden nur die Kopien der Nutzdaten abgespeichert. Der Metadaten Server verwaltet die Versionen der Kopien. Es werden nicht nur Metadaten zum Datenabgleich gespeichert, sondern auch inhaltliche Informationen zum Datensatz. Dies erlaubt es, dass man den Metadaten Server auch zur Suche von Datensätzen nutzen kann, um den aktuellsten Datensatz zu erhalten. Ist dieser gerade nicht verfügbar, weil das Gerät oder das Speichermedium nicht erreichbar ist, wird der Nutzer entsprechend darauf hingewiesen. Der Benutzer kann dann entscheiden, auf der alten Kopie weiter zu arbeiten oder die aktuelle Kopie verfügbar zu machen, indem eine Verbindung zu dem Gerät hergestellt wird.

Roma unterstützt jede Form von Datentypen, arbeitet aber hauptsächlich mit Dateien in einem hierarchischen Dateisystem. Intern werden die Datensätze über eine ID flach angeordnet. Jede ID ist mit mehreren Metadateien verknüpft. Eine Metadatei enthält eine Reihe von Attributen, die wiederum Schlüssel/Wert Paare darstellen. Da jede Kopie verschiedene Attribute besitzen kann und einen anderen Aufenthaltsort besitzt, der durch eine URI [BLFSM05] näher bezeichnet wird, existiert eine Metadatei für jede Kopie. Da jeder Datensatz eine global eindeutige ID besitzt, kann mit Hilfe des Metadaten Servers eine Datensynchronisation mit Zyklenunterstützung stattfinden. Roma sieht es vor, dass es sich bei dem Metadaten Server immer um dasselbe Gerät handelt. In weiteren Untersuchung wollen die Autoren von Roma aber untersuchen, in wie weit ein

verteilter Metadaten Server möglich ist, bei dem lokal ein Cache der Metadaten entsteht.

Bei Roma ist die Kopienanzahl beliebig, allerdings muss jede Kopie Kenntnis vom Roma Protokoll und dem Metadaten Server besitzen. Das Projekt sieht Roma-fähige Anwendungen vor, die von sich aus den Metadaten Server kontaktieren. Dies ermöglicht es, dass der Benutzer mit einer Anwendung arbeitet und diese kümmert sich automatisch darum, die neuste Version einen Datensatzes zu finden und bereitzustellen.

Änderungen werden pro Datensatz erkannt, indem für jede neue Änderung die Versionsnummer des Metadatensatzes im Server aktualisiert wird. Wird eine Kopie gelöscht oder ein neuer Datensatz angelegt, wird der entsprechende Metadatensatz gelöscht bzw. neu angelegt.

Die Architektur von Roma besteht aus einer Reihe von Benutzerapplikationen und ständigen Anwendungen im Hintergrund (*Agenten*). Die Agenten binden nicht Roma fähige Programme in den Datenabgleich mit ein. Die Beschränkung auf nur einen einzigen Benutzer und dessen persönliche Datensätze ist die größte Einschränkung von Roma. Andere Datenquellen wie zum Beispiel der Kontostand können aber durch Hintergrundanwendungen eingebunden werden.

Wann jeweils Abgleiche zwischen den Kopien erfolgen, steht noch zur weiteren Forschung innerhalb von Roma an. Auch wann die Änderungen an Datensätze an den Metadaten Server gemeldet werden sollen, wird noch untersucht. In der Beispiel-Implementierung geschehen die Aktualisierungen an den Metadaten bei jeder Änderung. Obwohl Roma (bzw. die Beispiel–Implementierung) im eigentlichen Sinne kein Abgleichverfahren darstellt, sind solche Funktionen durch einen Agenten angedacht. Ein solcher Agent greift auf den Metadaten Server zu, der alle Informationen liefert, die für einen vollständigen optimistischen Datenabgleich unter den Kopien nötig ist. Weil der eigentliche Agent noch fehlt, gibt [SKW+02] nicht an, ob Änderungen zwischen einzelnen Kopien komplett als Datensatz oder als Deltas übertragen werden. Ebenso fehlen Konzepte für eine dann nötige Konfliktresolution.

Roma ist ein sehr viel versprechendes Projekt. Es speichert alle Metadaten lokal in einem Gerät beim Nutzer. Es geht damit genau den anderen Weg vieler Abgleichverfahren, alle Daten zentral auf einem weit entfernten Server im Internet abzulegen. Benutzer könnten ein besseres Sicherheitsgefühl verspüren, wenn sie wissen, dass die Daten nur lokal vorliegen und nicht im weltweiten Internet. Die große Herausforderung ist die Schaffung dieses persönlichen Metadaten Servers, der über eine Vielzahl an Kommunikationsschnittstellen verfügen müsste. Dass Benutzer immer öfter ein persönliches Gerät mit sich führen, sieht man an der Nutzung von Mobiltelefonen oder MP3 Spielern. Diese Geräte verfügen bereits heute über die entsprechenden Speicherkapazitäten.

5.6. Tra

Die Autoren von Tra [CJ02; CJ04; CJ05] setzen sich mit den Problemen von Vektorzeiten auseinander und versuchen dafür Lösungen zu finden. Zum einen werden in Tra dynamische Vektoren eingesetzt, die nicht Vektoren mit starrer Größe sondern mathematische Mengen darstellen. In einer solchen Menge wird für jede bekannte Kopie ein Zeitstempel festgehalten. Dadurch ist es möglich, neue anfangs noch unbekannte Kopien nach und nach im Netzwerk aufzunehmen. Der Einsatz von Vektorzeiten führt zu der Fragestellung, wann Löschungen einzelner Datensätze allen Kopien bekannt sind, um gespeicherte Löschvermerke verwerfen zu können. Ansonst müsste für jeden gelöschten Datensatz dauerhaft ein Löschvermerk in allen Kopien verbleiben, weil eine vielleicht unbekannte Kopie die Löschaufforderung noch nicht erhalten hat. Das im Abschnitt 5.3 beschriebene Projekt Footloose löst diese Problematik durch ein Zwei-Phasen-Lösch Protokoll, bei dem alle Kopien den Erhalt einer Löschaufforderung bestätigen. Kopien die ausgefallen sind und längere Zeit keine physikalische Verbindung zum Netzwerk aufbauen, verhindern die Vollendung dieses Zwei-Phasen-Protokolls. Andere Projekte verwerfen Löschvermerke nach einer gewissen Zeit, zu der es wahrscheinlich scheint, dass alle anderen Kopien die Aufforderung erhalten und ausgeführt haben.

Tra löst dieses Problem dadurch, dass nicht nur ein Vektor sondern zwei Vektoren – ein *Vektorzeitpaar* – und ein zusätzlicher Zeitstempel pro Datensatz angelegt werden. Um Tra besser einordnen zu können, wird das Verfahren erst anhand der Kriterien untersucht. Danach wird genauer die Technik zur Änderungserkennung von Tra eingegangen, um zu erklären, warum Tra ohne spezielle Löschvermerke auskommt.

5.6.1. Einordnung

Tra ist ein Abgleichverfahren für Dateisysteme, entsprechend kann es hierarchische Datentypen verwalten. Hauptziel von Tra ist ein Abgleich innerhalb aller möglichen Netzwerktopologien mit besonderem Augenmerk auf das Vorhandensein von Zyklen. Die Anzahl der Kopien ist aufgrund der Verwendung von Mengen anstatt starrer Vektoren beliebig. Die Granularität der Änderungserkennung ist auf die Dateiebene beschränkt. Ansonsten müsste Tra Wissen über Dateiformate mitbringen, um Änderungen feiner auflösen zu können. Die Architektur der Beispiel-Implementierung ist eine Benutzerapplikation. Dies reicht aus, da Tra nur die letzte Änderung einer Datei benötigt und diese Informationen aus dem Dateisystem erhalten kann (z.B. die letzte Modifikationszeit). Die Benutzeranzahl ist nicht festgelegt. Es gibt keine Einschränkungen aber auch keine Optimierungen (z.B. Sicherheitsaspekte oder besondere Konfliktresolutionen) für eine

bestimmte Benutzeranzahl. Die Beispiel-Implementierung ist für genau einen Benutzer entworfen worden, der seine Dateien mit einer Reihe von anderen Kopien abgleichen will. Der Benutzer gleicht seine Dateien manuell ab. Für weitergehende automatische Abgleichregeln fehlen Hinweise in den vorliegenden Quellen. Existiert bereits eine Version einer Datei, wird sie nach einem abgewandelten rsync [SNT04; rsy93] Verfahren übertragen (Abbildung 5.1).

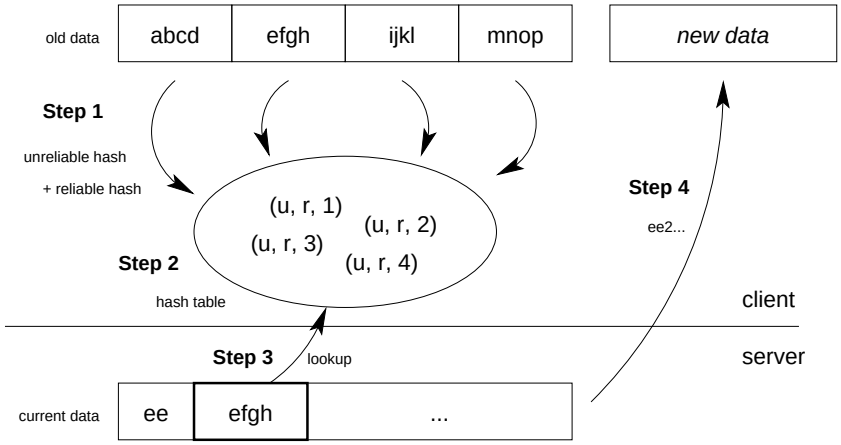

Abbildung 5.1.: rsync – Eine Datei wird in mehrere Blöcke unterteilt und darüber werden zwei Hashwerte gebildet. Diese werden an den Server übermittelt. Der Server geht jeden Block seiner Kopie durch und versucht dies mit *irgendeinem* Hash vom Klienten ausdrücken. Hat dies kein Erfolg wird das erste Zeichen des Blocks zum Klienten übertragen und der zu untersuchende Block wandert nur ein Zeichen weiter. Dadurch können fast nur die Hinzufügungen oder Ersetzungen innerhalb einer Datei übermittelt werden.

rsync selbst ist bereits ein Abgleichverfahren für Dateisysteme, allerdings unterstützt es nur Kopien in einer 1:1 Beziehung. Da Tra die zu ändernden Dateien ermittelt, können beim Übertragen von Dateien die Funktionen von rsync übernommen werden. In Tra werden die Dateien bei einem Konflikt erst auf eine eventuelle Gleichheit geprüft. Besteht der Konflikt weiterhin, wird dies dem Benutzer gemeldet.

5.6.2. Technik zur Änderungserkennung

Jede Kopie führt einen Kopie-globalen Zähler, der vor jedem lokalen Schreibzugriff (Hinzufügungen, Ersetzungen oder Löschungen) erhöht wird. Der Zähler sollte bei Eins be-

ginnen und bei jedem Schreibzugriff um Eins erhöht werden.[1]

Beispiel: *Der Zähler steht zurzeit auf Drei. Kopie B legt eine neue Datei mit dem Zeitstempel Vier an und ändert danach zwei weitere Dateien. Der Zähler steht danach bei Sechs.*[2] *Es findet ein Abgleich mit einer anderen Kopie A statt und A übermittelt zwei neue Dateien. Der lokale Zähler wird nicht erhöht. Erst wenn B wieder eine seiner Dateien ändert, erhöht sich der Zähler auf Sieben.*

Die Technik zur Änderungserkennung erstellt einen globalen Verlauf einer Datei über das Gesamtnetzwerk und basiert auf Vektorzeitpaaren. Der eine Vektor speichert in einem Zeitstempel den Zeitpunkt der letzten Änderung der Datei ab und wird im Folgenden mit $\vec{m}$ (für Englisch: *modification*) abgekürzt. $\vec{m}_A$ besagt, dass es sich um einen Vektor im Knoten A handelt und $\vec{m}_{B/System/}$ besagt, dass dies der $\vec{m}$ des Verzeichnisses *System* auf dem Knoten B ist.

Beispiel: *Kopie B legt eine Datei an. Der Zähler steht bei Eins. Entsprechend erhält der Datensatz* $\vec{m}_B = \{B1\}$. *Die Datei wird mit A abgeglichen und A ändert diese danach. Der Zähler von A steht bei Fünf, da vorher andere Dateien erzeugt worden waren. Daraus ergibt sich* $\vec{m}_A = \{A5\}$. *Der Zeitstempel von* $\{B1\}$ *kann verworfen werden, da nun* $\{A5\}$ *die neuste Modifikation ist.*

Es ist zu klären, wie Tra entscheiden kann, dass die Änderung von A eine neuere Änderung als die von B darstellt. Hierzu wird ein zweiter Vektor benötigt, der angibt, wann eine Datei zum letzten Mal mit welcher Kopie abgeglichen worden ist. Dieser Synchronisationsvektor wird mit $\vec{s}$ abgekürzt.

Beispiel: *Kopie B hat nach einer angelegten Datei noch zwei weitere Dateien geändert und der Zähler steht nun bei Drei. Die Dateien sind mit x, y und z bezeichnet und es ergeben sich folgende* $\vec{m}$*:* $\vec{m}_{B/x} = \{B1\}$, $\vec{m}_{B/y} = \{B2\}$ *und* $\vec{m}_{B/z} = \{B3\}$.[3] *Die* $\vec{s}$ *aller Dateien werden bei jeder Änderung mit angepasst und ergeben bei allen drei Dateien* $\vec{s}_B = \{B3\}$.

[1] Der Zähler kann für jeden Schreibzugriff um mehr als Eins steigen. Der Zähler muss nicht mit einer konstanten Zahl steigen.

[2] Wenn kein Abgleich zwischen den Schreibzugriffen stattgefunden hat, kann der Zähler am Ende auch Fünf sein, denn direkt aufeinander folgende Änderungen werden bereits durch *eine* Zählererhöhung abgebildet.

[3] Zusätzlich wird $\vec{m}_{B/} = \{B3\}$ gesetzt.

Da viele $\vec{s}$ im Dateisystem gleiche Werte besitzen, werden $\vec{s}$ immer als Delta zum übergeordneten Verzeichnis abgespeichert. Das Verzeichnis erhält dabei immer das Minimum aller Zeitstempel aus den $\vec{s}$ der enthaltenen Objekte (Dateien und Verzeichnisse). Ein Null-Zeitstempel besitzt keine weiteren Informationen und kann verworfen werden. Wird auf ein $\vec{s}$ verglichen, werden die $\vec{s}$ der übergeordneten Verzeichnisse wieder aufsummiert.

Beispiel: *Angenommen die Dateien x, y und z besitzen ein übergeordnetes Wurzelverzeichnis (Englisch:* root folder*; abgekürzt mit /), dann ist dessen* $\vec{s}_{B/} = \{B3\}$ *und die* $\vec{s}$ *der Dateien* $\vec{s}_B = \{B0\}$ *bzw. da Null-Zeitstempel verworfen werden führt dies zu* $\vec{s}_B = \{\}$.

Es stellt sich die Frage, wann Dateien im System jemals unterschiedliche $\vec{s}$ besitzen. Wenn alle Dateien den gleichen Vektor haben, dann könnte bereits ein $\vec{s}$ für das gesamte Dateisystem ausreichen. Für diese Erklärung muss ein wenig weiter ausgeholt werden.

Nach der Definition von *Datensynchronisation* aus dem ersten Kapitel wird immer der gesamten Datenbestand abgeglichen. Dies ist aus Sicht eines Benutzers auch sinnvoll, denn er erwartet nach einem Abgleich zweier Kopien, dass sie die gleichen Daten enthalten. Dies ist die Zwei-Wege Synchronisation. Eine Synchronisation in nur eine Richtung erzeugt eine aktuellste Kopie (Ein-Weg). Die Kopie besitzt sowohl die eigenen Änderungen als auch die Änderungen der anderen Kopie. Die andere Kopie besitzt aber nur ihre eigenen Änderungen.

Man kann sich bei beiden Synchronisationsarten vorstellen, dass nicht immer der gesamte Datenbestand abgeglichen wird. Da ein Abgleich eines großen Dateisystems durchaus sehr lange dauern kann, bietet Tra die Möglichkeit nur Teile abzugleichen. Dies ist besonders sinnvoll, wenn bestimmte Dateien z.B. der Inhalt der persönlichen Dokumente eines Benutzers sehr wichtig sind. Andere Dateien z.B. Systemdateien ändern sich dagegen regelmäßig ohne große Bedeutung. Diese Dateien immer mit abzugleichen dauert länger. Partielle Abgleiche ermöglichen es, dass ein ganzes Dateisystem synchron gehalten werden kann, man aber nicht gezwungen ist, immer das gesamte System abzugleichen.

Durch partielle Abgleiche entstehen in einer Kopie unterschiedliche $\vec{s}$ und es ist nötig, dass jede Datei einen eigenen $\vec{s}$ führt. Der Vektor zeigt an, welcher Änderungszustand dieser Datei bekannt ist. Ist ein andere Datei bei letzten Mal nicht mit abgeglichen worden, hat diese einen niedrigeren $\vec{s}$, da nicht unbedingt der letzte Zustand bekannt ist.

Ein zusätzlicher einzelner Zeitstempel für jede Datei gibt an, wann der Datensatz angelegt worden ist und wird im Folgenden c (für Englisch: *creation*) genannt. Der

Zeitstempel besteht aus der Kopie, die den Datensatz ursprünglich angelegt hat und dem Wert des lokalen Ereigniszählers auf dieser Kopie. Dieser Zeitstempel wird zur Unterscheidung von Hinzufügungen und Löschungen benötigt.

Tra nutzt folglich einen Modifikationsvektor $\vec{m}$, einen Synchronisationsvektor $\vec{s}$, einen Anlegezeitstempel c und einen Namen für jede Datei. Wie können daraus Ersetzungen, Hinzufügungen oder Löschungen erkannt werden?

5.6.2.1. Abgleich von Verzeichnissen

Angenommen es wird ein vollständiger Abgleich in eine Richtung von Kopie A nach B vorgenommen. In diesem Fall sollen alle Ersetzungen von A nach B gemeldet werden, darüber hinaus soll B erkennen können, welche Dateien A neu hinzugefügt und gelöscht hat. Dazu wird im Dateiverzeichnis bei der Wurzel begonnen:

```
if (m_A/ ≤ s_B/)
   do nothing
else
   for each child in directory
      sync(child)
```

In diesem Pseudocode Fragment wird der Modifikationsvektor von A mit dem Synchronisationsvektor von B (jeweils des Wurzelverzeichnisses) verglichen. B teilt A mit, wann beide Knoten zum letzten Mal abgeglichen worden sind, also bis wann B Kenntnis von Änderungen hat. Der Knoten A vergleicht dies mit den Änderungen die er lokal besitzt. Hierbei wird genau wie bei Vergleichen von normalen Vektorzeiten Elementweise vorgegangen, allerdings sind die Vergleichsoperatoren (neuer, älter und inkompatibel) anders definiert.[4]

Beispiel: $\vec{m}_A = \{A1, B6\}$ *und* $\vec{s}_B = \{A1, B6\}$
In diesem Fall kennt B alle Knoten die A kennt und A hat keine neueren Daten. Es müssen keinen weiteren Daten übertragen werden.

Beispiel: $\vec{m}_A = \{A1, B6\}$ *und* $\vec{s}_B = \{A1, B7\}$
B hat lokal neuere Daten als A. Da sich B die neusten Daten von A holen will und nicht umgekehrt, ist der Abgleich bereits jetzt beendet und es müssen keine weiteren Daten übertragen werden.

[4] Innerhalb von Vektorzeitenpaaren reicht die Beziehungen $\vec{m} \leq \vec{s}$ und $c \leq \vec{s}$ für alle Entscheidungen aus, weswegen andere Beziehungen nicht mathematisch aufgelöst werden.

Beispiel: $\vec{m}_A = \{A1, B6\}$ *und* $\vec{s}_B = \{A1, B6, C4\}$
In diesem Fall kennt B einen Knoten den A nicht kennt. Die beiden Vektoren wären bei einem Vergleich von normalen Vektorzeiten inkompatibel. Beim Vergleich von Vektorzeitpaaren ist dies nicht der Fall. Der Abgleich ist beendet, da B bereits alle und sogar noch mehr Daten kennt, als ihm A anbieten kann.

Beispiel: $\vec{m}_A = \{A1, B6, C4\}$ *und* $\vec{s}_B = \{A1, B6\}$
Hier deutet A an, dass es eine Änderung von C erhalten hat. B kennt diesen Knoten nicht. Der Abgleich muss für jedes Kind im Wurzelverzeichnis weiter laufen, um zu erkennen, welche Datei(en) der Knoten C geändert hatte.

Beispiel: $\vec{m}_A = \{A2, B6\}$ *und* $\vec{s}_B = \{A1, B6\}$
In diesem Fall kennt B alle Knoten die A kennt, aber A besitzt neuere Daten. Der Abgleich muss für jedes Kind im Wurzelverzeichnis weiter laufen, um zu erkennen, welche Datei(en) der Knoten A geändert hat.

Daraus ergibt sich:

$$\vec{m} \leq \vec{s} \Leftrightarrow \forall i \in \vec{m}, \exists j \in \vec{s} \text{ mit } \vec{m}_{i_1} = \vec{s}_{j_1} \text{ für die gilt } \vec{m}_{i_2} \leq \vec{s}_{j_2}.$$

$\vec{m}$ und $\vec{s}$ sind rechtseindeutige (binäre) Relationen als Teilmengen über dem kartesische Produkt $GUID \times \mathbf{N}$. GUID (Englisch: *Global Unique Identifier*) steht für eine Menge an global eindeutige Namen für jeden Knoten. Der Wertebereich dieser Menge ergibt je nach Wahl. Hier in den Beispielen sind es die Buchstaben A, B, C und D. Die Rechtseindeutigkeit besagt, dass keiner GUID mehr als eine natürliche Zahl zugewiesen ist. Dies stellt sicher das in $\vec{m}$ oder $\vec{s}$ kein Knoten mehrfach auftritt.[5] $\vec{m}$ kann daher auch als $\{(A, 2), (B, 6)\}$ geschrieben werden. Bei $i = 2$ wäre $\vec{m}_{2_1} = B$ und $\vec{m}_{2_2} = 6$.

Hat diese Entscheidung ergeben, dass A neue Änderungen vorliegen hat, müssen alle Kinder der Wurzel durchgegangen werden. Handelt es sich bei dem Kind um ein Verzeichnis, wird die gleiche Entscheidung wie bei der Wurzel ausgeführt. Dabei wird der Synchronisationsvektor mit den übergeordneten Verzeichnissen[6] aufsummiert und ver-

[5] Als anschauliches Beispiel sei auf die Programmiersprache Java verwiesen, in der sich die Klasse `java.util.Hashtable` für Modifikations- und Synchronisationsvektoren anbietet. Alle Schlüssel in $\vec{m}$ werden durchlaufen und der entsprechende Schlüssel wird in $\vec{s}$ gesucht. Enthält $\vec{s}$ den Schlüssel nicht, wird abgebrochen und die Bedingung $\vec{m} \leq \vec{s}$ wird verneint. Enthält $\vec{s}$ den Schlüssel werden die beide Wertepaare verglichen. Ist der Wert von $\vec{m}$ größer wird die Bedingung verneint, andernfalls wird der Vergleich für die nächsten Schlüssel von $\vec{m}$ weiter geführt.

[6] Bei der ersten Ebene ist dies nur die Wurzel.

glichen. Dies bedeutet es wird nicht nur der Delta-Vektor verglichen.

Beispiel: *Auf der Ebene der Wurzel sehen die Vektoren wie folgt aus:* $\vec{m}_{A/} = \{A2, B6\}$ *und* $\vec{s}_{B/} = \{A1, B6\}$. *Auf der ersten Ebene unter der Wurzel stehen sich* $\vec{m}_{A/test/} = \{A1, B6\}$ *und* $\vec{s}_{B/test/} = \{\}$ *gegenüber. B hatte beim letzten Abgleich mit A einen vollständigen Abgleich durchlaufen. A hat keine Änderung in dem Unterverzeichnis mit dem Namen* test/. *Es wird die Bedingung* $\{A1, B6\} \leq \{A1, B6\}$ *überprüft. Diese Bedingung trifft in diesem Fall zu, d.h. der Inhalt von* test/ *muss nicht weiter betrachtet werden. Das nächste Kind der Wurzel wird untersucht:* $\vec{m}_{A/System/} = \{A2, B6\}$ *und* $\vec{s}_{B/System/} = \{\}$. *Bei diesem Verzeichnis hat A eine Änderung, die B noch nicht kennt und die Bedingung trifft nicht zu.*

Tra verfährt nach dem Depth-First Muster, d.h. wird ein Verzeichnis gefunden und dessen Inhalt muss abgeglichen werden, wird erst das Verzeichnis mit seinen Kindern durchsucht (und womöglich dessen Unterverzeichnisse) bis das untersuchte Verzeichnis nur noch Dateien aber keine weiteren Verzeichnisse enthält. Dies erlaubt es, Arbeitspeicher zu sparen, denn ansonsten müssten für jede Verzeichnisebene die Metadaten aller enthaltenen Verzeichnisse für etwaige Aufsummierungen in einem Puffer gehalten werden.

Beispiel: *Die Wurzel enthält zwei weitere Verzeichnisse* System/ *und* test/. $\vec{s}_{/System/}$ *und* $\vec{s}_{/test/}$ *müssen zwischengespeichert (oder erneut angefragt) werden, wenn der Inhalt von* System/ *und danach der Inhalt von* test/ *durchlaufen wird. Wird aber erst* System/ *und dessen Kinder untersucht, anstatt das aktuelle Verzeichnis zu durchlaufen, dann muss der Vektor von* test/ *nicht die ganze Zeit im Speicher gehalten werden.*

5.6.2.2. Abgleich von Dateien

Bei Dateien muss entschieden werden, ob die Datei ersetzt, neu hinzugefügt oder gelöscht worden ist.[7] Existiert eine Datei nicht, wird für Vergleiche der $\vec{s}$ des übergeordneten Verzeichnisses herangezogen.

Hinzufügungen Angenommen die Datei existiert nicht in Kopie B, dann muss entschieden werden, ob B die Datei gelöscht hatte oder ob B die Datei noch nicht kennt. Hierzu wird folgende Abfrage verwendet:

`if` $(\vec{m}_A \leq \vec{s}_B)$

[7] Gleiches geschieht für neue oder gelöschte Verzeichnisse.

```
   do nothing
else if (c_A ⋠ s⃗_B)
   copy File_A to B
else
   report a conflict
```

Im ersten Fall hatte B die Datei gelöscht, da B laut dem übergeordneten Verzeichnis einen neueren Stand besitzt und alle Änderungen von A kennt. Da Änderungen aber nur von A nach B wandern, wird der Datenbestand von A erst einmal nicht geändert. Im zweiten Fall ist die Datei seit dem letzten Abgleich von A hinzugefügt worden. B kennt die Datei noch nicht und sie muss zu B übertragen werden. Im letzten Fall ist ein Schreib/Schreib Konflikt aufgetreten, denn A hat die Datei ersetzt und B hatte die Datei gelöscht. Das Abgleichverfahren kann nicht mehr entscheiden, was zu tun ist. Ein in Tra nicht näher erwähntes Verfahren zur Konfliktresolution muss diesen Fall lösen.

Löschungen Angenommen die Datei existiert nicht in Kopie A, dann muss entschieden werden, ob A die Datei gelöscht hatte. In diesem Fall wird eine ähnliche Abfrage wie oben herangezogen, nur die Knoten werden getauscht.

```
if (m⃗_B ≤ s⃗_A)
   delete File_B
else if (c_B ⋠ s⃗_A)
   do nothing
else
   report a conflict
```

A hat im ersten Fall den neusten Stand und da die Datei nicht mehr existiert, muss eine Löschung auf A statt gefunden haben. Im zweiten Fall wird überprüft, ob B die Datei neu hinzugefügt hat. Dies vermeidet Konfliktmeldungen, denn wenn A die Datei noch gar nicht kennt, liegt kein Schreib/Schreib Konflikt vor. Im letzten Fall hatte A die Datei gelöscht und B hatte die Datei ersetzt. Ein Konflikt wird gemeldet.

Hatten beide Kopien die Datei seit dem letzten Abgleich gelöscht, dann wird dies nicht mehr vom Abgleichverfahren erkannt, da beide Kopien keinen Verweis auf die Datei vorweisen können. Dies ist korrekt, denn zwei Löschungen auf der gleichen Datei ergeben einen Schreib/Schreib Konflikt, aber dessen Auflösung wäre immer auch eine Löschung. Deswegen muss dieser Konflikt nicht gemeldet oder erkannt werden.

Eine Datei wird gelöscht, indem c, $\vec{m}$ und die eigentlichen Daten gelöscht werden. Der $\vec{s}$ bleibt vorläufig erhalten und die Datei wird zu einem Löschvermerk. Wenn das gesamte Verzeichnis abgeglichen worden ist, sind alle $\vec{s}$ im Verzeichnis leer. Der Löschvermerk enthält keine Daten mehr und kann endgültig entfernt werden. Der $\vec{s}$ des Verzeichnisses

wird dann bei zukünftigen Entscheidungen herangezogen. Wenn ein partieller Abgleich eines Verzeichnisses stattgefunden hat, dann ist der $\vec{s}$ des Löschvermerks nicht leer und er verbleibt bis zum nächsten vollständigen Abgleich bestehen. Die beiden obigen Entscheidungen werden daher nicht nur bei nicht existierenden Dateien durchlaufen sondern auch dann, falls einer der Dateien ein Löschvermerk darstellt.

Ersetzungen Existiert die Datei sowohl auf A als auch auf B wird wie folgt die neuste Version ermittelt:

```
if (m_A ≤ s_B)
   do nothing
else if (m_B ≤ s_A)
   copy File_A to B
else
   report a conflict
```

Es wird erst überprüft, ob B bereits alle Änderungen von A kennt. Ist dies der Fall, müssen keine Daten an B übermittelt werden. Der zweite Fall überprüft, ob B seit dem letzten Abgleich ebenfalls Änderungen an der Datei vorgenommen hatte. Im letzten Fall wäre die Modifikationszeit von B größer oder inkompatibel zur Synchronisationszeit von A. In diesem Fall kennt A nicht alle Änderungen, die B besitzt, und es wird ein Schreib/ Schreib Konflikt gemeldet, da sowohl A als auch B zwei voneinander unabhängige Ersetzungen seit dem letzten Abgleich durchgeführt hatten.

5.6.2.3. Anpassungen der Zeitstempel

Um die Metainformationen einer Datei immer konsistent für den nächsten Abgleich zu erhalten, müssen zu gewissen Zeitpunkten und unter gewissen Regeln die Zeitstempel der Vektorzeiten an die neuen Zustände angepasst werden.

Anlegezeitstempel Der Zeitstempel der Erzeugung wandert immer mit seiner Datei und wird nur beim Entfernen der Datei gelöscht bzw. bei der Erzeugung der Datei auf den lokalen Change Counter und der lokalen Knoten GUID gesetzt.

Modifikationsvektor Wird eine Datei kopiert, dann wird $\vec{m}_A$ für $\vec{m}_B$ übernommen, da dies den Versionsstand angibt. Das Gleiche passiert im Konfliktfall, falls sich der Benutzer für die Version von A entschieden hat. Hat sich der Benutzer für die Version von B entschieden, wird keine Datei kopiert und $\vec{m}_B$ stellt immer noch die neuste Version dar. Hat der Konfliktfall ergeben, dass sowohl Teile von A als auch Teile von B übernommen

werden (eine *Zusammenführung*), dann muss $\vec{m}_B$ auf den aktuellen Change Counter von B gesetzt werden, denn B hat zu diesem Zeitpunkt die neuste Version. A wird davon nicht informiert, da nur Änderung von A nach B abgeglichen werden und nicht umgekehrt.

