Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3424

Konzepte und Mechanismen zur konsistenten nebenläufigen Aktualisierung der Weiterleitungstabellen in Software-defined Networks

Moritz von Pein

Studiengang: Informatik

Prüfer: Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer: Dr. rer. nat. Frank Dürr
begonnen am: 12.11.2012
beendet am: 14.05.2013

CR-Klassifikation: C.2.1, C.2.4

Abstract

In Software-defined Networks existieren zwei wichtige Gründe, die Kontrollebene auf mehrere Controller zu verteilen. Erstens ist es für die Skalierbarkeit des Systems wichtig, Zuständigkeiten flexibel auf mehrere Controller zu verteilen und zweitens muss ein Ausfall eines Controllers durch andere Controller toleriert und aufgefangen werden können. Außerdem ist eine Verteilungstransparenz der Kontrolllogik erwünscht, die es Kontrollanwendungen ermöglicht, Routen ohne große Kenntnisse des Netzwerks zu schreiben. Bei der Aktualisierung von Weiterleitungstabellen durch mehrere Controller treten durch asynchrone Kommunikation und konkurrierende Updates Inkonsistenzen in den Weiterleitungstabellen auf, die eine Koordination des Ablaufs nötig machen. Inkonsistenzen entstehen dadurch, dass innerhalb kurzer Zeit zwei Schreibaufträge für konkurrierende Routen gestartet werden und diese in unterschiedlicher Reihenfolge bei den Switchs verarbeitet werden, wodurch beide Flows unvollständig implementiert werden und Schleifen entstehen können.

In dieser Arbeit wird eine verteilte Control Coordination Middleware vorgestellt, die zwei Mechanismen anbietet, durch die diese Inkonsistenzen verhindert werden. In der Locking-Variante muss ein Agent, der einen Flow einrichtet, erst eine Sperre über das Matching-Kriterium beantragen, bevor er die Updates verschicken darf. In der Logical-Clock-Variante werden die Updates mit Zeitstempeln versehen, die es den Switchs ermöglichen, veraltete Updates zu verwerfen.

Inhaltsverzeichnis

1 Einführung 9

2 Grundlagen und Verwandte Arbeiten 11
2.1 Software-Defined Networking 11
2.2 OpenFlow 12
2.3 Transaktionsverarbeitung in verteilten Datenbanken 15
2.3.1 ACID-Eigenschaften 17
2.3.2 Eine-Kopie-Serialisierbarkeit 19
2.3.3 Zwei-Phasen-Commit 20
2.3.4 Locking-Verfahren 21
2.3.5 Deadlocks 21
2.3.6 Write-Ahead Log Protokoll 22
2.4. Ansätze für eine Kontrollebene mit mehreren Controllern 24
2.4.1 HyperFlow 24
2.4.2 Onix 26
2.4.3 Kandoo 29
2.5 Konsistente Routen-Aktualisierung 30
2.5.1 Frenetic 30
2.5.2 Konsistente Netzwerkaktualisierung bei Migration von virtuellen Maschinen 33
2.5.3 OpenFlow Safe Update Protokoll 34

3 Systemmodell und Problembeschreibung 37
3.1 Systemmodell 37
3.1.1 Datenebene 38
3.1.2 Kontrollebene 38
3.1.3 Anwendungsebene 40
3.2 Problembeschreibung 40
3.2.1. Konkurrierende Updates und Flows 40
3.2.2 Inkonsistenzen in Weiterleitungstabellen 44
3.3 Ziel dieser Arbeit 46
3.3.1 Konsistenz 46
3.3.2 Isolation 46
3.4 Weitere Eigenschaften der Aktualisierung 48
3.4.1 Atomizität 48
3.4.2 Dauerhaftigkeit 48

4 Entwurf 49
4.1 Architektur der Middleware 49
4.2 Locking-Agent 50
4.3 Logische-Uhr-Agent 55
4.4 Vergleich von Locking- und Logische-Uhr-Agenten 59

5 Implementierung 61
5.1 Übersicht 61
5.2 Inititiator 62
5.3 Locking-Service 62
5.4 Update-Agenten 63
5.5 Controller 64

6 Evaluation 65
6.1 Versuchsaufbau 65
6.2 Durchsatz 66

7 Zusammenfassung und Ausblick 69
7.1 Zusammenfassung 69
7.2 Ausblick 70

Literaturverzeichnis 71

1 Einführung

Die Belastungen des Internets wachsen immens, genauso wie dessen Bedeutung für die Wirtschaft und dessen Weiterentwicklung. Immer mehr mobile Geräte kommunizieren sowohl untereinander als auch mit großen Datenservern und produzieren dabei immer größere Datenmengen, die durch das Netz transportiert werden müssen. Benutzer speichern immer mehr Daten auf öffentlichen Portalen wie Google, Facebook oder Dropbox, die diese dann von Datenservern für eine ausgewählte Anzahl von Personen von überall zugänglich machen. Cloud-Dienste bieten Anwendungen an, die über das Internet kommunizieren und aus unterschiedlichen Modulen zusammengesetzt werden können.

Um für die wachsenden Anforderungen gerüstet zu sein, sucht die Industrie nach neuen Wegen, Netzwerke zu optimieren und flexibel zu konfigurieren. Optimierung wird durch die immer größer werdenden Datenmengen notwendig und Flexibilität wird benötigt, da z.B. Unternehmen Dienste wie private Clouds anbieten, die genau auf die Bedürfnisse des Kunden zugeschnitten werden müssen. Diese Anwendungen setzen sich aus unterschiedlichen Komponenten zusammen, die miteinander kommunizieren und bei denen sich der Nachrichtenverkehr dynamisch verändert, wobei dies schwer vorauszusehen ist. Die Anforderungen können sich hier über Nacht verändern und oft ist es erforderlich, dass ein Dienst innerhalb kurzer Zeit umgestaltet werden muss. Dies erfordert von einem Netzwerk, dass es flexibel und schnell modifizierbar ist, was bei der großen Komplexität heutiger Netzwerke eine zeitaufwendige und kostenintensive Aufgabe ist.

Software-Defined Networking [6] (SDN) bietet Werkzeuge zur Flexibilisierung und Optimierung von Netzen und basiert auf zwei Grundkonzepten. Erstens wird klar zwischen Daten- und Kontrollebene getrennt, um eine klare Trennung zwischen den beiden Ebenen zu haben, und zweitens wird die Kontrollebene logisch zentralisiert mit einer globalen Sicht über das Netz realisiert. In traditionellen Netzwerken war die Kontrollebene bisher auf den Switchs verteilt implementiert, die immer nur eine lokale Sicht von dem Netzwerk hatten. Bei SDN-Netzwerken hat die Kontrollebene Zugriff auf die Weiterleitungstabellen der Switchs und durch die globale Sicht über das Netz mehr Möglichkeiten, die richtigen Maßnahmen zur Optimierung zu treffen.

Für die Kommunikation zwischen Daten- und Kontrollebene wird ein gemeinsames standardisiertes Protokoll zwischen Kontrollebene und Datenebene wie z.B. OpenFlow[4,5,6] eingeführt, das die Kommunikation mit Switchs unabhängig vom Hersteller ermöglicht. Von den Switchs wird nur gefordert, dass sie die Nachrichten der Controller verarbeiten können und dadurch die Weiterleitungstabellen der Switchs durch die Controller zugänglich gemacht werden. Für die Skalierbarkeit

und Fehlertoleranz von größeren Netzwerken ist es nötig, die Kontrollebene auf mehrere Controller zu verteilen. Hier stellt sich die Frage nach der Konsistenz, wenn mehrere Controller nebenläufig die Tabelleneinträge der Switchs aktualisieren. Switchs können dadurch von mehreren Controllern Updates erhalten, die auf unterschiedlichen Routen basieren und die gleichen Pakete betreffen.

In dieser Arbeit soll ein formales Modell definiert werden, mit dem nebenläufige Updates der Weiterleitungstabellen und der Begriff der Konsistenz in einem Netzwerk beschrieben werden kann. Mit Hilfe des Modells sollen Mechanismen entwickelt werden, mit denen die Konsistenz in dem Netzwerk bei einer verteilten Kontrollebene garantiert werden kann, falls unterschiedliche Routen komplett installiert werden.

In Kapitel 2 werden Grundlagen erläutert, die für das Verständnis der Arbeit erforderlich sind. Außer SDN-Netzwerken und OpenFlow wird auf Transaktionen in verteilten Datenbanken eingegangen, da es hier viele Ähnlichkeiten zu dem Schreibvorgang eines Flows gibt. Außerdem werden verwandte Ansätze vorgestellt, die sich einerseits mit einer verteilten Kontrollebene oder andererseits mit der konsistenten Routenaktualisierung in einem Netzwerk mit einem Controller beschäftigten. In Kapitel 3 werden im Systemmodell die Komponenten definiert und ein formales Modell vorgestellt, über das Inkonsistenzen beschrieben werden können. Außerdem wird gezeigt, welche Fehler im Netzwerk durch sie entstehen können. In Kapitel 4 werden zwei Mechanismen entwickelt, die Konsistenz nach einer abgeschlossenen Aktualisierung im Netzwerk garantieren. Danach wird in Kapitel 5 eine Implementierung der entwickelten Mechanismen vorgestellt, mit Hilfe dieser in Kapitel 6 untersucht wird, wie viele Routen-Aktualisierungen pro Minute möglich sind. In Kapitel 7 wird die Arbeit noch zusammengefasst und ein Ausblick auf mögliche Weiterentwicklungen gegeben.

2 Grundlagen und Verwandte Arbeiten

In diesem Kapitel werden die Grundlagen von Software-defined Networking erläutert und OpenFlow als standardisiertes Kommunikationsprotokoll vorgestellt. Außerdem wird auf Transaktionsverarbeitung in verteilten Datenbanken eingegangen, da hier eine Verwandtschaft zu der Aktulisierung von Weiterleitungstabellen besteht. Am Ende werden noch Ansätze vorgestellt, die sich einerseits mit einer verteilten Kontrollebene, andererseits mit konsistenter Aktualisierung von Weiterleitungstabellen in Netzwerken mit einem Controller befassen.

2.1 Software-Defined Networking

Die Architektur des Software-Defined Networking (SDN) [6] wurde entwickelt, um Netzwerke den gestiegenen Anforderungen der heutigen Zeit anzupassen und durch programmierbare Switchs eine Grundlage für Innovationen zu schaffen. Bei der SDN-Architektur wurde die Kontrollebene, die bei traditionellen Netzwerken verteilt durch die Switchs implementiert wird, als eigenständige Komponente herausgearbeitet, die klar von der Datenebene getrennt und logisch zentralisiert wird. Die Kontrollebene besitzt eine globale Sicht auf das Netzwerk und kann auf den Switchs die Regeln für die Weiterleitung der Pakete ändern, indem sie Konfigurationsnachrichten an diese schickt. Zusätzlich kann der Status des Netzwerks überwacht werden und Überlastungen entdeckt werden, indem bei den Switchs ihr Status und die Anzahl der weitergeleiteten Pakete abgefragt wird.

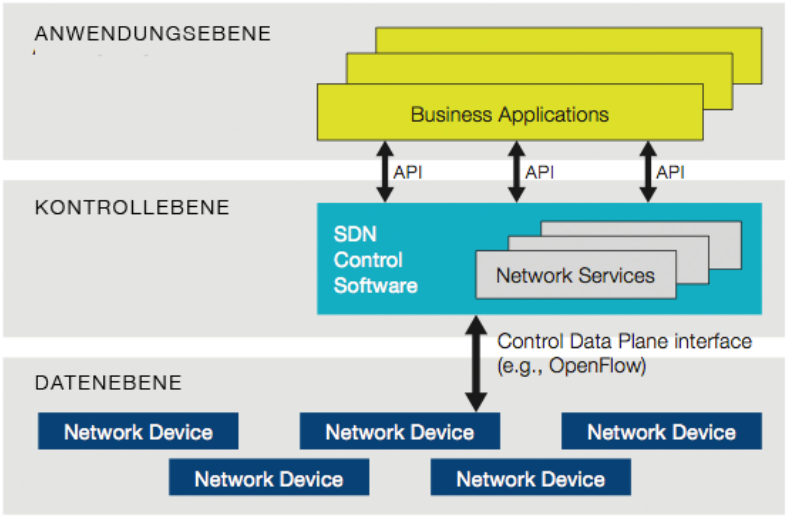


Abb. 2.1: Logischer Aufbau eines SDN-Netwerkes [6]

In Abb. 2.1 sieht man den logischen Aufbau eines SDN-Netzwerks mit der Anwendungs-, Kontroll- und Datenebene, wobei in letzterer sich die Infrastruktur des Netzes mit den Switchs befindet. In der Kontrollebene sitzen die Controller, die über die Switchs mit ihrem Wissen und der globalen Sicht über das Netzwerk Eingriffe in das Netzwerk ausführen können. Anwendungen, die bisher die Wahl einer Route nicht beeinflussen konnten, können nun über die Kontrollebene Routen schreiben, die an ihre Anforderungen angepasst sind und den Status des Netzwerks berücksichtigen. Die Northbound-Kommunikation erfolgt abhängig von den Controllern und kann an die Bedürfnisse der Anwendung angepasst werden. So ist es für eine Anwendung möglich über eine Schnittstelle eine Route im gesamten Netzwerk zu schreiben, wobei die Controller die Kommunikation mit den Switchs übernehmen. Die Southbound-Kommunikation läuft über das standardisierte OpenFlow Protokoll, was bei der Konfiguration von Switchs unterschiedlicher Hersteller eine Vereinfachung bedeutet, da hier nicht mehr mehrere unterschiedliche herstellerspezifische Protokolle berücksichtigt werden müssen.

Der größte Vorteil ist aber die Flexibilität, das Verhalten des Netzwerks in Echtzeit verändern zu können und neue Anwendungen oder Dienste innerhalb weniger Stunden oder Tage zu installieren. Routen können schrittweise im Netzwerk geschrieben werden, wobei die Kontrollebene über nur ein Protokoll mit den Switchs kommuniziert. Solche Umstellungen benötigen in herkömmlichen Netzwerken Wochen oder Monate und bei komplexeren Änderungen kann es auch nötig sein, dass die Hersteller der Switchs kontaktiert werden mussten, um die gewünschten Veränderungen in der nächsten Generation der Geräte einzubauen, was mehrere Jahre dauern konnte. Zusätzlich kann das Netzwerk durch die globale Sicht besser optimiert werden und Gegenmaßnahmen können vorgenommen werden, bevor überhaupt Probleme wie Überlastungen auftreten.

Die Open Network Foundation[13] hat dabei das OpenFlow-Protokoll standardisiert und achtet darauf, dass Switchs unterschiedlicher Hersteller untereinander kompatibel bleiben. Durch SDN gewinnen Netzwerke an Unabhängigkeit und müssen sich weniger an Anwendungen anpassen. Stattdessen dient die Kontrollebene als Vermittler zwischen Anwendungs- und Datenebene.

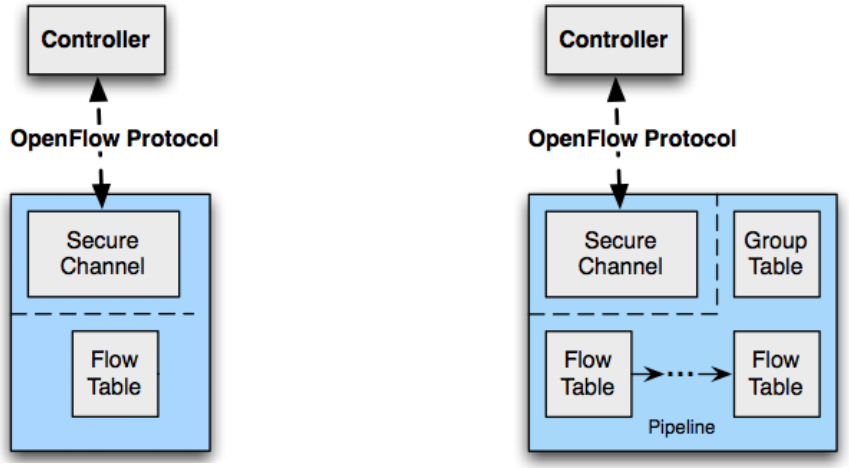
2.2 OpenFlow

Als erstes standardisiertes Kommunikationsprotokoll zwischen Kontroll- und Weiterleitungsebene wurde OpenFlow [4,5,6] entwickelt. OpenFlow ermöglicht das Schreiben von Weiterleitungstabellen von Switchs mit Hilfe von definierten Nachrichtenformaten unabhängig vom Hersteller dieser. Durch OpenFlow kann die Kontrolle über das Netzwerk auf eine logische zentralisierte Kontrollebene übertragen werden. Flows definieren Gruppen von Paketen und über die Zugehörigkeit eines Pakets zu einem Flow kann ein Switchs entscheiden, wie er dieses weiterleitet. Kennzeichen jedes Flows ist das Matching-Kriterium, das Header-Informationen aus Layer 2-4 verwendet, um zu entscheiden, ob ein Paket in dem Flow enthalten ist und deswegen wie alle anderen Pakete des Flows behandelt

werden muss.

Ein Switch benutzt Flow-Tabellen, in denen gespeichert wird, welche Aktionen mit einem Paket ausgeführt werden. Jeder Flow-Tabelleneintrag besteht dabei aus Matching-Kriterium und zugehöriger Aktion. Eine Aktion wäre z.B die Ausgabe des Packets an einem bestimmten Ausgangsport oder dass ein Paket verworfen wird. Über den Ausgangsport wird ein Paket an einen anderen Switch übertragen und zusammen mit der Konnektivität des Netzes der Weg eines Pakets durch das Netzwerk bestimmt. Der Switch untersucht ein ankommendes Paket auf Zugehörigkeit zu einem Flow und wendet die in der Flow-Tabelle zugewiesenen Aktionen darauf an. Geschrieben und verändert werden die Flow-Tabelleneinträge durch den Controller, der mit dem Switch über einen Kanal verbunden ist und mit Hilfe des OpenFlow-Protokoll mit ihm kommuniziert. In Abb. 2.2 sieht man, dass bei Switchs, die das OpenFlow-Protokoll V1.1.0 verwenden, mehrere Flow-Tabellen eingeführt wurden, die in einer Pipeline angeordnet sind. Der Matching-Prozess startet immer in der ersten Tabelle und falls keine Übereinstimmung gefunden wird, kann das Paket entweder an die nächste Flow-Tabelle übergeben, gelöscht oder auch an den Controller geschickt werden, was von den Einstellungen des Switchs abhängt. Durch das Schicken des Pakets an den Controller wird dieser darüber benachrichtigt, dass es keine Übereinstimmungen mit den Matching-Kriterien in der Flow-Tabelle eines Switch hat, was oft ein Hinweis auf eine unvollständige Installation eines Flows im Netzwerk ist. Er kann dann die notwendigen Schritte einleiten, was z.B. bedeutet, dass er einen Flow noch einmal neu installiert. Die in der Tabelle gespeicherten Instruktionen können unterschiedliche Auswirkungen auf die Pakete haben. Pakete können modifiziert, an einen Port ausgegeben oder auch an eine später folgende Flow-Tabelle weitergeleitet werden.

In Abb. 2.3 sieht man ein Beispiel für eine Flow-Tabelle eines Switch mit Einträgen zu bestimmten Matching-Kriterien. In den ersten beiden Spalten können Pakete nach ihrer Quell- oder Ziel-MAC-Adresse ausgewählt werden. In den darauf folgenden Spalten wird die IP-Adresse als Filterkriterium

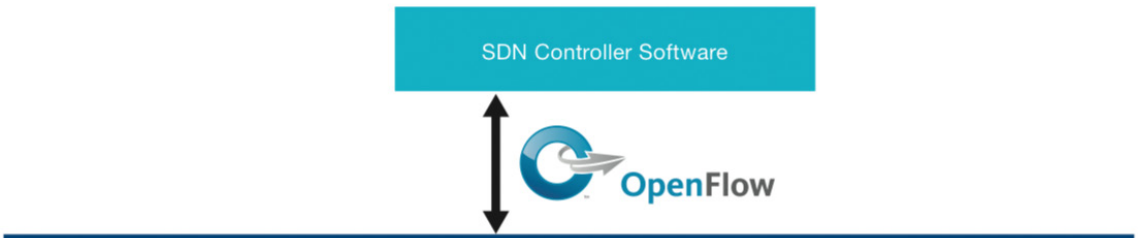


(a) Open Flow V1.0 (b) Open Flow V1.1.0

Abb. 2.2: Organisation eines OpenFlow-Switchs [5]

verwendet und in der fünften Spalte können Pakete nach ihrem Ziel-TCP-Port ausgewählt werden. Bei Einträgen, die nicht relevant fürs Matching sind, sind Wildcards gesetzt, die signalisieren, dass hier die Header-Informationen der Pakete nicht übereinstimmen müssen. Um strengere Kriterien einzuführen, kann man auch mehrere Matching-Kriterien vorgeben, die alle erfüllt sein müssen.

In der vorletzten Spalte wird dann den ausgewählten Paketen eine Aktion zugewiesen, die definiert, wie mit den Paketen verfahren wird. Wenn die Pakete an ein anderes Gerät weitergeleitet werden sollen, wird ihnen ein Ausgangsport zugewiesen, der mit einem anderem Gerät verbunden ist, wie zum Beispiel in den ersten beiden Zeilen. In der dritten Zeile hingegen werden alle Pakete mit dem drop-Befehl gelöscht, die an TCP-Port 25 adressiert sind. In der nächsten Zeile werden alle Pakete, die von IP-Adressen ab 192.0.0.0 verschickt wurden, lokal auf dem Netzwerk-Stapel des Switchs gespeichert, was eine weitere optionale Aktion eines OpenFlow-Switch ist. Dadurch wird eine Interaktion ermöglicht, die über das normale OpenFlow-Netzwerk anstatt über ein getrenntes Controller-Netzwerk läuft. Und in der letzen Zeile werden alle Pakete, für die bisher kein Matching erkannt wurde, an den Controller weitergeleitet. Für die Überwachung des Netzwerks durch die Controller spielt die letzte Spalte eine wichtige Rolle. Hier wird für jede Zeile ein Zähler gespeichert, der anzeigt, wie viel Pakete bisher über das Matching-Kriterium weitergeleitet wurden. Dadurch kann der Controller die Auslastung eines Switchs abhängig von Paketen überprüfen und bei einer Überlastung Maßnahmen ergreifen, den Switch zu entlasten.



MAC src	MAC dst	IP Src	IP Dst	TCP dport	...	Action	Count
*	10:20:.	*	*	*	*	port 1	250
*	*	*	5.6.7.8	*	*	port 2	300
*	*	*	*	25	*	drop	892
*	*	*	192.*	*	*	local	120
*	*	*	*	*	*	controller	11

Abb. 2.3: Beispiel für eine OpenFlow-Weiterleitungstabelle [6]

2.3 Transaktionsverarbeitung in verteilten Datenbanken

Konsistente Aktualisierung wurde in dem Bereich der verteilten Datenbanken schon intensiv untersucht und da hier viele Ähnlichkeiten mit der Aktualisierung von Weiterleitungstabellen existieren, werden Begriffe aus der Datenbanktheorie in dieser Arbeit übernommen. Deswegen wird hier auf die Grundbegriffe von Transaktionsverarbeitung eingegangen und Definitionen erklärt, die später verwendet werden.

Transaktionen sind in der realen Welt Vorgänge, bei denen zwischen unterschiedlichen Teilnehmern wie z.B. einer Person und einem Unternehmen etwas ausgetauscht wird. Sobald es dabei um kompliziertere und wirtschaftliche Vorgänge handelt, erfordert dies eine Buchhaltung, die heutzutage meistens von einem Computer durchgeführt wird. Ein Beispiel für eine Transaktionsverarbeitung (TP)[12] ist der Ablauf einer geschäftlichen Transaktion zwischen zwei Computern, die über das Internet verbunden sind. Wichtig dabei ist immer, dass der Zugriff auf die Datenbanken koordiniert wird, da hier anwendungs- und kundenspezifische Daten gespeichert werden. Falls hier z.B. Fehler beim Berechnen von Konto- oder Lagerbeständen gemacht werden, kann großer wirtschaftlicher Schaden entstehen. In der Informatik laufen Transaktionen auf vielen Ebenen ab und insbesondere dann, wenn zwischen unterschiedlichen Komponenten einer Software, Übereinkünfte getroffen werden müssen. Ob diese Unterkünfte von zwei lokalen Komponenten auf einem Rechner oder von zwei Diensten auf unterschiedlichen Kontinenten getroffen werden, ist dabei für die logische Betrachtung nicht wirklich relevant.

In Abb. 2.4 sieht man die Komponenten, in die ein TP-System aufgeteilt ist und die jeweils eine bestimmte Aufgabe erfüllen. Der Endbenutzer beauftragt das Ausführen einer Transaktion wie z.B. das Bestellen eines Buches oder eine Überweisung bei einer Bank. Dabei kann der Bestellvorgang von einem Webbrowser gestartet werden, der die Daten darstellt und bei der Bestätigung der Bestellung einen Auftrag an das Front-End Programm weiterleitet, das entweder auf einem anderen Server läuft oder aber in die Maschine integriert ist. Meistens ist das Front-End-Programm eine Anwendung, die auf einem WebServer sitzt und mit dem Browser des Endbenutzer über HTTP kommuniziert. Das Front-End-Programm überprüft die Eingaben des Benutzers und leitet den Auftrag weiter an der Request Controller. Die Weiterleitung erfolgt entweder direkt oder über einen Queue, in dem die Requests gespeichert und schrittweise abgearbeitet werden. Der Request Controller generiert aus dem Auftrag mehrere einzelne Transaktionen, die in einer bestimmten Reihenfolge auf möglicherweise unterschiedlichen Transaktions-Servern ausgeführt werden. Hier überwacht der Request Controller die erfolgreiche Ausführung der einzelnen Transaktionen und leitet die Teilergebnisse weiter. Ein Transaktions-Server ist ein Prozess, der die Anfragen einer lokalen oder verteilten Datenbank verwaltet und kontrolliert, so dass die gewünschten Änderungen in der Datenbank auch durchgeführt werden. Falls erforderlich kann er Werte aus der Datenbank ausgeben, die vom Request-Controller verarbeitet und weitergegeben werden. Die Datenbank besteht aus einem Datenbank-System, das die Schreibaufträge für eine spezielle Datenbank verwaltet, und der

Datenbank, in der die Daten gespeichert werden.

Im Softwarebereich gibt es mittlerweile Produkte, die diese Architektur in einer transaktionalen Middleware realisiert haben und die Kommunikation zwischen TP-Anwendungen und Komponenten auf niedrigeren Ebenenen erleichtern. Diese Middleware reduziert die Komplexität für die Anwendung, indem z.B. Anfragen für unterschiedliche Betriebssystem übersetzt werden und nach dem effizientesten Prozess eines Betriebssystems gesucht wird. Im Vergleich zu traditionellen Datenbankservern ist die Middleware flexibler, wenn es um das Skalieren und Erweitern von Datenbanksystemen geht, da Mechanismen zur Verfügung gestellt werden, mit deren Hilfe man zusätzliche Datenbanken einbauen kann.