Wird $\vec{m}_B$ geändert, wird dies den übergeordneten Verzeichnissen mitgeteilt, die das pro Element das Maximum aus dem neuen Wert und dem eigenen $\vec{m}$ bilden.

Beispiel:

Vor dem Abgleich:	*Nach dem Abgleich:*
$\vec{m}_{A/System/webserver.log} = \{A2\}$	$\vec{m}_{A/System/webserver.log} = \{A2\}$
$\vec{m}_{B/System/webserver.log} = \{B6\}$	$\vec{m}_{B/System/webserver.log} = \{A2\}$
$\vec{m}_{B/System/} = \{A1, B6\}$	$\vec{m}_{B/System/} = \{A2, B6\}$
$\vec{m}_{B/} = \{A1, B6\}$	$\vec{m}_{B/} = \{A2, B6\}$

B hat den Zeitstempel von A übernommen und teilt dies dem übergeordneten Verzeichnis mit, welches das Maximum aus A1 und A2 bildet.

Die übergeordneten Verzeichnisse zu informieren ist nötig, damit es bei zukünftigen Abgleichen ausreicht, den Zeitstempel der Wurzel zu betrachten, um entscheiden zu können, ob der Knoten auf Änderungen durchsucht werden muss oder nicht.

Die Aktualisierung der $\vec{m}_B$ der übergeordneten Verzeichnisse kann jederzeit geschehen[8], da während des aktuellen Abgleichs die $\vec{m}_B$ von Verzeichnissen nicht betrachtet werden. Die Aktualisierung kann während dem Abgleich erfolgen, z.B. wenn eine Datei geändert worden ist. Diese Aktualisierung kann aber auch erst nach dem Abgleich erfolgen.

Ähnliches geschieht mit dem $\vec{m}_B$ eines Verzeichnisses. Nachdem entschieden worden ist, dass das Verzeichnis durchsucht werden muss, wird das Maximum vom $\vec{m}_A$ und $\vec{m}_B$ gebildet. Dies ist nötig, damit keine Löschungen verloren gehen.

Beispiel: $\vec{m}_{A/User/} = \{A2, B6\}$
Angenommen im Verzeichnis User/ *befinde sich keine Datei mit einem* $\vec{m}_A = \{A2\}$*, dann war die zweite Änderung im Knoten A eine Löschung. Bei einer solchen Löschung werden alle Daten sofort entfernt. Dem übergeordneten Verzeichnis* User/ *wird die Löschung mitgeteilt und dieses setzt dann den aktuellen Change Counter des lokalen Knoten ein. D.h. vor der Löschung war* $\vec{m}_{A/User/} = \{A1, B6\}$*. Beim nächsten Abgleich erkennt B, dass sich das Verzeichnis geändert hat, da* $\vec{s}_{B/User/} = \{A1, B6\}$ *gilt: B kennt nicht alle Änderungen von A. Würde bei einer Löschung* nicht *das Maximum*

[8] Die Bedingung ist aber, dass es vor dem nächsten Abgleich geschieht.

über $\vec{m}_{A/User/}$ und $\vec{m}_{B/User/}$ gebildet, dann erhielte ein dritter Knoten C, der mit B abgleicht, nicht die Löschung, die A durchgeführt hat. Die Datei verbliebe in C bestehen. Wird dann ein Abgleich von C nach A durchgeführt, müsste A vermuten, eine neue Datei wäre an dieser Stelle hinzugefügt worden. Die Datei tauchte wieder bei A auf.

Synchronisationsvektor Um die Regeln für die Aktualisierungen von $\vec{s}$ nicht zu kompliziert zu gestalten, wird für jede untersuchte Datei in A, das Maximum vom $\vec{s}_A$ und $\vec{s}_B$ in B gebildet. Existierte die Datei nicht in A wird der $\vec{s}$ des übergeordneten Verzeichnisses herangezogen. Dabei werden nicht die gespeicherten Deltas verglichen, sondern die tatsächlichen Aufsummierungen.

Beispiel:

Vor dem Abgleich	*Nach dem Abgleich*[9]
$\vec{s}_{A/} = \{A2, B6\}$	$\vec{s}_{A/} = \{A2, B6\}$
$\vec{s}_{A/System/} = \{\}$	$\vec{s}_{A/System/} = \{\}$
$\vec{s}_{A/System/webserver.log} = \{\}$	$\vec{s}_{A/System/webserver.log} = \{\}$
$\vec{s}_{B/System/webserver.log} = \{\}$	$\vec{s}_{B/System/webserver.log} = \{A1\}$
$\vec{s}_{B/System/} = \{\}$	$\vec{s}_{B/System/} = \{\}$
$\vec{s}_{B/} = \{A1, B6\}$	$\vec{s}_{B/} = \{A1, B6\}$

Dieses Beispiel zeigt die gleiche Situation wie das vorhergehende Beispiel. Um Speicherplatz zu sparen, werden die $\vec{s}$ als Deltas abgespeichert. Die tatsächlichen Aufsummierungen sehen vor dem Abgleich wie folgt aus:

Kopie A	*Kopie B*
$\vec{s}_{A/} = \{A2, B6\}$	$\vec{s}_{B/} = \{A1, B6\}$
$\vec{s}_{A/System/} = \{A2, B6\}$	$\vec{s}_{B/System/} = \{A1, B6\}$
$\vec{s}_{A/System/webserver.log} = \{A2, B6\}$	$\vec{s}_{B/System/webserver.log} = \{A1, B6\}$

Es wird das Maximum über den $\vec{s}$ von A und B der Datei mit dem Namen webserver.log *aus dem Verzeichnis* System/ *gebildet. Nach dem Abgleich der Datei ergibt sich $\vec{s}_{B/System/webserver.log} = \{A2, B6\}$. Abgespeichert wird nur das Delta $\{A1\}$ zum übergeordneten Verzeichnis, denn $\{A1\} + \{\} + \{A1, B6\} = \{A2, B6\}$.*

Nachdem der Abgleich eines Verzeichnisses abgeschlossen ist[10], wird über den $\vec{s}$ der Kinder das Minimum aller $\vec{s}$ ermittelt. Danach wird das Maximum über diesem neu-

[9] Die Metadaten der Verzeichnisse sind noch nicht aktualisiert.
[10] Die Bedingung ist aber, dass es vor dem nächsten Abgleich geschieht.

en Vektor und dem untersuchten Verzeichnis gebildet. Die Deltas der Kinder werden entsprechend gesenkt.

Beispiel: *Nach dem Abgleich der Datei und der Verzeichnisse sind die gespeicherten* $\vec{s}$ *Deltas:*
$\vec{s}_{B/System/webserver.log} = \{A1\}$
$\vec{s}_{B/System/} = \{\}$
$\vec{s}_{B/} = \{A1, B6\}$

In diesem Beispiel wird angenommen, dass keine weiteren Dateien oder Verzeichnisse existieren. Nach der Aufsummierung der $\vec{s}$ *entspricht das Minimum innerhalb von* System/ *dem vom* $\vec{s}_{B/System/webserver.log}$ *mit dem Wert* $\{A2, B6\}$*. Das Maximum über diesem Wert und* System/ *mit dem Wert* $\{A1, B6\}$ *(nach der Aufsummierung) ergibt* $\{A2, B6\}$*. Da es nur ein Kind im Wurzelverzeichnis gibt, wird der (neue) Wert von* System/ *als Minimum genommen:* $\{A2, B6\}$*. Über diesem und dem* $\vec{s}$ *der Wurzel wird das Maximum gebildet. Als Ergebnis für die Wurzel entsteht erneut* $\{A2, B6\}$*.*

Da alle $\vec{s}$ *im Knoten den gleichen Wert besitzen, besitzt nach dem Senken der Delta Vektoren nur noch die Wurzel eine nicht leere Menge:*
$\vec{s}_{B/System/webserver.log} = \{\}$
$\vec{s}_{B/System/} = \{\}$
$\vec{s}_{B/} = \{A2, B6\}$
Bei einem kompletten Abgleich aller Daten, enthält der $\vec{s}$ *der Wurzel alle relevanten Daten und alle anderen* $\vec{s}$ *im Knoten können leer abgespeichert werden.*

Das Minimieren von $\vec{s}$ erfolgt nicht, wenn das Verzeichnis nur partiell abgeglichen worden ist. Dadurch gehen keine Änderungen verloren, da $\vec{s}$ nur den Kenntnisstand eines Knotens über eine Datei angibt. Der Knoten teilt anderen Knoten mit, dass er nicht alle Änderungen eines Verzeichnisses übermittelt oder kennt. Dies kann der sendende Knoten dadurch entscheiden, dass innerhalb des Verzeichnisses größere Modifikationszeiten als Synchronisationszeiten für einen Knoten existieren. Der sendende Knoten hat Änderungen, kennt aber nicht unbedingt alle. Der empfangende Knoten ändert dann den $\vec{s}$ des betroffenen Verzeichnisses nicht.[11]

[11] [CJ05, Abbildung 12] besagt, dass der $\vec{s}$ eines Verzeichnisses in jedem Fall auf das Elementweise Minimum aller enthalten $\vec{s}$ gesetzt wird. Dies macht allerdings dann keinen Sinn, wenn der $\vec{s}$ bereits einen Wert von A kennt z.B. $\{A4\}$. Wird eine neue Datei z.B. $\vec{m}_{A/Ordner/Datei.txt} = \vec{s}_{A/Ordner/Datei.txt} = \{A5\}$ nicht übertragen, so müsste der $\vec{s}_{B/Ordner/}$ wieder zur leeren Menge werden, da B eine Datei nicht kennt. Diese Unkenntnis führt bei der Bildung des Minimums zur leeren Menge. Dies bedeute,

5.6.3. Bewertung

Tra betrachtet sowohl das Speicherplatzproblem als auch das Zeitproblem von Vektorzeitpaaren.

5.6.3.1. Äußerungen zum Zeitproblem

Zum Zeitproblem schreiben [CJ02; CJ04], dass nicht für jede Datei die Vektorzeitpaare verglichen (und übertragen) werden, sondern nur im gleichen Verhältnis zu geänderten Dateien.

Dieser Argumentation kann nur bedingt gefolgt werden. Um Löschungen oder neue Dateien zu erkennen, müssen alle in einem geänderten Verzeichnis enthaltenen Metadaten übertragen werden. Metadaten sind in Tra der Dateiname, c, $\vec{m}$ und $\vec{s}$. Fehlt eine Datei in einer Kopie und existiert die Datei in einer anderen Kopie, dann handelt es sich um eine neue oder gelöschte Datei in der anderen Kopie.

```
if (m⃗_{A/} ≤ s⃗_{B/})
   do nothing
else
   for each child in directory
      if (m⃗_{A/child} ≤ s⃗_{B/})
         sync(child)
```

Dieser Algorithmus ist von einer alten Tra Version vorgeschlagen worden [CJ04], besitzt aber mehrere Fehler, die der Autor von Tra bestätigt hat. Der abgebildete Algorithmus für Verzeichnisse kann so nicht funktionieren, da A alle Kinder durchgeht, ein Löschvermerk in A aber kein Kind mehr darstellt. A hatte die Datei bereits gelöscht und weiß nichts mehr von ihr, d.h. es ist nicht möglich, dass A seine Kinder durchgeht ohne alle Kinder von B zu kennen. Kopie B könnte diesen Algorithmus auch nicht durchführen, da es dann nicht zwischen nicht geänderten und gelöschten Dateien unterscheiden kann. Daher findet der entsprechende und hier vorgestellte Algorithmus aus der ersten Tra Veröffentlichung [CJ02] wieder in der aktuellen Veröffentlichung [CJ05] Anwendung:

```
if (m⃗_{A/} ≤ s⃗_{B/})
   do nothing
else
   for each child in directory
      sync(child)
```

Dieser Algorithmus kann auf Verzeichnisebene nur entscheiden, dass Dateien geändert worden sind, aber nicht welche der Kinder dies betrifft. Daher müssen die Metadaten

B kenne den Knoten A gar nicht (mehr). Es ist daher sinnvoller den $\vec{s}_{B/Ordner/}$ bei einem partiellen Abgleich nicht zu aktualisieren statt zu minimieren, da B nach einem Abgleich nicht weniger über A weiß als vorher.

aller Kinder herangezogen werden. Wenn ein Verzeichnis sehr viele Dateien aber nur eine Änderung enthält bzw. wenn auf dem Pfad zu dieser Datei Verzeichnisse mit weiteren Kindern vorhanden sind, dann steigt die Anzahl der übertragenen Metadaten nicht im Verhältnis zu den Änderungen, sondern mit der Anzahl der Objekte auf der gleichen und den vorhergehenden Ebenen. Dies macht sich bemerkbar, wenn nur wenige Verzeichnisse aber viele Dateien vorhanden sind. Zum Beispiel werden bei Mobiltelefonen häufig alle Kontakte flach in einem Verzeichnis abgelegt. Bei nur einer Änderung müssen in Tra bereits Metadaten von vielleicht tausend Dateien übertragen werden.

5.6.3.2. Analyse zum Speicherplatzproblem

Es ist allerdings zu betonen, dass Tra nur Metadaten und nur tatsächlich geänderte Daten übertragen muss. Verzeichnisse die keine Änderungen enthalten, müssen keine Metadaten übermitteln, da der Algorithmus erkennen kann, ob eine Änderung (Hinzufügung, Ersetzung oder Löschung) im Verzeichnis vorliegt. Die Metadaten selbst sind sehr gering. [CJ05] zeigt dies ausführlich auf. Im Normalfall erfolgen vollständige Abgleiche, bei denen nur ein Synchronisationsvektor im Knoten übrig bleibt. Dies bedeutet, dass auch nur ein $\vec{s}$ übertragen werden muss. Die Anzahl der zu übertragenen $\vec{s}$ ist daher konstant, egal wie viele und wo die Änderungen vorliegen. Jede Datei hat einen Modifikationsvektor welcher die letzte Änderung speichert und maximal einen Zeitstempel enthält. Darüber hinaus hat jede Datei einen Anlegezeitstempel. Dieser Zeitstempel ist genau so groß wie ein Modifikationsvektor. Zusätzlich hat jede Datei einen Dateinamen. Folglich ist die Menge der zu übertragenen Metadaten pro Datei in der Regel gering (zwei Zeitstempel und den Dateinamen). Verzeichnisse besitzen das Elementweise Maximum aller Modifikationsvektoren der Kinder. Entsprechend besitzen diese Vektoren meist mehr als nur einen Zeitstempel.

5.6.4. Fazit

Der große Vorteil von Vektorzeitpaaren liegt nicht nur in der sehr guten Konflikterkennung sondern auch darin, dass gelöschte Dateien sofort keinen Speicherplatz in Form von Metadaten verbrauchen, wie dies gewöhnliche Abgleichverfahren basierend auf Vektorzeiten tun müssten. Tra löst das Speicherplatzproblem von Vektorzeiten sehr effizient. Jede Datei hat nach einer vollständigen Synchronisation zwei Zeitstempel und pro Knoten gibt es nur einen (Synchronisations-) Vektor, der mit der Anzahl der Knoten im Gesamtsystem wächst.

6. Industriestandards

6.1. IrMC

Die Infrared Data Association (IrDA) [IR94] ist ein Zusammenschluss mehrerer Hersteller um eine kabellose Datenübertragung über Kurzstrecken zu ermöglichen. Dabei werden Infrarotlicht Sender und Empfänger genutzt, die es erlauben mobile Endgeräte untereinander aber auch mit Personal Computer zu verbinden, ohne Kabel und Steckerkombinationen nutzen zu müssen. Die Reichweite beschränkt sich auf wenige Meter und im Idealfall sollten die Geräte nicht weiter als wenige Zentimeter voneinander entfernt sein. Die IrDA definiert ähnlich dem ISO/OSI Referenzmodell [Tan03] die Schnittstellen von der physikalischen Schicht (Schicht 1) über die Transportschicht bis hin zurSchnittstelle für Anwendungen (Schicht 7). Da IrDA für ein breites Spektrum an mobilen Geräten entwickelt worden ist, ist es wichtig, dass die Geräte die Funktionalitäten der jeweiligen anderen Geräte schnell erkennen können. Ein Drucker nach IrDA sollte sich mit einem Personal Computer verstehen. Der Personal Computer sollte erkennen, dass Dateien auf dem Drucker ausgedruckt werden können. Ein Mobiltelefon soll für den mobilen Internetzugang nutzbar sein. Der Speicher einer Digitalkamera sollte komfortabel mit einer Vorschau versehen durchsucht werden können.

Ausgehend von diesen Anforderungen hat die IrDA eine Vielzahl an Anwendungsschnittstellen für Dienste definiert und bietet die Möglichkeit, dass Hersteller standardkonform eigene Dienste bereitstellen. Diese Dienste werden über eine einheitliche Schnittstelle aufgefunden.

Für die mobile Datensynchronisation entstanden dabei mehrere Dienste. Eine Dienstleistung beschäftigt sich mit dem Object Exchange (OBEX) [MSK03], welches ermöglicht einzelne Dateien zwischen IrDA Geräten auszutauschen. OBEX ist dem Hypertext Protocol (HTTP) [FGM+99] nachempfunden. Es werden fast alle HTTP Funktionen nachgebildet, OBEX benutzt aber nicht wie HTTP klar lesbare Befehle und Kopfinformationen, sondern kodiert diese weitestgehend Binär.

Beispiel: *Minimaler HTTP 1.1 Befehl:*
`GET / HTTP/1.1`
`host: medien.informatik.uni-ulm.de`
Der entsprechende OBEX Befehl unter der Annahne, dass eine Verbindung zu dem OBEX Server medien.informatik.uni-ulm.de besteht:
`0x83 0x00 0x03`

Die obigen Hexadezimal-Zahlen werden jeweils als ein Byte übertragen. Es werden daher genau drei Byte innerhalb von OBEX gesendet, wobei der entsprechende HTTP Befehl für jedes Zeichen ein Byte im obigen Fall mehr als 50 Byte benötigt. Da OBEX für mobile Geräte mit begrenzten Speichern entworfen worden ist, sieht es sehr viele Längenindikatoren vor, so dass ein OBEX Parser wenig rechnen muss und mit nur einer geringen Menge an Arbeitspeicher auskommen kann.

OBEX kann als binäres HTTP bezeichnet werden. Es erlaubt Daten von einem anderen Gerät (GET) zu erhalten und Daten an andere Geräte zu senden (PUT). Diese zwei Befehle sind die Basis für viele weitere Dienstleistungen. Eine davon ist die Möglichkeit Dateiverzeichnisse und die darin enthaltenen Metainformationen auszutauschen. So besteht die Möglichkeit als sei es ein Dateisystem durch den Speicher eines mobilen Gerätes zu navigieren.

Hinter dem Dienst mit dem Namen Ir Mobile Communications(IrMC) [LSS00] in Version 1.1 aus dem Jahr 1999 versteckt sich ein Abgleichprotokoll für den mobilen Datenabgleich. Es basiert ebenfalls auf OBEX und nutzt dessen Möglichkeiten Zusatzinformationen wie Dateipfade, Dateinamen und eigene Informationsfelder zu definieren. IrMC gibt an, wie Dateien aufzufinden sind, d.h. es werden Pfadnamen[1] vorgegeben, durch die das komplette Adressbuch, Kalenderdaten, Notizzettel, Aufgabenlisten und E-Mail Verzeichnisse zugreifbar sind.

Beispiel: *Alle Pfade beginnen mit* telecom/*, so befindet sich das Adressbuch unter* telecom/pb.vcf *und der Kalender unter* telecom/cal.vcs.

6.1.1. Einordnung

IrMC besitzt mehrere Funktionsstufen (Level), die an die Möglichkeiten mobiler Endgeräten angepasst sind. IrMC Level 4 bietet ein vollständiges Synchronisationsprotokoll, welches 1:n Abgleiche ermöglicht. Bei allen Daten handelt es sich um Tupeldaten, d.h.

[1] Es wird nicht der Pfad vorgegeben, wo die Dateien liegen müssen, sondern nur der Name, d.h. die Dateien können in einem anderen Pfad liegen, müssen aber über den relativen IrMC Pfadnamen ansprechbar sein.

es werden die vCard 2.1 [ver96b] und vCalendar 1.0 [ver96a] Datenformate genutzt. Für Notizen, E-Mail Nachrichten und Internet Lesezeichen sind eigene Datenformate entwickelt worden, die sich an die Struktur und Konstrukte von vCard und vCalendar Daten anlehnen.

```
BEGIN:VCARD
VERSION:2.1
   N;CHARSET=ISO-8859-1:Müller;Steffen
   TEL;HOME;VOICE:+1 (210) 555-1357
   TEL;HOME;FAX:+1 (210) 555-0864
   ADR:123 Cliff Ave.;Big Town;CA;97531
END:VCARD
```

IrMC bietet wie OBEX und IrDA selbst die Möglichkeit, dass Hersteller eigene Befehle und Datenformate hinzufügen können ohne eine Hersteller übergreifende Funktionalität aufgeben zu müssen. Die Anzahl der Geräte bzw. Kopien sind beliebig. IrMC ist besonders für Mobiltelefone, Pager und PDAs geeignet. In der Regel sind die mobilen Geräte der IrMC Server, was zur Folge hat, dass man sein Mobiltelefon mit dem Rechner zu Hause, dem Rechner auf der Arbeit und einem PDA abgleichen kann. Synchronisiert man allerdings den Rechner zu Hause und den PDA können Duplikate entstehen, da die Technik zur Änderungserkennung Log basiert ist und Zyklen in der Netzwerktopologie nicht unterstützt. Die Log Datei erfasst die Zeitpunkte von Änderungen pro Datei. Die Zeitpunkte können sowohl logische Zähler aber auch physikalische Uhren sein.

Beispiel: *Change Log Datei mit logischen und physikalischen Zeitstempeln:*
SN:1218182THD000001-2
DID: 03df30423
Total-Records:4
Maximum-Records:50
M:123:19990104T180000Z:0A456566
N:124:19990114T180100Z:0FED4101
D:126:19990222T000320Z:133DEFDE
Die ersten beiden Zeilen geben die ID (Seriennummer) des Geräts und die ID der Datenbank an. Danach wird beschrieben wie viele Datensätze vorhanden und wie viele maximal möglich sind. Die Zeilen danach geben die Änderungen an, wobei M *für Ersetzung,* N *für Hinzufügung und* D *für eine Löschung steht. Danach erscheinen durch einen Doppelpunkt abgetrennt der logische und der physikalische Zeitstempel der Änderung. Das letzte Feld in jeder Zeile gibt die LUID des Datensatzes an.*

Da die Dateiformate fest vorgegeben sind, kann im Konfliktfall eine sehr viel genauere

Auflösung des Konflikts stattfinden und zur Entscheidung dem Benutzer entsprechend präsentiert werden. IrMC ist für nur einen Benutzer gedacht und entsprechend gibt es keine Optimierungen für mehrere Benutzer. Der Abgleich erfolgt in der Regel manuell, d.h. der Benutzer muss diesen selbst starten. Der IrMC Server ist fest in das Mobiltelefon integriert und stellt eine Middleware zur Datensynchronisation dar. Dies hat zur Folge, dass alle Änderungen mit deren Zeitpunkten direkt in die Log Datei übernommen werden müssen. Wird ein Abgleich nicht erfolgreich beendet, werden neue erhaltene Änderungen nicht übernommen.

6.1.2. Details

Der empfohlene Nachrichtenablauf von IrMC ist vergleichsweise einfach und ist in [LSS00, Kapitel 5.6] beschrieben. Die vollständige Beschreibung benötigt nicht einmal eine DIN-A4 Seite. Der IrMC Klient – in der Regel ein Personal Computer – sendet den Zeitstempel des letzten Abgleichs und erhält darauf vom Server den Inhalt der Log Datei bis zu diesem Zeitpunkt. Die Log Datei enthält für jede Änderung sowohl die Art (Hinzufügung, Ersetzung oder Löschung) als auch eine LUID (*Local Unique ID*). Der Klient verarbeitet diese Information und bezieht wenn nötig die entsprechenden Datensätze. Danach sendet der Klient die eigenen Änderungen seit dem letzten Abgleich, worauf der Server bei neuen Dateien eine LUID zum Klienten zurück übermittelt, damit der Server den Datensatz beim nächsten Abgleich wieder dem eigenen Datensatz zuordnen kann. Jede übermittelte und gespeicherte Änderung wird am Ende des Abgleichs in die lokale Log Datei geschrieben, damit andere Klienten, diese Änderungen ebenfalls mitbekommen. (ähnliche Abbildung 6.2)

Anstatt GUIDs benutzt ein Klient LUIDs, da diese nur lokal eindeutig sein müssen und dadurch sehr kurz sein bzw. eine feste Größe aufweisen können. Dies ist erneut ein Kompromiss an die Speicherkapazitäten von mobilen Geräten.

Erkennt der Server, dass die Log Datei vom Klienten inkompatibel zu seinen Daten ist, dann wird eine Slow Sync durchlaufen, bei der alle Datensätze des Klienten innerhalb der Datenbank (z.B. Adressbuch oder Kalender) vom Klienten geladen werden.

Wird kein Abgleich von zwei Kopien einer Datenbank benötigt, reicht bereits IrMC Level 2. Hier wird immer das gesamte Adressbuch oder der gesamte Kalender abgefragt bzw. ersetzt. Einzelne Datensätze befinden sich in einer großen Datei, die den Adressbuch- oder Kalenderinhalt darstellt.

6.1.2.1. Datenfelder Reduktion

Da IrMC die Datenformate spezifiziert und genau kennt, kann das Protokoll eine Funktion aufweisen, die von der Forschung und Wissenschaft selten oder scheinbar nach den Recherchen zu dieser Arbeit gar nicht untersucht werden. Wenn zum Beispiel ein Foto an eine vCard angehängt ist oder die vCard selbst viele Felder umfasst (z.B. ein Geburtstag oder verschiedene Lieferanschriften für Pakete und normale Post), kann diese sehr viel Speicherplatz beanspruchen. Ein mobiles Gerät hat womöglich nicht den Speicherplatz oder gar die Ein- und Ausgabemöglichkeiten diese Daten zu verwalten. Falls nun eine übermittelte vCard nur zum Teil gespeichert wird, stellt sich die Frage, was passiert, wenn die Datei geändert und wieder zurück geschickt wird. Geht dann das angehängte Bild verloren?

IrMC bietet dazu die Funktion, dass jeder IrMC Server seine Fähigkeiten anderen Geräten mitteilt. Hierbei werden nicht nur die unterstützen Dateiformate und deren Version aufgezeigt, sondern auch welche Felder des Dateiformats unterstützt werden und wie viel Byte jedes Feld maximal aufnehmen kann.

```
Total-Records:27
Maximum-Records:*
Free-Records:271
IEL:0x08
HD:NO
SAT:CC
DID:1
X-IRMC-FIELDS:
<Begin>
   VERSION:2.1
   N[1=20;2=20]:
   X-TEL-FIELDS:5
      TEL;TYPE=WORK;VOICE:(5){1,10,18,25,32}
      TEL;TYPE=HOME;VOICE:(5){2,11,19,26,33}
      TEL;TYPE=CELL:(5){3,12,20,27,34}
      TEL;TYPE=WORK;FAX:(5){4,13,21,28,35}
      TEL;TYPE=MODEM:(1){5}
      TEL;TYPE=FAX:(5){6,14,22,29,36}
      TEL;TYPE=HOME;FAX:(5){7,15,23,30,37}
      TEL:(5){8,16,24,31,38}
      TEL;TYPE=PAGER:(2){9,17}
   EMAIL:=50
   NOTE:=50
   X-IRMC-LUID:=5
<End>
ICL:NO
OCL:NO
MCL:NO
```

Dies ist das *telecom/pb/info.log* eines Philips Fisio 820 Mobiltelefons für Global System for Mobile Communications (GSM) Netzwerke [ETS04a]. Neben den eigentlichen Fähigkeiten des Geräts, auf die hier nicht weiter eingegangen wird, erhält man zusätzlich die maximalen Längen des Namensfelds.

```
N[1=20;2=20]
```

Dies besagt, dass ein Vorname und Nachname unterstützt wird, und die jeweiligen Felder sind nicht länger als 20 Byte sein sollten. Im diesem Mobiltelefon sind eine E-Mail Adresse und eine Notiz mit maximal 50 Byte pro vCard möglich.

```
X-TEL-FIELDS:5
...TEL;TYPE=MODEM:(1){5}
```

Es sind maximal fünf Felder für Telefonnummern verfügbar. Maximal eines der Felder darf eine Datennummer angeben und dieses Feld wird vom Server übernommen, sollten mehr als fünf Telefonnummern vorhanden sein.

IrMC hat sich in Bezug auf die Datenreduktion bei der Synchronisation sehr große Mühe gegeben. Dies ist allerdings nur dadurch möglich, da die Datenformate genau spezifiziert werden.

6.1.3. Status von IrMC

Bluetooth [BT01] hat IrMC übernommen. Bluetooth ermöglicht wie IrDA kabellose Datenverbindungen über Kurzstrecken, benötigt aber im Gegensatz zu IrDA keine direkte Sichtverbindung zwischen den Geräten. Trotzdem und gleichwohl die IrMC Spezifikation von vielen Herstellern mitentwickelt worden ist, setzt sich IrMC nicht in der Industrie durch. Bereits wenige Monate nach der Veröffentlichung von der IrMC Version 1.1 mit den aktuellen Synchronisationsfähigkeiten ist ein neuer Standard entwickelt worden: SyncML. Innerhalb der IrMC Spezifikation 1.1 sind zwei Betreuer genannt. E-Mail Kontakte mit Rob Lockhart ergaben, dass IrMC nicht mehr weiter entwickelt wird [Loc04]. Letzte wichtige Korrekturen und Änderungen der Spezifikation sind 2000/2001 durchgeführt worden. Fast alle Änderungen sind bei [LSS00] frei zugänglich, nur zwei weitere kleinere Korrekturen benötigen eine IrDA Mitgliedschaft.

6.2. SyncML

SyncML ist ein Hersteller übergreifender Zusammenschluss und hat die theoretischen Ansätze aus IrMC übernommen. Es ist immer noch nur für 1:n Netzwerktopologien optimiert, ist Log Datei basiert, nutzt logische oder physikalische Zeitstempel als Change

Counter und nutzt aus Speicherplatzgründen LUIDs für Datensätze. SyncML ist als IrMC Level 5 [HMPT03] in die IrMC Spezifikation eingeflossen und kann als Fortsetzung der IrMC Bemühungen um ein Hersteller übergreifendes Datenabgleichverfahren angesehen werden. Dazu ist im Jahr 2000 die SyncML Initiative gegründet worden, welche 2003 in der Open Mobile Alliance [OMA00a] aufgegangen ist und die Spezifikation in *OMA (SyncML) DS* umbenannt hat, wobei DS für *Data Synchronization* steht. Die Spezifikation wird regelmäßig in Details korrigiert und erweitert [OMA04b]. Aktuell ist die Version 1.2 vom Juni 2004.

SyncML und IrMC Level 4 unterscheiden sich in den Punkten des Nachrichtenaufbaus und der Server- Klient Beziehung, was in den folgenden Abschnitten genauer beschrieben ist.

6.2.1. Server – Klient

In IrMC ist meist das mobile Gerät z.B. ein Mobiltelefon der Server und ein Personal Computer ist der Klient. In SyncML ist der Server ein größerer Rechner meist im Internet. Alle Geräte gleichen sich über das Internet mit diesem zentralen Server ab. Wird lokal (über Bluetooth, Infrarot oder Kabel) ein Datenabgleich vollzogen, ist das schwächere Gerät z.B. ein Mobiltelefon in der Regel der Klient und der Personal Computer als rechenstärkeres Gerät der SyncML Server. Wird dann ein Abgleich mit dem zentralen Rechner im Internet durchgeführt, ist der lokale Personal Computer wieder der SyncML Klient.

6.2.2. Nachrichtenaufbau in XML

Statt die erweiterten Funktionen von OBEX zu nutzen, werden alle SyncML Nachrichten durch XML beschrieben. Die Gründe für diese Umstellungen zeigen [SD99; ETS01; ETS02] auf.

6.2.2.1. Selbstbeschreibend

Einer der Kritikpunkte an IrMC ist, dass es einen eigenen Parser benötigt um die Tupeldaten Name:Wert aus den Log und Informations-Dateien auszuwerten. Als Alternative wird XML [W3C04b] und seine Fähigkeiten besonders die hierarchische Anordnung der Elemente aufgezeigt. Eine Hierarchie wird bei einem Name:Wert Tupel nicht durch die Datei selbst deutlich, sondern muss durch eine externe Spezifikation beschrieben werden.

Als Gegenargument ist dazu anzuführen, dass die Bedeutung und die Anwendung der XML Element muss aber immer noch in einer separaten Spezifikation festgehalten

werden.

6.2.2.2. Transport unabhängig

Eine Anpassung an XML erlaubt es, sowohl die Synchronisationsbefehle und die jeweiligen Daten in einem Dokument zu beschreiben. Dies macht XML unabhängig vom Transportmedium. IrMC ist nur auf OBEX definiert und nutzt erweitere Funktionalitäten, die es in HTTP [FGM+99] oder anderen Transportmechanismen z.B. SMTP [ITF01] oder IMAP4 [Cri03] nicht gibt.

Als Gegenargument ist dazu anzuführen, dass OBEX nicht auf die Verwendung innerhalb von IrDA beschränkt ist. OBEX ist eine Schlüsseltechnologie in der Bluetooth Spezifikation und wird innerhalb der USB Spezifikation in der Geräte Klassen Beschreibung für USB CDC WMC [USB01] genutzt. USB CDC WMC ermöglicht es, dass mobile Geräte über ein USB Datenkabel den internen OBEX Server und alle Dienste die darauf aufbauen z.B. IrMC als eigenständige Dienst Hersteller übergreifend im Betreibsystem anzumelden. Die IANA hat OBEX den TCP Port 650 zugewiesen, um Nachrichten über das Intranet oder Internet zu senden [IAN05].

Obwohl man OBEX als ein binäres HTTP Äquivalent bezeichnen kann, fehlen noch die Verschlüsselungsmöglichkeiten wie sie HTTP bietet.

6.2.2.3. Nachrichten Reduktion

Da XML sowohl die Synchronisationsbefehle und die jeweiligen Daten in einem Dokument beschreiben kann, erlaubt es sowohl Informationen bezüglichen der Synchronisationszustände (Change Counter) als auch die eigentlichen Daten in einer Nachricht zu übertragen. Dadurch wird es möglich, dass mehrere Daten auf einmal übertragen werden können und die Anzahl der Nachrichten insgesamt verringert wird. Besonders in mobilen Netzwerken in denen Nachrichten durch schlechten Empfang oder Überlastungen verloren gehen können, sind wenige Nachrichten vorteilhafter. Wenn keine Änderungen vorliegen, kann SyncML bereits mit zwei Nachrichten einen Abgleich durchlaufen. Bei einem Abgleich mit Änderungen, reichen bereits vier Nachrichten aus, egal wie viele Änderungen vorliegen. Normalerweise und beim ersten Abgleich sind bis zu sechs Nachrichten erforderlich. Bricht die Verbindung ab, sendet der SyncML Klient die letzte Nachricht erneut. SyncML ist daher unempfindlicher gegenüber Verbindungsabbrüche.

Im Gegensatz dazu wird in IrMC jede Änderung durch einen OBEX Befehl angefordert oder übertragen. Je nach Anzahl der Änderungen entstehen hierdurch sehr viele Nachrichten. Für fehlerfreie Verbindungen z.B. über lokale Verbindungen (Bluetooth,

IrDA, Kabel) ist dies nicht weiter relevant. Für Internetverbindungen kann dies in mobilen Netzwerken bei schlechten Empfangsbedingungen zu Zeitüberschreitungen führen. IrMC muss in diesem Fall die Synchronisation vom Anfang an neu starten.

In den IrMC Korrekturen vom Oktober 1999 wird dieses Problem durch ein neues zusätzliches Log Datei Format gelöst. Der IrMC Server sendet nach dem Erhalt des Change Counters nicht nur die IDs der Änderungen zurück, sondern zusätzlich noch den Eintrag selbst. Damit der IrMC Klient nur eine Nachricht senden muss, wird ein neues Format angegeben, welches erlaubt mehrere Änderungen in einer Nachricht zum Server zu senden. Durch diese Anpassung reduziert sich die Anzahl der Nachrichten in IrMC ähnlich wie in SyncML.

6.2.2.4. Unabhängigkeit vom Dateninhalt

SyncML nutzt MIME Media Types [FB96], um Daten näher zu bestimmen. Media Types können für Nutzerspezifische Daten erweitert werden. SyncML Geräte, welche die gleichen Media Types verstehen, können dadurch die entsprechenden Daten austauschen. Um eine minimale Kompatibilität zu gewährleisten, sieht SyncML die Formate vCard 2.1 [ver96b] und vCalendar 1.0 [ver96a] vor.

In IrMC können ebenfalls eigene Dateiformate eingebunden werden, IrMC schreibt aber nicht das sehr ausgereifte Media Types Konzept vor. In IrMC sind rein binäre Datenobjekte angedacht worden [ETS01, Kapitel 5.2 Absatz 5], diese Überlegungen fanden aber keinen Einzug in die IrMC Spezifikation. SyncML besitzt mit der Unterstützung beliebiger Daten eine wichtige Funktionalität und macht dies für eine größeres Spektrum als nur den Abgleich mit Mobiltelefonen interessant.

XML und binäre Daten XML gibt einen Wertebereich [W3C04b, Abschnitt 2.2] der enthaltenen Zeichen vor, welcher es nicht erlaubt binäre Daten 1:1 in XML einzubinden. Zur Lösung dieses Problems werden binäre Daten außerhalb des XML Dokuments gespeichert und das XML Dokument selbst enthält einen Verweis auf diese Daten. Dies ist in SyncML nicht vorgesehen. Die beschrittene Alternaive bettet die binären Daten in das Dokument mit ein, wobei keine verbotenen Zeichen vorkommen dürfen. Hierzu bietet sich eine Base64 [Int03] Kodierung der binären Daten an. SyncML unterstützt dies durch ein entsprechend definiertes XML Element, welches die Kodierung angibt. Auch das Carriage Return Zeichen (0x0D) wird in XML gesondert behandelt, da XML vorsieht, dass alle Zeilenenden zu einem einfachen Line Feed (0x0A) konvertiert werden [W3C04b, Abschnitt 2.11]. vCard und vCalendar [ver96b; DH98; ver96a; DS98] definieren ihre Zeilenenden allerdings fest mit einem Line Break: 0x0D 0x0A. Dies hat zur Folge,

dass diese Dateiformate als binäre Daten in XML gesondert kodiert werden müssten, um das Carriage Return zu erhalten. In der Praxis werden diese Dateiformate daher auch mit einem einfachen Line Feed als Zeilenende akzeptiert.

Datenfelder Reduktion Obwohl SyncML keine Datenformate kennt, werden die Datenreduktionsfähigkeiten von IrMC (Datenfelder weglassen oder deren Längen zu beschränken) für Dateiformate mit Tupeldaten nicht aufgegeben. Jedes SyncML Gerät verfügt über eine Geräteinformationsdatei, in der die unterstützten Dateitypen aufgelistet werden. Basiert eines der Dateiformate auf Tupeldaten (Name:Wert Paar), dann werden zusätzlich die unterstützten Feldernamen und deren maximalen Wertlängen angebeben.

6.2.2.5. WBXML

Da XML Dokumente vergleichsweise groß werden und sich daher für geringe Empfangspuffer nicht eignen, in einem XML Dokument aber viele wiederholende Textbausteine enthalten sind, bietet sich eine Komprimierung von XML an. HTTP bietet bereits drei verlustlose Komprimierungsverfahren an [FGM$^+$99, Abschnitt 3.5], die dynamisch erzeugte Wörterbücher nutzen. Diese Komprimierungen könnten den Empfangspuffer und die Dauer der Übertragung minimieren. Da die XML Elemente in SyncML durch drei DTDs fest vorgegeben sind, bietet sich eine Kombination aus statischem Wörterbuch für die XML Elemente und einem dynamischen Wörterbuch für Elementinhalte an. Das WAP Binary XML Format (WBXML) [OMA01] ist eine solche Kodierung und entstand im Rahmen der Wireless Application Protocol Familie (WAP) [OMA00b], die ebenfalls wie SyncML von der Open Mobile Alliance verwaltet wird. Ein Vorteil von WBXML liegt darin, dass es die XML Struktur erhält. Jedes XML Element wird durch das entsprechende Byte aus der statischen Tabelle ersetzt. Ein WBXML Leser braucht diese Ersetzung nicht wieder rückgängig zu machen, sondern kann direkt auf dieser Datei arbeiten. Dies hat zur Folge, dass die Datei nicht vom Empfangspuffer kopiert werden muss, sondern direkt im Puffer bearbeitet werden kann. Beim Empfänger entstehen daher keine Laufzeit oder Speichernachteile. Aus Laufzeitgründen kann der Sender darauf verzichten das dynamische Wörterbuch zu erstellen.

Das World Wide Web Consortium (W3C) zuständig für die XML Spezifikation hat eine Arbeitsgruppe für binäres XML eingerichtet [W3C04a]. Diese Arbeitsgruppe untersucht im jetzigen Stadium die Nutzungsszenarien von binärem XML. Die langsame Fortschreitung zu diesem Thema liegt unter anderem daran, dass die Lesbarkeit von Dokumenten durch Menschen eines der Ziele von XML darstellt. Ein binäres Format gibt

dieses Ziel auf. Darüber hinaus existiert eine Vielzahl an binären XML Formaten, wovon jedes seine Stärken und Schwächen besitzt. WBXML zum Beispiel basiert auf einem statischen Wörterbuch. Ein Wechsel des Wörterbuchs aufgrund von eines Namensraumwechsels innerhalb des XML Dokuments ist nicht definiert.[2]

6.2.3. Gliederung der SyncML Spezifikation

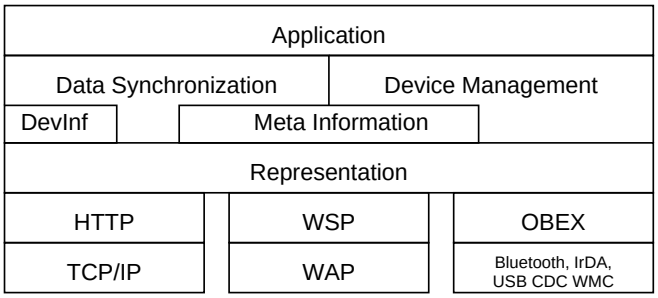

Abbildung 6.1.: SyncML Protokoll Architektur

Die SyncML Spezifikation definiert drei *Transport Bindings*, die angeben wie Daten über OBEX, HTTP oder WSP[3] versendet werden. Für OBEX wird zusätzlich beschrieben, wie der SyncML Dienst in IrDA oder Bluetooth auffindbar ist. Aufgrund der XML Unabhängigkeit von einem Transportprotokoll, sind hier weitere herstellerspezifische Transporte denkbar.

Über den Transportweg werden XML Elemente versendet. Diese Elemente sind in der SyncML *Representation* Spezifikation aufgelistet. Es werden Abhängigkeiten der Elemente untereinender und der Sinn und Zweck also die Einsatzmöglichkeiten jedes Elements beschrieben. Für diesen allgemeinen SyncML Namensraum wird eine entsprechende DTD definiert. Darüber hinaus werden zwei weitere Namensräume beschrieben:

1. Meta Information
   Diese Elemente geben weitere Informationen über Dateninhalte (z.B. Typ oder Kodierung).

2. Device Information (DevInf)
   Diese Elemente ermöglichen den Austausch von unterstützten Dateiformaten.

[2]Dies ist ein Problem innerhalb von SyncML, da die Elemente der Device Information DTD ein anderes Wörterbuch nutzen als die restlichen SyncML Elemente. Die Device Information wird in der Regel bei jedem Datenabgleich durch einen Namensraumwechsel übertragen. Bei Benutzung von WBXML führt dies zu Inkompatibilitäten innerhalb von SyncML.

[3]Ein Hersteller übergreifendes Transportprotokoll aus dem Mobilfunkbereich aus der Wireless Access Protocol (WAP) Familie.

Die nächste Ebene beschreibt den Nachrichtenablauf und konkretisiert die SyncML XML Elemente ihrer Bedeutung. Die Aufgabe und mögliche Wertebelegungen der Elemente innerhalb einer Nachricht werden festgelegt. Darüber hinaus werden Anforderungen aufgezeigt, die ein Klient und Server erfüllen müssen.

Aktuell basieren zwei Spezifikationen auf der SyncML Representation. Ein Protokoll dient zum Verwalten von Geräteeinstellungen: Device Management; abgekürzt OMA DM. *Mobilfunkanbieter* sollen auf diese Weise die mitunter für einen Benutzer sehr komplexen Einstellungen eines Mobiltelefons aus dem Mobilfunknetz warten können. *Hersteller* sollen dieses Protokoll nutzen, um Teile der Software/Firmware Module über das Mobilfunknetz zu warten. Das zweite Protokoll an dem sich diese Ausarbeitung orientiert und seinen Schwerpunkt legt, dient zum Datenabgleich der Benutzerdaten. Dieses *Data Synchronization* (abgekürzt *OMA DS*) wird im weiteren Verlauf dieser Ausarbeitung der Einfachheit halber mit *SyncML* bezeichnet. Der *Device Information* Namensraum beschreibt die Dateiformate beim Datenabgleich und wird nur von OMA DS genutzt. Der *Meta Information* Namensraum wird von beiden Protokollen genutzt.

6.2.4. Nachrichtenablauf

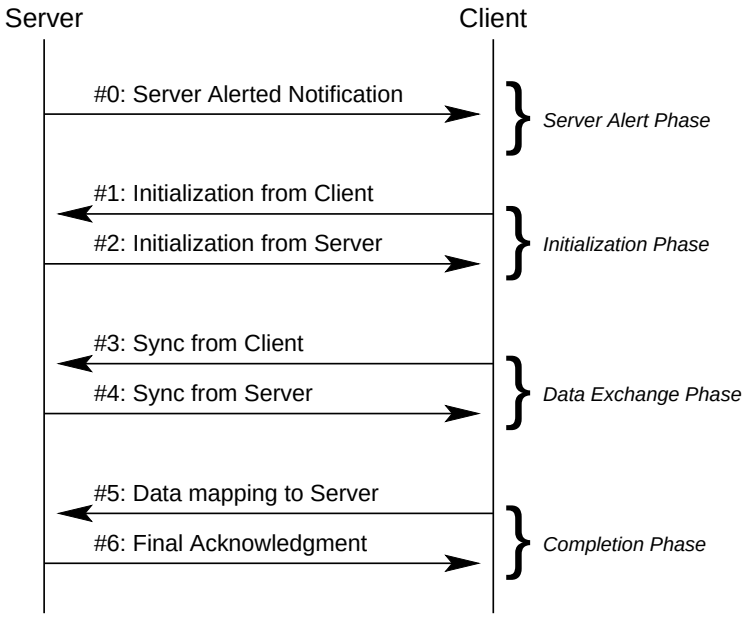

Abbildung 6.2.: OMA SyncML DS – Nachrichtenablauf

Der Nachrichtenablauf gestaltet sich ähnlich wie in IrMC (Abbildung 6.2). Baut der Server eine Verbindung zum Klienten auf, dann wird die optionale Server Alert Pha-

se durchlaufen. Danach verläuft der Nachrichtenabgleich genau so, als hätte der Klient den Abgleich gestartet. In der Phase der Initialisierung werden wie bei IrMC die Change Counter und die Gerätefähigkeiten ausgetauscht. In SyncML wird darüber hinaus die aktuelle Sitzung des Abgleichs authentifiziert. IrMC hätte dies während des Aufbaus der OBEX Verbindung durchgeführt. Danach werden die geänderten Daten ausgetauscht. Bei der Komplettierung des Abgleichs werden die genutzten LUIDs im Klienten übermittelt, welcher der Server nochmals bestätigt. IrMC nutzt für die LUID Übertragung erweiterte Funktionen von OBEX.

6.2.5. SyncML XML Elemente

Durch die Verwendung von XML, ist der Aufbau der Nachrichten im Vergleich zu IrMC verschieden, weswegen ein kurzer Überblick über die wichtigsten XML Elemente und den Paketaufbau selbst gegeben wird. SyncML unterscheidet darüber hinaus Nachrichten und Pakete.[4] Eine Nachricht kann aus mehreren Paketen bestehen, wenn die Nachricht für den Transportweg oder die Puffer des empfangenden Geräts zu groß wäre. Daher kann es sein, dass sich die im Ablauf gezeigten Nachrichten und Phasen überlappen.

Beispiel: *Der Server sendet immer noch Änderungen aus der Datenabgleichphase, der Klient antwortet bereits mit den LUIDs, um den Abgleich bald zu komplettieren.*

Ist das Element *Final* enthalten, handelt es sich um das letzte Paket innerhalb einer Nachricht. Im Folgenden wird der obige Nachrichtenablauf beginnend mit einem Server Alerted Notification Paket [OMA03a, Kapitel 13] aufgezeigt und die wichtigsten Elemente werden beschrieben. Vereinfacht werden Nachrichten gezeigt, die nur aus einem Paket bestehen. In diesem Beispiel hat vorher zwischen Server und Klient noch kein Abgleich stattgefunden (Slow Sync).

Jedes SyncML Paket besteht aus Kopfinformationen `SyncHdr` und einem Paketinhalt `SyncBody`. Dieser Ausschnitt zeigt die Kopfinformationen in denen die Versionsnummer, Sitzungsnummer und Paketnummer dem SyncML Partner mitgeteilt werden. Danach folgen das Ziel und die Quelle des Pakets, die in einer URI [BLFSM05] kodiert sind.

```
<SyncML>
   <SyncHdr>
      <VerDTD>1.1</VerDTD>
      <VerProto>SyncML/1.1</VerProto>
      <SessionID>1</SessionID>
      <MsgID>1</MsgID>
```

[4] Im Englischen der SyncML Spezifikation lautet der Begriff *Nachricht* in English *package* und *Paket* wird *message* genannt.

```
        <Target>
           <LocURI>/</LocURI>
        </Target>
        <Source>
           <LocURI>OBEX:SYNCML-SYNC/</LocURI>
        </Source>
     </SyncHdr>
     <SyncBody>
        ...
     </SyncBody>
</SyncML>
```

Der Paketinhalt enthält die Befehle, die der SyncML Partner auszuführen hat. In diesem Fall soll der Klient durch ein `Alert` mit dem Dateninhalt 206 dazu bewogen werden, seinerseits eine SyncML Sitzung zu starten. Dabei soll nur die Adressbuch Datenbank (Dateiformat `text/x-vcard`) abgeglichen werden. `Source` gibt dabei die Datenbankquelle auf dem Server an. Diese URI wird mit der URI aus den Kopfinformationen verknüpft und ergibt die absolute URI: `OBEX:SYNCML-SYNC/Contacts`. Das Paket enthält das Element `Final` und stellt damit eine vollständige Nachricht dar.

```
<SyncML>
   <SyncHdr>
      ...
   </SyncHdr>
   <SyncBody>
      <Alert>
         <CmdID>1</CmdID>
         <Data>206</Data>
         <Item>
            <Source>
               <LocURI>Contacts</LocURI>
            </Source>
            <Meta>
               <Type xmlns='syncml:metinf'>text/x-vcard</Type>
            </Meta>
         </Item>
      </Alert>
      <Final/>
   </SyncBody>
</SyncML>
```

Der Klient antwortet und startet damit die Initialisierungsphase. Die Quell- und Zieladressen werden aus Sicht des Klienten gesendet. Darüber hinaus wird dem Server mitgeteilt, dass die Pakete nicht größer als 3584 Bytes sein dürfen.

```
<SyncML>
   <SyncHdr>
      <VerDTD>1.1</VerDTD>
```

```
      <VerProto>SyncML/1.1</VerProto>
      <SessionID>1</SessionID>
      <MsgID>1</MsgID>
      <Target>
         <LocURI>OBEX:SYNCML-SYNC/</LocURI>
      </Target>
      <Source>
         <LocURI>IMEI:54746288917</LocURI>
      </Source>
      <Meta>
         <MaxMsgSize xmlns='syncml:metinf'>3584</MaxMsgSize>
      </Meta>
   </SyncHdr>
   <SyncBody>
      ...
   </SyncBody>
</SyncML>
```

Jeder Befehl und der Erhalt der Kopfinformationen werden durch `Status` Element mit einem Fehlercode den HTTP Fehlercodes ähnlich quittiert. Eine 200 bedeutet wie in HTTP einen Erfolg.

```
<SyncML>
   <SyncHdr>
      ...
   </SyncHdr>
   <SyncBody>
      <Status>
         <CmdID>1</CmdID>
         <MsgRef>1</MsgRef>
         <CmdRef>0</CmdRef>
         <Cmd>SyncHdr</Cmd>
         <TargetRef>IMEI:54746288917</TargetRef>
         <SourceRef>OBEX:SYNCML-SYNC/</SourceRef>
         <Data>200</Data>
      </Status>
      <Status>
         <CmdID>2</CmdID>
         <MsgRef>1</MsgRef>
         <CmdRef>1</CmdRef>
         <Cmd>Alert</Cmd>
         <SourceRef>Contacts</SourceRef>
         <Data>200</Data>
      </Status>
      ...
   </SyncBody>
</SyncML>
```

Im Paketinhalt folgen weitere Kommandos, die der Server auszuführen hat. Hier for-

dert der Klient den Server zu einem Slow Sync auf, d.h. alle Dateien der Adressbuch Datenbank beider Kopien müssen übertragen werden. Der Klient teilt darüber hinaus seinen aktuellen Change Counter in den Elementen `Anchor` und `Next` mit.

```
<SyncML>
   ...
   <SyncBody>
      ...
      <Alert>
         <CmdID>3</CmdID>
         <Data>201</Data>
         <Item>
            <Target>
               <LocURI>Contacts</LocURI>
            </Target>
            <Source>
               <LocURI>telecom/pb.vcf</LocURI>
            </Source>
            <Meta>
               <Anchor xmlns='syncml:metinf'>
                  <Last>0</Last>
                  <Next>63</Next>
               </Anchor>
            </Meta>
         </Item>
      </Alert>
      ...
   </SyncBody>
</SyncML>
```

Damit der Server über die möglichen Datenfelder des Klienten informiert ist, werden die entsprechenden Geräte Informationen vom Klienten übermittelt. Hierzu dient der Befehl `Put`, durch den es möglich ist, direkt einzelne Dateien an den SyncML Partner zu senden. Um einzelne Dateien zu erhalten wird ein `GET` mit entsprechender Pfadangabe gesendet.

```
<SyncML>
   ...
   <SyncBody>
      ...
      <Put>
         <CmdID>4</CmdID>
         <Item>
            <Source>
               <LocURI>./devinf11</LocURI>
            </Source>
```

```
            <Meta>
               <Type xmlns='syncml:metinf'>
                  application/vnd.syncml-devinf+xml
               </Type>
            </Meta>
            <Data>...</Data>
         </Item>
      </Put>
      <Final/>
   </SyncBody>
</SyncML>
```

Die Antwort vom Server enthält nur die entsprechenden Status Informationen und bestätigt den langsamen Abgleich des Klienten. Da keine neue XML Elemente verwendet werden, ist für die Betrachtungen erst die nächste Nachricht vom Klienten aus der Datenabgleich Phase wieder interessant. Der Klient sendet alle seine Dateien in einem `Sync` Element. Dieses zeigt einen Datenabgleich an. Darin enthalten sind Datenbefehle wie `Add` (Hinzufügungen), `Replace` (Ersetzungen) und `Delete` (Löschungen), die dem Partner anzeigen sollen, was mit den Daten passieren soll. Da dies der erste Abgleich ist, erscheinen nur `Add` Elemente. Außerdem wird die Anzahl der noch freien Speicherplätze mitgeteilt.

```
<SyncML>
   ...
   <SyncBody>
      <Sync>
         <CmdID>3</CmdID>
         <Target>
            <LocURI>Contacts</LocURI>
         </Target>
         <Source>
            <LocURI>telecom/pb.vcf</LocURI>
         </Source>
         <Meta>
            <Mem xmlns='syncml:metinf'>
               <FreeMem>48500</FreeMem>
               <FreeID>485</FreeID>
            </Mem>
         </Meta>
         <Add>
            <CmdID>4</CmdID>
            <Meta>
               <Type xmlns='syncml:metinf'>text/x-vcard</Type>
            </Meta>
            <Item>
               <Source>
                  <LocURI>1</LocURI>
               </Source>
```

```
                <Data>
                   BEGIN:VCARD
                      VERSION:2.1
                      N:Alexander Traud
                      ADR;HOME:;;Briegelweg 7;Darmstadt;;64287;D
                      EMAIL;INTERNET:alex@traud.de
                      NOTE:Author of Enkel DS
                      X-IRMC-LUID:1
                   END:VCARD
                </Data>
             </Item>
          </Add>
       </Sync>
       ...
    </SyncBody>
 </SyncML>[5]
```

Der Server sendet seine Daten mit den gleichen XML Elementen `Add`, `Replace` oder `Delete` an den Klienten. Der Klient bestätigt den Empfang der Daten und sendet die lokal zugewiesenen LUIDs in einem `Map` Element zurück, so dass der Server beim nächsten Abgleich die geänderte Dateien seinen gespeicherten Dateien zuordnen kann. In diesem Beispiel hatte der Server eine neue Datei an den Klienten gesendet. Lokal im Klienten ist die nächste freie LUID `2` zugeordnet worden. Die GUID des Datensatzes beim Server lautet `10536681`. Das `MapItem` zeigt dies durch die entsprechenden `Target` (Server GUID) und `Source` (Klient LUID) an. Es muss kein absoluter URI angegeben werden, da diese sich aus dem umschließenden `Map` Befehl und den Kopfinformationen des Pakets zusammen bauen lassen.

```
<SyncML>
   ...
   <SyncBody>
      <Map>
         <CmdID>3</CmdID>
         <Target>
            <LocURI>./Contacts</LocURI>
         </Target>
         <Source>
            <LocURI>/telecom/pb.vcf</LocURI>
         </Source>
         <MapItem>
            <Target><LocURI>10536681</LocURI></Target>
            <Source><LocURI>2</LocURI></Source>
         </MapItem>
      </Map>
      ...
   </SyncBody>
```

[5] In der übermittelten vCard taucht ein IrMC Feld auf. Dieses Beispiel Gerät unterstützt sowohl IrMC als auch SyncML. Beiden Technologien können in einem Gerät unterstützt werden.

```
</SyncML>
```

Der Server bestätigt durch entsprechende `Status` Befehle, dass die `Map` angekommen ist und gespeichert werden konnte. Eine Verbindung kann abgebaut werden und die Daten werden auf beiden Seiten gespeichert. Die Change Counter werden im Server für den nächsten Abgleich gesichert. Der Datenabgleich war erfolgreich.

Die URI Rountinginformationen und die `Status` Befehle sind in SyncML nötig, da im Gegensatz zu IrMC diese Informationen nicht mehr eindeutig durch das Transportprotokoll übermittelt werden können, da eine Verbindung während des Abgleichs verloren gehen kann und ein Klient in diesem Fall nur die unbestätigten Befehle erneut Senden soll.

7. Standardisierungsgremien

Innerhalb von offiziellen Standardisierungsgremien gibt es zurzeit noch keine Spezifikation für ein Abgleichverfahren. Ziel eines offiziellen Standards wäre, dass sich eine Vielzahl an verschiedenen Gerätetypen untereinander abgleichen. Die Aufgabe des Standard wäre das Format und die Abfolge der Nachrichten zu bestimmen, die zwischen den einzelnen Kopien beim Abgleich ausgetauscht werden.

Das Problem eines solchen Standards ist, dass es sehr viele unterschiedliche Gerätetypen gibt und eine große Anzahl an Kriterien existiert, die zu berücksichtigen sind. Innerhalb der Kriterien eine Auswahl zu treffen, heißt auch meist die möglichen Fähigkeiten des Systems vorzugeben. Allerdings steigt mit höheren Fähigkeiten des Systems (z.B. Unterstützung von Zyklen im Netzwerk, Mehr-Benutzer System usw.) die Rechenleistung und der Speicherbedarf. Ebenso steigt auch der Programmieraufwand. Dies ist nicht unerheblich, da man davon ausgehen kann, dass viele Programme in der Zukunft eine Datensynchronisation bereitstellen wollen. Hier werden viele Programmierer weiterhin auf eigene Lösungen setzen, die womöglich schneller zu implementieren und zu testen sind. Es wäre daher nötig, dass ein Standard gefunden wird, der für eine große Anzahl an Gerätetypen einsetzbar ist und einfach und schnell zu implementieren ist.

Im Folgenden wird ein Standard aus dem Themenfeld Adressbuch und Kalenderdatenbank aufgezeigt, die sich auch für einen Abgleich verschiedener Datenbestände eignet.

7.1. AT Kommandos

7.1.1. ITU-T V.250

Ursprünglich hatte die Firma Hayes in ihren Produkten eine sehr einfache Schnittstelle entworfen, um ein Modem anzusteuern, es wählen zu lassen und schließlich eine Verbindung mit einem anderen Modem aufzubauen. Alle Hersteller von analogen Modems haben diese Schnittstelle übernommen, damit ein Austausch der Modem ohne große Konfigurations-Änderungen im Computer möglich war. Diese Schnittstelle ist von ITU-T formal spezifisiert worden. [ITU03]

AT steht für *Attention* (Englisch für *Achtung*) und signalisierte dem Modem, dass ein

Befehl folgt der auszuführen ist. Der Befehl wird immer mit einem Zeichen abgeschlossen, welches dem Modem anzeigt, dass der Befehl komplett eingeben worden ist und zur Bearbeitung vorliegt. Dies ist üblicherweise ein *Carriage Return*, kann allerdings auf beliebiges anderes Zeichen umgestellt werden. Ein Interpreter im Modem wertet diesen Befehl aus und liefert eine entsprechende Antwort. Das Format des Befehls und der Antwort sind durch eine Backus Naur Form genau spezifisiert. Weitere Details der Syntax sind in [ITU03, Kapitel 5] zu finden.

Darüber hinaus wird jedem Befehl eine Bedeutung zugeordnet. Es wird nicht angegeben *wie* der Befehl umgesetzt wird, sondern nur welches Ergebnis nach Ausführung geliefert werden soll. Da sich ITU-T bewusst ist, dass Hersteller und andere Gremien auf dieser Spezifikation aufbauen und erweiterte AT Kommando Schnittstellen mit weiteren Befehlen anbieten wollen, gibt es hierfür entsprechende Mechanismen. [ITU03, Abschnitt 5.8][ITU01]

7.1.2. 3GPP 27.007

Das European Telecommunications Standards Institute (ETSI) hat entsprechend in seinem Verantwortungsbereich ITU-T V.250 für Mobilfunkgeräte in Global System for Mobile Communications (GSM) und Universal Mobile Telecommunications System (UMTS) angepasst und erweitert. [ETS04a] Dabei entstanden nicht nur Befehle, um eine mobile Datenverbindung über das Mobiltelefon aufzubauen oder die Empfangsstärke des Mobiltelefons auslesbar zu machen, sondern es sind eine Reihe weiterer AT Kommandos entworfen worden. Für das Thema Datensynchronisation sind die folgenden Befehle von Bedeutung:

- `AT+CPBR=?`
- `AT+CPBS=?`
- `AT+CPROT=0`
- `AT+CLCK=?`

Die ersten beiden Befehle erlauben es, Telefonbücher auszulesen. Dabei ist zu beachten, dass die ETSI selbst ein Telefonbuch auf der Subscriber Identity Module (SIM) [ETS03] Karte vorschlägt, welches einer Telefonnummer einen Namen zuweißt. Eine SIM ist für GSM Mobiltelefone erforderlich, um sich im Netzwerk anzumelden und zu authentifizieren. Da die ETSI genau wie die ITU nicht den Funktionsumfang eines Adressbuches im Mobiltelefon vorgeben kann und will, beschränken sich diese Befehle immer darauf

maximal eine Nummer pro Namen zu verwalten. Entsprechend sind die Befehle bei modernen Geräten mit komplexen Adressbüchern (pro Name mehrere Datenfelder) wenig geeignet.[1] Der letzte Befehl erlaubt das Auslesen und Setzen der Uhrzeit in einem Mobiltelefon. Dies ist für den Befehl `AT+CPROT=0` interessant, denn dieser erlaubt es OBEX und damit IrMC und SyncML over OBEX zu starten. Wenn in diesen Protokollen nicht Change Counter sondern absolute Zeitangaben verwendet werden, muss die Uhrzeit des Mobiltelefons entsprechend beachtet werden.