Um nebenläufige Transaktionen und ihre Eigenschaften genauer untersuchen zu können, wurde eine abstrahierte Schreibweise entwickelt. Dabei wird bei einer Transaktion die Reihenfolge von Schreib- und Lesevorgängen dokumentiert und wann eine Transaktion mit Commit beendet wird.

Beispiel für zwei Transaktionen:
$T_1 = r_1[x]\ r_1[y]\ w_1[x]\ c_1$
$T_2 = r_2[y]\ r_2[x]\ w_2[y]\ c_2$

In T_1 wird zuerst die Variablen x und y gelesen und dann ein berechneter Wert in der Variable x gespeichert. Bei T_2 werden die Variablen x und y in anderer Reihenfolge gelesen und dann auf y ein Wert gespeichert. Wichtig für die in den Variablen gespeicherten Werte ist nun die Reihenfolge, in der die einzelnen Schritte ausgeführt werden. Dazu betrachtet man einen Schedule der beiden Transaktionen

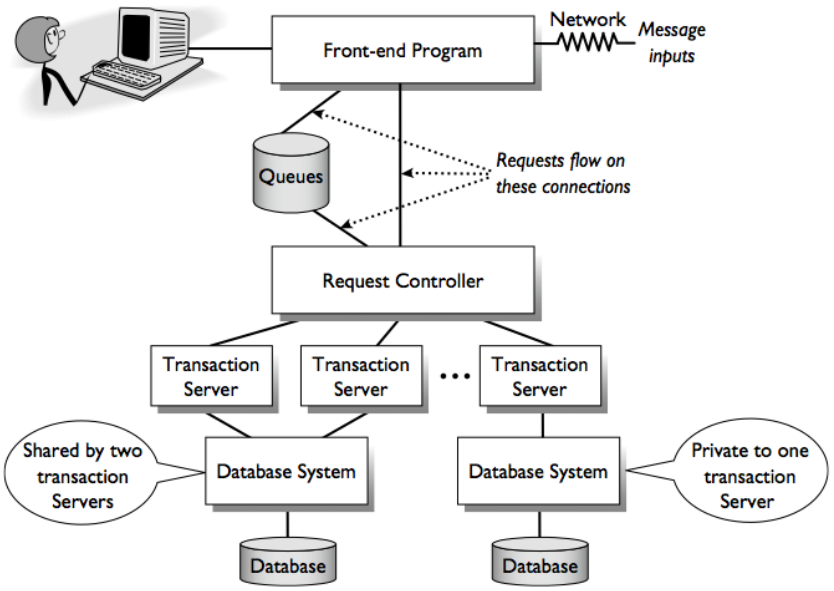


Abb. 2.4: Architektur einer TP-Anwendung [12]

Beispiel für zwei Schedules:

$S_1 = r_1[x]\ r_1[y]\ w_1[x]\ c_1\ \ r_2[y]\ r_2[x]\ w_2[y]\ c_2 = T_1\ T_2$

$S_2 = r_1[x]\ r_1[y]\ r_2[y]\ r_2[x]\ w_1[x]\ c_1\ w_2[y]\ c_2$

Der Schedule S_1 entspricht einer seriellen Ausführung der Transaktionen, in der zuerst alle Opertationen von T_1 und dann alle von T_2 ausgeführt werden. Im zweiten Schedule S_2 hingegen werden die Operationen nebenläufig ausgeführt und die Leseoperationen von T_2 starten schon, bevor T_1 den Wert in x gespeichert hat und beendet wurde. Dies bedeutet, dass die Berechnungen von T_1 nicht in der Berechnung von T_2 berücksichtigt werden und am Ende der beiden Schedules ein unterschiedlicher Werte in y durch T_2 gespeichert werden kann. Falls zwei Schedules die gleiche Auswirkung auf die Variablen haben, spricht man von zwei equivalenten Schedules. In dem Beispiel unterscheiden sich aber die Berechnung der Variablen, deswegen sind S_1 und S_2 nicht equivalent.

2.3.1 ACID-Eigenschaften

Um Eigenschaften von Transaktionen zu definieren, benutzt man die Begriffe Atomizität, Konsistenz, Isolation und Dauerhaftigkeit. Falls ein TP-System alle Eigenschaften erfüllt, spricht man von ACID-Transaktionen, wobei jeweils der erste Buchstabe des entsprechenden englischen Begriffs genommen wurde.

2.3.1.1 Atomizität

Als erstes garantiert die Atomizität, dass eine Transaktion ganz oder gar nicht ausgeführt wird (all-or-nothing). Zum Beispiel muss bei einer Transaktion von 100€ von einem Konto A auf ein Konto B sicher gestellt sein, dass der Betrag entweder bei beiden Kontos abgezogen bzw. hinzugefügt wurde oder von keinem. Es darf auf keinen Fall passieren, dass der Betrag zum Beispiel nur von Konto A abgezogen, aber nicht Konto B hinzugefügt wurde.

Das TP-System garantiert Atomizität dadurch, dass sie die Ausführung jedes Schrittes einer Transaktion überprüft und reagiert, falls einer der Transaktionen fehlschlägt. Dann müssen alle Transaktion, die davor schon abgeschlossen wurden, wieder rückgängig gemacht und die Datenbanken in den Zustand überführt werden, den sie vor dem Start der Transaktion hatten. Dieser Vorgang wird das Kompensieren von Transaktionen genannt und bei komplexeren TP-Anwendungen sollte jede Transaktion eine entsprechende kompensierende Transaktion enthalten, um das Abbrechen von Transaktionen zu erleichtern. Bei einer erfolgreichen Durchführung einer Transaktion spricht man von einem *Commit*, bei einer fehlgeschlagenen von einem *Abort*.

2.3.1.2 Konsistenz

Als zweite Eigenschaft erfordert die Konsistenz, dass eine Transaktion eine Datenbank aus einem konsistenten wieder in einen neuen konsistenten Zustand überführt. Bei der Konsistenz einer

Datenbank spricht man von bestimmten internen Einschränkungen wie zum Beispiel, dass alle Primary Keys einzigartig bleiben müssen oder dass nur auf existierende Objekte verwiesen wird. Im Gegensatz zu Atomazität, Isolation und Dauerhaftigkeit bezieht sich Konsistenz auf eine Vereinbarung zwischen den Transaktionsprogrammen und den TP-Systemen untereinander und es liegt in der Verantwortung des Anwendungsprogrammierers, dass das Programm die Konsistenz in der Datenbank erhält.

2.3.1.3 Isolation

Die Isolation als dritte Eigenschaft ist gegeben, falls Transaktionen, die gleichzeitig ausgeführt werden, auf das System die gleichen Auswirkungen haben, als wenn man sie nacheinander ausführen würde. Bei einer Kombination von Transaktionen spricht man deswegen auch von der Serialisierbarkeit. Eine Ausführung ist serialisierbar, falls deren Ergebnis identisch mit dem Ergebnis mit einem seriellen, schrittweisen Abarbeiten der Transaktionen ist, ohne dass sich zwei Transaktionen überschneiden.

Zur Veranschaulichung wird wieder ein Beispiel aus dem Bankgewerbe betrachtet, bei dem zwei Parteien von einem Konto 100€ abheben wollen, auf dem sich genau noch 100€ befindet und dass nicht überzogen werden darf. Falls beide Parteien erst den Kontostand abfragen, bevor sie das Geld abheben, und dies zum gleichen Zeitpunkt passiert, nehmen beide Parteien an, dass sich genug Geld auf dem Konto befindet und übertragen jeweils 100€ auf ihr Konto. Offensichtlich ist das ein für die Bank unerwünschtes Ergebnis und entspricht auch nicht dem Ergebnis, das erfolgen würde, falls beide Anwendungen strikt hintereinander ausgeführt werden. Die erste Anwendung würde nämlich die 100€ abheben und die zweite würde leer ausgehen. Anhand dieses Beispiels kann man auch den Unterschied zur Atomizität feststellen, da bei der fehlerhaften Ausführung die Atomizität im Gegensatz zur Isolation gewahrt wäre, da bei beiden Anwendungen alle Transaktionen komplett ausgeführt würden.

Einem Endbenutzer erscheint es durch die Isolation, dass er der einzige Teilnehmer ist, der gerade mit dem TP-System kommuniziert und Transaktionen startet. In der Realität gibt es natürlich in jedem TP-System viele parallele Anfragen und viele Transaktionen müssen parallel ablaufen, um effizientes Abarbeiten zu ermöglichen. Um Isolation zu gewährleisten, arbeiten TP-Systeme mit Sperren, bei denen verhindert wird, dass andere Anwendungen auf Werte zugreifen können, die noch nicht bestätigt wurden. Bei unserem Beispiel würde die Anwendung erst eine Sperre über das Konto beantragen, bevor es den Kontostand lesen würde, so dass die andere Anwendung erst gar nicht den Kontostand lesen könnte und warten müsste, bis die erste Anwendung das Konto wieder freigegeben hat.

2.3.1.4 Dauerhaftigkeit

Als letzte Eigenschaft bedeutet Dauerhaftigkeit, dass alle von Transaktionen durchgeführten Änderungen auf stabilem Speicher dauerhaft gespeichert werden, so dass sie nicht durch einen

Absturz des Betriebssystems verloren gehen können. Stabiler Speicher sind heutzutage Festplatten oder Flash-Karten, auf die zugegriffen werden kann, wenn das Betriebssystem wieder hochfährt. So kann auch bei einem Absturz einer unvollständigen Transaktion nachvollzogen werden, welche Schritte schon durchgeführt wurden und welche nicht.

TP-Systeme realisieren Dauerhaftigkeit, indem sie während einer Transaktion eine Log-Datei schreiben, in der alle fertig gestellten Schritte auf stabilem Speicher gespeichert werden. Bevor ein Transaktionsprogramm ein finales Commit herausschickt, überprüft es, ob die Log-Datei wirklich gespeichert wurde und damit alle Änderungen dauerhaft sind.

2.3.2 Eine-Kopie-Serialisierbarkeit

Die Eine-Kopie-Serialisierbarkeit [12] ist als Erweiterung zur Serialisierbarkeit formuliert worden, falls eine Datenbank verteilt realisiert wird und Duplikate von Dateneinträgen auf unterschiedlichen Instanzen existieren. In diesem Fall ist es für mache Anwendungen wichtig, dass bei einem Schreibvorgang wirklich alle Kopien aktualisiert werden, da sonst veraltete Werte gelesen werden. Eine Ausführung von Transaktionen ist Eine-Kopie-serialisierbar, falls das Ergebnis identisch mit dem Ergebnis einer nicht verteilten Datenbank wäre, in der keine Duplikate von Einträgen existieren. In einem Beispiel mit drei Transaktionen T_1,T_2 und T_3 und zwei Kopien von x, die auf zwei unterschiedlichen Orten A und B gespeichert werden, wird gezeigt, welche Probleme auftreten können:

$$T_1 = r_1[x_A]\ w_1[x_A]\ w_1[x_B]\ c_1$$
$$T_2 = r_2[x_B]\ w_2[x_B]\ c_2$$
$$T_3 = r_3[x_A]\ w_3[x_A\ ]\ w_3[x_B]\ c_3$$

Die drei Transaktionen setzen sich aus Lese-, Schreib- und Bestätiguns-Operationen zusammen. Bei der Transaktion T_1 wird zuerst von Speicherort A gelesen, danach auf beiden Kopien geschrieben und mit einem Commit bestätigt. In T_2 wird x auf Speicherort B gelesen und dann auch nur auf B wieder abgespeichert. Dies kann daran liegen, dass die Ressource B gerade nicht verfügbar war und bei der Ausführung der Transaktion aber nicht gewartet werden konnte. Deswegen wurde die Transaktion trotzdem bestätigt. Im nächsten Schritt lädt T_3 den Wert von x nur von der Ressource A, führt mit diesem eine Berechnung durch und speichert den neuen Wert auf beiden Orten ab. Deshalb wird der Wert, den T_2 gespeichert hatte, nicht berücksichtigt und wieder von T_3 überschrieben. In einem Szenario, in dem die Variable x ohne Duplikate nur auf einem Ort gespeichert ist, wäre bei der Berechnung von x die Transaktion T_2 berücksichtigt worden, und das Endergebnis der Transaktionen kann sich unterscheiden. Die Transaktionen sind zwar serialisierbar, aber erfüllen nicht die Voraussetzung für die Eine-Kopie-Serialisierbarkeit.

Falls man sicher bei seinen Transaktionen die Eine-Kopie-Serialisierbarkeit erreichen will, ist eine offensichtliche Lösung, dass jede Transaktion bei einem Schreibvorgang die Werte aller Duplikate einer Variable aktualisieren muss. Doch dies ist in einer verteilten Umgebung oft nicht möglich,

da manchmal manche Ressourcen nicht erreichbar sind und das Warten die Anwendung zu sehr verzögern würde. Ein anderer Ansatz wäre, dass Transaktionen immer den frischesten Wert einer Variable lesen, da in diesem der aktuellste Wert abgespeichert wurde.