7.1.3. Weitere Entwicklungen

Viele Hersteller haben sich von diesen Befehlen inspirieren lassen und eigene Erweiterungen entwickelt. Auch heute noch arbeiten fast alle Hersteller im Mobilfunkbereich (z.B. Motorola, Samsung, Panasonic) mit eigenen AT Kommandos, um das komplexe Adressbuch und Kalenderinformationen der Mobiltelefone durch Personal Computer oder PDAs verwalten zu können. Dies ermöglich den Herstellern Schnittstellen für den eigenen Funktionsumfang des Mobiltelefons zu schaffen. So hat weder die ITU noch ETSI Befehle direkt für die Verwaltung von Kalenderdaten.

Durch die Integration von Bluetooth und dessen IrMC Level 4 und die weitere Verbreitung von SyncML Klienten, werden auf absehbare Zeit Hersteller spezifische AT Kommandos obsolet. Die Hersteller übergreifenden Möglichkeiten, die Bluetooth, Infrarot, USB CDC WMC und `AT+CPROT=0` bieten, halten immer mehr Einzug und ermöglichen es über Geräte Generation aber auch über Hersteller hinweg, Programme zum Datenabgleich für Personal Computer und PDAs zu entwickeln.

Die ETSI hat im Umfeld der Entwicklungen für die Mobilfunknetze der nächsten Generation IrMC aber besonders SyncML als Standard für den mobilen Datenabgleich aufgegriffen. [ETS00; ETS02]

[1] Die Einführung der USIM [ETS04c] für UMTS und die aktuelle Stufe von 27.007 [ETS04b] vom Dezember 2004 erlauben nun auch mehrere Felder pro Adressbuch Kontakt.

8. Problemstellungen

Die Kriterien der Datenabgleichverfahren zeigen bereits viele Problemstellungen auf und geben mögliche Ziele vor.

8.1. Netzwerktopologie

Idealerweise unterstützt ein Verfahren Netzwerktopologien mit Zyklen. Die Projekte CIPSync und Tra beschäftigen sich intensiv mit den Problemen der langsamen Synchronisation oder erhöhtem Speicherplatzverbrauch für Metadaten, wenn Zyklen unterstützt werden. Das Unison und Harmony Projekt beschäftigt sich mit der Korrektheit eines Datenabgleichsverfahrens. Unison beschränkt sich dabei noch auf Netzwerktopologien mit einer reinen 1:1 Beziehung.

8.2. Gründe für Abgleiche

Einen Datenbestand abzugleichen ist Zeit- und Ressourcenaufwendig. Tra bietet bereits die Möglichkeiten nur Teile, IrMC und SyncML ermöglichen es, nur bestimmte Datenfelder eines Datenbestandes abzugleichen. Diese Möglichkeiten werfen weitere Fragen auf. Angenommen die Daten wandern über mehrere Knoten und Datenfelder gehen dabei verloren, obwohl der empfangene Knoten an den restlichen Datenfeldern interessiert ist. Wie kann er diese Informationen wieder auffinden?

8.3. Benutzeranzahl

OceanStore beschäftigt sich mit der Problematik von vielen verschiedenen Benutzern. Viele andere Projekte beschäftigen sich maximal einem Benutzer. Bei der Unterstützung von mehr als einem Benutzer stellen sich Fragen zur Weitergabe von Daten. Sind die Daten tatsächlich von diesem Autor oder sind sie auf Weg des Datenabgleichs verfälscht worden? Nicht jeder soll den persönlichen Kalender einsehen können. Wie kann der Empfängerkreis von bestimmten Daten eingeschränkt werden?

8.4. Benutzerschnittstelle

Wenn ein Konflikt auftritt, bietet es sich an, den Konflikt durch den Benutzer aufzulösen zu lassen. Es stellt sich die Frage, wie eine Benutzerschnittstelle aussehen könnte, die bereits soweit wie möglich Vorschläge macht und eine Entscheidung unterstützt. Sind an dem Konflikt mehrere Benutzer beteiligt, dann stellt sich die Frage, wie die Benutzerschnittstelle helfen kann, das Problem zu verdeutlichen und einen Konsens für alle Benutzer zu finden.

8.5. Zusammenfassung

Diese Bereiche machen deutlich, dass noch einige Fragestellungen offen sind und viele Fragestellungen nur vereinzelt angegangen werden. Kein Projekt beschäftigt sich mit allen aufgezeigten Kriterien. Es werden nur punktuelle Lösungen angeboten, wobei es die Projekte aufgrund der verschiedenen Zielsetzungen kaum oder gar nicht erlauben, kombiniert zu werden.

9. Unterstützung von Zyklen in SyncML

Der Hersteller übergreifende Standard SyncML unterstützt keine Zyklen in der Netzwerktopologie. Um Konflikte und Duplikate von Datensätzen zu vermeiden, müssen Benutzer diese Netzwerktopologie meiden und immer direkt mit einem zentralen Datenabgleichserver kommunizieren. Dieser Server befindet sind meist im Internet und ist nicht unter der Kontrolle des Benutzers. Ziel dieser Arbeit ist es, SyncML um die Unterstützung von Zyklen zu erweitern, damit Benutzer jedes Gerät direkt mit jedem abgleichen können und damit die günstigsten Netzwerkverbindungen ausgenutzt werden. Dies ermöglicht auf einen zentralen Server im Internet zu verzichten, auf dem persönliche Daten zwischengespeichert werden müssten. Der Benutzer kann dadurch seine Daten in seinem persönlichen Umfeld halten und muss diese nicht auf einem entfernten Server abspeichern. Es ist zu erwarten, dass dies das Vertrauen von Benutzern in einen Datenabgleich erhöht. Darüber hinaus dürfen so wenige Konflikte wie nur möglich auftreten, damit ein Benutzer nicht mit sich wiederholenden Konfliktlösungen beschäftigt ist.

9.1. Vorgeschlagene Lösung

Die Beschreibung des Netzwerktopologie Kriteriums in dieser Ausarbeitung zeigt, dass ein Log Datei basiertes Verfahren wie SyncML auch für n:n Szenarien geeignet sein kann, wenn eine rein hierarchische Anordnung der Knoten im Gesamtsystem gegeben ist. [HMPT03, Kapitel 1, Seite 8–9] behauptet, dass SyncML jede n:n Beziehung also auch Zyklen innerhalb von beliebigen Graphen unterstützt. Die Autoren dieses Buches meinen, dass dies mit dem bereits beststehenden OMA SyncML DS möglich sei:

> SyncML, however, *allows* many-to-many synchronization. It allows each data item to have an associated version which could actually be a version vector required for many-to-many synchronization. It also does not specify the format of the sync anchor explicitly and therefore that could also be a version vector. Furthermore, a SyncML device can play dual roles of a server or a client.

Um diesen Vorschlag besser verstehen zu können, werden die Definitionen [OMA03b] der genannten SyncML Elemente benötigt:

Version Usage: Specifies the revision identifier of a data object. [...]
Restrictions: The value MUST specify either an UTC based date/time stamp or a monotonically increasing numeric integer string.

Ein Versionen Vektor müsste daher flach als eine Zahl repräsentiert werden. Dies könnte man dadurch erreichen, dass jeder Eintrag im Vektor eine feste Größe erhält. Bei einem Datensatz mit vielen Änderungen kann dies zu einem Überlauf der fest zugeordneten Zahl führen. Zum Beispiel wenn man jeder Kopie maximal drei Dezimalstellen zuordnet, entsteht bereits nach 999 Änderungen ein Überlauf.

Beispiel: *999 001 004* → *000 001 004*
Der Datensatz ist drei Kopien bekannt. An der ersten Kopie sind 999, an der zweiten Kopie eine und an der dritten Kopie sind nur vier Änderungen vorgenommen worden.

Dies ist kein monoton steigender Zähler. Eine Lösung wäre eine zusätzliche Zahl, die angibt wie viele Dezimalstellen jeder Kopie folgen. Dies entspricht einem Längenindikator ähnlich dem der Programmiersprache Pascal für Strings. Dadurch entsteht ein monoton wachsender Integer.

Beispiel: *3999 11 14* → *41000 11 14*
Die erste Kopie besitzt eine dreistellige Zahl und 999 Änderungen sind auf der Kopie durchgeführt worden. Für die zweite Kopie folgte eine einstellige Zahl und es ist eine Änderung vorgenommen worden. Für die dritte Kopie folgt ebenfalls nur eine einstellige Zahl. Auf dieser Kopie sind vier Änderungen vorgenommen worden.

In dieser Lösung dürfen die Kopien ihre Position im Vektor nicht ändern, da z.B. die Umordnung `11 14 3999` die Monotonie zerstört. Bei festen Vektoren ändert sich die Position nicht. Die Anzahl der Knoten und deren Position müssten am Anfang bestimmt werden.

Anchor Usage: Specifies the the synchronization state information (i.e., sync anchor) for the current synchronization session. [...]
Restrictions: The OPTIONAL Last element type specifies the synchronization anchor for the previous synchronization session. The REQUIRED Next element type specifies the synchronization anchor for the current synchronization session.

Das angesprochene `Anchor` Element besitzt die Unterelemente `Last` und `Next`. Da die letzten Abgleichvektoren aus Speicherplatzgründen nicht mehr vorliegen dürften, reicht die Beschreibung des Unterelements `Next`:

Next Usage: Specifies the synchronization state information (i.e., sync anchor) for the current synchronization session. [...]
Restrictions: The value MUST specify either an UTC based date/time stamp or a monotonically increasing numeric integer String.

Es bestehen die gleichen Probleme wie beim Element `Version`.

Problematisch ist, dass SyncML nicht festlegt, wie ein solcher Vektor repräsentiert werden soll. Wenn sich eine Implementierung für eine Lösung entscheidet, dann entsteht automatisch eine Insellösung. Es ist aber festzuhalten, dass die Behauptung von [HMPT03] richtig zu sein scheint. Man kann Vektoren in SyncML unterbringen, auch wenn dies Insellösungen darstellen werden. Die Kopien sind von Anfang an fest vorgegeben und können sich nicht mehr ändern.

Eine beliebige Graphenunterstützung entsteht dadurch allerdings nicht. Zwei Einträge werden in der Kopie A erzeugt und an die Kopien B und C weitergegeben. Wenn B und C nun untereinander synchronisieren, den Graphen schließen und damit einen Zyklus entstehen lassen, wie sollen B und C erkennen, dass es sich bei den für sie neuen Einträgen um dieselben handelt? Dieses Problem könnte auf Applikationsebene abgefangen werden. Neue Datensätze werden mit allen vorhandenen Datensätzen verglichen. Was geschieht wenn einer der Knoten, den neuen Datensatz in der Zwischenzeit geändert hat, so dass kein Konflikt entsteht, er aber nicht mehr zugeordnet werden kann? Um dieses Problem zu lösen, müsste jeder Datensatz eine Systemweit eindeutige ID besitzen. Auf dieses Problem wird in der vorgeschlagenen Lösung von [HMPT03] nicht eingegangen.

Bei der weiteren Lektüre von OMA SyncML DS [OMA04c, Data Sync Protocol, Abschnitt 6.2.2] findet man:

> Sync Anchors for Data Items
> This protocol does not specify the functionality to transfer the sync anchors associated with individual data items. If this functionality is desired, it MUST be provided inside the data items if it is included. An example is the Sequence Number property of vCalendar [...].

Ein Anker für jeden Datensatz ist nichts anderes als der vorgeschlagene Vektor im Element `Version`. Der zweite Satz besagt, dass eine solche gewollte Funktionalität nur in einem Datensatz vorkommen darf. Das angeführte vCalendar Beispiel zeigt, dass

hier tatsächlich der Inhalt eines Datensatzes gemeint ist und nicht die möglichen Meta Informationen (und damit das Element `Version`) des Elements `Item`.

Dies hat zur Folge, dass jeder Datensatz komplett übertragen werden müsste oder bei der Verwendung eines Sync Ankers im Element `Version` wird die Spezifikation nicht eingehalten.

Die Motivation hinter dieser Formulierung der Spezifikation bleibt verborgen. Diese Formulierung ist bereits seit SyncML Version 1.0 [OMA00a] vorhanden und macht den vorgeschlagenen Ansatz für die Zyklenunterstützung zunichte.

9.2. Mögliche Lösung

IrMC und SyncML sehen vor, dass mit unbekannten Knoten ein *Slow Sync* durchzuführen ist. Alle Datensätze des Knotens werden mit denen des anderen Knotens verglichen. Inhaltlich gleiche Datensätze werden als Kopie desselben Datensatzes wahrgenommen. Aufgrund der Tupel Gestalt von vCard und vCalendar kann ein Vergleich Feldweise stattfinden. Ein Datensatz ohne eine Kopie wird dem anderen Knoten als Hinzufügung übermittelt. Daraus kann eine LUID-LUID Umsetzungstabelle erstellt werden und auf eine eindeutige ID im Gesamtsystem kann verzichtet werden.

Beispiel: *Datensatz 56 hat sich in Knoten A geändert. A übermittelt diese Änderung an B. B ruft seine Umsetzungstabelle auf und findet die ID 56 des Knotens A lokal als Datensatz mit der ID 823.*

Angenommen es wird beim jedem Abgleich eine Slow Sync eingesetzt, dann kann die Netzwerktopologie Zyklen enthalten.

Diese Technik ist vergleichsweise langsam, da alle Datensätze übertragen und verglichen werden müssen. Die Lösung dafür ist eine *partial Slow Sync*. Diese erfolgt nur dann, falls ein Datensatz noch keinen Eintrag in der LUID Umsetzungstabelle besitzt. Dies hat zur Folge, dass nur ein vermeidlich neuer Datensatz mit allen Datensätzen des Servers verglichen werden muss. Auf allen anderen Datensätzen kann der schnellere Abgleich über die Log Datei und den Change Counter erfolgen. Existiert bei der partial Slow Sync ein Datensatz mit gleichem Inhalt, wird angenommen, dass es sich um denselben Datensatz handelt und eine LUID Beziehung wird aufgebaut. Dies vermeidet ein doppeltes Erscheinen des Datensatzes. Alle Änderungen an diesem Datensatz werden bei zukünftigen Datenabgleichen korrekt erkannt und übernommen.

Obwohl diese Lösung nicht in der Spezifikation zu finden ist, wird sie von vielen Implementierungen bereits angewendet. Ein Test mit Apples Programm für Datenabgleiche

iSync [App03b] und dessen IrMC Level 4 Modus haben dies bestätigt. Auch in IrMC Level 4 ist diese partial Slow Sync denkbar. Die Java Implementierung eines SyncML Servers *Sync4j* [CFF05, Abschnitt 2.2] merkt diese Funktionalität wie folgt an:

> If for example the client sends a new item, that has never been mapped, this item will be in Am [Modified items belonging to source A], but not in A [Items belonging to source A (as known via LUID-GUID mapping)]. In order to be sure that the new item is not equal to some existing item in B [Items belonging to source B], it must be looked up in B. If an item in B represents the same item as in Am, A is virtually augment of such item, so that at the end, Am will be a sub-set of A.

Ein weiteres Problem besteht allerdings darin zu erkennen, wann zwei Datensätze wirklich dieselben sind.

Beispiel: *A erstellt einen neuen Datensatz und leitet diesen an die Kopie B und C weiter. B ändert den Datensatz und leitet diesen an die Kopie C weiter. Da der Datensatz auf jeder Kopie nur eine LUID besitzt und C noch keine LUID-LUID Beziehung zu B aufbauen konnte, ist der modifizierte Datensatz für C neu. Da dieser geändert worden ist, stimmt der Datensatz mit der ursprünglichen Version von A nicht mehr überein. Es entsteht weiterhin ein Duplikat.*

Dieser Ansatz erlaubt eine vollständige Konformität zu den IrMC und SyncML Spezifikationen und funktioniert auch, wenn andere Kopien selbst keine Unterstützung für Zyklen bieten. Es ist daher empfehlenswert, dass jede IrMC und SyncML Implementierung die Funktion nutzt, auch wenn nur ein Teil an Duplikaten vermieden werden kann.

9.3. Auswahl eines anderen Verfahrens

Da eine Slow Sync ohne global eindeutige IDs nicht in allen Duplikate vermeiden kann, muss ein anderes Verfahren gefunden werden, um SyncML um Zyklen zu erweitern. Die Verwendung von Vektorzeiten bietet sich an, allerdings müssten diese anders in SyncML integriert werden, als es [HMPT03] vorschlägt. Im Abschnitt 4.9.2.3 zur Technik der Änderungserkennung sind drei Probleme von Vektorzeiten genannt worden. Vektorzeiten benötigen vergleichsweise viel Speicherplatz, da für jeden Datensatz Metadaten bezüglich jeder Kopie gespeichert werden müssen. In derzeitigen SyncML müssen nur geänderte Datensätze und wenige Metadaten übertragen werden. Alle Metadaten von Vektorzeiten

müssten übertragen werden, um entscheiden zu können, welche Datensätze geändert worden sind. Vektorzeiten brauchen daher länger für einen Abgleich. Sind die Vektoren inkompatibel müssen die tatsächlichen Datensätze übertragen werden, auch wenn diese vielleicht gar nicht ungleich sind. Wird ein Datensatz gelöscht verbleibt ein Löschvermerk im System und beansprucht zusätzlichen Speicherplatz. Ein weiteres Problem stellt die Auflösung von Konflikten dar. Wird in einer Kopie ein Konflikt vom Benutzer aufgelöst, dann haben anderen Kopien davon keine Kenntnis, und es erstehen erneut Konflikte beim Abgleich. Konflikte setzten sich durch das System durch, anstatt dass sich die Konfliktlösung im System durchsetzt und der Benutzer nur einmal den Konflikt auflöst.

Reine Vektorzeiten lösen zwar das Problem der Zyklenunterstützung, bringen aber diese Probleme mit sich. Speicherplatz wird in naher Zukunft billiger und dadurch können die Speicher in mobilen Geräten größer werden. Bereits heute ist es dem Benutzer möglich, Mobiltelefone mit Gigabyte Speichermodulen aufzurüsten. Die Netzwerkverbindungen werden immer schneller und Netzbetreiber suchen nach Datendiensten, die aktuelle Bandbreiten sinnvoll ausnutzen können. Wirklich problematisch ist die mangelhafte Konfliktresolution innerhalb von Vektorzeiten. Ein Benutzer wird ein System nicht akzeptieren, falls er sich wiederholt für eine Datensatzkopie entscheiden muss.

Es ist daher erstrebenswert nach eine Lösung zu suchen, welche besser ist als Vektorzeiten oder diese stark verbessert. Einige der vorgestellten Projekte erfüllen diese Anforderung.

Roma (Abschnitt 5.5) stellt ein neues Konzept für die optimistische Datenreplikation dar und geht davon aus, dass der Benutzer immer ein Gerät mich sich führt. Dieses Konzept ist für zukünftige Entwicklungen sehr interessant und viel versprechend. Es ist allerdings zum SyncML Konzept sehr verschieden. SyncML könnte zwar für Roma sowohl für den Metaserver als auch für den Abgleich zwischen den Datenbeständen angepasst werden, aktuell existieren aber noch keine Geräte, die alle Anforderungen an einen Metaserver erfüllen. Idealerweise besitzt ein solcher Metaserver verschiedene Kommunikationstechnologien, einen großen Datenspeicher, ist tragbar, leicht und durch Anwendungen erweiterbar. Solche Geräte entwickeln sich zurzeit erst. Roma sieht den Metaserver als den Abgleich diktierende Komponente. In einem Netzwerk mit nur gleichgestellten Kopien, harmoniert dieses Konzept nicht. Trotzdem ist Roma für die optimistische Datenreplikation mit nur einem Benutzer eine sehr interessante Alternative.

Footloose (Abschnitt 5.3) ist dagegen SyncML sehr viel ähnlicher. Footloose löst bezüglich der Löschvermerke die Speicherplatzproblematik von Vektorzeitpaaren durch ein Zwei-Phasen-Löschprotokoll, welches alle Kopien durchlaufen müssen. Footloose benötigt dafür allerdings sehr viele Abgleiche und falls eine Kopie ausgefallen ist, kann der

Löschvermerk nicht verworfen werden.

Tra (Abschnitt 5.6) eignet sich dagegen sehr für den Einsatz in SyncML. Es bearbeitet sehr detailliert die Speicherplatz- und Zeitdauerproblematiken von Vektorzeitpaaren. Ein weiterer Vorteil von Vektorzeitpaaren ist die Möglichkeit partielle Abgleiche zu ermöglichen. Dadurch könnte SyncML erweitert werden, so dass ein Abbruch während eines Abgleich keine Auswirkungen auf die Konsistenz des System hätten. In SyncML werden die bereits erhaltenen Daten bei einem Abbruch verworfen. Vektorzeitpaare könnten in vielen Fällen die bereits erhaltenen Daten direkt speichern ohne die Konsistenz des Gesamtsystems zu gefährden. Zwischenspeicher könnten eingespart werden und Abgleiche könnten automatisch starten, ohne die Funktionalität des Geräts während dem Abgleich einzuschränken.[1] Auch wenn die Geräte nur kurze Zeit miteinander kommunizieren, können bei einem automatischen Abgleich bereits einzelne Daten abgeglichen werden. Automatische Abgleiche, selbst wenn diese nur partiell geschehen, können bereits im Vorfeld viele Konflikte vermeiden.

Beispiel: *Herr Maier betritt sein Büro um die aktuelle Briefpost abzuholen, danach muss er gleich wieder zu einer Sitzung mit dem Chef. Während dem kurzen Besuch in seinem Büro konnte der PC zum Mobiltelefon eine Verbindung aufbauen und die neusten Kalenderdaten übertragen. Der Chef von Herrn Maier möchte die nächste Sitzung am Freitag einrichten. Herr Maier sitzt bei seinem Chef und sieht, dass er bereits für diesen Zeitraum heute Morgen einen neuen Termin erhalten hat und bittet um einen anderen Termin.*

Die Fähigkeit von Vektorzeitpaaren, dass einmal gefällte Konfliktlösungen im System dominieren, so lange kein neuer Konflikt auftritt, macht Vektorzeitpaaren zu einem idealen Kandidaten.

Aufgrund der Recherche von verschiedenen Abgleichverfahren, sind Vektorzeitpaare ausgewählt worden, um SyncML für Zyklen in der Netzwerktopologie zu erweitern. Die folgenden Abschnitte beschäftigen sich mit der Anpassung von Vektorzeitpaaren an die SyncML Umgebung. Tra als Abgleichverfahren eines reinen Dateisystems muss für alle Arten von Datensätzen angepasst, die Vektorzeitpaare von Tra müssen in XML Elemente verpackt und ein Nachrichtenablauf muss gefunden werden, der mit ähnlich wenigen Nachrichten wie OMA SyncML DS auskommt.

[1] Viele IrMC Level 4 und OMA SyncML DS Implementierungen beschränken die Benutzerschnittstelle zu den Datenbanken, damit keine Änderungen während einem Abgleich vorgenommen werden können.

Teil III.

Implementierung

10. Herangehensweise

Im Folgenden wird die Umsetzung von Vektorzeitpaaren innerhalb von SyncML geschildert. Es entsteht ein neues Protokoll mit dem Namen *Enkel (SyncML) DS*[1], welches die XML Elemente und deren Einschränkungen aus der SyncML Common Spezifikation [OMA03b] nutzt, um mit bestehenden SyncML Systemen kompatibel zu bleiben. Ein eigner Nachrichtenablauf wird verwendet, der OMA (SyncML) DS ersetzt. Ebenfalls wird eine Strategie entwickelt Kompatibilität mit OMA DS Geräten zu gewährleisten, so dass ein Enkel DS Server sowohl OMA DS als auch Enkel DS Klienten bedienen kann.

Im ersten Abschnitt erfolgt eine Anpassung von Vektorzeitpaaren, so dass diese in den Kontext von SyncML aufgenommen werden können. Da Tra für den Abgleich von einem Dateisystem dient, wird es entsprechend erweitert, um alle Formen von Datensätzen abgleichen zu können. Tra besitzt keinen eigenen Nachrichtenablauf, d.h. für jede Anfrage von Daten wird eine Nachricht über *Remote Procedure Calls* (RPC) gesendet. Dies hat zur Folge, dass Tra sehr viele Nachrichten austauschen muss und dadurch bereits auf einem PC Performanzverluste verzeichnet werden [CJ04, Abschnitt 5.3]. Einer der Entwicklungsgründe von OMA DS war gerade der Versand von wenigen Nachrichten. Ein Nachrichtenablauf ist zu entwerfen, der es erlaubt genau wie OMA DS mit so wenigen Nachrichten wie möglich, Datenbanken von Geräte abzugleichen. Danach wird festgelegt wie die Vektorzeitpaare durch SyncML XML Elemente repräsentiert werden und welche SyncML Befehle welche Rolle in Enkel DS spielen.

Nach diesen theoretischen Überlegungen werden bestehende SyncML Implementierungen untersucht, welche sich am Besten für eine Anpassung an Enkel DS eignen. Aufbauend auf der gewählten Implementierung wird dann Enkel DS umgesetzt und dessen Funktionalität durch verschiedene Testfälle überprüft. Am Ende findet eine Bewertung statt, in wie weit Enkel DS bzw. Vektorzeitpaare die Anforderungen an ein Abgleichverfahren erfüllen.

[1] Der Name Enkel DS soll einen Gegensatz zu OMA DS verdeutlichen, da im Deutschen *Oma* für Großmutter steht und ein *Enkel* eine Familienbeziehung zu dieser besitzt. Die Wahl fiel auf die Familienbeziehung Enkel, da Enkel DS jung und dynamisch aber auch ein wenig vorlaut dem OMA DS Protokoll entgegen steht.

11. Anpassung an SyncML

Im ersten Schritt müssen die Tra Konzepte für Dateisysteme für allgemeine Datensätze und Datenbanken angepasst werden. Der zweite Schritt legt einen Nachrichtenablauf fest, der es ermöglicht, mit der gleichen Anzahl an Nachrichten wie OMA SyncML DS auszukommen. Im dritten Schritt wird bestimmt wie Vektorzeitpaare innerhalb von XML Elementen verschickt werden.

11.1. Eindeutige Erkennung von Daten

In der Beschreibung der Vektorzeitpaare [CJ05] wird nicht ausdrücklich darauf eingegangen, wie Kopien, Datenbanken und Datensätze eindeutig erkannt werden. Da ein Abgleich für Dateisysteme aufgezeigt wird, ist ein Datensatz auf Kopie A derselbe Datensatz auf Kopie B, falls die Pfadangabe dieselbe ist. In der Datensynchronisation z.B. bei Tupel basierten Systemen (IrMC und SyncML) und klassischen Datenbanksystemen gibt es mitunter keine eindeutigen Pfadangaben im Gesamtsystem. Um die Vektoren von Datensätzen zuordnen zu können, müssen alle Datensätze in der angestrebten Implementierung eindeutige Identifikationen besitzen.

Universal Unique Identifier (UUID) [Ope97] bieten sich in diesem Zusammenhang an. Bei der Erzeugung einer UUID werden eine sechs Zeichen lange Adresse mit dem aktuellen pysikalischen Zeitstempel (10 Oktetts) abgespeichert. Entsprechend wären pro Datensatz zusätzlich 16 Byte anzunehmen. Auch die Bestimmung der Zeit gestaltet sich problematisch. Andere Lösungen müssten gefunden werden.

In SyncML hat bereits jede Datenbank einen Change Counter. Es handelt sich um einen positiven Integer, der bei jeder Änderung um eins erhöht wird. Dies stellt eine logische Uhr dar. Innerhalb der Vektorzeitpaare kann der Wert dieser Uhr bei einer Modifikation in den Modifikationsvektor $\vec{m}$ des jeweiligen Datensatzes eingetragen werden. Zu Beginn einer Synchronisation kann der Wert dieser Uhr in den Synchronisationsvektor $\vec{s}$ der Datenbank eingetragen werden. Beim Anlegen eines Datensatzes kann der Wert dieser Uhr in den Anlegezeitstempel c eingetragen werden. Bei diesem Anlegezeitstempel handelt es sich dann um ein Wertepaar aus der Kopie, die den Datensatz angelegt hat,

und dem logischen Zeitstempel, wann der Datensatz lokal in dieser Kopie erzeugt worden ist. Wenn man davon ausgeht, dass die Kopie eindeutig im System erkannt werden kann, und dass bei jeder Änderung (Hinzufügung, Ersetzung oder Löschung) innerhalb einer Kopie der Zähler erhöht wird, ist das Wertepaar aus der Kopie und dessen lokalen Zeit Systemweit eindeutig.

Beispiel: *IMEI:54746288917:67*
Der Datensatz ist auf einem Mobiltelefon mit der Seriennummer 54746288917 *zum Zeitpunkt* 67 *angelegt worden.*

Auf diese Weise kann c als eine abgewandelte UUID angesehen werden, die nicht mehr auf UTC angewiesen ist. Da c zusammen mit einem Datensatz von einer Kopie zur nächsten wandert und nicht mehr geändert wird, bis der Datensatz gelöscht wird, bleibt dieser Anlegezeitstempel eindeutig.

Der vorgestellte Algorithmus zu Vektorzeitpaaren schlägt vor, dass gelöschte Datensätze nur einen Synchronisationsvektor $\vec{s}$ besitzen. Dieser fällt weg, wenn das übergeordnete Verzeichnis komplett abgeglichen worden ist, wobei das Delta zwischen dem Datensatz und dem Verzeichnis Null wird. Der $\vec{s}$ ist leer und kann gelöscht werden. Wenn sich ein solcher Löschvermerk sich in einem Verzeichnis befindet, dann besteht das Problem, dass c fehlt und der Datensatz nicht mehr eindeutig gefunden werden kann. Daher wird in Enkel DS, c erst gelöscht, wenn auch der $\vec{s}$ gelöscht werden kann. Da einem Löschvermerk immer der $\vec{m}$ fehlt, kann über dessen Abwesenheit ermittelt werden, dass es sich tatsächlich um einen Löschvermerk handelt und es muss nicht der Inhalt jedes Datensatzes betrachtet werden.

Folglich kann ein c als Global Unique Identifier (GUID) genutzt werden. Dieser Zeitstempel ist sowohl lokal als auch systemweit für die jeweilige Datenbank eindeutig.

11.2. Geplanter Nachrichtenverlauf

[CJ04, Abschnitt 5.3] gibt bei der Beschreibung der Vektorzeitpaare nicht an, in welcher Reihenfolge zwei Knoten Daten miteinander austauschen:

> The current implementation of Tra does not perform aggressive batching of writes: many threads are executing RPCs in parallel, but the small RPC packets (tens of bytes each) are being written one at a time via the write system call. We believe that batching the many outgoing RPCs into a small number of larger write calls will help this.