2.3.3 Zwei-Phasen-Commit

Falls bei einer Transaktion zwei oder mehr Ressourcen wie zum Beispiel Datenbanksysteme im Spiel sind, ist es nötig neue Verfahren einzuführen, um Atomizität garantieren zu können. Denn falls eine Datenbank eine Transaktion abbricht, darf diese auch auf den anderen Ressourcen nicht ausgeführt werden. Um die Transaktionen zu koordinieren, benötigt man ein Modul namens Transaktionsmanager, welches mit Hilfe des Zwei-Phasen-Commits-Protokolls sicher stellt, dass alle oder keine Operationen der Transaktionen durchgeführt werden. Grundidee dabei ist, dass die Bestätigung einer Transaktion in zwei Phasen unterteilt wird. Die erste Phase wird vom Transaktionsmanager gestartet, indem er allen Teilnehmern eine Nachricht schickt, damit diese die Transaktion vorbereiten und eine Kopie der Updates auf stabilen Speicher sichern. Nachdem dies bei allen Teilnehmer erfolgreich durchgeführt und der Transaktionsmanager von allen Teilnehmern benachrichtigt wurde, startet dieser die zweite Phase, in der die Transaktion umgesetzt wird. Sobald aber der Transaktionsmanager in der ersten Phase ein negatives Feedback eines Teilnehmers erhält, der die Transaktion nicht durchführen kann, muss er die anderen Teilnehmer über den Abbruch der Transaktion benachrichtigen und alle machen die Transaktion rückgängig. Wichtig dabei ist auch, dass in der ersten Phase eine Kopie der Updates auf stabilem Speicher gesichert wird, denn falls ein Teilnehmer abstürzt, kann er auch immer Inhalte des Hauptspeichers verlieren. Dadurch dass er eine Kopie auf stabilem Speicher besitzt, kann er bei einem Neustart überprüfen, ob ein unvollständig durchgeführtes Update vorhanden ist und ob der Transaktionsmanager noch über eine vorbereitetes Update benachrichtigt werden muss.

Als Koordinator hat der Transaktionsmanager eine wichtige Rolle und kommuniziert mit den Ressourcemanagern, die jeweils für die Durchführungen der Transaktion auf einer Ressource

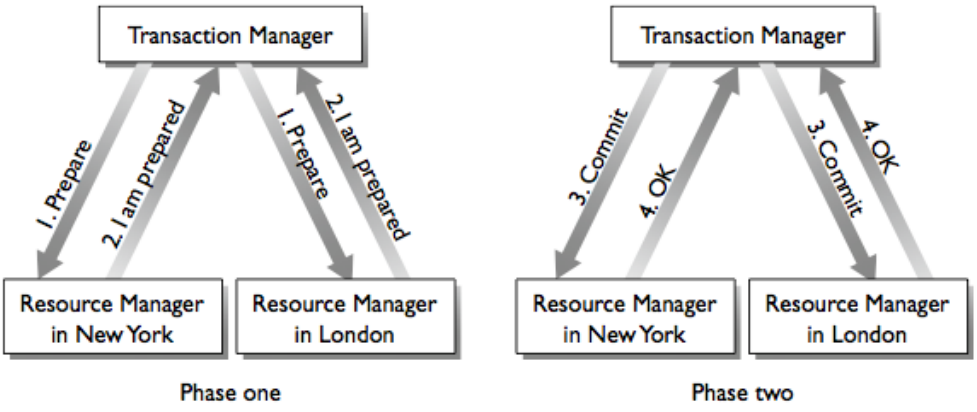


Abb. 2.5: Ablauf des Zwei-Phasen-Commit-Protokolls [12]

zuständig sind. In einem verteilten System sitzt in der Regel auf jedem Knoten ein Transaktionsmanager, der jeder Transaktion beim Start einen eindeutigen Identifikator zuweist, damit keine Verwechslungen zustande kommen. Außerdem muss der Transaktionsmanger informiert werden, falls von Teilnehmern zusätzliche Ressourcenmanager in eine Transaktion involviert werden, da er in der zweiten Phase wirklich alle teilnehmenden Ressourcenmanager über ein Commit benachrichtigen muss. Ein Ressourcenmanager, der auf ein Commit wartet und nicht benachrichtigt wird, macht die Änderungen wieder rückgängig und die Atomizität der Transaktion ist nicht mehr gewährleistet. In Abb. 2.5 sieht man einen Transaktionsmanager, der eine Transaktion mit zwei Ressourcen steuert. In der ersten Phase wird die Ressource vorbereitet, in der zweiten Phase wird das endgültige Commit verschickt und vom Ressourcemanager bestätigt.

2.3.4 Locking-Verfahren

Falls in einem verteilten System mehrere Transaktionen auf unterschiedliche Daten zugreifen, müssen Verfahren angewendet werden, um die Isolation der Transaktionen zu gewährleisten, was gleichbedeutend damit ist, dass der Effekt der Transaktion gleichbedeutende mit einer seriellen Abfolge von Transaktionen ist. Das am weitesten verbreitete Verfahren dafür ist Locking[12], bei dem Transaktionen über alle Daten, auf die sie zugreifen wollen, Sperren beantragen. Dabei unterscheidet man zwischen Schreibsperren für zu schreibende Daten und Lesesperren für zu lesende Daten. Eine Transaktion kann eine Lesesperre für Daten setzen, falls noch keine aktive Schreibsperre existiert. Eine Schreibsperre kann nur gesetzt werden, falls weder eine Lesesperre noch eine Schreibsperre aktiv ist. Man sagt auch, eine Lesesperre steht in Konflikt mit einer Schreibsperre und eine Schreibsperre steht sowohl im Konflikt mit einer Lese- als auch Schreibsperre. Ein Konflikt zwischen zwei Operationen existiert, falls sie mit den gleichen Daten arbeiten und davon mindestens einer dieser Operationen eine Schreiboperation ist.

Um beim Locking die Serialisierbarkeit zu erhalten, ist es wichtig, dass Transaktionen die Sperren erst aufheben, wenn Sie schon alle Sperren erhalten haben. Man spricht deshalb vom Zwei-Phasen-Locking, bei in der ersten Phase die Sperren beantragt und in der zweiten Phase die Sperren wieder aufgehoben werden. Es darf nicht erlaubt sein, eine Sperre zu beantragen, falls man schon ein Sperre aufgehoben hat.

2.3.5 Deadlocks

Falls mehrere Transaktionen Sperren für die gleichen Datenquellen beantragen, kann es zu Konflikten führen, wodurch Transaktionen blockiert werden. Falls zwei Transaktionen aufeinander warten, dass sie ihre Transaktion abschließen und die Sperre auflösen, können deswegen beide Transaktionen nicht mehr weiterverarbeitet werden und man spricht von einem Deadlock[12].

In dem Beispiel in Abb. 2.6 sieht man zwei Transaktionen, die beide eine Sperre über x und y

beantragen wollen. Die Transaktion T_1 hat zuerst eine Sperre über x beantragt und erhalten, die Transaktion T_2 hingegen über y. Im zweiten Schritt wollen beide Transaktionen nun eine Sperre über die andere Quelle erhalten, wobei dies bei beiden aber abgelehnt wird. Beide Transaktionen warten jetzt, bis die andere Transaktion beendet wird. Solange nun nicht von einer externen Partei eingegriffen und eine Transaktion abgebrochen wird, werden beide Transaktionen blockiert sein.

Deadlocks zu verhindern ist sehr schwierig und führt dazu, dass Transaktionen fast nur seriell ausgeführt werden können. Deswegen ist es effektiver, Deadlocks zu entdecken, falls sie auftreten und dann aufzulösen. Dabei gibt es zwei unterschiedliche Techniken, die benutzt werden: Timeout-basierte Entdeckung und Graphen-basierte Entdeckung. Bei der Timeout-basierten Entdeckung wird einfach die Zeit begrenzt, die eine Transaktion dauern kann und falls ein Timeout nach z.B. 10 sec noch nicht beendet ist, wird sie abgebrochen. Dadurch wird bei Deadlocks eine Transaktion beendet und die andere Transaktion kann auf die Ressource zugreifen. Der Vorteil ist, dass dieser Ansatz relativ leicht umzusetzen ist, der Nachteil, dass manchmal Transaktionen abgebrochen werden, bei denen kein Deadlock besteht und dass Deadlocks, die z.B. gleich nach 2 sec auftreten, erst relativ spät aufgelöst werden. Bei den Graphen-basierten Methoden werden die Abhängigkeiten zwischen wartenden Transaktionen in Graphen gespeichert und regelmäßig auf Zyklen untersucht. Wie schnell ein Deadlock entdeckt wird, ist abhängig davon, wie oft die Graphen untersucht wird, aber im Gegensatz zu Timeout-basierten Methode sind alle entdeckten Deadlocks wirkliche Deadlocks.

2.3.6 Write-Ahead Log Protokoll

Von transaktionsverarbeitenden Systemen wird erwartet, dass sie jeden Tag 24 Stunden verfügbar sind und die gewünschten Operationen ausführen können. Da aber nie ausgeschlossen werden kann, dass Fehler auftreten, werden Mechanismen benötigt, wie man im Fall von Abstürzen oder Ausfällen für die Korrektheit der Daten garantieren kann. Dabei sollte die Zeit, die das System benötigt, um den Fehler zu erkennen und zu reparieren, möglichst gering sein. Das Verhältnis dieses Zeitraums zu dem Zeitraum, in dem das System korrekt gearbeitet hat, definiert die Verfügbarkeit eines Systems.

Datenbanken speichern ihre Einträge sowohl auf stabilem als auch auf flüchtigem Speicher, dem Cache. Ein Zugriff auf die Daten von außen erfolgt immer über den Cache, auf den die benötigten Daten vom stabilen Speicher kopiert werden. Der Cache Manager registriert dabei, welche Daten sich als Kopien im Cache befinden und sorgt dafür, dass bei Schreibvorgängen die Daten auf den

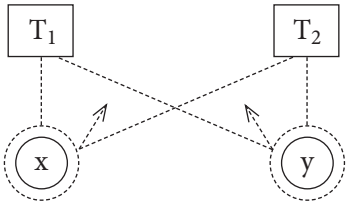


Abb. 2.6: Beispiel für ein Deadlock bei zwei Transaktionen T_1 und T_2 [12]

stabilen Speicher übertragen werden. Bei häufigen Zugriffen auf populäre Daten bedeutet der Cache eine erhebliche Erhöhung der Performance, da ein Zugriff auf den Cache sehr viel schneller realisiert werden kann. Außerdem befindet sich ein Log auf dem stabilen Speicher, der Einträge mit den letzten Änderungen in der Datenbank enthält, inklusive des Werts der Daten vorher und danach. Immer wenn eine Komponente Einträge in den Cache schreibt, muss sie auch einen dazugehörigen Log-Eintrag ablegen, der die Änderung dokumentiert. Zusätzlich wird hier auch dokumentiert, wann eine Transaktion ein Commit oder ein Abort geschickt hat.

Der Recovery Manager ist dafür zuständig, Commit- oder Abort-Operationen auszuführen und eine Datenbank in einem konsistenten Zustand neuzustarten, falls dieser abgestürzt ist. Bei einem Commit übernimmt er alle durch die Transaktion geänderten Werte in den stabilen Speicher. Eine Abort-Nachricht bedeutet, dass wieder die Werte der Daten hergestellt werden müssen, die vor der Transaktion gespeichert waren. Bei einem Neustart müssen alle Transaktionen abgebrochen werden, die zu der Zeit des Absturzes aktiv waren, und alle bestätigten Daten, die im Cache gespeichert waren, müssen auf stabilen Speicher kopiert werden, wofür die Einträge im Log verwendet werden können.

Falls nun z.B. eine Transaktion T abgebrochen wird, die einen Wert x geschrieben hat, muss der Recovery Manager kontrollieren, ob x auf dem stabilen Speicher gespeichert wurde. Falls nicht, kann der Wert von x im Cache einfach gelöscht werden und wenn eine neue Transaktion ihn lesen will, müsste der alte Wert aus dem stabilen Speicher geholt werden. Falls aber der neue Wert von x auch schon auf dem stabilen Speicher übernommen wurde, müsste der Recovery Manager im Log nachschauen, den alten Wert ermitteln und mit diesem den neuen überschreiben. In Abb 2.7 sieht man, wie trotz des Logs Daten verloren gehen können. Nämlich genau dann, wenn die Datenbank abstürzt, nachdem der stabile Speicher beschrieben und bevor der Log-Eintrag gespeichert wurde. Dadurch wäre der neue Wert von x auf stabilen Speicher und der Recovery Manager hätte keine Möglichkeit nachzuvollziehen, dass dieser von einer nicht vollständig ausgeführten Transaktion

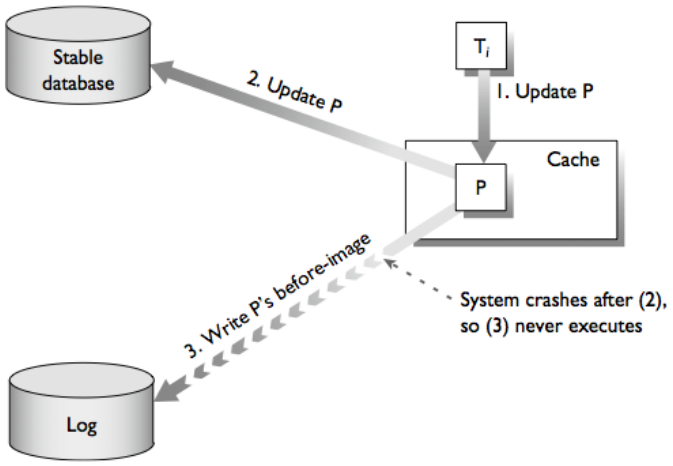


Abb. 2.7: Verlust von Daten beim Schreibvorgang ohne Write-Ahead Log Protokoll [12]

stammt. Das Write-Ahead Log Protokoll verhindert dies, indem es vorschreibt, dass immer bevor ein Dateneintrag auf stabilem Speicher abgelegt wird, schon ein Eintrag dazu im Log gespeichert sein muss.