Es scheint, dass Tra für jeden Datenaustausch einen eigenen Thread anlegt und den Remote Procedure Call (RPC) ausführt. Dies ist die Ursache für sehr viele kleine Nachrichten.

In OMA DS hat man sich absichtlich für sehr wenige Nachrichten entschieden. Bereits vier bis sieben Nachrichten reichen für alle sieben Synchronisationstypen aus. Diese Nachrichten können allerdings in mehrere Packte aufteilt werden, damit auch Geräte mit kleinen Zwischenspeichern die womöglich sehr großen Nachrichten empfangen und verarbeiten können. Um dieser Grundphilosophie von OMA DS zu folgen, sollten daher in Enkel DS so wenige Nachrichten ausgetauscht werden wie möglich.

Die auszutauschenden Datenmengen zwischen zwei Knoten sind ebenfalls gering zu halten. Alle (Meta-) Daten sollten maximal einmal über das Netzwerk verschickt werden und es sollten keine unnötigen Anfragen an den Kommunikationspartner geschickt werden. Die Ergebnisse von Berechnungen sollen lokal gehalten werden und nicht noch zusätzlich über das Netzwerk versendet werden.

Beim Entwurf verschiedener Nachrichtenabläufe ist die Komplexität des Vektorzeitpaar Algorithmus deutlich geworden. Die Metadaten der beiden Kopien müssen in vielen Situationen der anderen Kopie bekannt sein.

Die Ersetzungen von Datensätzen festzustellen ist relativ einfach. Ein Knoten benötigt das Wissen über alle $\vec{s}$ und $\vec{m}$ der jeweiligen Datensätze der anderen Kopie. Um $\vec{s}$ richtig zu berechnen, ist allerdings $\vec{s}$ des übergeordneten Datensatzes (bei hierarchischen Systemen) und der Datenbank selbst nötig, da diese die Grundlage für das Delta beim selbst Datensatz bilden.

Beispiel: *Es existieren die Kopien A, B und C, welche alle eine Adressbuch Datenbank besitzen. Diese Datenbank besitzt ein Unterverzeichnis für private Einträge. In diesem Unterverzeichnis sind die Kontaktdaten z.B. die Telefonnummer der Eltern gespeichert. Datenbank: Adressbuch Verzeichnis (z.B. private Einträge) Datensatz: Eltern A: 3 1 0 B: 78 0 0 C: 2 7 0 Um den Datensatz der Eltern zu übertragen, könnte man alle drei Vektoren inklusive der übergeordneten Verzeichnisse einzeln oder nur einen Vektor übertragen: A: 4 (3+1+0) B: 78 (78+0+0) C: 9 (2+7+0)*

Auf den ersten Blick wäre es nicht sinnvoll Delta Vektoren zu übertragen sondern nur den kombinierten Vektor. Um Zeit beim Abgleich zu sparen, nutzte der Vektorzeitpaar Algorithmus bereits die übergeordneten Vektoren, um zu ermitteln, ob überhaupt eine Synchronisation mit dem Inhalt des übergeordneten Verzeichnisses nötig war. Dies bedeutet, dass alle übergeordneten $\vec{s}$ bereits der anderen Kopie vorliegen. Der Algorithmus über Vektorzeitpaare geht auch davon aus, dass Vektoreinträge mit einer Null

verworfen werden. In diesem Beispiel wäre $\vec{s}$ des Datensatzes der Eltern die leere Menge: $\{A3, B78, C2\}, \{A1, C7\}, \{\emptyset\}$. Da partielle Abgleiche nicht die Regel sondern die Ausnahme sind, kann man davon ausgehen, dass eine große Anzahl leerer $\vec{s}$ existieren und nur der $\vec{s}$ der Datenbank tatsächlich Vektordaten enthält. Entsprechend wird beim Austausch der $\vec{s}$ in Enkel DS nur das gespeicherte Delta $\vec{s}$ übertragen.

Nach mehreren Entwürfen entstand der in Abbildung 11.1 dargestellte Nachrichtenablauf von Enkel DS. Der Knoten der den Abgleich startet, ist der Klient. Der Klient fragt einen anderen Knoten den Server an. Der Server ist in den Tra Codefragmenten vom Abschnitt 5.6 der Knoten mit der Beizeichnung A und der Klient hatte die Bezeichnung B.

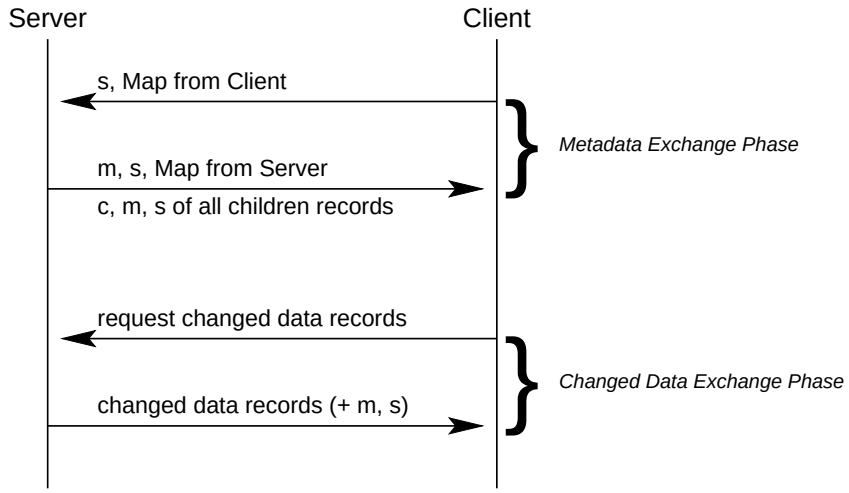

Abbildung 11.1.: Enkel DS Nachrichtenablauf

Der Klient sendet den $\vec{s}$ seines Wurzelverzeichnisses (in der Regel der Datenbanken) an den Server. Der Server kann daraus ermitteln, ob der Klient aktuell ist oder ob der Server neue Änderungen enthält.

```
if (m_Server/ ≤ s_Client/)
   do nothing
else
   send all children
```

Ist der Klient aktuell wird nur der $\vec{m}$ der Wurzel des Servers zurückgesendet. Der Klient muss diesen Vektor kennen, um entscheiden zu können, ob alle Datensätze in Server gelöscht worden sind oder ob der Klient die aktuellsten Daten besitzt.

```
if (m_Server/ ≤ s_Client/)
   do nothing
elseif (‚no children received' == true)
```

```
   delete all children
else
   sync each child
```

Hat der Server ermittelt, dass neue Änderungen für den Klient vorliegen, werden alle Vektoren aller direkten Kinder der Wurzel in einer Nachricht übermittelt. Der Zeitstempel c_{Server} muss für jeden Datensatz übermittelt werden, da sonst der Klient diesen Datensatz seiner lokalen Kopie nicht eindeutig zuweisen könnte. Damit der Klient die Entscheidungen über Ersetzungen und Hinzufügungen treffen kann, werden die $\vec{m}_{Server}$ aller Datensätze benötigt.

```
if (m⃗_Server ≤ s⃗_Client)
   do nothing
else // update or conflict
   request data in next package
```

Ebenfalls für neue Server Datensätze (Hinzufügungen) sind die $\vec{m}_{Server}$ nötig. Der folgende Code wird durchlaufen, falls der Klient keinen entsprechenden Datensatz mit c besitzt. Für $\vec{s}Client$ wird das übergeordnete Verzeichnis herangezogen.

```
if (m⃗_Server ≤ s⃗_Client)
   do nothing
else // new or conflict
   request data in next package
```

Um Löschungen zu erkennen, sind die $\vec{s}_{Server}$ der Server Datensätze nötig.

```
if (m⃗_Client ≤ s⃗_Server)
   delete File_B
else if (c_Client ≰ s⃗_Server)
   do nothing
else
   report a conflict
```

Aus diesen Gründen werden alle drei Vektoren übermittelt.[1]

Die Vektoren aller direkten Kinder des Server Wurzelverzeichnisses sind nötig, damit der Klient entscheiden kann, ob die Datei gelöscht worden ist oder ob der Server die Datei noch gar nicht kennt.[2]

Hat der Klient alle Datensätze innerhalb dieses Verzeichnisses verglichen, fordert er durch eine Nachricht die Inhalte der Datensätze des Servers an, die der Klient noch nicht

[1] Spätere Betrachtungen haben gezeigt, dass es ausreichend c und $\vec{m}_{Server}$ zu übermitteln. Nur wenn es sich bei dem Datensatz um einen Löschvermerk des Servers (aus einem partiellem Abgleich) handelt, muss der $\vec{s}_{Server}$ vorliegen. Da der $\vec{s}$ in der Regel leer ist, ist dieser Fehler vernachlässigbar.

[2] Gelöschte Datensätze werden dem Klienten in der Regel nur dann explizit durch Löschvermerke angezeigt, wenn der Server vorher (als Klient) einen partiellen Abgleich durchlaufen hat und der $\vec{s}$ des übergeordneten Verzeichnisses noch nicht aktualisiert werden konnte.

kennt oder der Server durch neue Versionen ersetzt hat. Der Server sendet dann diese Datensätze in einer Nachricht mit den aktuellen c, $\vec{m}$ und $\vec{s}$ zurück. [3] Der Klient nutzt nun alle drei Vektoren, um Hinzufügungen, Ersetzungen und Konflikte voneinander zu unterscheiden. Die Konflikte werden dem Benutzer gemeldet, der dann aus den vorliegenden Versionen des Servers und des Klienten entscheiden kann, ob eine der beiden Versionen gewinnt oder ob eine Mischung aus beiden Datensätzen angebracht ist.

Der Nachrichtenablauf in 11.1 erfolgt nur in eine Richtung, d.h. die Inhalte der Datensätze wandern nur vom Server zum Klienten. Es handelt sich nicht um einen Zwei-Wege Abgleich. Damit beide Kopien auf den neusten Stand kommen, wird der Nachrichtenablauf umgekehrt, d.h. ein Klient muss auch die Serverrolle übernehmen können und ein Server die Klienten Rolle. Nachdem der Abgleich erfolgreich war, wird aus dem Server A ein Klient, welcher dann den ehemaligen Klient B als Server nach den neusten Änderungen befragt.[4]

Ein Nachrichtenablauf für einen direkten Zwei-Wege Abgleich ist daher nicht nötig und würde nur unnötig kompliziert werden.[5] Abbildung 11.1 zeigt nur eine Verzeichnisebene. Eine Verzeichnisebene ist bei flachen Datenbanken wie Kalender und Adressbüchern in SyncML und IrMC üblich. In solchen Datenbanken gibt es keine weiteren Verzeichnisebenen. Trotzdem lässt sich der obige Nachrichtenablauf leicht um weitere Verzeichnisebenen erweitern (Abbildung 11.2). In der dritten Nachricht wird für alle Verzeichnisse innerhalb des Wurzelverzeichnisses der $\vec{s}_{Client}$ übertragen. Der Server entscheidet dann genau wie bei der Wurzel, ob das Verzeichnis untersucht werden muss, und sendet zusätzlich zu den angeforderten Daten in der vierten Nachricht den $\vec{m}_{Server}$ der Verzeichnisse und wenn Änderungen vorliegen auch deren direkte Kinder.

Bei mehreren Verzeichnisebenen kann die Implementierung entscheiden, ob jedes Verzeichnis einzeln übermittelt wird und ein Tiefendurchlauf erfolgt. Bei einem Tiefendurchlauf wird ein Verzeichnisast solange abgeglichen, bis es keine weiteren Verzeichnisse enthält, erst dann wird das nächste Verzeichnis auf der Ebene abgeglichen. In diesem Fall

[3] Die aktuellen $\vec{m}$ und $\vec{s}$ müssen hier erneut gesendet werden, da der Server potentiell nicht nur als Server fungiert, sondern selbst auch wieder Klient sein kann. Gleicht der Server seine Datenbank gerade mit einer anderen Datenbank ab, dann könnten neue Versionen der Datensätze während der zweiten und vierten Nachricht den Server erreichen. Erlaubt man dem Server nicht gleichzeitig Server und Klient zu sein, kann auf die wiederholte Übertragung von $\vec{m}$ und $\vec{s}$ verzichtet werden.

[4] Nur wenn ein Server zur gleichen Zeit Klient sein kann, dann ist es nötig, dass die $\vec{m}$ und $\vec{s}$ in der vierten Nachricht mit jedem Datensatz erneut übermittelt werden.

[5] Knoten die sich in Netzwerken mit Network Address Translation (NAT) [HS01] befinden, können nicht oder nur schwer von anderen Knoten außerhalb dieses Netzwerks als Server angesprochen werden. Netzwerke von Mobilfunkprovidern arbeiten in der Regel mit NAT, damit die Mobiltelefone nicht unaufgefordert mit Datenanfragen vom Internet überschwemmt werden können. Es ist daher in zukünftigen Untersuchungen zu überlegen, ob nicht doch ein Zwei-Wege Nachrichtenablauf nötig ist.

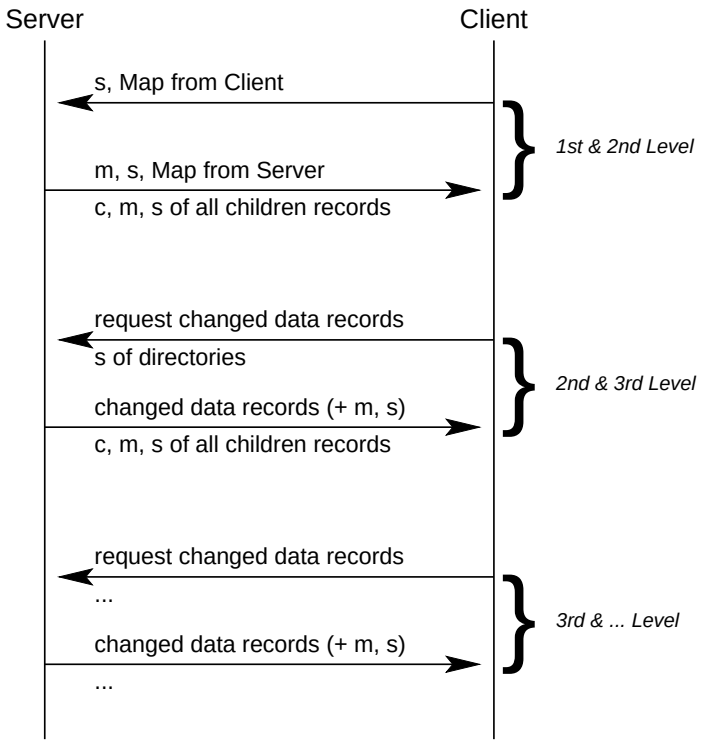

Abbildung 11.2.: Enkel DS – mehr als eine Verzeichnisebene

werden für jedes abzugleichende Verzeichnis zwei weitere Nachrichten benötig. Alternativ kann ein Breitendurchlauf erfolgen, bei dem alle Verzeichnisse einer Ebene übermittelt werden. Dadurch sind zwei weitere Nachrichten nur für jede Ebene nötig. Die Nachrichten sind allerdings größer. Da SyncML immer gleich große Pakete aus den Nachrichten erstellen kann, ist ein Breitendurchlauf sinnvoller.

Der Nachrichtenablauf für eine Ebene begnügt sich mit der gleichen Anzahl an Nachrichten, mit denen OMA DS auskommt. OMA DS benötigt am Anfang zwei zusätzliche Nachrichten für die Authentisierung des Benutzer und ob dieser Zugriff auf die Datenbank erhalten darf (Initialisierungsphase). In Enkel DS sind diese beiden Nachrichten ebenfalls vorgesehen. Enkel DS wird die Initialisierungsphase von OMA DS übernehmen und die Nachrichten als Enkel DS Nachrichten markieren (Abbildung 11.3). Dadurch kann ein Enkel DS Server sowohl OMA DS als auch Enkel DS Klienten bedienen und die Sicherheitskonzepte von einem bestehenden OMA DS Server wiederverwenden. Erkennt der Enkel DS Server, dass die Nachricht von einem Enkel DS Klienten stammt, wird entsprechend geantwortet. Der Enkel DS Klient kann sich dann entscheiden ob er nur nach OMA DS verfährt oder wahrscheinlicher den obigen Enkel DS Nachrichtenablauf startet.

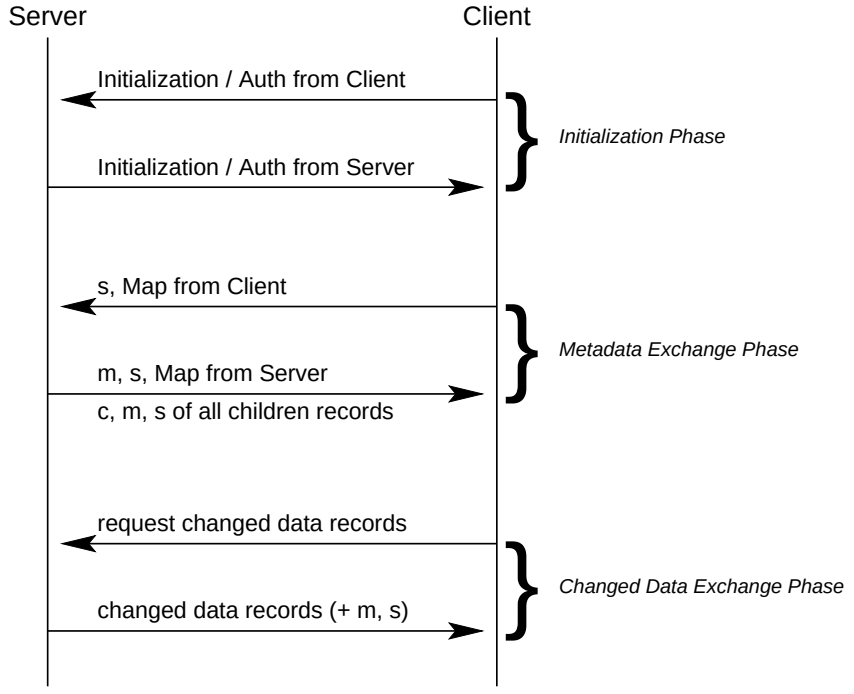

Abbildung 11.3.: Enkel DS mit Initialisierungsphase von OMA DS

Ein Benutzer startet einen Abgleich immer vom Klienten Knoten aus. Da am Klienten Konflikte gemeldet und auch dort entschieden werden, müssen keine Datensätze unnötigerweise kommuniziert werden, weil der Benutzer gerade mit dem Klienten Knoten arbeitet. Alle Konflikte werden lokal aufgelöst.

Der aufgezeigte Nachrichtenablauf hat den weiteren Vorteil, dass keine Daten vom Klienten an den Server übermittelt werden. Private Datensätze werden dem Server nicht bekannt gemacht. Dies erlaubt Architekturen in denen es öffentliche Datenserver gibt, in denen der Klient nur Daten abholt, selbst aber keine übermittelt. Private nicht öffentliche Datensätze müssen nicht einmal als Metadaten dem Server bekannt gemacht werden. Nur der Synchronisationsvektoren der Wurzel wird übertragen. Für höchsten Datenschutz kann ein leerer Vektor übertragen werden, der einen vollständigen Abgleich startet.

Der Nachrichtenablauf zeigt im den ersten beiden Nachrichten noch ein *Map* Objekt. Hierbei handelt es sich um eine Tabelle, die global eindeutige IDs von Knoten (GUID) in lokal eindeutige (LUID) umwandelt. Intern in einem Knoten wird jede GUID nur einmal abgespeichert, da eine GUID sehr lang werden kann. Für jede GUID wird eine LUID erzeugt, die innerhalb der Vektoren abgespeichert wird. Auf diese Weise kann weiterer Speicherplatz gespart werden. Die Umsetzungstabelle bietet sich nicht nur zum Sparen von Speicherplatz an, sondern kann auch während dem Abgleich als *Map* genutzt werden. So können die Zeitstempel mit nur kurzen LUIDs gesendet werden.[6]

11.3. Anpassung der SyncML XML Elemente

Der Nachrichtenablauf hat die nötigen Daten aufgezeigt, die übermittelt werden müssen. In OMA DS werden die Befehle `Anchor`, `Sync`, (`Add`,) `Replace`, `Delete` und `Map` verwendet. Die Definition dieser Elemente erlaubt meist keine Wiederverwendung in Enkel DS. Einzig der `Map` Befehl kann übernommen werden.

Document Type Definition von `Map`:

```
CmdID, Target, Source, Cred?, Meta?, MapItem+
```

Ein `Map` Befehl kann mehrere `MapItem` Elemente enthalten. Die Elemente `Target` und `Source` geben dabei die Datenbank der Quelle und des Zielknotens an. Die `MapItem`

[6] Bei einer Kompression der XML Packete oder bei Verwendung von WBXML, ist die Einsparung minimal. In der Regel werden aber XML Packte in SyncML nicht komprimiert. Es ist auch unüblich die dynamische Wörterbuch Funktionalität von WBXML zu nutzen, da diese einen zweiten Durchlauf des WBXML Generators erfordert. Daher macht es Sinn diese Datenreduktion bereits auf der Ebene des Enkel DS Protokolls vorzunehmen.

Elemente enthalten jeweils ein `Target` und ein `Source`. `Source` gibt die LUID an und `Target` die GUID jedes Datensatzes der Umsetzungstabelle des Knotens. Map besitzt kein optionales `NoResp`. Ist dieses Element vorhanden, müsste kein `Status` als Antwort zurückgesendet werden. `Map` benötigt in OMA DS aber auch in Enkel DS immer eine Antwort, da sonst der Abgleich nicht erfolgreich fortgesetzt werden kann, daher ist das Fehlen von `NoResp` gewollt. Cred erlaubt eine optionale Authentifizierung auf die Umsetzungstabelle dieser Datenbank. Dieses Feld können sicherheitssensitive Anwendungen nutzen. Es ist daher unabhängig von OMA DS oder Enkel DS und des Nutzung hängt von den Sicherheitsregeln des Knotens ab. Das `Meta` Element von `Map` wird in Enkel DS nicht benötigt.

Da jede Übermittlung des Klienten eine direkte Antwort des Servers erwartet, wird der Befehl `Get` genutzt. Ein `Get` besitzt immer eine Antwort in durch mindestens ein `Results`. `Add`, `Replace` und `Delete` hätten sich nicht für Enkel DS geeignet, da diese Befehle keine Antwort sind oder verlangen. Der `Get` Befehl ist ähnlich wie der Get Befehl in HTTP oder OBEX ein sehr abstrakter Befehl von SyncML und es bedarf genauerer Erklärungen, wie dieser in Enkel DS verwendet wird.

Document Type Definition von `Get`:

```
CmdID, NoResp?, Lang?, Cred?, Meta?, Item+
```

In der ersten Nachricht wird der $\vec{s}_{Client}$ mittels eines `Get` Befehls übermittelt und der entsprechende $\vec{m}_{Server}$ angefordert. Dabei wird das `Item` Element im `Get` entsprechend befüllt.

Document Type Definition von `Item`:

```
Target?, Source?, Meta?, Data?
```

Das `Meta` eines `Item` enthält den Enkel DS spezifischen MIME Media Type [FB96] *application/x-EnkelDS-VectorTimePair*, der angibt, dass in `Data` ein Vektorzeitpaar dem Enkel DS Format entsprechend enthalten ist. Wenn die Umsetzungstabelle erfolgreich ausgetauscht worden sind, dann können folgende c eines Datensatzes eine Knoten ID enthalten, die nicht mehr Systemweit eindeutig ist, sondern nur noch lokal zu dem Knoten.

Beispiel: *GUID = IMEI:54746288917:67 wäre LUID = B:67*

Der Empfänger ermittelt die GUID mit Hilfe der LUID und der Umsetzungstabelle.

Ist die LUID nicht vorhanden, dann ist die gesendete ID eine GUID. Ist der Datensatz lokal auf dem Knoten entstanden wird in einem `Source` Element nur der Zeitstempel ohne LUID oder GUID gesendet. Die Knoten ID ergibt sich aus dem `SyncHdr`. Der Server sendet daraufhin die entsprechenden Vektordaten aus seinem Bestand in einem `Results` zurück.

Document Type Definition von `Results`:

```
CmdID, MsgRef?, CmdRef, Meta?, TargetRef?, SourceRef?, Item+
```

Die `MsgRef` enthält den Wert der `MsgID` des Klienten, also eine Referenz auf das Paket in dem das zu beantwortende `Get` gesendet worden ist. `CmdRef` enthält den Wert der `CmdID` und damit eine Referenz auf den `Get` Befehl selbst.[7] `TargetRef` und `SourceRef` bleiben leer. Fehlt ein `Data` im `Get` und der MIME Typ entspricht nicht der Enkel DS Vektorzeitpaare, dann werden vom Server die eigentlichen Daten in einem `Results` zurück gesendet.[8] Falls der Klient ein Verzeichnis anfordert und der Server erkennt, dass der Inhalt des Verzeichnisses abgeglichen werden muss, wird als erstes `Item` des `Results` die Vektordaten des Verzeichnisses und danach die Vektordaten der Kinder übermittelt. Es ist nötig ein Verzeichnis komplett[9] in einem `Results` zurückzugeben, da $\vec{s}_{Server}$ nur als Delta übermittelt werden und die Daten der übergeordneten Verzeichnisse – dem ersten `Item` – benötigt werden.

Eine Enkel DS Implementierung müsste daher nur die Befehle `Get`, `Map` und `Results` aufweisen.[10] Die Beschränkung auf diese wenigen Befehle erleichtert die Umsetzung von Enkel DS.

11.4. Beispielhafter Nachrichtenablauf

Um die vorherigen Abschnitte zusammenzufassen wird der Enkel DS Nachrichtenablauf exemplarisch in XML dargestellt. Um die Beispiele einfach zu halten wird nicht mit `Map` Elementen gearbeitet und es wird immer das `Target` Element anstatt dem `Source` Element genommen.

[7] Diese Referenzen sind nötig, da ein `Results` nicht im darauffolgenden Paket vollständig erfüllt muss und ein Paket mehrere `Get` enthalten könnte.

[8] Für den Fall dass der Server gleichzeitig Server und Klient ist und um Race Conditions zu vermieden, werden zusätzlich die aktuellen Vektordaten des Datensatzes übermittelt.

[9] In SyncML Version 1.2 [OMA04e] ist es möglich, direkt ein übergeordnetes Verzeichnis für jedes `Item` anzugeben. Die Enkel DS Lösung spart sich aber wiederholende Verweise auf dieses Verzeichnis und ist kompatibel mit SyncML in Version 1.0.

[10] Zusätzlich ist noch `MapItem` zu unterstützen, da dies in SyncML auch als *Befehl* spezifiziert wird.

In den HTTP bzw. OBEX Kopfinformationen wird ein OMA DS Paket mit dem Media Type *application/vnd.syncml+xml* für XML oder *application/vnd.syncml+wbxml* für WBXML beschrieben. Da Enkel DS die Initialisierungsphase von OMA DS übernimmt, wird derselbe Media Type verwendet. Um Enkel DS Pakete als solche kenntlich zu machen, wird ein Parameter an den Media Type angehängt und ergibt dann für XML: *application/vnd.syncml+xml; EnkelDS=1.0*.[11] Es werden zuerst Pakete mit dem Enkel DS Media Type gesendet. Akzeptiert der Server die Pakete nicht, weil die Implementierung einen Bug enthält und Media Types mit Parametern nicht unterstützt werden, wird das Enkel DS Paket mit einem `EMI` im `SyncHdr` gesendet. Falls die Implementierung auch Probleme mit `EMI` hat und der Versand dieser Nachricht fehl schlägt, wird nach OMA DS verfahren, da anzunehmen ist, dass der Server kein Enkel DS unterstützt. Diese zusätzlichen Pakete sind nur nötig, wenn der Server Implementierungsfehler aufweist. Ein bestehender SyncML Server sollte mit dem Media Type umgehen können und den neuen Parameter ignorieren [FB96, Abschnitt 5, Absatz 5].