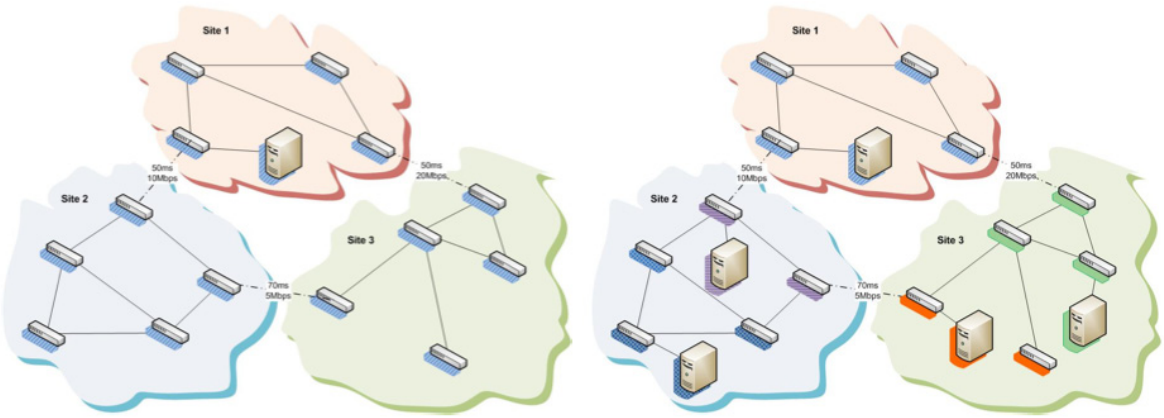
2.4. Ansätze für eine Kontrollebene mit mehreren Controllern

Um die Skalierbarkeit und Robustheit von OpenFlow-Netzwerken zu erhöhen, muss der Controller auf mehrere Server verteilt werden, um einen Single-Point-of-Failure zu verhindern. Auf den nachfolgenden Seiten werden drei Ansätze vorgestellt, die unterschiedliche Ansätze für eine Kontrollebene haben.

2.4.1 HyperFlow

Bei HyperFlow[8] werden die Switchs in Teilnetze unterteilt, die jeweils einem Controller zugeordnet werden. Dabei wird die Entfernung der Switchs zu den Controllern möglichst gering gehalten, damit die Kommunikation das Netzwerk wenig belastet. In Abb. 3.1(a) sieht man ein OpenFlow-Netzwerk mit einem Controller, der alle Switchs steuert. Bei einem Netzwerk mit großem Durchmesser entsteht dadurch viel Verkehr durch OpenFlow-Nachrichten an entfernte Switchs und falls eine wichtige Verbindung mal ausfällt, die Teilnetzwerke verbindet, kann dies nicht mehr gesteuert werden, da keine Verbindung mehr zum Controller existiert. Bei einem HyperFlow-Netzwerk wie in Abb. 3.1(b) werden mehrere Controller initialisiert, denen man Switchs eindeutig zuordnet. Alle Controller haben eine globale Sicht über das komplette Netzwerk, aber können nur die lokalen Switchs, die ihnen zugeordnet sind, direkt ansprechen. Falls sie z.B. einen Flow schreiben wollen, der auch Switchs betrifft, die sich nicht in seinem Zuständigkeitsbereich befinden, muss er die zugehörigen Controller kontaktieren, die wiederum die zugehörigen Konfigurationsnachrichten an die Switchs schicken.

Als Voraussetzung für ein HyperFlow-Netzwerk muss auf allen Switchs OpenFlow installiert sein und



(a) Netzwerk mit einem Controller (b) Netzwerk wendet HyperFlow an

Abb. 2.8: Gruppierung von OpenFlow-Netzwerk [8]

auf den Controllern die HyperFlow-Anwendung laufen. Um Informationen über Änderungen im Netzwerk und Weiterleitungstabellen auszutauschen, benutzen die Controller ein Publish/Subscribe-System. In Abb. 3.2 sieht man die Controller, auf denen die HyperFlow-Anwendungen laufen und die jeweils 3 Kanäle abonnieren, um alle für sie relevante Ereignisse zu erhalten: den Daten-Kanal, den Kontroll-Kanal und jeweils einen eigenen Kanal für jeden Controller. Der Daten-Kanal wird genutzt, um alle Ereignisse zu veröffentlichen, die Auswirkungen auf den Zustand des Netzwerks haben und deswegen für andere Controller wichtig sind. Falls ein Controller eine Weiterleitungstabelleneintrag eines nicht lokalen Switch beschreiben will, veröffentlicht er auf dem Daten-Kanal ein Ereignis mit der zugehörigen Switch-Id. Der lokale Controller erkennt die Id eines ihm zugeordneten Switchs und leitet das Ereignis an den zugehörigen Switch weiter. Nach der positiven Rückmeldung des Switch veröffentlicht der lokale Controller wieder ein Ereignis auf dem Daten-Kanal, das signalisiert, dass die gewünschte Änderung erfolgreich ausgeführt wurde.

Über den Kontroll-Kanal wird überprüft, ob alle Controller noch verfügbar sind. Jeder Controller muss hier regelmäßig ein Ereignis veröffentlichen, in dem auch Informationen über die ihm zugeordneten Switchs gespeichert sind. Falls ein Controller eine bestimmte Zeit keine Kontroll-Ereignisse mehr veröffentlicht hat, wird davon ausgegangen, dass er das Netz verlassen hat und die ihm zugeordneten Switchs werden neuen Controllern zugeordnet. Außerdem kann ein Kontroller über seinen eigenen Kanal direkt angesprochen werden.

Partitionierung und Aggregation ist bei einem HyperFlow-Netzwerk möglich, falls das zugehörige Publish/Subscribe-System dies unterstützt. Wenn das Netzwerk partitioniert wird, sind bei verteilten Publish/Subscribe-System nur noch Ereignisse von Controllern sichtbar, die sich innerhalb des Teilnetzwerks befinden. Die HyperFlow-Anwendung kann dadurch auf allen Controllern des Teilnetzwerks weiterlaufen und diese können weiter miteinander kommunizieren. Falls sich Netzwerke zusammenschließen, finden die Controller alle notwendigen Informationen über hinzuge-

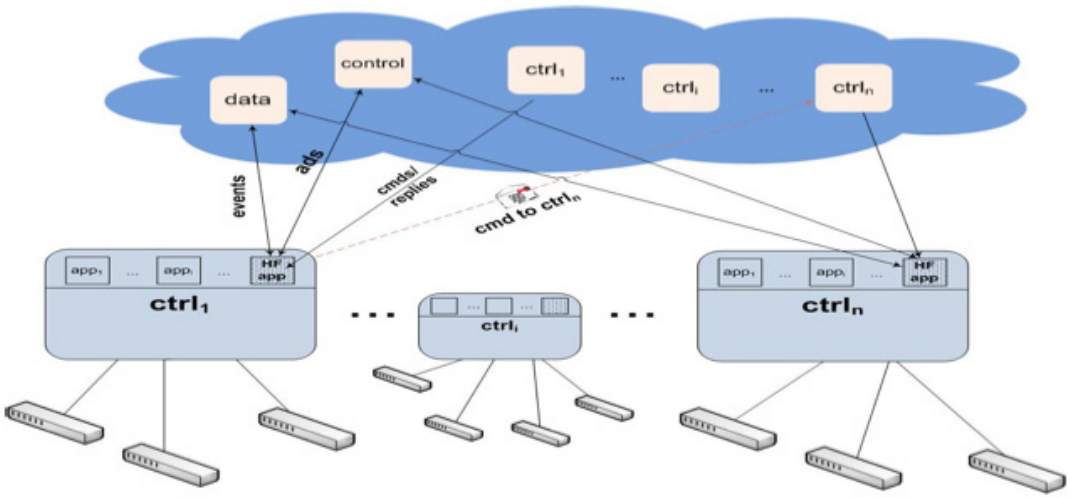


Abb. 2.9: Überblick über HyperFlow inklusive des Publish/Subscripe-Systems[8]

kommene Switchs in den Ereignissen auf dem Daten-Kanal, da hier alle nötigen Informationen abgelegt werden.

Dadurch dass Ereignisse bei den Controllern unterschiedlich schnell verarbeitet werden, entstehen vorübergehende Inkonsistenzen, die bei manchen Entscheidungen der Controller zu Konflikten führen können. Um dies zu verhindern, muss ein Controller ausgewählt werden, der bei diesen Konflikten die finale Entscheidung trifft. Bei der Erzeugung eines Flows hat diese Aufgabe der Switch, auf dem der Flow startet und andere Anfragen bezüglich des Flows müssen an diesen Controller weitergeleitet werden.

In Ab. 2.10 ist ein Beispiel dargestellt, in dem nach der Einrichtung von zwei Routen in HyperFlow eine Schleife entsteht, die die Pakete nicht verlassen können. Falls Controller $Ctrl_1$ die in roten Pfeilen dargestellte Route nach der blauen und $Ctrl_2$ die blaue nach der roten installiert, ist auf S_4 die rote Weiterleitung eingerichtet und auf S_5 die rote. Problematisch ist auch die Skalierbarkeit von HyperFlow für den Fall, dass viele Controller sich in dem Netzwerk befinden. Durch die zahlreichen Nachrichten würden die Kanäle des Publish/Subscribe-Systems unübersichtlich werden und die Controller müssten einen enormen Aufwand betreiben, um die Nachrichten zu verarbeiten und jederzeit eine globale Sicht über das Netzwerk zu erhalten. Besser ist es, Controller nur mit der Pflege ihres Teilnetzwerks zu beauftragen und die Überwachung der Topologie des Netzes einer anderen Komponente zu übertragen, die Controller kontaktieren können, falls sie Informationen über das Netzwerk benötigen würden. Aber auch das Berechnen der Flows kann eine Belastung sein, die die Controller von der Weiterleitung von Updates abhalten kann, und sollte deswegen von einer anderen Komponente übernommen werden.

2.4.2 Onix

Onix [2] ist eine Plattform, die es Anwendungen in der Kontrollebene ermöglicht, Informationen über den Status der Elemente im Netzwerk zu erhalten und deren Einstellungen mit festgelegten primitiven Befehlen zu verändern, wie z.B. das Schreiben in Weiterleitungstabellen. Dafür müssen die Anwendungen nicht direkt mir den Netzwerkelementen wie z.B. den Switchs kommunizieren,

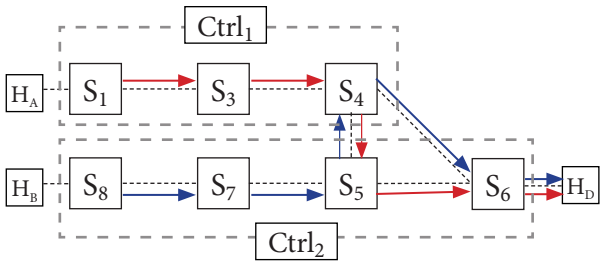


Abb. 2.10: Beispiel für entstehende Schleife bei HyperFlow

sondern benutzen APIs von Onix. Onix übernimmt die Kommunikation mit der physikalischen Ebene und garantiert, dass die gewünschten Änderungen ausgeführt werden. Dadurch erhalten Anwendungen eine globale Sicht auf das Netzwerk, ohne aber selber den Aufwand betreiben zu müssen, der für eine Überwachung und Steuerung eines komplexen Netzwerks nötig ist. Onix arbeitet dabei mit einem Datenmodell, das die Struktur des Netzwerkes abbildet und in dem die Statusänderungen der Entitäten gespeichert werden. Dieses Datenmodell erlaubt es auch Onix als verteilte Anwendung auf unterschiedlichen Servern zu laufen, wobei sich die unterschiedichen Instanzen über Änderungen im Netzwerk gegenseitig informieren und das Datenmodell abgleichen. Um sich an unterschiedliche Anforderungen von Umgebungen anzupassen, kann die Kontrollebene das Datenmodell nach ihrer Vorstellung konfigurieren. Wichtig dabei ist, dass Inkonsistenzen zwischen unterschiedlichen Onix-Instanzen von den Anwendungen in der Kontrollebene entdeckt werden und bei Konflikten anwendungsspezifische Lösungen angeboten werden.

In Abb. 3.3 sieht man die vier Komponenten, aus denen ein Onix-Netzwerk besteht. Über die obere Kontrollebene, die sogenannte Network Control Logic, können Anwendungen wie in einem normalen SDN-basiertem Netzwerk auf die einzelnen Elemente des physikalischen Netzwerks zugreifen. Die unteren beiden Ebenen stellen das physische Netzwerk und die verbindende Infrastruktur dar. Onix sitzt zwischen diesen beiden Ebenen und erleichtert den Anwendungen in der Kontrollebene die Verwaltung und Aktualisierung des Netzwerks.

Die physikalische Infrastruktur besteht aus den Switchs und den Routern sowie allen anderen Netzwerkelementen, die eine Schnittstelle unterstützen, so dass Onix ihren Zustand lesen und dadurch ihr Verhalten zu steuern. Über die Verbindungsinfrastruktur kommuniziert Onix über Kontrollnachrichten mit den physikalischen Netzwerkgeräten. Dieser Kontrollkanal kann entweder die gleichen Verbindungen wie der gewöhnliche Datenverkehr nutzen (in-band) oder über eine gesondertes physikalisches Netzwerk laufen, das nur Onix-Nachrichten überträgt (out-of-band). Voraussetzung für die Infrastruktur ist, dass eine bidirektionale Verbindung zwischen den Switchs und den Onix-Instanzen besteht. Um die Kontrollnachrichten im Netzwerk zu verbreiten, können bekannte Routingverfahren wie IS-IS oder OSPF verwendet werden.

Als eigenständige Komponente vermittelt Onix zwischen dem physikalischem Netzwerk und der Kontrollebene, die über Onix auf das Netzwerk zugreift. Onix kann als verteiltes System auf einem Cluster von mehreren physikalischen Servern laufen, wobei jeder Server auch mehrere Onix-Instanzen beherbergen kann. Damit eine Skalierung auf Netzwerke mit Millionen von Ports möglich ist, müssen die unterschiedlichen Onix-Instanzen ihren Netzwerk-Status mit anderen Instanzen austauschen, die in einem Cluster beheimatet sind. Die Kontrolllogik kann über die API von Onix auf das Netzwerk zugreifen und das gewünschte Verhalten des Netzwerks bewirken.

Den Kern der Onix API bildet eine Datenstruktur, die sogenannte Network Information Base (NIB). Sie speichert die Infrastruktur des Netzwerks als Graph von Entitäten mit ihrem Zustand

und über sie können die verteilten Onix-Instanzen Informationen über das Netzwerk austauschen. Die Kontrollebene kann die NIB lesen und schreiben, wodurch die gewünschten Eigenschaften des Netzwerks hergestellt werden können und auch überprüft werden kann, ob die diese eintreten und erhalten bleiben. Die NIB unterstützt keine feinkörnigen oder verteilten Locking-Mechanismen, jedoch kann einer Anwendung garantiert werden, dass sie als Einzige Zugang zur Datenstruktur einer einzelnen lokalen Instanz hat. Sobald diese Instanz aber ihre Statusänderungen mit anderen Instanzen wieder austauscht, kann nicht vermieden werden, dass der Zustand der überarbeiteten Netzwerkelemente wieder durch die Kommunikation mit anderen Instanzen verändert wird. Anwendungen können sich dann über einen Subscribe-Service darüber informieren lassen, falls sich der Zustand eines Elements ändert, aber müssen eigene Mechanismen wie das Consensus-Verfahren oder Distributed Locking benutzen, um mehrere Instanzen zu koordinieren.

Die Kommunikation zwischen Onix und dem Netzwerk erfolgt asynchron, wodurch garantiert wird, dass die NIB-Operationen irgendwann ausgeführt werden, jedoch ohne eine Angabe des Zeitpunktes. Manchmal ist es aber erforderlich, dass Operationen in einer bestimmten Reihenfolge vollzogen werden müssen, wenn z.B. Weiterleitungstabellen aktualisiert werden. In diesem Fall bietet die API eine Callback-Methode an, die die Kontrolllogik informiert, wenn eine Operation ausgeführt wurde. Daraufhin kann die Kontrolllogik die nächste Operation ausführen. Dieser Mechanismus funktioniert aber nicht für verteilte Onix-Instanzen, weswegen die Kontrollebene dann zusätzliche Verfahren anwenden muss.

Bei großen Netzwerken mit mehreren Onix-Instanzen steigt auch die Anzahl an Nachrichten zwischen den Instanzen und die Größe der benötigten Ressourcen, um die NIB abzuspeichern und aktuell zu halten. Um zukünftigen Anforderungen gerecht zu werden, bietet Onix unterschiedliche Strategien um Skalierbarkeit und Verfügbarkeit zu erhöhen.

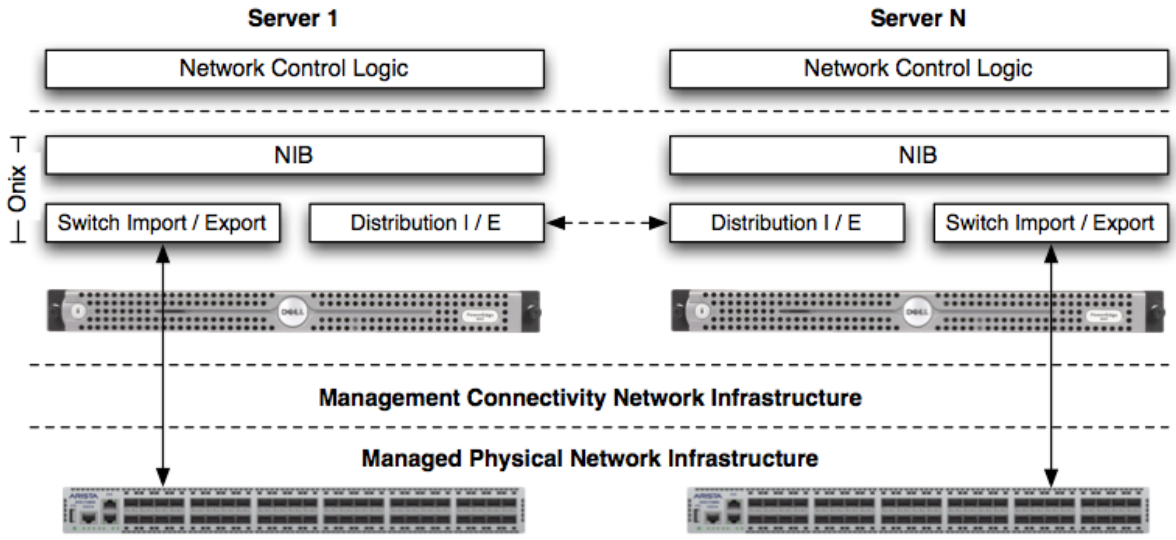


Abb. 2.10: Die vier Komponenten im Onix-Netzwerk [2]

Partitionierung ermöglicht den Controller-Instanzen, nur eine Teilmenge des NIB im Speicher aktuell zu halten, wodurch der Aufwand für die Controller deutlich reduziert werden kann. Außerdem können die zu verwaltenden Elemente eines Teilnetzwerks auf zwei Onix-Instanzen aufgeteilt werden, falls eine einzelne Instanz mit dieser überfordert ist. Als nächste Strategie bietet die Aggregation, dass mehrere Teilnetzwerke als ein einzelnes logisches Element zu einer anderen Onix-Instanz hinzugefügt werden. Dadurch kann die Größe der NIB reduziert werden und große Netzwerke werden als Baumstruktur dargestellt, wie z.B. bei einem Campus-Netzwerk, bei dem das Netzwerk eines einzelnen Gebäudes durch einen einzelnen Knoten angesprochen werden kann.

Um Konsistenz und Dauerhaftigkeit zu erreichen, bietet Onix unterschiedliche Strategien an, bei denen die Kontrolllogik vorschreiben muss, welche Anforderungen das Netzwerk hat. Bei einer verteilten Onix-Plattform dient die NIB als verteilte Datenbank, bei der zwei unterschiedliche Möglichkeiten existieren, um Status-Updates in der NIB zwischen Onix-Instanzen abzugleichen. Falls Konsistenz und Dauerhaftigkeit gefordert ist, kann eine replizierte transaktionale Datenbank genutzt werden. Falls Schnelligkeit gefordert ist und inkonsistente Zustände toleriert werden können, steht eine speicher-basierte verteilte Hashtabelle zur Verfügung.

Onix liefert eine Lösung für die Synchronisierung von mehreren Instanzen in der Kontrollogik, aber macht keine konkreten Vorschläge, wie Anwendungen eine Route im Netzwerk schreiben können, für die mehrere Onix-Instanzen zuständig sind. Hier treten Probleme auf, da die Anwendungen keinen Einfluss darauf haben, in welcher Abfolge die Updates ausgeführt werden. Falls Konflikte mit anderen Flows auftreten, kann es passieren, dass Flows unvollständig installiert werden. Eine Möglichkeit wäre, dass sie die Updates aufteilen und jeweils die Onix-Instanzen kontaktieren, die für die Switchs direkt zuständig sind, was aber einen großen Aufwand bedeutet. Besser wäre es, wenn man über eine Onix-Instanz lokationstransparent einen Flow im Netzwerk schreiben könnte und bei Bedarf garantiert bekommen würde, dass dieser auch im Netzwerk komplett installiert wurde.

2.4.3 Kandoo

Kandoo [9] verfolgt einen anderen Ansatz, um das Problem zu lösen, dass ein zentraler Controller nur eine begrenzte Anzahl von Nachrichten bearbeiten kann. Kandoo will diese Anzahl reduzieren, indem man zwei Ebenen von Controllern einführt und lokale Ereignisse von globalen Ereignissen unterscheidet, die bei der Verarbeitung Wissen über das Netzwerk benötigen. Der zentrale Root-Controller kennt dabei weiterhin die Struktur des gesamten Netzwerks und die Zustände der einzelnen Elemente, aber wird nur noch mit globalen Ereignissen konfrontiert. Lokale Ereignisse werden von den lokalen Controllern verarbeitet, die zwischen den Switchs und dem Root-Controller sitzen und nur lokales Wissen besitzen. Falls jedoch die Verarbeitung Kenntnisse über das gesamte Netzwerk erfordert, wird ein Ereignis erzeugt, dass der Root-Controller abonniert hat.

Als Voraussetzung für Kandoo sollte in unmittelbarer Nähe zu den Switchs Rechenleistung verfügbar

sein, um lokale Controller zu verwenden und Anwendungen zu installieren, die nur in einem lokalen Bereich arbeiten und kein weiteres Wissen über das Netzwerk benötigen. In Abb. 3.4 sieht man in der untersten Ebene die Switchs, die jeweils einem lokalen Controller zugeordnet werden. Mit diesem tauschen sie wie in einem herkömmlichen OpenFlow-Netzwerk Nachrichten aus. Der Controller kann diese Nachrichten mit seinem lokalen Wissen bearbeiten und nur die Nachrichten, die globales Wissen benötigen, werden an den Root-Controller weitergeleitet. Lokale Controller haben dabei auch keine Kenntnisse darüber, wie viele andere lokale Controller noch in dem Netzwerk existieren und kennen nur den Root-Controller.

Um Switchs zu überwachen, müssen auf den Controller Anwendungen installiert werden. Hierbei handelt es sich um OpenFlow-Anwendungen, die OpenFlow-Nachrichten versenden und verarbeiten. Zusätzlich können sie Kandoo-Ereignisse veröffentlichen und abonnieren, um mit anderen Kandoo-Anwendungen im Netzwerk zu kommunizieren. In Abb. 3.5 sieht man ein Netzwerk mit vier Controllern, die reagieren, falls Elefanten-Flows in dem Netzwerk entdeckt werden. Auf den drei lokalen Controllern ist die Anwendung App_{detect} installiert, die jeweils die zugeordneten Switchs überwachen. Diese lokale Anwendung benötigt keinerlei Informationen über das gesamte Netzwerk, sondern untersucht nur die Nachrichten des einen zugewiesenen Switchs. Falls ein Elefanten-Flow entdeckt wird, wird ein Ereignis vom Typen $E_{elephant}$ erzeugt, der von der Anwendung $App_{reroute}$ auf dem Root-Controller abonniert wurde. Diese globale Anwendung überwacht nun keine lokalen OpenFlow-Nachrichten, sondern reagiert auf entdeckte Elefanten-Flows, indem sie mit Hilfe ihres Wissens über die Topologie des Netzes neue Flows in die Weiterleitungstabellen der Switchs einträgt.

2.5 Konsistente Routen-Aktualisierung

In den folgenden Arbeiten geht es darum, wie man in Netzwerken mit einem Controller die Aktualisierung der Weiterleitungstabellen so organisieren kann, dass während der Aktualisierung keine vorübergehenden Inkonsistenzen entstehen und gewünschte Eigenschaften des Netzwerks erhalten bleiben.

2.5.1 Frenetic

Für Frenetic [7] wurde untersucht, wie Änderungen in einem Netzwerk implementiert werden und dabei bestimmte gewünschte Bedingungen eingehalten werden. Diese beinhalten, dass z.B. bestimmte Filtereigenschaften erhalten bleiben, keine Pakete verloren werden oder Verbindungen nicht unterbrochen werden. Frenetic bietet Software-Abstraktionen, die dem Entwickler garantieren, dass bestimmten Invarianten bei einer Umstellung eines Netzwerks gültig bleiben, ohne dass dieser sich mit Low-Level-Details auseinandersetzen muss. Dabei wird auch berücksichtigt, dass die Nachrichten über ein verteiltes Netzwerk versendet werden.

In Abb. 3.6 sieht man ein Beispiel für ein Netzwerk, das eine verteilte Zugangskontrolle realisiert. Die vier Switchs realisieren hier eine Firewall, die den Zugang zum Netzwerk abhängig vom Host und dem Traffic-Typen beschränkt. Studenten (S) und Fakultätsmitglieder (F) haben uneingeschränkten Zugang, wohingegen unbekannte Teilnehmer (U) und Gäste (G) keine SSH-Verbindung aufbauen können. Switch I erhält dabei alle Anfragen und leitet diese je nach Host an die drei Filter weiter, die nach dem Traffic-Typen entscheiden, ob die Anfrage weiter gesendet oder verworfen wird. In Konfiguration I werden dabei Anfragen von nicht vertrauenswürdigen Teilnehmern (U,G) an F1, von Studenten an F2 und von Fakultätsmitgliedern an F3 weitergeleitet. Nun kann sich aber die Belastung so verändern, das mehr Ressourcen benötigt werden, um die Anfragen von nicht vertrauenswürdigen Teilnehmern zu verarbeiten, welche in der Konfiguration II vorhanden wären. Wenn bei der Konfigurationsumstellung nun die Einstellungen der einzelnen Switchs verändert werden, muss man aber darauf achten, dass jederzeit die gewünschte Eigenschaften des Firewall erhalten bleiben. Falls z.B. zuerst die Weiterleitungstabelle von I und erst später die Filtereigenschaft von F2 geändert wird, würden eine Zeit lang auch SSH-Nachrichten von nicht vertrauenswürdigen Teilnehmern

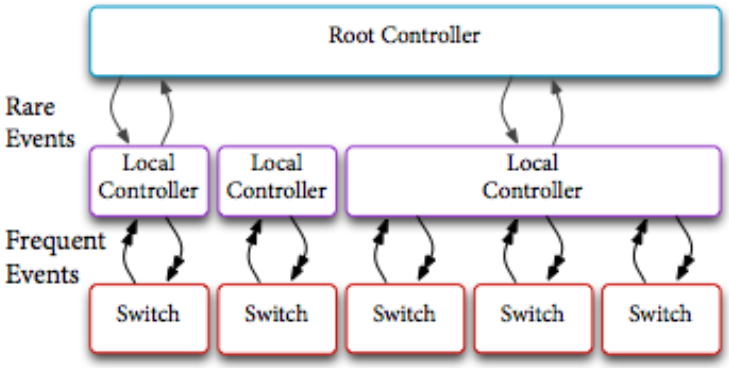


Abb. 3.4: Unterschiedliche Ebenen von Controllern [9]