11.4.1. Initialisierungsphase

Anfrage des Klienten

Die erste Nachricht wird vom Klienten an den Server gesendet und startet mit einer OMA DS Autorisierung durch ein `Cred` Element im `SyncHdr`. In diesem Beispiel wird der Einfachheit halber die *Basic Authentication* [OMA03a, Kapitel 8] verwendet, die das Passwort im Klartext[12] übermittelt. Außerdem enthält die Nachricht ein `Alert` Element zum Start einer Slow Sync, dadurch findet auch dann ein Abgleich statt, wenn der Server nur das OMA DS Protokoll unterstützt.

```
<SyncML>
   <SyncHdr>
      <VerDTD>1.1</VerDTD>
      <VerProto>SyncML/1.1</VerProto>
      <SessionID>1</SessionID>
      <MsgID>1</MsgID>
      <Target>
         <LocURI>OBEX:SYNCML-SYNC/</LocURI>
      </Target>
      <Source>
         <LocURI>IMEI:54746288917</LocURI>
      </Source>
      <Cred>
         <Meta>
```

[11] Die genaue Spezifikation der Syntax dieses Enkel DS Parameters befindet sich in der Implementierung im Quellcode der Java Klasse `de.traud.EnkelDS.framework.util.Helper`.
[12] SyncML beinhaltet aber eine Reihe weiterer sicherer Authentifizierungsverfahren.

```
            <Format xmlns='syncml:metinf'>b64</Format>
            <Type xmlns='syncml:metinf'>syncml:auth-basic</Type>
         </Meta>
         <Data>QTAwMTpndWVzdA==</Data>
      </Cred>
      <Meta>
         <EMI xmlns='syncml:metinf'>EnkelDS=1.0</EMI>
      </Meta>
   </SyncHdr>
   <SyncBody>
      <Alert>
         <CmdID>1</CmdID>
         <NoResp/>
         <Data>201</Data>
         <Item>
            <Target>
               <LocURI>Contacts</LocURI>
            </Target>
            <Source>
               <LocURI>telecom/pb.vcf</LocURI>
            </Source>
            <Meta>
               <Anchor xmlns='syncml:metinf'>
                  <Last>0</Last>
                  <Next>1</Next>
               </Anchor>
            </Meta>
         </Item>
      </Alert>
      <Final/>
   </SyncBody>
</SyncML>
```

Antwort vom Server

Die Nachricht vom Server zum Klienten bestätigt die Authentifizierung des `SyncHdr` durch das erste `Status` im `SyncBody` und schickt die eigenen Change Counter in einem `Anchor` zurück. Damit der Klient entscheiden kann, ob der Server Enkel DS unterstützt, wird das EMI Element oder der Media Type entsprechend gesetzt.[13]

```
<SyncML>
   <SyncHdr>
      <VerDTD>1.1</VerDTD>
      <VerProto>SyncML/1.1</VerProto>
```

[13] Spätere Betrachtungen stellten fest, dass es Server Implementierung gibt, die das `Meta` Element und damit auch das `EMI` Element des Klienten übernehmen und damit `EMI` zurücksenden, obwohl keine Enkel DS Unterstützung vorliegt. Daher sollte in einer nächsten Ausbaustufe der Inhalt von `EMI` z.B. auf *EnkelDSServer=1.1* durch den Server geändert werden.

```
        <SessionID>1</SessionID>
        <MsgID>1</MsgID>
        <Target>
           <LocURI>IMEI:54746288917</LocURI>
        </Target>
        <Source>
           <LocURI>OBEX:SYNCML-SYNC/</LocURI>
        </Source>
        <RespURI>OBEX:SYNCML-SYNC/?sid=W0JAMWM1YzVjLTExMDc5MzU5Nzg1Nj</RespURI>
        <Meta>
           <EMI xmlns='syncml:metinf'>EnkelDS=1.0</EMI>
        </Meta>
     </SyncHdr>
     <SyncBody>
        <Status>
           <CmdID>1</CmdID>
           <MsgRef>1</MsgRef>
           <CmdRef>0</CmdRef>
           <Cmd>SyncHdr</Cmd>
           <TargetRef>OBEX:SYNCML-SYNC/</TargetRef>
           <SourceRef>IMEI:54746288917</SourceRef>
           <Data>212</Data>
        </Status>
        <Alert>
           <CmdID>2</CmdID>
           <Data>201</Data>
           <Item>
              <Target>
                 <LocURI>telecom/pb.vcf</LocURI>
              </Target>
              <Source>
                 <LocURI>Contacts</LocURI>
              </Source>
              <Meta>
                 <Anchor xmlns='syncml:metinf'>
                    <Last>1107935979362</Last>
                    <Next>1107935979362</Next>
                 </Anchor>
              </Meta>
           </Item>
        </Alert>
        <Final/>
     </SyncBody>
  </SyncML>
```

11.4.2. Metadaten Austausch Phase

Anfrage des Klienten

Die nächste Nachricht vom Klienten an den Server enthält die $\vec{s}$ der abzugleichenden Datenbanken. Der Server hatte ein Verzeichnis zu seinem Zeitpunkt `1` angelegt.

```
<SyncML>
   ...
   <SyncBody>
      <Get>
         <CmdID>1</CmdID>
         <Meta>
            <Type>
               application/x-EnkelDS-VectorTimePair
            </Type>
         </Meta>
         <Item>
            <Target>
               <LocURI>OBEX:SYNCML-SYNC/:1</LocURI>
            </Target>
            <Data>OBEX:SYNCML-SYNC/:1</Data>
         </Item>
      </Get>
      ...
   </SyncBody>
</SyncML>
```

Antwort vom Server

Der Server sendet seine $\vec{m}$ und $\vec{s}$ seiner Datenbank Kopie zurück. Da es sich um ein Verzeichnis handelt und der Server erkannt hat, dass das Verzeichnis neue Änderungen für den Klienten enthält, werden die Vektordaten der direkten Kinder des Verzeichnisses übermittelt. In diesem Fall existiert nur ein Datensatz in diesem Verzeichnis, welches zum Zeitpunkt `400` auf dem Server erstellt worden ist.

```
<SyncML>
   ...
   <SyncBody>
      <Results>
         <CmdID>1</CmdID>
         <MsgRef>2</MsgRef>
         <CmdRef>1</CmdRef>
         <Meta>
            <Type xmlns='syncml:metinf'>
               application/x-EnkelDS-VectorTimePair
            </Type>
            </Meta>
```

```
            <Item>
               <Target>
                  <LocURI>OBEX:SYNCML-SYNC/:1</LocURI>
               </Target>
               <Data>OBEX:SYNCML-SYNC/:400OBEX:SYNCML-SYNC/:400</Data>
            </Item>
            <Item>
               <Target>
                  <LocURI>OBEX:SYNCML-SYNC/:400</LocURI>
               </Target>
               <Data>OBEX:SYNCML-SYNC/:400</Data>
            </Item>
         </Results>
         <Final/>
      </SyncBody>
   </SyncML>
```

11.4.3. Übertragung der Änderungen

Anfrage des Klienten

Der Klient kann anhand der Vektordaten erkennen, dass es sich um eine neue Hinzufügung des Servers handelt und fordert entsprechend den Inhalt des Datensatzes an.

```
<SyncML>
   ...
   <SyncBody>
      <Get>
         <CmdID>1</CmdID>
         <Item>
            <Target>
               <LocURI>OBEX:SYNCML-SYNC/:400</LocURI>
            </Target>
         </Item>
      </Get>
      ...
   </SyncBody>
</SyncML>
```

Antwort vom Server

Der Server erkennt, dass der Datensatz selbst und nicht nur seine Metadaten angefragt worden ist, da der Media Type *application/x-EnkelDS-VectorTimePair* in der Anfrage des Klienten fehlte. Der Server sendet daraufhin den Inhalt des Datensatzes zurück. Da die Datenbank der Serverkopie gleichzeitig von Servern und Klienten genutzt werden kann, werden zur Sicherheit die aktuellen Vektordaten erneut mitgeschickt.

```
<SyncML>
```

```
...
<SyncBody>
   <Results>
      <CmdID>1</CmdID>
      <MsgRef>3</MsgRef>
      <CmdRef>1</CmdRef>
      <Item>
         <Target>
            <LocURI>OBEX:SYNCML-SYNC/:400</LocURI>
         </Target>
         <Meta>
            <Type xmlns='syncml:metinf'>text/x-vCard</Type>
         </Meta>
         <Data>
            BEGIN:VCARD
            VERSION:2.1
            N:Alexander Traud
            ADR;HOME:;;Briegelweg 7;Darmstadt;;64287;D
            EMAIL;INTERNET:alex@traud.de
            NOTE:Author of Enkel DS 1.0
            END:VCARD
         </Data>
      </Item>
      <Item>
         <Target>
            <LocURI>OBEX:SYNCML-SYNC/:400</LocURI>
         </Target>
         <Meta>
            <Type xmlns='syncml:metinf'>
               application/x-EnkelDS-VectorTimePair
            </Type>
         </Meta>
         <Data>OBEX:SYNCML-SYNC/:400</Data>
      </Item>
   </Results>
   <Final/>
</SyncBody>
</SyncML>
```

11.4.4. Abschluss

Der Datenabgleich ist nach dieser Nachricht abgeschlossen. Der Klient kann die neuen Daten speichern und seine Vektordaten entsprechend aktualisieren. Der Server ändert keiner seiner Vektordaten, da er als Server keine neuen Daten von einem Klient erhält.

11.5. Auswahl der Basisimplementierung

Da Enkel DS auf der SyncML Common Spezifikation [OMA03b] aufbaut und dessen XML Elemente nutzt und da der Nachrichtenablauf so gewählt werden konnte, dass er anfänglich OMA DS kompatibel ist, bietet es sich an eine bestehende OMA DS Implementierung aufzugreifen und um den Enkel DS Nachrichtenablauf zu erweitern. Im Idealfall stellt diese Basisimplementierung sowohl einen SyncML Klienten als auch einen Server.

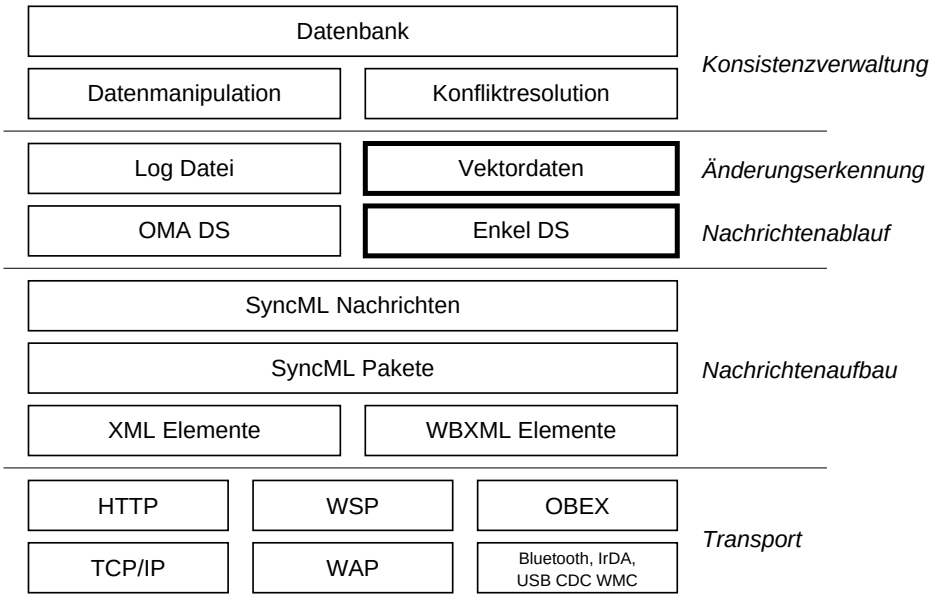

Abbildung 11.4.: Aufbau einer idealen Basisimplementierung – nur die Bestandteile *Enkel DS* und *Vektordaten* sind zu implementieren.

Eine ideale Basisimplementierung in Abbildung 11.4 macht es nicht nötig die verschiedenen Transporte erneut zu programmieren. Sowohl HTTP als auch OBEX werden unterstützt, d.h. die idelae Implementierung kommt bereits mit einem HTTP Server, HTTP Klienten, OBEX Server und einem OBEX Klienten. Der SyncML Nachrichtenaufbau wird durch einfache Programmierschnittstellen vorgegeben und erleichtert die Erzeugung der SyncML Elemente. Die Aufteilung in SyncML Pakete wird automatisch unterstützt. Die Pakete werden dann wahlweise in XML oder WBXML umgesetzt. Bei der Implementierung von Enkel DS muss daher in einer solchen idealen Implementierung nur der Nachrichtenablauf programmiert und die Nachrichten entsprechend aufgebaut werden. Die unteren Ebenen sind durch die Implementierung bereits abstrahiert.

Darüber hinaus muss eine Enkel DS Implementierung die Änderungen in den Knoten

erkennen. Dies wird durch den Austausch der Vektordaten innerhalb von Nachrichten möglich. Anhand der erkannten Änderungen wird möglicherweise eine Konfliktresolution nötig. In jedem Fall müssen die Änderungen in Form von Hinzufügungen, Ersetzungen oder Löschungen gespeichert werden und führen zu Datenmanipulationen der Datenbank. Eine ideale Basisimplementierung bietet daher zusätzlich die Möglichkeit verschiedene Datenbanken einheitlich anzusteuern und es wird zwischen Meta- und Nutzdaten unterschieden. Dies hat den Vorteil dass die Datenbank abstrahiert wird und dadurch austauschbar durch andere Implementierungen ist. Idealerweise bietet die Basis bereits eine Datenbank Implementierung, um Aufwand für eine Enkel DS Implementierung zu sparen.

Bei der Wiederverwendung von Implementierungen und Code spielen in der Regel weitere Faktoren eine Rolle. Bei einer kommerziellen Anwendung des fertigen Produkts ist das wichtigste Kriterium Softwarelizenzen, die eine Nutzung nicht verbieten oder unmöglich machen. Im Rahmen von Enkel DS spielt dies keine Rolle, da keine kommerzielle Nutzung des finalen Codes angedacht ist, bzw. der Quellcode von Enkel DS 1.0 soweit möglich als *Public Domain* veröffentlicht werden soll. Aus diesem Grund sollte die Basisimplementierung ebenfalls öffentlich zugänglich sein und als Open Source Projekt vorliegen.

Bei der Wiederverwendung stellen die Stabilität und die Verständlichkeit des bestehenden Quellcodes dar zwei weiter Faktoren dar. Ergeben sich Fehler in der Basisimplementierung müssen diese schnell auffindbar sein. Wichtig ist auch, dass die Basisimplementierung weiterhin gewartet wird. SyncML entwickelt sich laufend weiter und neue Klienten kommen auf den Markt. Die Basisimplementierung sollte daher aktiv weiter programmiert werden, damit sie in Zukunft nicht an Kompatibilität einbüßt. Enkel DS wird sich so schnell nicht ändern, allerdings wäre bei einem Stillstand der Basisimplementierung auch die Kompatibilität mit zukünftigen OMA DS Klienten gefährdet.

Im Folgenden werden eine Reihe frei zugänglicher Open Source Projekte aufgezeigt und an dieser für eine Enkel DS Umsetzung idealen Basisimplementierung gemessen. [Dum03]

11.5.1. SyncML tools

[Bou03] implementiert einen OMA DS Server in Version 1.0 mit Hilfe der Skriptsprache PHP [PHP97]. Der Server unterstützt nur XML und HTTP. Das Projekt versucht den Abgleich mit einem Sony Ericsson P800[SEP03] Mobiletelefon zu ermöglichen. Als Datenanbindung wird eine MySQL [MyS94] Datenbank vorausgesetzt. Der Server ist im Mai 2003 entstanden und seit dem nicht weiter entwickelt worden. Der Autor bezeich-

net es selbst als Hack und es sei nicht als Software zu klassifizieren. Das Projekt zeigt trotzdem eindrucksvoll das man selbst mit einer Skriptsprache und um die 1500 Zeilen Code einen OMA DS Server (für ein bestimmtes Gerät) schreiben kann.

11.5.2. kSync

Bei kSync [Oks01] handelt es sich um eine OMA DS Implementierung in der Programmiersprache Java. Es wird sowohl ein Klient als auch ein Server geliefert. Der Klient unterstützt Java 2 Micro Edition [Sun00b] und der Server fordert eine Java 2 Enterprise Edition Umgebung. Der öffentlich zugängliche Code stammt von der Firma Reaxion [Rea02] und ist im Dezember 2001 als Version 1.0 veröffentlich worden. Bis Mitte 2003 sind Entwicklungen in der Mailing Liste zu verzeichnen und es wird eine Version 2.0 angekündigt. Version 1.0 unterstützt als Transport nur HTTP, Pakete können nicht als WBXML versendet werden und es werden nicht alle SyncML 1.0 XML Elemente unterstützt. Der für Enkel DS benötigte `Get` Befehl ist nicht implementiert. Da das Projekt zwischen 2003 und 2004 endgültig geschlossen worden ist, wird es öffentlich nicht weiter programmiert. Bei Reaxion selbst findet sich kein Produkt, welches den Code enthalten könnte. Das Projekt scheint tot zu sein.

11.5.3. LibSyncML

Bei dieser Diplomarbeit an der Technischen Universität München handelt es sich um einen OMA DS Server in der Programmiersprache C++. [Ber02; Ber01; Buc02] Die Entwicklung stoppte im April 2002 und ein Folgeprojekt in der Programmiersprache Objective-C ist gestartet worden. Aus diesem Grund wird dieses Projekt nicht näher betrachtet.

11.5.4. SyncML support for OGo

[Ber04] ist in der Programmiersprache Objective-C geschrieben und ein OMA DS Server in Version 1.1 mit HTTP und XML Unterstützung. Der LibSyncML Quellcode ist angepasst übernommen worden. Die Datenanbindung erfolgt an das OGo Adressbuch und das Apple Adressbuch ab Mac OS X v10.2 [App03a]. Die Implementierung unterstützt noch nicht den vollen Umfang an SyncML Elementen. Dem TODO des Quellcodes ist zu entnehmen, dass es noch Probleme sowohl mit dem Slow Sync als auch der Fast Sync gibt. Letztere scheint noch nicht umgesetzt zu sein. An dem Projekt arbeiten zwei Programmierer in ihrer Freizeit.

11.5.5. SyncML C Reference Toolkit

[OMA04a] stellt die Referenzimplementierung der SyncML Initiative bzw. der OMA in der Programmiersprache C dar. Die Implementierung diente als Basis für die SyncML Conformance Test Suite (SCTS) bis OMA DS in Version 1.1.1.[14] Die Aufgabe der Referenzimplementierung ist es, einen schnelleren Einstieg in SyncML zu ermöglichen, ohne die XML und WBXML Verarbeitung erneut programmieren zu müssen. Dieses Projekt ermöglicht den Nachrichtenaufbau und generiert daraus dann die entsprechende XML/WBXML Elemente. Allerdings kann man mit Hilfe dieser API nur die Pakete zusammenbauen. Die Aufteilung von Nachrichten in Pakete erfolgt nicht automatisch. Es existieren keine vorgefertigten Schnittstellen um der Spezifikation entsprechend einen Nachrichtenablauf aufzubauen. Die Referenzimplementierung bietet absichtlich keinen vollständigen Nachrichtenablauf, um nicht mit einer *SyncML Server Industrie* zu konkurrieren. [HMPT03] Die Referenzimplementierung bietet nur SyncML Pakete zu erzeugen und beim Parsen von Paketen die eigentliche Anwendung über auftretende SyncML Elemente zu informieren. Zusätzlich wird eine Schnittstelle für den Transport angeboten und Musterimplementierungen für HTTP, OBEX und WSP werden geliefert. Eine aktuelle Implementierung für SyncML 1.2 [OMA04e] ist nicht öffentlich verfügbar. [OMA04a; HMPT03; Buc02; Ber02]

11.5.6. Sync4j

[CFF05] stellt einen OMA DS Server für die Java 2 Enterprise Edition (J2EE) zur Verfügung. Es werden sowohl fertige Programmumgebungen für Java 2 Standard Edition (J2SE) in Version 1.4 inklusive aller J2EE Bestandteile angeboten, als auch nur die entsprechenden J2EE Erweiterungen, um einen bestehenden J2EE Server weiter nutzen zu können. Darüber hinaus arbeitet Sync4j an einer Reihe weiterer Unterprojekte. Es wird ein Java 2 Micro Edition (J2ME), ein J2SE und ein C++ Klient angeboten. Letzterer ist für eine Reihe von Plattformen optimiert. Auf der Transportebene wird nur HTTP aber kein OBEX unterstützt. Auf der Nachrichtenebene wird XML und WBXML unterstützt. Die API bietet die Möglichkeit Pakete zusammenzubauen. Eine Abstraktion zwischen Paketen und Nachrichten erfolgt innerhalb des Nachrichtenablaufs. Beim Nachrichtenablauf werden alle wichtigen Abgleichtypen von OMA DS in Version 1.1 unterstützt. Eine Erweiterung für SyncML 1.2 ist noch nicht geplant, allerdings ist eine OMA Device Management [OMA04d] Erweiterung in Planung.

[14] Eine SyncML Implementierung sollte fest vorgeschriebene Tests durchlaufen, um die Zusammenarbeit zwischen verschieden Implementierungen zu verbessern.

Der SyncML Server ist mit einer Vielzahl an Klienten gestestet worden und eine Produktunterstützung ist durch vergleichsweise aktive Mailinglisten [Sou; Yah] gewährleistet. Das Projekt besitzt neben der sehr restriktiven GPL Lizenz auch eine kommerzielle Lizenz, die bezahlte Produktunterstützung durch die Firma Funambol per direkten E-Mail Kontakt erlaubt. Der Server enthält eine API, um verschiedene Datenbanken und Datenquellen anzusteuern. Einige Musterimplementierungen werden mitgeliefert. Für Metadaten werden Konfigurationen für eine Vielzahl an Datenbank Produkten bereitgestellt. Für die Nutzdaten werden Datei basierte Datenbaken mitgeliefert, aber auch fertige Module, um Microsoft Outlook/Exchange Daten anzusteuern.

11.5.7. JSR–75 und JSR–230

Diese beiden Projekte haben eine Sonderstellung in dieser Aufzählung, da es sich nicht primär um Implementierungen handelt, sondern es sind Spezifikationen für eine einheitliche Java Programmierschnittstelle. Implementierungen die diese API befolgen, können daher beliebig ausgetauscht werden. Es reicht eine Programmierung mit Hilfe der Schnittstelle und im Idealfall ist dann der entstanden Code auf jedem Gerät lauffähig, welches die APIs durch eigene Implementierungen anbietet.

JSR–75: FileConnection & PIM API

Diese Schnittstellenbeschreibung ermöglicht den Zugriff auf ein Adressbuch und einen Kalender, welche durch ein Personal Information Manager Modul (PIM) bereitgestellt werden. Ebenfalls kann darüber ein Dateisystem (File) angesprochen werden. Diese beiden Pakete File und PIM können einzeln oder zusammen bereitgestellt werden. Gerade in J2ME sind dieses APIs interessant, da es sonst keine einheitlichen Dateizugriffmöglichkeiten gibt, wie dies das `java.io` Package in J2SE bietet. Diese Schnittstellen könnten für eine SyncML oder IrMC Implementierung in J2ME genutzt werden, falls das Gerät nicht bereits selbst eine solche besitzt. Es werden aber keine Hilfestellungen für SyncML oder IrMC angeboten. [Sun04]

JSR–230: Data Sync API

Diese Schnittstellenbeschreibung wird zurzeit erarbeitet und liegt in einem frühen Entwurfsstadium vor. Ziel ist es eine bestehende OMA DS und IrMC Level 4 Implementierung zu kapseln, so dass ein Anwendungsprogrammierer nur die Nutzdaten liefern muss und selbst den Datenabgleich starten kann. Zusammen mit JSR–75 ergibt sich dadurch die Möglichkeit alle Daten des Geräts abzugleichen. Da sich die Spezifikation noch in

der Entwicklung befindet, existieren noch keine Implementierungen. Nach dem jetzigen Stand des Entwurfs kann Enkel DS höchstens selbst eine JSR–230 Implementierung sein, da die API von einer Log Datei basierten Anwendung ausgeht und nur `Add`, `Replace` und `Delete` Befehle bereitstellt.

11.5.8. Entscheidung

Die Projekte SyncML tools, kSync, LibSyncML und SyncML for OGo sind jeweils sehr interessante Projekte aber werden meist nicht mehr weiter entwickelt. Alle diese Projekte haben noch keinen sehr hohen Reifegrad erreicht. Die Zusammenarbeit mit anderen SyncML Implementierungen ist kaum gegeben. Die Referenzimplementierung ist dagegen sehr nahe an einer gewünschten idealen Implementierung. Im Bereich des Transports und im Nachrichtenaufbau – bis auf die Unterscheidung Paket und Nachricht, die kein Projekt bietet – entspricht sie einer idealer Basisimplementierung. Allerdings wäre für Enkel DS eine bestehende OMA DS Authentifizierung sinnvoll. Ebenso wären bereits vorgefertigte Datenbankschnittstellen für Metadaten und Nutzdaten von Vorteil. Beides bietet die Referenzimplementierung nicht, sondern diese Funktionen müssen selbst programmiert werden. Sync4j entspricht bis auf das Fehlen vom OBEX Transport einer idealen Basisimplementierung. Das Sync4j ist sehr aktiv und die Programmiersprache Java erlaubt durch die Möglichkeiten der Codewiederverwendung eine schnelle Erweitung und viele neue mobile Geräte gerade im Bereich von Mobiltelefonen bieten keine C API sondern nur J2ME an. Allerdings ist zu bedenken, dass die öffentlichen Schnittstellen in Sync4j noch nicht sehr stabil sind. Dies hat zur Folge, dass sich zwischen verschiedenen Versionen die API noch ändert und alte Erweiterungen inkompatibel werden.

12. Umsetzungsdetails

12.1. Sync4j Klassen

In Sync4j verwaltet das Java Interface `SessionHandler` im Package `sync4j.framework.server.session` eine Abgleichsitzung. Die Klasse `sync4j.server.session.SyncSessionHandler` implementiert diese Schnittstelle und es wird für jeden Abgleich eine neue Instanz erzeugt. Die einzelnen SyncML Pakete werden vom HTTP Server empfangen, gegebenenfalls von WBXML in XML umgewandelt und dann durch das XML/Java Object Binding Framework *JiBX* [Sos04] konvertiert. Daher muss nicht durch eine XML Parser API (z.B. SAX oder DOM) auf die SyncML Pakete zugegriffen werden, sondern das gesamte SyncML Paket liegt als ein `SyncML` Java Objekt im Speicher.

Die XML Elemente und deren Inhalte werden durch Java Objekte aus dem Package `sync4j.framework.core` ansprechbar (Abbildung 12.1). So kann aus einem SyncML Paket dessen `SyncHdr` und `SyncBody` als Java Objekt erhalten werden. Ein `SyncBody` Java Objekt enthält einen oder mehrere Befehle und zeigt an, ob es das letzte Paket einer Nachricht ist. Aus einem Befehl können dessen `ID`, `NoResp`, `Meta` und `Cred` Inhalte ausgelesen werden. Alle Befehle erweitern die gemeinsame Oberklasse `AbtractCommand` um ihre speziellen XML Elemente. XML Befehle die `Item` Elemente enthalten, werden in der Java Oberklasse `ItemizedCommand` zusammengefasst.

Die SyncML Pakete werden nach der Umwandlung in Java Objekte einem `SyncSessionHandler` übergeben, welcher die Daten analysiert und wieder in Form eines `SyncML` Java Objekts antwortet. Diese Antwort wird in (WB)XML umgewandelt und danach dem HTTP Server übergeben, um das Paket an den Klienten zurückzusenden.

Abbildung 12.2 zeigt wie jedes Paket den Zustand im `SyncSessionHandler` ändern kann. Nach dem Zustand *Init Processed* ist die Sitzung autorisiert. Die nachfolgenden Pakete bewirken einen Zustandübergang und es beginnen die Datenmanipulationen entsprechend OMA DS.

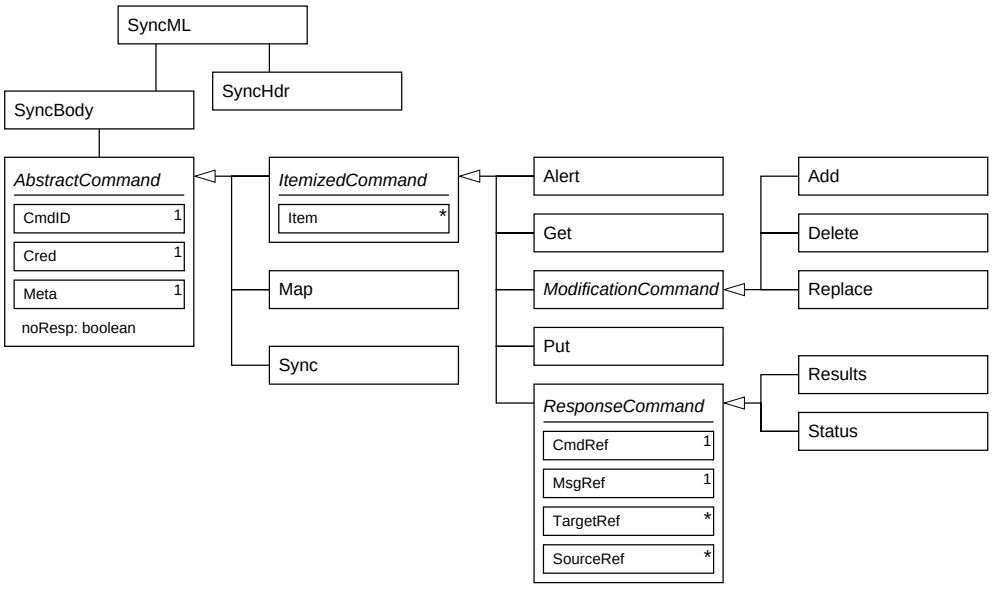

Abbildung 12.1.: Klassendiagramm – Auszug Sync4j Core Framework

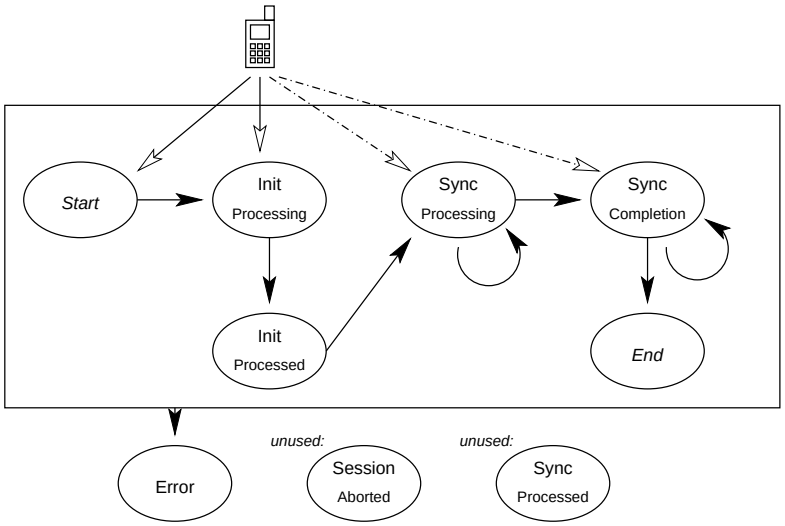

Abbildung 12.2.: Zustandsdiagramm – Sync4j SyncSessionHandler

12.2. Enkel DS Server

In der Enkel DS Implementierung wird ein neuer `SyncSessionHandler` eingeführt, der sich analog zum Sync4j Package und den Java Package Bennenungsregeln in `de.traud.EnkelDS.server.session` befindet. Die Enkel DS Implementierung erzeugt als erstes einen Sync4j `SyncSessionHandler` und leitet alle Pakete an diesen weiter, bis der Nutzer autorisiert ist. Die folgenden Pakete werden dann direkt von Enkel DS verarbeitet, falls der Klient die Enkel DS Spezifikation versteht. Ansonsten verläuft die gesamte Sitzung wie in OMA DS und über den original Sync4j `SyncSessionHandler`. Im Enkel DS `SyncSessionHandler` werden die Nachrichten dem Kapitel 11 entsprechend verarbeitet und beantwortet.

Datenbankzugriff

Der Zugriff auf die Datenbanken erfolgt über die Klasse `sync4j.framework.server.store.PersistentStoreManager`. In Sync4j dient diese Klasse als zentraler Zugriff auf alle dauerhaft gespeicherte Daten, indem man der Klasse die Objekte übergibt, die man erhalten möchte. Damit dieser Manager auf die Daten zugreifen kann, enthält dieser mehrere `PersistentStore` Objekte, wobei jedes für ein oder mehrere Klassen zuständig ist.

Nutzdaten

Sync4j Um die Datenquellen zu erhalten, welche die Nutzdaten verwalten, wird dem Manager die Klasse `sync4j.framework.server.Sync4jSource` übergeben, der die entsprechenden Objekte zurückgibt. Diese `Sync4jSource` Objekte können dazu genutzt werden, Hinzufügungen, Ersetzungen und Löschungen durchzuführen. Sync4j enthält eine Reihe an Datenquellen, die auf vCard, vCalendar und vNote [LSS00] Objekte spezialisiert sind. Die Datenquellen selbst legen die Nutzdaten im Dateiverzeichnis des Betriebsystems ab.

Enkel DS In Enkel DS sind diese Datenquellen durch die Klasse `de.traud.EnkelDS.framework.engine.source.EnkelSource` gekapselt. Diese Klasse dient dazu, die durch eine OMA DS Sitzung geänderten Daten, mit aktualisierten Vektordaten in Enkel DS verfügbar zu machen.[1][2]

[1] In der aktuellen Implementierung fehlt diese Funktionalität noch, daher werden alle `EnkelSource` direkt an die entsprechende Datenquellen von Sync4j weitergeleitet.

[2] `EnkelSource` behebt zusätzlich in den Sync4j Datenquellen Bugs, so dass diese nicht in einem `SyncSessionHandler` umgangen werden müssen.

Metadaten

Für die Metadaten in Enkel DS ist die Klasse `de.traud.EnkelDS.framework.store.EnkelPersistentStore` im Manager zuständig. Die folgenden Klassen befinden sich ähnlich wie in Sync4j im Package `de.traud.EnkelDS.framework.util`. Bei diesen Klassen ist auf CLDC 1.0 [Sun00a] Kompatibilität geachtet worden. CLDC ist die Grundlage von Java 2 Micro Edition und stellt eines der Java Umgebungen mit dem geringsten API Umfang dar. Zweck dieser CLDC Kompatibilität ist es, mit so wenigen Anpassungen wie möglich eine Java 2 Micro Edition kompatible Version von Enkel DS entwickeln zu können.

ChangeCounter Ein `ChangeCounter` wird als Metadatei gespeichert, da Sync4j selbst nicht mit wieder verwendbaren logischen Zeitstempeln sondern mit physikalischen Uhrzeiten arbeitet.

NodeMap Zusätzlich wird eine `NodeMap` gespeichert, die eine LUID-GUID Umsetzungstabelle darstellt. Die Tabelle wird während des Abgleichs zur Findung von GUIDs genutzt und am Anfang des Datenabgleichs als `Map` übermittelt.