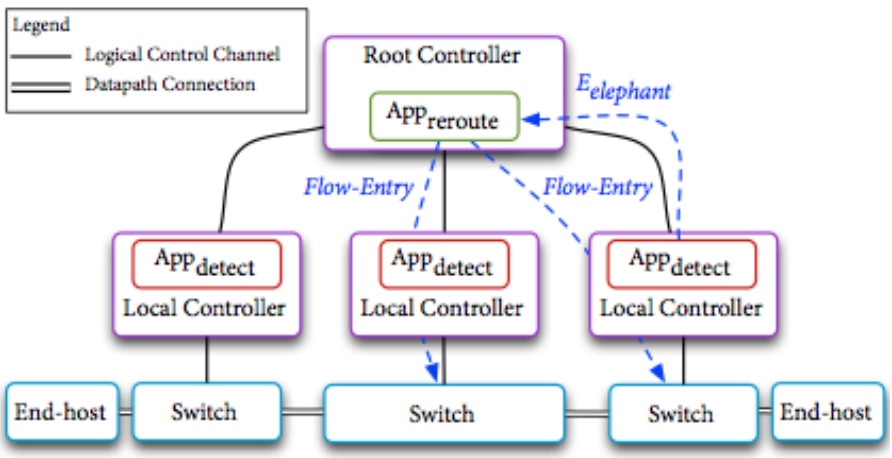
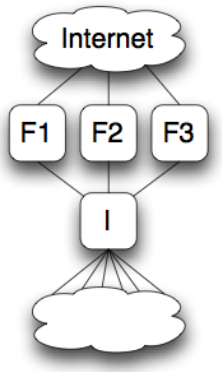


Abb. 2.11: Beispiel für die Entdeckung eines Elefanten-Flows[9]

zugelassen werden. Es gibt nur eine zulässige Reihenfolge, wie man von der einen auf die andere Konfiguration wechseln kann. Zuerst muss I den Traffic von Studenten auf F3 weiterleiten. Nachdem alle in-flight-Pakete von F2 verarbeitet wurden, muss F2 so umprogrammiert werden, dass der Switch alle SSH-Pakete ablehnt. Und als letztes muss I so umgestellt werden, dass er den Traffic von Gästen zu F2 weiterleitet. Bei dieser Abfolge wäre sichergestellt, dass zu keinem Zeitpunkt SSH-Pakete von nicht vertrauenswürdigen Teilnehmern weitergeleitet werden.

In dem Beispiel ist es recht einfach, eine Lösung zu finden, doch bei komplexeren Netzwerken kann dies komplizierter sein und manchmal sogar unmöglich. Als formale Eigenschaft garantiert die Pro-Paket-Konsistenz dafür, dass Pakete während der Umstellung korrekt weitergeleitet werden. Wenn man in dem Beispiel nur das SSH-Pakete betrachtet, das in dem unerwünschten Fall von F2 weitergeleitet wird, kann man beobachten, dass dies in zwei unterschiedlichen Konfigurationen passiert. Zuerst wird es von Switch I, auf dem Konfiguration II installiert ist, weitergeleitet und dann von Switch F2 verarbeitet, auf dem noch Konfiguration I installiert ist. Die dadurch auftretenden unerwünschte Eigenschaften des Netzwerks können vermieden werden, indem man bei der Verarbeitung von Paketen die Pro-Paket-Konsistenz einfordert, die garantiert, dass ein Paket auf seinem Weg durch das Netzwerk immer nur von Switchen in einer Konfiguration verarbeitet wird. Ein Update des Netzwerks würde dann in zwei Phasen ablaufen. Zuerst würde auf allen Switchs neben der Konfiguration I auch die Konfiguration II installiert werden, die jedoch nicht aktiv wäre. Wenn das Update auf allen Switchs abgeschlossen wäre, würden Pakete beim Senden oder Eintreten in das Netzwerk mit Versionsnummern markiert werden, die den Switchs signalisieren würde, dass die Pakete nur in der Konfiguration II verarbeitet werden sollen. Nun würde die Konfiguration II in den Switchs aktiv werden und sobald alle Pakete, die vor dem Update gestartet wurden, angekommen sind, kann man die Konfiguration I deaktivieren. Als weitere Eigenschaft wird außerdem die Pro-Flow-Konsistenz eingeführt, bei der garantiert wird, dass alle Pakete in einem Flow mit der gleichen Einstellung weitergeleitet werden.

Das Ziel von Frenetic ist es, dass der Entwickler sich aber erst gar nicht mit solchen Abläufen bei einer



Configuration I

	Type	Action
I	U, G	Forward F_1
	S	Forward F_2
	F	Forward F_3
F_1	SSH	Monitor
	$*$	Allow
F_2	$*$	Allow
F_3	$*$	Allow

→

Configuration II

	Type	Action
I	U	Forward F_1
	G	Forward F_2
	S, F	Forward F_3
F_1	SSH	Monitor
	$*$	Allow
F_2	SSH	Monitor
	$*$	Allow
F_3	$*$	Allow

Abb. 2.12: Beispiel für eine Zugangskontrolle[7]

Umstellung befassen muss. Stattdessen schreibt er einfach den Befehl: *per_packet_update(config2)*. Der Befehl wird kompiliert und automatisch ein Ablauf generiert, der alle Low-Level-Befehle beinhaltet, die benötigt werden, um die Konfiguration zu wechseln. Dabei wird auch versucht, die Kosten für eine Umstellung möglichst gering zu halten, indem das richtige Verfahren ausgewählt wird. Zusätzlich ist es möglich zu überprüfen, ob bestimmte Sicherheits-Richtlinien in einem Netzwerk eingehalten werden, indem folgender Befehl ausführt wird: *ok = verify (config2, topo, pol)*. Dabei wird in der Variable *ok* ein boolescher Wert gespeichert, der aussagt, ob die gewünschten Richtlinien eingehalten wurden. Bei unserem Beispiel kann z.B. kontrolliert werden, dass wirklich keine SSH-Pakete von nicht vertrauenswürdigen Hosts weitergeleitet wurden.

Frenetic geht darauf ein, wie Konsistenz in einem Netzwerk mit einem Controller erreicht werden kann, aber macht keine Aussagen für Netzwerke mit mehreren Controllern.

2.5.2 Konsistente Netzwerkaktualisierung bei der Migration von virtuellen Maschinen

In [1] wird untersucht, wie man virtuelle Maschinen (VM) auf andere Knoten umziehen kann, um dabei die Belastung für das Netzwerk gering zu halten. Dabei wird von einem vorhandenen Startzustand des Netzwerks ausgegangen, das über mehrere Schritte in einen gewünschten Endzustand transformiert werden soll. Die VMs befinden sich auf Servern, die untereinander über Switchs verbunden sind. Da die Flows zwischen VMs erhalten bleiben sollen, müssen zusätzlich zu dem Umzug der Knoten die Einträge in den Weiterleitungstabellen aktualisiert und die dafür zugehörigen Open-Flow-Nachrichten verschickt werden. Da man bei einem Netzwerk nicht garantieren kann, dass diese Änderungen zeitgleich passieren, muss man dafür sorgen, dass die Reihenfolge der Schritte keine Inkonsistenzen oder Fehler produziert.

Um die Transformation des Netzwerks möglichst optimal durchzuführen, wird eine Heuristik vorgeschlagen, mit der man die Schritte der Änderung erhält. Das Finden der optimalen Lösung resultiert in einem NP-harten Problem und wäre deshalb bei einem schnell benötigten Übergang nicht praktikabel. Der Controller eines SDN-Netzwerks muss die Heuristik ausführen können und die gewünschten Schritte einleiten. In Abb. 3.6 sehen sie ein Netzwerk mit neun Servern, die über vier Switchs verbunden sind. Falls man nun mit V4 auf den Server S5 umziehen will, um die Entfernung zwischen den beiden VMs zu reduzieren, muss man auch die Einträge in den Weiterleitungstabellen von X1, X3 und X4 verändern.

Wenn Weiterleitungstabellen in einer falschen Reihenfolge beschrieben werden, können Schleifen entstehen. In Abb. 3.7 sehen sie die virtuelle Maschine V, die von Knoten s auf den Knoten d gewandert ist und der alte Flow P1 soll mit dem neuen Flow P2 ersetzt werden. Wenn zum Zeitpunkt t_1 die Weiterleitungstabelle von X2 beschrieben wird, noch bevor zum Zeitpunkt t_2 die von X1 aktualisiert wird, entsteht in dem Zeitintervall $[t_1,t_2)$ eine Schleife und die Pakete von X2 und X1 werden hin und hergeschickt. Der Größe des Zeitintervalls ist abhängig von der Zeit, die

ein Controller braucht, um die Weiterleitungstabellen zu beschreiben. Da aber ein Schleife auch bei Pfaden entstehen kann, bei denen zahlreiche Switchs bearbeitet werden müssen, ist es wichtig, diesen möglichst klein zu halten. Um die Möglichkeit, dass Kreisläufe entstehen, zu minimieren, wird in der Arbeit vorgeschlagen, dass man immer mit dem Switch anfängt, der sich am nächsten vom Zielknoten befindet, und dann schrittweise einen Knoten weiter geht. In Abb. 3.7 würde dann zum Zeitpunkt t_1 der Switch X1 aktualisiert werden und zum Zeitpunkt t_2 der Switch X2. Dann würden nur die Pakete, die vor t_1 X1 verlassen und vor t_2 X2 erreichen einmal im Kreis laufen.

Auch wenn in dieser Diplomarbeit vorübergehende Inkonsistenzen toleriert werden, ist es erwünscht, möglichst keine Kreisläufe während der Aktualisierung im Netzwerk zu generieren. Deswegen übernehmen die Controller die in dieser Arbeit empfohlene Abfolge von Updates und fangen immer bei den Switchs in der Nähe des Zielknotens mit der Aktualisierung an.

2.5.3 OpenFlow Safe Update Protokoll

Das Safe Update Protokoll [3] bietet eine Lösung, wie bei einer Aktualisierung von Switchs eine Pro-Paket-Konsistenz garantiert werden kann. Dabei wird im Gegensatz zu Frenetic[7] vermieden, dass zwei unterschiedliche Flows gleichzeitig auf einem Switch installiert sind. Stattdessen wird ein Flow eingeführt, der zwischen beiden Konfigurationen aktiv wird und bei dem alle Pakete an den Controller weitergeleitet werden. Dieser wird nur bei Switchs installiert, bei denen sich etwas in der Weiterleitungstabelle ändert, und garantiert, dass keine Pakete von zwei unterschiedlichen Konfigurationen verarbeitet werden.

Bei jedem Paket, das während dem Übergang von zwei Flows zum Controller geschickt wird, kann der Controller den Flow zuordnen und zuverlässig unter diesen Regeln das Paket zu seinem Ziel weiterleiten. Dabei kann das Paket entweder vom Controller direkt zu seinem Ziel geschickt werden, oder an einer anderen Stelle wieder in das Netzt injiziert werden, von der es dann weitergeleitet wird. Der Controller kennt den Status des Netzwerks und kann deswegen entscheiden, wohin das Paket am besten geschickt wird. Die Aktualisierung einer Weiterleitungstabelle erfolgt auf einem Switch atomar, so dass ein Paket immer durch einen Flow weitergeleitet wird und nie durch eine Mischung.

Falls ein Switch bei der Installation eines Flows die Einträge seiner Weiterleitungstabelle ändern muss. weiss er nicht, ob die nachfolgenden Switchs das Paket nach dem alten oder dem neuen Flow behandeln. Deshalb kann er das Paket nicht nach seiner Weiterleitungstabelle verarbeiten, sondern muss es es an den Controller schicken, der einen Überblick über das Netzwerk hat und garantieren kann, dass das Paket sein Ziel erreicht. Falls sich die Einträge in einer Weiterleitungstabelle eines Switchs nicht verändern, kann der Switch das Paket jedoch nach seiner Tabelle weiterleiten.

Um einen Übergang zu realisieren, wird dieser in 4 Epochen aufgeteilt, die man anhand der Weiterleitungsfunktionen der Switchs beschreiben kann. Am Anfang in Epoche 1 ist auf jedem Switch

der Flow mit der Weiterleitungsfunktion f^{s}_{1} installiert. Der Controller startet die zweite Epoche, indem er die Updates für Übergangsfunktion f^{s}_{12} an die beteiligten Switchs schickt. Jetzt beginnt die Übergangszeit, in der die Tabellen der Switchs beschrieben werden und in der ein Switch das Paket entweder nach der Funktion f^{s}_{1} oder nach f^{s}_{12} weiterleitet. Sobald die Umstellung der Funktion abgeschlossen ist, sendet der Switch eine Bestätigungsnachricht an den Contoller, dass der Übergang der Funktionen abgeschlossen ist. Nachdem der Controller alle Bestätigungen der teilnehmenden Switchs erhalten hat, wartet er die maximale Netzwerk-Latenz ab und startet Epoche 3. Der Controller schickt nun die Updates für f^{s}_{2} an alle beteiligten Switchs und gibt außerdem alle Pakete heraus, die ihm in der vorherigen Epoche zugeschickt wurden. Die Switchs bestätigen wieder die Umstellung auf f^{s}_{2} mit einer Nachricht an der Controller und dieser wartet wieder die maximale Latenzzeit ab, bevor Epoche 4 beginnt, in der alle Switchs f^{s}_{2} übernommen haben. Jetzt kann wieder jederzeit eine neuer Flow installiert werden.

In Abb. 2.13 sieht man ein Netzwerk, in dem der in durchgezogenen Linien gezeigte Flow installiert ist und ein Wechsel auf den Flow in gestrichelten Linien durchgeführt werden soll. Auf Abb. 2.14 sieht man, wie nach Abschluss von Episode 2 die Übergangsfunktion auf den Switchs u und v installiert wurde, auf denen sich durch den neuen Flow in der Weiterleitungstabelle etwas ändert. Diese schicken die Pakete des Flows an den Controller, der diese an t weiterleitet.