MetaRecord Die Vektordaten $\vec{m}$ und $\vec{s}$ werden in der Klasse `MetaRecord` zusammengefasst und dauerhaft gespeichert. `MetaRecord` Objekte können eindeutig über deren c angesteuert werden. Die Vektordaten sind ebenfalls Objekte und besitzen jeweils ein `java.util.Hashtable` Objekt, wobei der Schlüssel die Knoten ID und der Wert der Zeitstempel auf diesem Knoten ist. Die Knoten IDs sind `java.lang.String` Objekte und die Zeitstempel sind Unterklassen von `java.math.BigInteger`. Anstatt die Datentypen `Integer` oder `Long` von Java zu nutzen, fiel die Wahl für den Change Counter und die Zeitstempel auf die Klasse `BigInteger`, da in Vektorzeitpaaren keine Zahlenüberläufe stattfinden dürfen. Die Klasse `BigInteger` vermeidet jegliche Überläufe und verbraucht dennoch nur soviel Speicherplatz wie nötig, da intern ein Byte Array zum Einsatz kommt.[3]

[3] Die Klasse `BigInteger` ist nicht in CLDC enthalten, es gibt allerdings eine CLDC kompatible Implementierung im Bouncy Castle Crypto Projekt [Bou00], welche aufgrund seiner Lizenz leicht wieder verwendet werden kann.

12.3. Enkel DS Klient

Der Quellcode für den Klienten in Enkel DS nutzt die Packages des Enkel DS Servers mit und vermeidet so eine doppelte Implementierung. Darüber hinaus verwendet der Klient die gleiche `SyncML` Java Objekt Darstellung wie der Sync4j Server. Da der Klient ebenfalls eine Sitzung steuert, befindet sich die gesamte Logik des Enkel DS Klienten in der Klasse `de.traud.EnkelDS.client.SessionLayer`. Das entsprechende Objekt dieser Klasse nutzt das Sync4j JiBX Object Binding zur Konvertierung der SyncML Pakete von der Java Objektform in (WB)XML. Der Enkel DS Klient nutz darüber hinaus den Sync4j HTTP Klienten aus dem Package `sync4j.syncclient.spds`.

12.4. Konfliktresolution

Da der Enkel DS Klient Konfliktresolutionen durchzuführen hat, existiert das Interface `ConflictResolver`, welches bei jedem Konflikt aufgerufen wird. Dabei wird näher bestimmt, ob beide Knoten die Datei geändert haben oder ob ein Knoten den Datensatz geändert, aber in der Zwischenzeit der andere Knoten den Datensatz gelöscht hat. Damit eine entsprechende Entscheidung vom Benutzer getroffen werden kann, werden die Nutzdaten angezeigt.[4] Da es nicht Ziel dieser Diplomarbeit war, eine ideale Benutzerschnittstelle für die Konfliktresolution zu bieten, ist auf eine graphische Version verzichtet worden und eine entsprechend leistungsfähige Kommandozeilenabfrage als Unterklasse der `ConflictResolver` Schnittstelle implementiert worden. Zukünftige Arbeiten können diese Schnittstelle implementieren, um verbesserte graphische Benutzerschnittstellen anzubieten.

12.5. Übersetzung und Ausführung

Der Quellcode liegt auf der CD-ROM bei und die Datei `terminal.txt` gibt die weiteren Befehle an, um den Quellcode zu übersetzen und das Resultat auszuführen.

Die Klasse `Compiler` im Package `de.traud.EnkelDS.test` erlaubt die Übersetzung des Quellcodes sowohl für den Enkel DS Klienten als auch den Server. `Changer` und `Client` sind Beispielimplementierungen für einen Enkel DS Klienten und einen Daten-

[4] [CJ04] hat in einem solchen Fall dem Nutzer zusätzlich Änderungszeitstempel aber nicht die Nutzdaten angeboten. In Enkel DS ist alles bereits dafür vorgesehen, auch die Änderungszeitstempel mit anzuzeigen. Die aktuelle Sync4j Implementierung der Datenquellen erlaubt dies nicht. Eine entsprechende Funktionalität ist angefordert und für die nächste Sync4j Version akzeptiert worden: Feature Request #301219.

manipulierer. `Tester` startet den Klienten mit den Testfällen, die im nächsten Kapitel 13 beschrieben sind.

13. Testfälle

Unsion beweist die Korrektheit eines Abgleichverfahrens, welches zwei Knoten unterstützt. In Enkel DS können potentiell unendlich viele Knoten unterstützt werden. Der Beweis einer Korrektheit von Enkel DS oder Vektorzeitpaaren ist nicht Gegenstand dieser Ausarbeitung. Es gilt die Implementierung durch verschiedene Testfall Klassen weitestgehend auf seine Funktionalität zutesten und damit aufzuzeigen, dass der Algorithmus der Vektorzeitpaare verstanden worden ist und in eine funktionsfähige SyncML Implementierung für Zyklen entwickelt werden konnte.

13.1. Grundlegende Funktionalität

Die erste Reihe mit acht Fällen testet die Funktionen von Enkel DS und stellt sicher, dass die drei Arten von Änderungen (Hinzufügungen, Ersetzungen und Löschungen) von einem Knoten zum nächsten wandern können. In diesen Tests werden drei Knoten A, B und C simuliert. Die Pfeile zwischen den Knoten geben die Richtung der Wanderung einer Änderung an. Der Pfeil ist mit der Art der Änderung beschriftet, wobei N für *New* (Hinzufügung), R für *Replace* (Ersetzung) und D für *Delete* (Löschung) steht. Ein x deutet an, dass zwar ein Abgleich stattfindet, aber keine Nutzdaten übertragen werden sollen. Die Zahl vor dem Buchstaben gibt die Reihenfolge der Änderungen an.

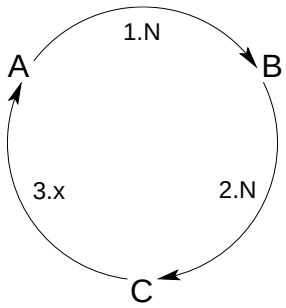

Abbildung 13.1.: Testfall 01

Im Testfall 1 wandert als erste eine Änderung von A nach B. B arbeitet als Klient und bezieht die neusten Daten von A. Danach arbeitet B als Server und C bezieht als Klient

die neusten Daten. Im dritten Schritt findet ein Abgleich zwischen C und A statt, wobei diesmal A die neusten Änderung von C bezieht. Besitzt A wie in diesem Testfall eine Hinzufügung, dann sollte B diesen neuen Datensatz erkennen und anfordern. C sollte diesen neuen Datensatz auf B erkennen, obwohl A die Hinzufügung erzeugt hatte. Am Ende soll A erkennen, dass C auf dem gleichen Stand ist, und es sollen keine Metadaten bzw. Nutzdaten für diesen Datensatz ausgetauscht werden.

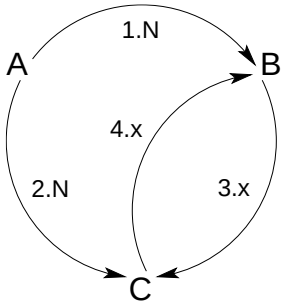

Abbildung 13.2.: Testfall 02

Da A im ersten und zweiten Schritt die Hinzufügung bereits B und C bekannt gemacht hat, testet Fall 2, ob B und C auch ohne A entscheiden können, dass der Datensatz von A bereits existent ist. B und C durchlaufen untereinander einen Abgleich, dabei müssen aber keine Änderungen oder Nutzdaten übertragen werden.

Ähnliches wird in Fall 3 und 4 getestet. In diesen Fällen kennen bereits alle drei Knoten denselben Datensatz. Es handelt sich folglich um die Situation nach dem Fall 1 oder 2. A hat nun eine Änderung an dem Datensatz durchgeführt und diese Änderung soll zu B und C wandern. Findet ein Abgleich von C nach A statt, d.h. im Testfall 3 ist C der Server und A der Klient, dann soll A erkennen, dass keine Nutzdaten zu übertragen sind. Gleiches gilt für die Knoten B und C im Testfall 4.

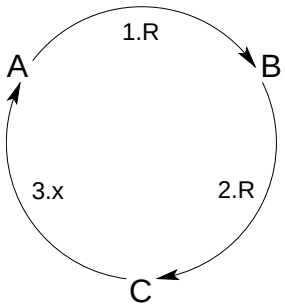

Abbildung 13.3.: Testfall 03

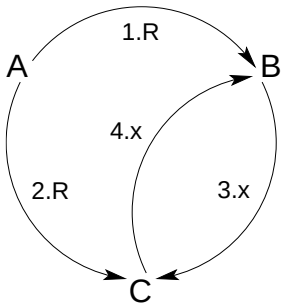

Abbildung 13.4.: Testfall 04

In Testfall 5 und 6 wird ein bereits allen bekannter Datensatz gelöscht. Ausgangspunkt

kann hier Testfall 1, 2, 3 oder 4 sein.

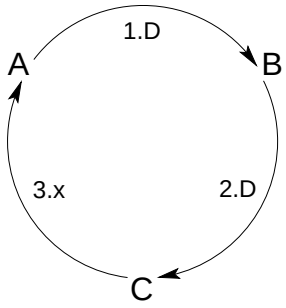

Abbildung 13.5.: Testfall 05

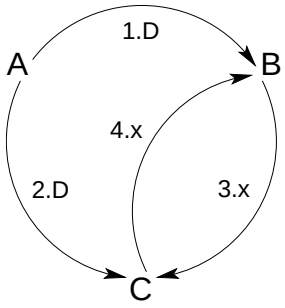

Abbildung 13.6.: Testfall 06

In Testfall 7 wird erst ein allen bekannter Datensatz bei A geändert und diese Änderung wandert zu B. B ändert daraufhin den Datensatz ebenfalls und Diese zweite Änderung stellt die aktuellste Version dar. Wandert die Änderung zu C und danach zurück zu A, soll kein Konflikt entstehen, da die zweite Änderung die erste Änderung *dominiert*, d.h. die zweite Änderung ist die Maß gebende Instanz.

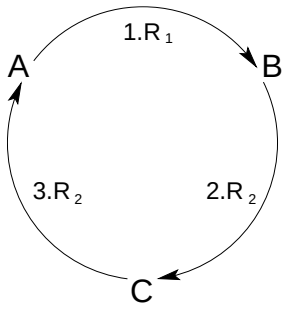

Abbildung 13.7.: Testfall 07

In Testfall 8 wird ein ähnlicher Aufbau angenommen. Hier ändert allerdings auch C den Datensatz (3. Änderung), nachdem er von B empfangen worden ist.

Diese Testfälle dienten zur Überprüfung der Funktionalität der Enkel DS Implementierung. Falls Fehler in diesem Zusammenhang aufgetreten sind, dann handelte es sich meist einfach um Implementierungsfehler entweder im Sync4j Framework oder in Enkel DS selbst.[1] Die Testfälle sind Stück für Stück durchgegangen worden. Fand sich ein

[1] In diesem Zusammenhang sind etwa ein dutzend Fehler und ein halbes dutzend Verbesserungsvorschläge in Sync4j 2.2b2 aufgefunden und an die Sync4j Entwickler gemeldet worden. Da die jeweiligen Fehler im vorliegenden Quellcode genau untersucht und korrigiert werden konnten, sind die Lösungen für einen Großteil der Fehler bereits in die nächste Version von Sync4j eingeflossen. Größtes Problem besteht darin, dass die online verfügbare Version weiterhin das EMI Element nicht unterstützt. Auf der beigelegten CD-ROM finden sich entsprechende Anpassungen an dem original Sync4j Quellcode.

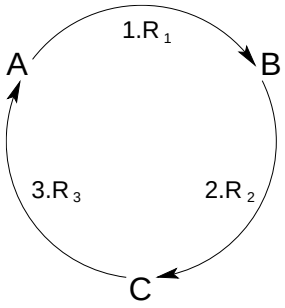

Abbildung 13.8.: Testfall 08

Fehler, ist dieser zuerst korrigiert worden. Keiner der acht Tests lief beim ersten Mal durch. Dies zeigt, dass die Testfälle gut gewählt waren. Da die Enkel DS spezifischen Fehler schnell gefunden werden konnten, stellte dies die Qualität der Implementierung nicht völlig in Frage.

13.2. Komplexe Beziehungen

Testfall 9 sah einen vierten Knoten im Netzwerk mit der Begründung vor, dass A und C keine Kommunikationsbeziehung besitzen und sich gegenseitig nicht kennen. Allerdings besitzt bereits C bei drei Knoten zu A keine Änderungsbeziehung. C liefert nur Daten aus, merkt sich allerdings nicht an wen. Dies kommt daher, da ein Enkel DS Server keine Änderungen an den lokalen Datensätzen durchführt. Nur Klienten oder Datenmanipulierer ändern Metadaten und Nutzdaten. Es ist nicht nötig, mit mehr als drei Knoten Testfälle aufzubauen. Testfall 9 entfiel daher.

Testfall 10 beschäftigt sich mit der Kombination von Hinzufügungen, Ersetzungen und Löschungen und obwohl A für C eine Hinzufügung hat, dominiert die Löschung von C, da diese Löschung eine Ersetzung von B dominiert, die wiederum die Hinzufügung von A dominiert. Bei A darf daher kein Konflikt auftreten und der Datensatz wird automatisch gelöscht.

13.3. Mehrere Datensätze pro Knoten

Die bisherigen Testfälle umfassten nur einen Datensatz im Wurzelverzeichnis. Es muss ebenfalls überprüft werden, dass mehrere Datensätze unter der Wurzel korrekt verarbei-

Ein weiteres Problem ist die Missachtung von Namensräumen innerhalb eines SyncML XML Pakets, die vom Sync4j Server gesendet werden. Dies Problem konnte nur teilweise gelöst werden. Eine vollständige Lösung hätte Änderungen an zu vielen Sync4j Klassen bedeutet.

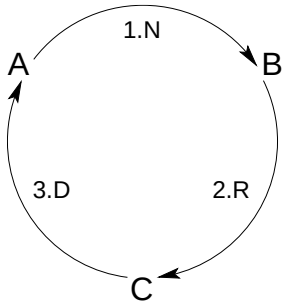

Abbildung 13.9.: Testfall 10

tet werden. Dies betrifft im Besonderen die Aktualisierung der Vektordaten nach einem erfolgreichem Abgleich. Entsprechend werden in diesem Testfall zwei Datensätze in A erzeugt, wovon einer bereits B bekannt ist. B ersetzt den bekannten Datensatz und A sendet seine Hinzufügung an B. B sendet seine Ersetzung an A. Werden die Vektorzeiten über beide Datensätze korrekt aktualisiert, ist dieser Testfall erfolgreich durchlaufen. Mehr als zwei Datensätze sind nicht nötig, da es nur wichtig ist, dass verschiedene Vektorzeiten korrekt aktualisiert werden. Weiter verschaltete Verzeichnisstrukturen werden nicht getestet, da die Enkel DS Implementierung entsprechend SyncML Geräten nur flache Datenbanken unterstützt.[2]

13.4. Konflikte

Die Testfälle 12 bis 15 beschäftigen sich mit der Fragestellung, ob ein Konflikt korrekt erkannt wird und ob die Konfliktresolution funktioniert. Dies ist für die nachfolgenden Testfälle wichtig, da diese komplexere Szenarien darstellen.

In Testfall 12 hat B eine Ersetzung vorgenommen, bekommt nun aber eine andere Ersetzung von A. Es besteht ein Schreib/Schreib Konflikt seit dem letzten Abgleich. Ein Konflikt muss gemeldet werden und die Konfliktresolution muss je nach Implementierung des `ConflictResolver` bzw. in der Beispielimplementierung je nach Wahl des Nutzers, die Version von A oder die Version von B abspeichern.

In Testfall 13 hat B den Datensatz gelöscht, erhält von A aber eine Ersetzung die B noch unbekannt war. Es besteht ein Lösch/Schreib Konflikt. Wenn der Benutzer sich dafür entscheidet, den Datensatz zu löschen, dann soll die von A kommende Ersetzung verworfen werfen. Entscheidet sich der Benutzer für die Version A, dann muss der Datensatz in B wieder angelegt werden.

[2] Eine nächste Ausbaustufe wäre die Unterstützung von komplexen Verzeichnisstrukturen. Innerhalb von Enkel DS ist dafür bereits alles vorbereitet und der Server sollte dies bereits unterstützen.

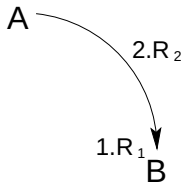

Abbildung 13.10.: Testfall 12

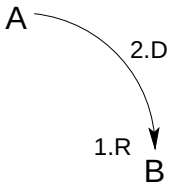

Abbildung 13.11.: Testfall 13

Testfall 14 beschäftigt sich mit einer ähnlichen Frage. In diesem Fall hat B eine Änderung vorgenommen aber A sendet eine Löschung. Erneut liegt ein Konflikt vor, da zwei konkurrierende Änderungen seit dem letzten Abgleich vorgenommen worden sind und keine Änderung dominiert die andere.

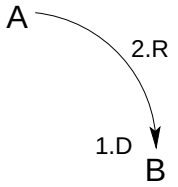

Abbildung 13.12.: Testfall 14

In Testfall 15a hat seit dem letzten Abgleich sowohl A als auch B eine Löschung durchgeführt. Obwohl beide eine Änderung aufweisen und ein Schreib/Schreib Konflikt besteht, stehen die Änderung nicht in Konkurrenz, da egal welche Version übernommen wird, beide Versionen führen zu dem selben Ergebnis, nämlich dass eine Löschung auf B stattfinden soll.

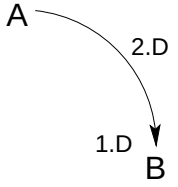

Abbildung 13.13.: Testfall 15a

In Testfall 15b hat B eine Löschung vorgenommen. A hat aber weiterhin einen Datensatz, da A noch nichts von der Löschung weiß. Wenn B die Änderungen von A bezieht, dann muss B erkennen, dass seine Version gelöscht worden ist und diese Löschung die bestehende Version in A dominiert. Dieser Test ist wichtig, da Vektorzeitpaare keine Löschvermerke besitzen und nur anhand der Vektordaten des übergeordneten Verzeich-

nisses die Löschung erkannt wird.

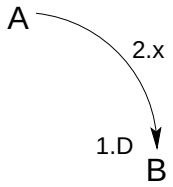

Abbildung 13.14.: Testfall 15b

13.5. Komplexe Szenarien

Die folgenden Testfälle zeigen komplexe Szenarien auf, in denen Vektor basierte Systeme scheitern, indem falsche Konflikte gemeldet werden. Obwohl die Fälle komplex sind, konnten diese Tests bereits beim ersten Mal erfolgreich durchlaufen werden. Dies zeigt die Qualität von Vektorzeitpaaren. Sind diese einmal richtig implementiert, dann bieten diese für den Benutzer einen sehr hohen Komfort, da er alle Knoten untereinander abgleichen kann, ohne sich Gedanken über Zyklen machen zu müssen. Darüber hinaus wird der Benutzer mit weniger Konflikten als in anderen Systemen konfrontiert, die nur mit einem Vektor arbeiten.

In Testfall 16 kennen alle drei Knoten einen Datensatz. A sendet eine Ersetzung an B und C. Danach löscht B den Datensatz. Beim Abgleich zwischen B und C wird zwar die Ersetzung von A übermittelt. Da die Löschung von nach der Ersetzung geschehen war, erkennt B, dass seine Löschung dominiert. Es tritt kein Konflikt auf.

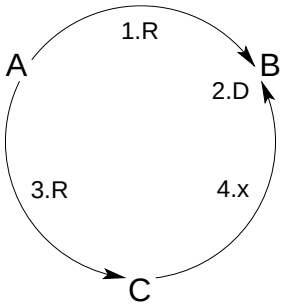

Abbildung 13.15.: Testfall 16

In Testfall 17 sendet A eine Ersetzung an C. C ändert ebenfalls den Datensatz und sendet die aktuelle Ersetzung an B. B bezieht die neusten Änderungen von A, wobei B die dominierende Ersetzung von C beibehält. Wieder entsteht kein Konflikt.

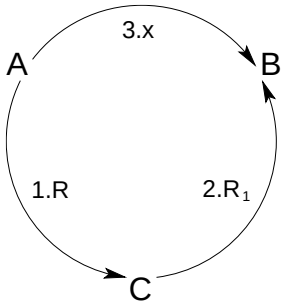

Abbildung 13.16.: Testfall 17

In Testfall 18 besitzen alle drei Knoten den gleichen Datensatz. C ersetzt den Datensatz. A sendet danach eine eigene Ersetzung an B und C. Bei C entsteht ein Schreib/ Schreib Konflikt, da zwei voneinander unabhängige Ersetzungen aufgetreten sind. Der Konflikt wird durch die Konfliktresolution (durch den Benutzer) aufgelöst. Sendet nun C seine Version an B, dann wird kein Konflikt gemeldet, egal für welche Version sich C entschieden hatte. Wenn sich C dafür entscheidet die Ersetzung von A zu übernehmen, dann entsteht kein Konflikt, da C und B dieselbe Version besitzen. Entscheidet sich C für seine Änderung, dann entsteht ebenfalls kein Konflikt, da die aktualisierten Vektorpaare verdeutlichen, dass C bereits mit A abgeglichen hat. C muss die aktuellste Version besitzen, denn er hat eine für B unbekannte Änderung, hat aber den gleichen Wissenstand wie B. B wird dann die neue Version von C übernehmen und die Nutzdaten anfordern.

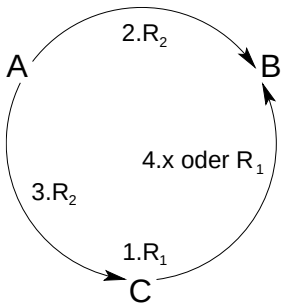

Abbildung 13.17.: Testfall 18

Die Testfälle 19 bis 22 sind anderen Tests sehr ähnlich und dienten nur zum weiteren Testen. Testfall 23 entfiel, da er doppelt war.

Im Testfall 24 besitzen alle drei Knoten dieselbe Version. A, B und C ersetzen ihre Version unabhängig voneinander. Nun sendet A seine Ersetzung zu B und C. Bei beiden Knoten muss eine Konfliktresolution stattfinden. Sendet C seine aktuelle Version an B, dann entsteht in zwei möglichen Konfliktresolutionen kein Konflikt. Falls sich B für die

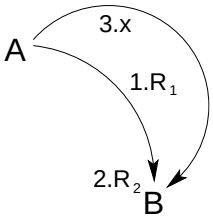

Abbildung 13.18.: Testfall 19

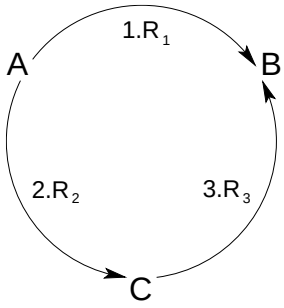

Abbildung 13.19.: Testfall 20

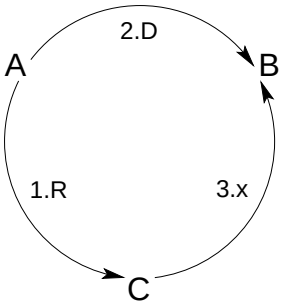

Abbildung 13.20.: Testfall 21

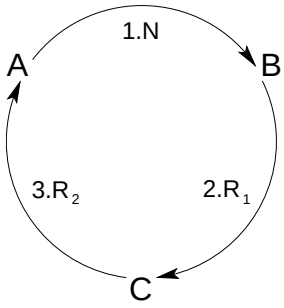

Abbildung 13.21.: Testfall 22

Version von A entschieden hat, dann ist es egal welche Version C genommen hat, es wird kein Konflikt auftreten. Angenommen C hat sich für die Version von A entschieden, dann kann gar kein Konflikt vorhanden sein, da beiden Knoten dieselben Nutzdaten besitzen. Hat sich C für seine Version entschieden, dann dominiert diese Version gegenüber A. Hat sich nun B für die Version von A entschieden, wird diese durch die Version von C ebenfalls dominiert und B fordert die Nutzdaten von C an, da dieser Knoten die aktuellste Version besitzt. Die andere Möglichkeit in der kein Konflikt auftritt, sieht wie folgt aus: B hat sich für seine Version entschieden und dominiert die Version von A. C hat sich für die Version von A entschieden und nun seine Änderungen an B. B wird keine Nutzdaten anfordern, da seine Version die Version von C und von A dominiert. Haben sich B oder C für andere Konfliktresolutionen entschieden, dann entsteht weiterhin ein Konflikt, denn in diesen Möglichkeiten haben B und C konkurrierende Versionen.

Mit nur einem Vektor wären in jedem Fall drei Konflikte gemeldet worden, da nur der Modifikationsvektor von B und C betrachtet wird, welche immer inkompatibel sind. Inkompatibilität bedeutet, dass ein Konflikt vorliegt. Der Synchronisationsvektor in Vektorzeitpaaren ermöglicht die Bestimmung einer Dominanz eines Datensatzes auch nach einer Konfliktresolution.

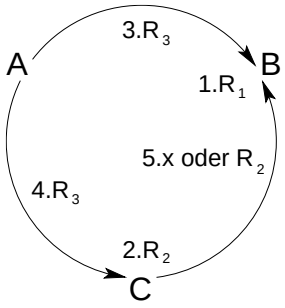

Abbildung 13.22.: Testfall 24

Testfall 25 bildet die Situation aus [CJ05, Abbildung 3] nach und stellt diese in der hier gewählten Notation dar. Dieses Beispiel dient in [CJ05] zur Verdeutlichung der Fähigkeit der Versionsdominanz innerhalb von Vektorzeitpaaren. Dieser Fall ist eine Kombination aus bereits vorangegangen Testfällen ist, zeigt aber, dass der Algorithmus von Vektorzeitpaaren korrekt an SyncML angepasst worden ist.

A besitzt eine neue Version und sendet diese im Schritt 1 und 2 an B und C. A und B ersetzen danach voneinander unabhängig die Version und B bezieht erneut die aktuellsten Daten von A. Es liegt ein Konflikt vor. Danach bezieht B erneut die aktuellsten Daten von A. In diesem fünften Schritt sollten keine Konflikte auftreten oder Nutzdaten übertragen werden. Das Gleiche gilt für den sechsten Schritt, in dem C nur die Hinzufügung von A mitteilen will, die B schon lange kennt. Im siebten Schritt hatte A erneut eine Ersetzung durchgeführt und gleicht diese mit B ab. Bei B entsteht kein Konflikt, falls sich B in der vorhergehenden Konfliktresolution für die Version von A entschieden hatte. Hatte sich B für seine Version entschieden, wird nun wieder ein Konflikt gemeldet, da die aktuelle Version von A noch unbekannt ist und B diese nicht dominiert.

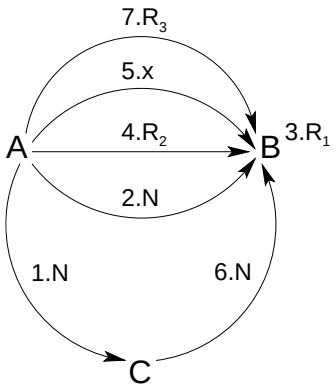

Abbildung 13.23.: Testfall 25

14. Ausblick

Die Implementierung von Enkel DS zeigt, dass OMA SyncML DS um eine Zyklenunterstützung erweitert werden kann, ohne dass dem Benutzer zusätzliche Konfliktresolutionen zugemutet werden. Da die Erweiterung mit OMA SyncML DS kompatibel ist, kann ein Server sowohl OMA DS als auch Enkel DS Klienten unterscheiden und entsprechend bedienen. Die Vektorzeitpaare selbst bieten aber noch mehr, als bis jetzt in der Implementierung umgesetzt ist.

14.1. Einbindung von OMA DS

Die aktuelle Implementierung unterstützt zwar OMA DS Klienten, allerdings werden deren Änderungen am Server nicht in Enkel DS aufgenommen. Es ist noch nicht möglich ein Netzwerk aus Enkel DS und OMA DS Knoten aufzubauen, bei dem alle Daten zwischen den verschiedenen Protokollen umherwandern. In Enkel DS ist dies allerdings bereits durch die Klasse `EnkelSource` vorgesehen, die eine Umsetzung der IDs von OMA DS Datensätzen auf Enkel DS bietet. In weitere Arbeiten müssten die Änderungen in OMA DS erkannt und die Vektoren in Enkel DS entsprechend aktualisiert werden. In dieser Arbeit war aber wichtig, dass OMA DS für Zyklen erweitert werden kann. Dabei ist auf die Kompatibilität zum jetzigen OMA DS geachtet worden, so dass diese Erweiterung in der Zukunft leicht möglich ist.

14.2. Unterstützung von Unterverzeichnissen

IrMC Level 4 und OMA DS 1.1 Implementierungen unterstützen nur eine Datenbank mit einer flachen Struktur. Weitere Unterverzeichnisse werden erst durch OMA DS in Version 1.2 sinnvoll. Es wird angenommen, dass der Enkel DS Server bereits tief verschachtelte Unterverzeichnisse unterstützt, dies ist allerdings nicht getestet worden, da der Enkel DS Klient genau wie eine OMA DS Implementierung zurzeit nur für eine flache Struktur ausgelegt ist. Das Kapitel über den Nachrichtenablauf von Enkel DS zeigt bereits die mögliche Umsetzung für tiefere Verzeichnisstrukturen.

14.3. Partielle Abgleiche

Vektorzeitpaare erlauben im Gegensatz zu IrMC Level 4 und OMA DS partielle Abgleiche. Partielle Abgleiche sind unüblich, da der Benutzer nach einem Abgleich auf beiden Knoten die gleichen Daten erwartet und die Datenmengen von einem Adressbuch nicht so groß sind, als dass partielle Abgleiche sinnvoll sind. Die aktuelle Implementierung unterstützt dies noch nicht. Weitere Arbeiten müssten zeigen, in wie weit Abbrüche unterstützt werden. In OMA DS muss der ganze Abgleich gelingen. In Enkel DS sollte es bereits möglich sein, nur Teile einer XML Datei zu Empfangen und diese soweit zu verarbeiten. Weitere Arbeiten sollten untersuchen, wie viel eines SyncML Pakets benötigt wird, um darauf bereits zu arbeiten. Gegebenfalls muss die Implementierung bis auf die Ebene des Transports angepasst werden, so dass bereits Teile der Transportpakete für einen partiellen Abgleich ausreichen. Dadurch wären Abbrüche optimal unterstützt.

SyncML sieht für jeden Befehl eine Antwort in Form eines `Status` vor. Im OMA DS kompatiblen Nachrichtenabschnitt wird dies eingehalten. Im reinen Enkel DS Teil wird bei jedem Befehl kein Status verlangt, da auf ein `Get` entweder ein `Results` folgt oder nicht. Für eine optimale Unterstützung von Verbindungsabbrüchen sollte untersucht werden, ob `Status` Antworten nicht auch im Enkel DS Teil des Nachrichtenablaufs sinnvoll sind.

Da partielle Abgleiche in Enkel DS fehlen, hätte bereits ein Synchronisationsvektor pro Verzeichnis und nicht zusätzlich für jede Datei ausgereicht. Da Vektorzeitpaare diese Möglichkeit aber bieten und zukünftige Arbeiten die Funktionalität schnell unterstützen sollen, ist trotzdem die aufwendigere Implementierung mit Deltas und Synchronisationsvektoren für jeden Datensatz umgesetzt worden. Darüber hinaus sollte die Umsetzung aufzeigen, mit welchem Aufwand der vergleichsweise komplexe Algorithmus zu Vektorzeitenpaaren umgesetzt werden kann. Es fehlt nur noch die Untersuchung bis zu welchem Grad an Abbrüchen die Vektoren aktualisierte werden können oder nicht.

14.4. Slow Synchronization

Enkel DS bietet im Gegensatz zu IrMC Level 4 und OMA SyncML DS keine Slow Sync an. Diese ist dann nötig, wenn die Log Dateien inkompatibel sind, sich die Knoten noch nicht oder nicht mehr untereinander kennen. In Enkel DS können aufgrund der Vermeidung von Überläufen durch die Klasse `java.util.BigInteger` keine Situationen entstehen, in denen Vektordaten verloren gehen. Auch ist anzunehmen, dass aktuelle Geräte genügend Speicherplatz für die Vektordaten mitbringen, damit keine Daten ver-

worfen werden müssen. Es stellt sich daher nur die Problematik des ersten Abgleichs, bei dem sich die Knoten nicht kennen. In diesem Fall entstehen Duplikate der Daten. Der Benutzer ist gezwungen diese zu löschen oder vor dem Abgleich eins der Geräte zu Leeren. Normalerweise wird auch ein neues Gerät mit einer leeren Datenbank starten. Daher ist dieser Fall in Enkel DS abgedeckt. Weitere Untersuchungen müssten zeigen, ob das Benutzerverhalten und dessen Erwartungshaltung eine Slow Sync für die restlichen Fälle nötig macht. Eine Slow Sync könnte dadurch abgefangen werden, dass jede Hinzufügung erst noch mit jedem Datensatz verglichen wird, bevor diese aufgenommen wird. Der andere Knoten müsste dann über eine Löschung der Dublette informiert werden.

14.5. SyncML Paket vs. Nachricht

In SyncML wird zwischen Paketen und Nachrichten unterschieden. Diese Unterscheidung fehlt in der Enkel DS Implementierung, da Sync4j diese Unterscheidung nicht abstrahiert. In den Recherchen hat keine mögliche Basisimplementierung eine Schnittstelle geboten, in der man einfach SyncML Nachrichten aufbaut und diese werden dann automatisch in Pakete unterteilt. Weitere Arbeiten sollten untersuchen, wie eine mögliche Schnittstelle und Funktionsweise aussehen könnte, dass sich eine Implementierung nur mit dem Nachrichtenablauf beschäftigt, aber keine weiteren Gedanken über SyncML Pakete machen muss.

14.6. Zwei-Wege Abgleich

Wie bereits im Abschnitt 11.2 beim Nachrichtenablauf erwähnt, sollte untersucht werden, ob Zwei-Wege Abgleiche nötig sind und wie der Nachrichtenablauf und besonders die Aktualisierung der Vektordaten in diesem Fall aussehen sollte.

14.7. Java 2 Micro Edition

Um die Dateimengen (Arbeitspeicher, Dateispeicher, Transportmengen) von Enkel DS besser einschätzen zu können, sollte eine vollständig CLDC 1.0 kompatible Version entstehen. Aufgrund der Verwendung von Sync4j als Framework kann der tatsächliche minimale Arbeitspeicherverbrauch und die Dateispeicher nur schwer eingeschätzt werden.

15. Fazit

Vektorzeitpaare zeigen sehr deutlich eine Vormachtstellung gegenüber anderen Abgleichverfahren auf. Es werden Zyklen in der Netzwerktopologie unterstützt aber nicht die Speicherplatzprobleme von Abgleichverfahren mit nur einem Vektor für jeden Datensatz übernommen. Darüber hinaus bieten Vektorzeitpaare Unterstützung für Abbrüche und verbesserte Konflikterkennungen. [CJ05] zeigt Tests von Vektorzeitpaaren bezüglich des Speicherplatzverbrauchs und der Dauer eines Abgleichs. Gerade die Dauer des Abgleich und die Menge an zu übertragenden Daten sollte mit anderen Abgleichverfahren weiter untersucht werden. Es sollten geeignete Szenarien entwickelt werden (durchschnittliche Verzeichnistiefe, Anzahl an Datensätzen) und Häufigkeit des Datenabgleichs.

Allerdings sind Vektorzeitpaare nur schwer mit anderen Verfahren vergleichbar, da deren Mächtigkeit höher ist. Eine ständige Slow Sync als Möglichkeit Zyklen im Netzwerk zu erlauben, erzeugt bei verschiedenen Datensätzen Duplikate oder bei Verwendung von eindeutigen Datensatz IDs Konflikte. Beides tritt in Vektorzeitpaaren nicht auf. Vektorzeitpaare sind daher mächtiger als eine Slow Sync.

Ein ganz anderes Problem ist die Umsetzbarkeit von einem Abgleichverfahren.

IrMC Level 4 ist vergleichsweise schnell und einfach zu implementieren. Wenn bereits eine OBEX Implementierung vorliegt, dann sind nur die entsprechenden Pakte, Informationsdaten und Log Dateien aufzubauen. Zusätzlich ist ein Parser für vCard, vCalendar und vNote nötig.

OMA SyncML DS ist vergleichsweise komplizierter. Es wird auf der Transportebene OBEX, WSP und HTTP benötigt. HTTP benötigt wieder TCP/IP. Für die Nachrichten sind WBXML und XML Parser sinnvoll. In einem SyncML Klienten zum Beispiel einem Mobiltelefon existieren bereits TCP/IP, HTTP, OBEX und WBXML Parser, da diese Protokolle für andere Funktionalitäten im Mobiletelefon bereits benötigt werden (WAP 2.0 und IrDA bzw. Bluetooth). Zusätzlich muss der Nachrichtenablauf programmiert werden. Dieser ist vergleichsweise einfach und es können vorgefertigte WBXML Dateien genutzt werden, die nur noch befüllt werden müssen.

Enkel DS benötigt nur einen neuen Nachrichtenablauf, allerdings sind die Aktualisierungen der Vektordaten nicht trivial und leicht verständlich. Wenn Vektorzeitpaare nicht

verstanden werden, werden diese auch nicht weiter untersucht oder finden gar nicht erst in die Industrie. Es besteht daher die Hoffnung, dass diese Ausarbeitung Vektorzeitpaare in ihrem Vorgehen und mit ihren Möglichkeiten verständlicher machen konnte.

Literaturverzeichnis

App03a Apple: *Adressbuch*. http://www.apple.com/de/macosx/features/addressbook/. Version: 2003

App03b *Apple iSync*. https://www.apple.com/de/isync/. Version: 2003

BCG+00 Butrico, Maria ; Cohen, Norman ; Givler, John S. ; Mohindra, Ajay ; Purakayastha, Apratim ; Shea, Dennis G. ; Cheng, Josephine M. ; Clare, Don ; Fisher, Gerry ; Scott, Rob ; Sun, Yudong ; Wone, May ; Zondervan, Quinton: Enterprise Data Access from Mobile Computers: An End-to-end Story IEEE Computer Society (International Workshop on Research Issues in Data Engineering, 10), 9-16

BD95 Beuter, Thomas ; Dadam, Peter: Prinzipien der Replikationskontrolle in verteilten Systemen / Universität Ulm, Fakultät für Informatik. Version: November 1995. http://www.informatik.uni-ulm.de/dbis/01/dbis/downloads/UIB-1995-11.pdf (UIB-1995-11). – Ulmer Informatik-Berichte. – Online–Ressource

Ber01 Berger, Maximilian: *LibSyncML*. http://libsyncml.sourceforge.net/. Version: Dezember 2001. – C++ Implementierung eines OMA SyncML DS 1.0 Servers

Ber02 Berger, Maximilian: *Integrated PIM data management with SyncML*. Fakultät für Informatik, Informatik XI: Angewandte Informatik / Kooperative Systeme, Technische Universität München, Diplomarbeit, Juni 2002. http://max.berger.name/syncml/. – Online–Ressource

Ber04 Berger, Maximilian: *SyncML support for OGo*. http://www.opengroupware.org/en/projects/syncml/. Version: 2004. – Objective-C Implementierung eines OMA SyncML DS 1.1 Servers

BLFSM05 Berners-Lee, T. ; Fielding, R. ; Software, Day ; Masinter, L.: *Uniform Resource Identifiers (URI): Generic Syntax*. Network Working

Group: Internet Engineering Task Force (IETF), Januar 2005. `http://www.ietf.org/rfc/rfc3986.txt`

Bou00 *Bouncy Castle Crypto*. `http://www.bouncycastle.org/`. Version: 2000

Bou03 Bougues, Nicolas: *SyncML tools*. `http://nicolas.bougues.net/syncml/`. Version: Mai 2003. – PHP Implementierung eines OMA SyncML DS 1.0 Servers

BT01 Bluetooth Special Interest Group: *Synchronizaton Profile*. 1.1. Februar 2001. `https://www.bluetooth.org/spec/`

BT05 Bluetooth Special Interest Group, Printing Working Group: *Basic Printing Profile*. 1. Februar 2005. `https://www.bluetooth.org/spec/`

Buc02 Buchmann, David: *SyncML and its Java Implementation Sync4j*. Telecom Research Group, Universität Freiburg, Schweiz, Diplomarbeit, 2002. `http://sync4j.funambol.com/RandD/David%20Buchmann.pdf`. – Online–Ressource

Byo93 *Bayou*. `http://www.parc.xerox.com/bayou/`. Version: 1993

CFF05 Capobianco, Fabrizio ; Fassina, Luigia ; Fornari, Stefano: *Sync4j*. `http://www.sync4j.org/`. Version: März 2005. – Java Implementierung eines OMA SyncML DS Servers und Klienten

CGSB02 Campana, J. ; Gmelin, M. ; Schöchlin, J. ; Bolz, A.: XML-based synchronization of mobile medical devices, Universität Karlsruhe, Institut für Biomedizinische Technik (Biomedizinische Technik 47 2), 857-859

CJ02 Cox, Russ ; Josephson, William: *File System Synchronization Using Vector Time Pairs*. `http://web.archive.org/web/20031205084442/pdos.lcs.mit.edu/~rsc/tra/osdi.pdf`. Version: Mai 2002

CJ03 *tra*. `http://web.archive.org/web/20031205084442/pdos.lcs.mit.edu/~rsc/tra/`. Version: 2003

CJ04 Cox, Russ ; Josephson, William: *Optimistic Replication Using Vector Time Pairs*. `http://www.pdos.lcs.mit.edu/6.824/papers/tra.pdf`. Version: 2004

CJ05 Cox, Russ ; Josephson, William: File Synchronization with Vector Time Pairs / MIT Laboratory for Computer Science. Version: Februar 2005. http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TM-650.pdf (MIT-LCS-TM-650). – Memo. – Online–Ressource

Cod87 *Coda*. http://www.coda.cs.cmu.edu/. Version: 1987

Coh00 Cohen, Norman H.: Design and implementation of the MNCRS Java framework for mobile data synchronization / IBM Research. Version: August 2000. http://www.research.ibm.com/sync-msg/RC21774.pdf (RC 21774). – Forschungsbericht. – Online–Ressource. Beinhaltet Designentscheidungen von MNCRS

CP00 Cohen, Norman H. ; Purakayastha, Apratim: Toward Interoperable Data Synchronization with COSMOS IEEE Computer Society (IEEE Workshop on Mobile Computing Systems and Applications 3), 138-150

Cri03 Crispin, M.: *Internet Message Access Protocol (IMAP4)*. Network Working Group: Internet Engineering Task Force (IETF), März 2003. http://www.ietf.org/rfc/rfc3501.txt

CVS85 *Concurrent Versions System (CVS)*. http://www.cs.vu.nl/~dick/CVS.html. Version: 1985

DBMP01 Dahlin, Mike ; Brooke, Aslan ; Muralidhar ; Porter, Narasimhan B.: Data Synchronization for Distributed Simulations / University of Texas at Austin Department of Computer Sciences. Version: Juni 2001. http://www.uxxi.org/docs/siso.pdf. – Forschungsbericht. – Online–Ressource

DH98 Dawson, F. ; Howes, T.: *vCard MIME Directory Profile*. 3.0. Network Working Group: Internet Engineering Task Force (IETF), September 1998. – vCard 3.0. http://www.ietf.org/rfc/rfc2426.txt

DS98 Dawson, F. ; Stenerson, D.: *Internet Calendaring and Scheduling Core Object Specification (iCalendar) (RFC 2445)*. 2.0. Network Working Group: Internet Engineering Task Force (IETF), November 1998. http://www.ietf.org/rfc/rfc2445.txt

Dum03 Dumbill, Edd: *SyncML toolkits*. http://www-106.ibm.com/developerworks/xml/library/x-syncml3.html. Version: Juni 2003. – Übersicht von SyncML Implementierungen

EMP+97 Edwards, W. K. ; Mynatt, Elizabeth D. ; Petersen, Karin ; Spreitzer, Mike J. ; Terry, Douglas B. ; Theimer, Marvin M.: Designing and implementing asynchronous collaborative applications with Bayou. In: *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, ACM Press, 1997. – ISBN 0897918819, S. 119–128

ETS00 European Telecommunications Standards Institute: *Wide Area Network Synchronization*. 3.1.0. Oktober 2000. – Basiert auf IrMC. `http://webapp.etsi.org/key/key.asp?GSMSpecPart1=27&GSMSpecPart2=103`

ETS01 Discussion of Synchronization Standards. Version: 4.0.0, März 2001. `http://webapp.etsi.org/key/key.asp?GSMSpecPart1=27&GSMSpecPart2=903` (3GPP TR 27.903). – Marktanalyse. – Online–Ressource. Vergleich zwischen IrMC und SyncML

ETS02 European Telecommunications Standards Institute: *Wide Area Network Synchronization*. 5.0.0. Juni 2002. – Basiert auf SyncML 1.0. `http://webapp.etsi.org/key/key.asp?GSMSpecPart1=27&GSMSpecPart2=103`

ETS03 European Telecommunications Standards Institute: *Specification of the Subscriber Identity Module - Mobile Equipment Interface*. 8.9.1. Juni 2003. `http://webapp.etsi.org/key/key.asp?GSMSpecPart1=11&GSMSpecPart2=11`

ETS04a European Telecommunications Standards Institute: *AT command set for User Equipment*. 5.5.0. Dezember 2004. `http://webapp.etsi.org/key/key.asp?GSMSpecPart1=27&GSMSpecPart2=007`

ETS04b European Telecommunications Standards Institute: *AT command set for User Equipment*. 6.7.0. Dezember 2004. – Version 6 mit erweiterter Adressbuchunterstützung in +CPBx. `http://webapp.etsi.org/key/key.asp?GSMSpecPart1=27&GSMSpecPart2=007`

ETS04c European Telecommunications Standards Institute: *USIM and IC card requirements*. 6.1.0. Juni 2004. `http://webapp.etsi.org/key/key.asp?GSMSpecPart1=21&GSMSpecPart2=111`

FB96 Freed, N. ; Borenstein, N.: *Multipurpose Internet Mail Extensions (MIME)*. Network Working Group: Internet Engineering Task Force (IETF), November 1996. `http://www.ietf.org/rfc/rfc2045.txt`

FGM+99 Fielding, R. ; Gettys, J. ; Mogul, J. ; Frystyk, H. ; Masinter, L. ; Leach, P. ; Berners-Lee, T.: *Hypertext Transfer Protocol (HTTP).* 1.1. Network Working Group: Internet Engineering Task Force (IETF), Juni 1999. http://www.ietf.org/rfc/rfc2616.txt

Fid91 Fidge, Colin: Logical Time in Distributed Computing Systems. In: *Computer* 24 (1991), Nr. 8, S. 28–33. http://dx.doi.org/10.1109/2.84874. – DOI 10.1109/2.84874. – ISSN 0018–9162

For04 Fornari, Stefano: *Sync4j Architecture.* 1.21. : , August 2004

GGM+03 Greenwald, Michael B. ; Gunden, Owen ; Moore, Jonathan T. ; Pierce, Benjamin C. ; Schmitt, Alan: *Harmony: A Synchronization Framework For Tree-Structured Data.* http://www.cis.upenn.edu/~bcpierce/papers/harmonyslides-2003aug.pdf. Version: September 2003. – Vortrag

Hac01 Hacklin, Fredrik: *A 3G Convergence Strategy for Mobile Business Middleware Solutions*, Helsinki University of Technology, Marktanalyse, September 2001. http://www.hacklin.com/fredrik/hacklin/thesis/. – Online–Ressource. – 67–76 S. – Kapitel 3.5 gibt einen Überblick von Dateisystemen für mobile Geräte

HMPT03 Hansmann, Uwe ; Mettälä, Riku ; Purakayastha, Apratim ; Thompson, Peter: *SyncML: Synchronizing and managing your mobile data.* Pearson Education, 2003. – ISBN 0130093696

Hof94 Hofmann, Richard: Kausalität und Zeit in parallelen und Verteilten Systemen, H. Wedekind, 1994 (Verteilte Systeme), S. 79–106

HPGP92 Heidemann, John S. ; Page, Thomas W. ; Guy, Richard G. ; Popek, Gerald J.: Primarily Disconnected Operation: Experiences with Ficus. In: *Proceedings of the Second Workshop on Management of Replicated Data* University of California, Los Angeles, IEEE, 2–5

HS01 Holdrege, M. ; Srisuresh, P.: *Protocol Complications with the IP Network Address Translator (NAT).* Network Working Group: Internet Engineering Task Force (IETF), Januar 2001. http://www.ietf.org/rfc/rfc3027.txt

IAN05 The Internet Corporation for Assigned Names and Numbers (IANA): *Port numbers*. Februar 2005. `http://www.iana.org/assignments/port-numbers`

IMA99 *Terminvereinbarung unter mobilen Benutzern*. `http://www.informatik.uni-stuttgart.de/ipvr/vs/lehre/ws9899/studienprojekt/index.html#Aufgabe`. Version: 1999

Int03 Internet Engineering Task Force (IETF): *The Base16, Base32, and Base64 Data Encodings*. Juli 2003. `http://www.ietf.org/rfc/rfc3548.txt`

IR94 *Infrared Data Association (IrDA)*. `http://www.irda.org/`. Version: 1994

ITF01 Internet Engineering Task Force (IETF): *Simple Mail Transfer Protocol (SMTP)*. 1.1. April 2001. `http://www.ietf.org/rfc/rfc2821.txt`

ITU01 International Telecommunication Union: *Serial asynchronous automatic dialling and control (ITU-T V.250) - Supplement 1: Various extensions to V.250 basic command set*. Juni 2001. `http://www.itu.int/ITU-T/publications/recs.html`

ITU03 International Telecommunication Union: *Serial asynchronous automatic dialling and control (ITU-T V.250)*. Juli 2003. `http://www.itu.int/ITU-T/publications/recs.html`

KWK03 Kang, Brent B. ; Wilensky, Robert ; Kubiatowicz, John: The Hash History Approach for Reconciling Mutual Inconsistency IEEE Computer Society (International Conference on Distributed Computing Systems 23), 670-678

LKC04 Lee, YoungSeok ; Kim, YounSoo ; Choi, Hoon: Conflict Resolution of Data Synchronization in Mobile Environment, Springer-Verlag GmbH. – ISBN 3540220569, 196-205

LLW03 Lam, Franky ; Lam, Nicole ; Wong, Raymond: Update Synchronization for Mobile XML Data / University of New South Wales, School of Computer Science & Engineering. Version: Juni 2003. `ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/0310.pdf` (UNSW-CSE-TR-0310). – Forschungsbericht. – Online–Ressource

Loc04 Lockhart, Rob: *E-Mail Korrespondenz.* mailto:rob.lockhart@systeklabs.com. Version: August 2004

LSS00 Lockhart, Robert K. ; Scales, James ; Stossel, John: *Specifications for Ir Mobile Communications (IrMC).* 1.1+. P.O. Box 3883, Walnut Creek, California, U.S.A. 94598: Infrared Data Association, November 2000. – IrMC 1.1 inklusive aller Änderungen bis November 2000. http://www.irda.org/

Mat89 Mattern, Friedemann: Verteilte Basisalgorithmen. In: *Springer-Verlag* (1989). http://www.inf.ethz.ch/~mattern/. ISBN 3540518355

MD97 Munson, Jonathan P. ; Dewan, Prasun: Sync: A Java Framework for Mobile Collaborative Applications. In: *Computer* 30 (1997), Nr. 6, 59–66. http://www.cs.unc.edu/~dewan/sync/. – Java Implementierung eines Abgleichverfahrens. – ISSN 0018–9162

Mob99 Mobile Network Computing Reference Specification Consortium: *Mobile Network Computing Reference Specification (MNCRS).* 1.1. März 1999. http://web.archive.org/web/19990418100051/www.mncrs.org/

Mro01 Mrowczynski, Mirko: *Synchronisation von Terminplanern mittels XML*, TU Chemnitz, Fakultät für Informatik, Diplomarbeit, November 2001. http://www.misterm.de/diplom/. – Online–Ressource. – Überblick über SyncML Implementierungen

MSK03 Megowan, Pat ; Suvak, Dave ; Kogan, Doug: *Object Exchange Protocol (OBEX).* 1.3+. P.O. Box 3883, Walnut Creek, California, U.S.A. 94598: Infrared Data Association, April 2003. – OBEX 1.3 inklusive aller Änderungen bis April 2003. http://www.irda.org/

Mus02 Musolesi, Mirco: *XMIDDLE*, University College London, Dept. of Computer Science, Diplomarbeit, 2002. http://www.cs.ucl.ac.uk/staff/m.musolesi/tesi.pdf. – Online–Ressource

MyS94 *MySQL.* http://www.mysql.de/. Version: 1994

Oks01 Oksyuk, Oleg: *kSync.* http://ksync.objectweb.org/. Version: Dezember 2001. – Java Implementierung eines OMA SyncML 1.0 Servers und Klienten

OMA00a *SyncML Initiative*. http://www.syncml.org/. Version: 2000

OMA00b Open Mobile Alliance: *Wireless Application Protocol*. 1.2.1. Juni 2000. http://www.openmobilealliance.org/tech/affiliates/wap/wapindex.html

OMA01 Open Mobile Alliance: *Binary XML Content Format Specification (WBXML)*. 1.3. Juli 2001. http://www.openmobilealliance.org/tech/affiliates/wap/wap-192-wbxml-20010725-a.pdf

OMA03a Open Mobile Alliance: *Data Synchronization (OMA SyncML DS)*. 1.1.2. Juni 2003. http://www.openmobilealliance.org/release_program/ds_v112.html

OMA03b Open Mobile Alliance: *SyncML Common Specification (OMA SyncML)*. 1.1.2. Juni 2003. http://www.openmobilealliance.org/release_program/SyncML_v112.html

OMA04a Open Mobile Alliance: *SyncML C Reference Toolkit*. http://sourceforge.net/projects/syncml-ctoolkit/. Version: Mai 2004. – C Implementierung der SyncML Initiative

OMA04b *White Paper on OMA SyncML Data Synchronisation Changes*. http://member.openmobilealliance.org/ftp/Public_documents/DS/2004/OMA-DS-2004-0166-WP-DataSync-Changes.zip. Version: November 2004

OMA04c Open Mobile Alliance: *Data Synchronization (OMA SyncML DS)*. 1.2. Juni 2004. http://www.openmobilealliance.org/release_program/ds_v12.html

OMA04d Open Mobile Alliance: *Device Management (OMA SyncML DM)*. 1.1.2. Januar 2004. http://www.openmobilealliance.org/release_program/dm_v112.html

OMA04e Open Mobile Alliance: *SyncML Common Specification (OMA SyncML)*. 1.2. Juni 2004. http://www.openmobilealliance.org/release_program/SyncML_v12.html

Ope97 The Open Group: *Universal Unique Identifier*. 1997. http://www.opengroup.org/onlinepubs/9629399/apdxa.htm

Pab02 Pabla, Chandandeep: *SyncML intensive*. `http://www-106.ibm.com/developerworks/xml/library/wi-syncml2/`. Version: April 2002. – Einstieg und umfassender kurzer OMA SyncML DS Überblick

PHP97 *PHP*. `http://www.php.net/manual/en/`. Version: 1997

PSYC03 Paluska, Justin M. ; Saff, David ; Yeh, Tom ; Chen, Kathryn: Footloose: A Case for Physical Eventual Consistency and Selective Conflict Resolution. In: *5th IEEE Workshop on Mobile Computing Systems and Applications*. IEEE Computer Society Press, 170–180

PV98 *Unison*. `http://www.cis.upenn.edu/~bcpierce/unison/`. Version: 1998

PV04 Pierce, Benjamin C. ; Vouillon, Jérôme: What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer / Dept. of Computer and Information Science, University of Pennsylvania. Version: 2004. `http://www.cis.upenn.edu/~bcpierce/papers/unisonspec.pdf` (MS-CIS-03-36). – Forschungsbericht. – Online–Ressource

RC01 Ramsey, Norman ; Csirmaz, Előd: An algebraic approach to file synchronization. In: *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM Press. – ISBN 1581133901, 175–185

Rea02 *Reaxion*. `http://www.reaxion.com/news_28022002.html`. Version: Februar 2002. – Pressemitteilung zu kSync

Rov95 *Rover*. `http://www.pdos.lcs.mit.edu/rover/`. Version: 1995

RRP97 Ratner, D. ; Reiher, P. ; Popek, G.: Dynamic Version Vector Maintenance / University of California. Version: Juni 1997. `http://fmg-www.cs.ucla.edu/roam98/welcome.html`. Computer Science Department, Juni 1997 (CSD-970022). – Forschungsbericht. – Online–Ressource

rsy93 *rsync*. `http://rsync.samba.org/`. Version: 1993

RU01 Roth, Jörg ; Unger, Claus: Using Handheld Devices in Synchronous Collaborative Scenarios. In: *Personal Ubiquitous Comput.* 5 (2001), Nr. 4, 243–252. `http://dx.doi.org/10.1007/s007790170003`. – DOI 10.1007/s007790170003. – ISSN 1617–4909

Sai02 Saito, Yasushi: Unilateral version vector pruning using loosely synchronized clocks / Hewlett-Packard Labs. Version: März 2002. `http://www.hpl.hp.com/techreports/2002/HPL-2002-51.pdf`. Storage Systems Department, März 2002 (HPL-2002-51). – Forschungsbericht. – Online–Ressource

Sca00 Scales, James: *SyncML Over Bluetooth*. `https://www.bluetooth.org/foundry/sitecontent/document/SyncML/en/1/SyncML.pdf`. Version: August 2000

SD99 Scales, James ; Dawson, Frank: *External Standards Impact On The IrMC Specification*. `http://www.3gpp.org/ftp/tsg_t/WG2_Capability/TSGT2_06/Docs/t2-99830.doc`. Version: 1999. – Übergang von IrMC zu SyncML

SEP03 *Sony Ericsson P800 – GSM Mobiltelefon*. `http://www.sonyericsson.com/p800/`. Version: 2003

SKW+02 Swierk, Edward ; Kiciman, Emre ; Williams, Nathan C. ; Fukushima, Takashi ; Yoshida, Hideki ; Laviano, Vince ; Baker, Mary: The Roma personal metadata service, Kluwer Academic Publishers, 407-418

SNT04 Suel, Torsten ; Noel, Patrick ; Trendafilov, Dimitre: Improved File Synchronization Techniques for Maintaining Large Replicated Collections over Slow Networks. In: *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, IEEE Computer Society, 2004. – ISBN 0769520650, S. 153–164

Sos04 Sosnoski, Dennis M.: *JiBX: Binding XML to Java Code*. `http://www.jibx.org/`. Version: 2004

Sou SourceForge.net: *Sync4j*. `http://sourceforge.net/mailarchive/forum.php?forum=sync4j-users`. – Mailingliste für Sync4j bei SourceForge.net

STA02 Starobinski, David ; Trachtenberg, Ari ; Agarwal, Sachin: On the Scalability of Data Synchronization Protocols for PDAs and Mobile Devices. In: *IEEE Network (Special Issue on Scalability in Communication Networks)* 16 (2002), August, Nr. 4, 2228. `http://people.bu.edu/staro/pda_review.pdf`. – Vergleich zwischen verschiedenen Protokollen

STA03 Starobinski, David ; Trachtenberg, Ari ; Agarwal, Sachin: Efficient PDA Synchronization. In: *IEEE Transactions on Mobile Compu-*

ting 2 (2003), März, Nr. 1, 40-51. `http://ipsit.bu.edu/documents/efficient_pda_web.pdf`

Sun00a Sun Microsystems: *Connected Limited Device Configuration (CLDC).* `http://java.sun.com/products/cldc/overview.html`. Version: 2000

Sun00b Sun Microsystems: *Java 2 Platform, Micro Edition (J2ME).* `http://java.sun.com/j2me/`. Version: 2000

Sun04 Java Community Process: *PDA Optional Packages for the J2MET Platform (JSR-75).* `http://www.jcp.org/en/jsr/detail?id=75`. Version: Juni 2004. – FileConnection und Personal Information Manager API

Sun05 Java Community Process: *Data Sync API (JSR-230).* 0.4. Januar 2005. – Aktuell existiert nur ein Entwurf. `http://www.jcp.org/en/jsr/detail?id=230`

SZ92 Sandhu, Harjinder S. ; Zhou, Songnian: Cluster-based file replication in large-scale distributed systems. In: *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, ACM Press, 1992. – ISBN 0897915070, S. 91–102

Tan03 Tanenbaum, Andrew S.: *Computernetzwerke.* Pearson, 2003. – ISBN 3827370469

TTP+95 Terry, D. B. ; Theimer, M. M. ; Petersen, Karin ; Demers, A. J. ; Spreitzer, M. J. ; Hauser, C. H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, ACM Press, 1995. – ISBN 0897917154, S. 172–182

USB01 Universal Serial (USB) Bus Implementers Forum: *Class Definitions for Communication Devices (CDC) Subclass Specification for Wireless Mobile Communication Devices (WMC).* 1.0. November 2001. `http://www.usb.org/developers/devclass_docs`

ver96a versit Consortium: *vCalendar - The Electronic Calendaring and Scheduling Exchange Format.* 1.0. September 1996. – vCalendar 1.0. `http://www.imc.org/pdi/pdiproddev.html`

ver96b versit Consortium: *vCard - The Electronic Business Card*. 2.1. September 1996. – vCard 2.1. `http://www.imc.org/pdi/pdiproddev.html`

W3C04a *XML Binary Characterization Working Group*. `http://www.w3.org/XML/Binary/`. Version: 2004

W3C04b World Wide Web Consortium (W3C): *Extensible Markup Language (XML)*. 3. Februar 2004. `http://www.w3.org/TR/REC-xml/`

WPS99 Wiesmann, M. ; Pedone, F. ; Schiper, A.: A Systematic Classification of Replicated Database Protocols based on Atomic Broadcast. In: *Proceedings of the 3rd Europeean Research Seminar on Advances in Distributed Systems (ERSADS'99)*. Madeira Island, Portugal, April 1999

XMI01 *XMIDDLE*. `http://xmiddle.sourceforge.net/`. Version: 2001

Yah Yahoo! Groups: *SyncML*. `http://groups.yahoo.com/group/SyncML/`. – Mailingliste für Sync4j bei Yahoo!

Glossar

Innerhalb der Ausarbeitung werden eine Reihe von Begriffe verwendet, die sich auf den ersten Blick inhaltlich kaum unterschieden. Die folgenden Erklärungen dienen als Nachschlagewerk für die genaue Bedeutung dieser Begriffe innerhalb dieser Ausarbeitung.

Abgleichverfahren Verfahren um Datenkopien zu erstellen und zu pflegen. Wobei die Bedeutung von Pflegen durch das Verfahren definiert wird. Einige Verfahren versuchen alle Kopien auf dem gleichen, aktuellen Stand zu halten und Konflikte wieder in Einklang zu bringen.

Datenabgleichverfahren siehe Abgleichverfahren.

Gerät Physische Einheit, die eigenständig Berechnungen durchführt und über einen Datenspeicher verfügt. Beispiele wären ein PC, Mobiltelefon oder PDA. Ein Gerät kann mehrere Kopien eines Datenbestands besitzen. In der Regel besitzt ein Gerät aber nur eine Kopie. Es kann nur eine oder mehrere Versionen einer Datenkopie besitzen. Dies ist vom Abgleichverfahren abhängig.

Knoten Kopie innerhalb einer Netzwerktopologie, wobei die Topologie logische Verbindungen beschreibt, zum Beispiel: Kann ein Gerät mehrere Kopien besitzen, dann kann ein Gerät mehrere Knoten umfassen.

Kopie Stellt eine Version dar (siehe Gerät). In der Regel wird das Erstellen einer aktuellsten Kopie eines Datenbestands angestrebt.

Replik siehe Kopie.

Synchronisationsverfahren siehe Abgleichverfahren.

Version Abbild eines Datenbestandes (eines Datensatzes oder der gesamten Datenbank) zu einem bestimmten Zeitpunkt.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Ulm, den 31. März 2005

Alexander Traud