Das OpenFlow Safe Update Protokoll reduziert die Aufgaben für einen Switch bei der Installierung eines Flows, falls Pro-Paket-Konsistenz gewünscht ist, da dieser keine zwei parallelen Regeln installieren muss. Aber für einen Controller wird der Aufwand erhöht, das bei größeren Netzwerken sehr viele Pakete anfallen, die er weiterleiten muss. Dadurch ist die Methode nicht gut skalierbar und

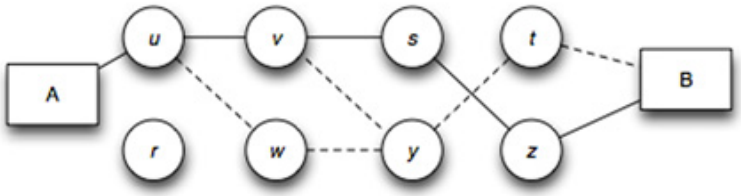


Abb. 2.13: Installierter und gewünschter, gestrichelter Flow[3]

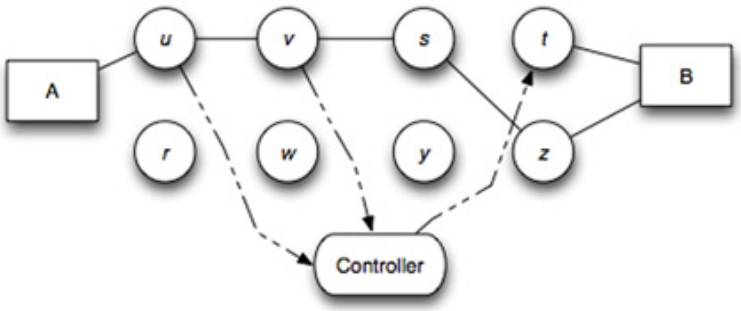


Abb. 2.14: Übergangszustand beim OpenFlow Safe Update Prokoll [3]

nur für kleinere Netzwerke praktikabel. Hinzu kommt, dass bei großen Netzwerken die maximale Latenzzeit durch einen großen Durchmesser relativ groß sein kann und so der Controller zwischen den Episoden länger warten muss, bis er die nächste Episode starten kann.

3 Systemmodell und Problembeschreibung

In dem Systemmodell werden alle Komponenten des Netzwerks vorgestellt und ihre Rollen erläutert. Eine Anwendung, die einen Flow schreiben will, wendet sich an die Middleware, der die Topologie des Netzwerks bekannt ist und die notwendigen Schreibaufträge an die Controller weiterleitet. Danach werden die Probleme formuliert, die durch die asynchrone Kommunikation zwischen Middleware und Controller auftreten können und die als Ziel dieser Arbeit verhindert werden sollen.

3.1 Systemmodell

In Abb. 3.1 sieht man die drei Ebenen unseres Systemmodells. Kontrollanwendungen aus der Anwendungsebene beauftragen die Control Coordination Middleware (CoCoMi) in der Kontrollebene mit dem Schreiben von Flows in dem Netzwerk. CoCoMi besitzt eine globale Sicht auf das Netzwerk und berechnet die nötigen Updates, die für die Erzeugung des Flows im Netzwerk nötig sind. Die Updates werden an die Controller versendet, die für die zugehörigen Switchs zuständig sind. Hier werden keine Annahmen darüber getroffen, wie viele Switchs den Controllern zugeordnet werden, sondern die Beziehungen zwischen Controller und Switchs flexibel definiert, so dass z.B. auf die Belastungen des Netzwerks reagiert werden kann, indem Zuständigkeiten für Switchs unter

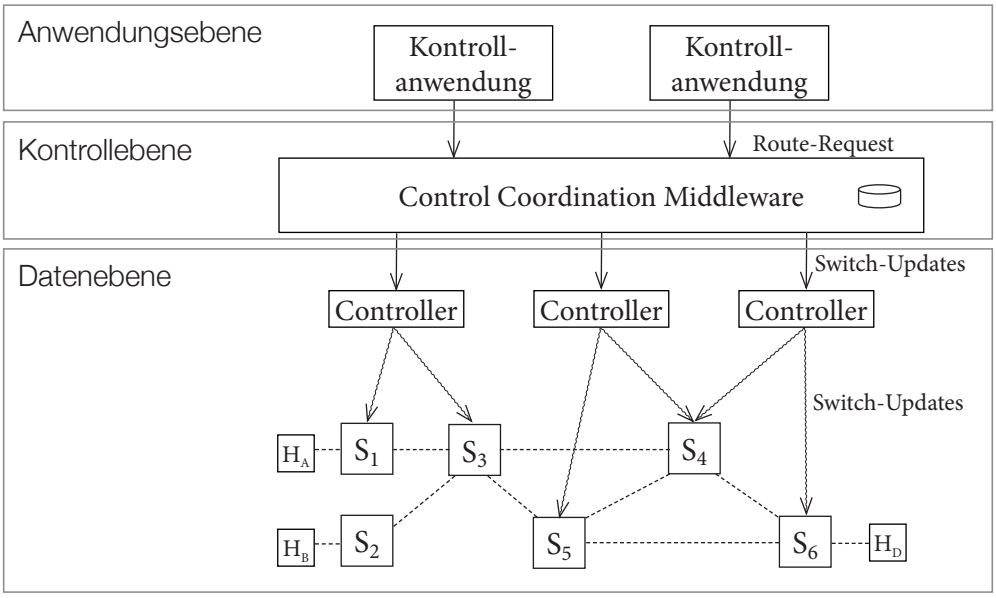


Abb. 3.1 Übersicht der unterschiedliche Ebenen unseres Systemmodells

Controllern neu aufgeteilt werden. Dabei ist auch möglich, dass ein Switch von mehreren Controllern Update-Nachrichten empfängt. Die Controller erzeugen mit Hilfe dieser Nachrichten von den Agenten OpenFlow-Nachrichten, die an die Switchs in der Datenebene geschickt werden und die Weiterleitungstabellen auf die gewünschte Art aktualisieren. Mit Hilfe der Weiterleitungstabellen werden die Pakete von den Switchs im Netzwerk transportiert.

3.1.1 Datenebene

Die grundlegenden Elemente, die in unserem Netzwerk-Modell transportiert werden, sind Pakete als Sequenzen von Bits, die aus einer 0 oder 1 bestehen. Pakete enthalten in ihrem Header Informationen zu Ihrem Adressaten, welche von den Switchs benutzt werden, um sie weiterzuleiten. Pakete werden während des Transports durch das Netzwerk in einem Port p gespeichert und warten dort darauf, weiterverarbeitet zu werden.

Ein Port ist jeweils einem Switch S zugeordnet und optional über eine physische Netzwerkverbindung mit einem anderen Port verbunden. Ein Switch benutzt Ports, um Pakete an andere Switchs weiterzuleiten und von anderen Switchs zu empfangen. Falls ein Switch ein Paket über einen Port weitergibt, wird es von diesem über die Verbindung an den entsprechenden Port übertragen und dort vom dazugehörigen Switch verarbeitet.

Bei dem Switch handelt es sich um ein vereinfachtes Modell eines OpenFlow-Switchs, bei dem nur die Weiterleitungsfunktion übernommen wurde, da dies für diese Arbeit ausreichend ist. Switchs entscheiden anhand der Weiterleitungsfunktion, an welchen Port ankommende Pakete ausgegeben oder gelöscht werden. Die Weiterleitungsfunktion sucht nach Übereinstimmungen mit den Filterkriterien, die in einer Tabelle gespeichert werden. Wird ein passendes Filterkriterium gefunden, das sogenannte Matching-Kriterium m, wird das betreffende Paket an den in der Weiterleitungstabelle gespeicherten Port übergeben und von diesem an den verbundenen Port eines benachbarten Switchs übertragen. Wie bei einer OpenFlow-Tabelle wird einem Matching-Kriterium durch die Weiterleitungsfunktion ein Port zugewiesen: $m \rightarrow p$. Zusätzlich kann in der Weiterleitungstabelle auch gespeichert werden, dass bestimmte Pakete gelöscht und nicht mehr weiterverarbeitet werden: $m \rightarrow drop$.

3.1.2 Kontrollebene

Die Einträge in den Weiterleitungstabellen werden mit Hilfe des OpenFlow-Protokolls durch Controller in der Kontrollebene geändert, die dadurch die Weiterleitungsfunktion steuern. Ein Controller kann auf dem Switch Schreib-, Lösch- und Leseoperationen mit Hilfe von OpenFlow-Nachrichten ausführen, die sich immer über das Matching-Kriterium auf einen speziellen Flow beziehen. Mit Hilfe der Switch-Updates $\mathcal{U}_{\text{Switch}}$ aktualisiert ein Controller die Weiterleitungstabellen der Switchs und legt dadurch fest, wie ankommende Pakete behandelt werden. $\mathcal{U}_{\text{Switch}}{}^{+}$-Updates

realisieren Schreiboperationen, bei denen auch bestehende Einträge überschrieben werden. $\mathcal{U}_{\text{Switch}}^{-}$-Updates führen Löschoperationen von Tabelleneinträgen aus, wodurch diese aktuell gehalten werden können und überflüssige Einträge vermieden werden. Zusätzlich besitzt jeder Switch noch einen Timeout-Mechanismus, durch den Einträge gelöscht werden, über die einen bestimmten Zeitraum lang keine Pakete weitergeleitet wurden.

Definiton: Switch-Update

Ein Update $\mathcal{U}_{\text{Switch}} = (S_i, m_i, p_i)$ setzt sich aus dem Switch S_i, auf dem sich die Weiterleitungstabelle befindet, dem Matching-Kriterium m_i und dem zugewiesenen Ausgangsport p_i zusammen. Je nachdem, ob es sich dabei um einen schreibenden oder löschenden Update handelt, wird der Eintrag erstellt, modifiziert oder gelöscht.

Statusnachrichten St_{Switch} werden an den Switch verschickt, um zu erfahren, ob dieser zu einem bestimmten Matching-Kriterium einen Eintrag gespeichert hat.

Definiton: Status-Request und Status-Report

Ein Status-Request $St_{\text{Request}} = (S_i, m_i)$ beinhaltet den Switch und das Matching-Kriterium des gesuchten Flows. Als Antwort versendet der Switch einen Status-Report $St_{\text{Report}} = (S_i, m_i, p_i)$, bei dem der Ausgangsport p_i enthalten ist, falls ein Eintrag existiert, oder p_i=-1 ist, falls kein Eintrag existiert.

In unserem Systemmodell kann ein Switch mit beliebig vielen Controllern in Kontakt stehen und von diesen auch Nachrichten empfangen. Genauso kann ein Controller mit mehreren Switchs eine Verbindung aufbauen, und an diese OpenFlow-Nachrichten versenden. CoCoMi kennt die Topologie des Netzwerks und erzeugt mit diesem Wissen Flows für Pakete im Netzwerk, indem sie gezielt Update-Nachrichten an die zuständigen Controller schickt, die über eine Verbindung zu den jeweiligen Switchs verfügen. Die Controller leiten diese über das OpenFlow-Protokoll an die Switchs weiter.

Als Flow-Udate wird eine Liste von Updates bezeichnet, die eine Weiterleitung für Pakete eines Flows einrichten:

Definiton: Flow-Update

Ein Flow-Update $\mathcal{U}_{\text{Flow}} = \{\mathcal{U}_{\text{Switch 1}}, \ldots, \mathcal{U}_{\text{Switch n}}\}$ ist eine Liste von Switch-Udates, die zusammen den Flow erzeugen, über den Pakete durch das Netzwerk geleitet werden.

Die Kommunikation zwischen CoCoMi, Controllern und Switchs erfolgt asynchron, weshalb die Übertragungsdauer der Nachrichten variieren kann. Deshalb kann bei einem Switch die Reihenfolge der ankommenden Updates sich von der Reihenfolge unterscheiden, in der sie von CoCoMi verschickt wurden.

Damit CoCoMi jederzeit eine globale Sicht über das Netzwerk hat, speichert es die aktuelle Topologie des Netzwerks in einer Datenbank. Dazu gehört die Konnektivität unter den Switchs und über welche Controller die Switchs erreichbar sind. Falls sich die Topologie des Netzwerk verändert, wird die Middleware über die Controller benachrichtigt und die Datenbank wird aktualisiert. Zusätzlich werden noch in einer weiteren Datenbank alle installierten Flows nach dem Matching-Kriterium geordnet gespeichert. So kann CoCoMi bei der Berechnung der Route eines Flows Rücksicht auf bestehende Flows nehmen. CoCoMi ist eine verteilte Anwendung, die auf mehreren Servern laufen kann und bei der die Daten über das Netzwerk zwischen den Instanzen synchronisiert werden. Dabei kann es vorkommen, dass vorübergehende Inkonsistenzen in den Datenbanken der Instanzen auftreten, da die Synchronisation noch nicht abgeschlossen ist.

3.1.3 Anwendungsebene

CoCoMi enthält von Kontrollanwendungen Aufträge, Routen im Netzwerk zu schreiben. Dabei macht es keinen Unterschied, welche Instanz von CoCoMi kontaktiert wird, da jede Instanz eine globale Sicht über das Netzwerk hat und über die Controller die Weiterleitungstabellen aller Switchs im Netzwerk aktualisieren kann. Die Anwendung kann entweder einen expliziten Wunsch für die Route des Flows angeben, oder sie kann den Controller nur mit dem Schreiben einer Route beauftragen, der zwei Knoten verbindet. So muss sie nicht die aktuelle Topologie des Netz kennen, aber kann zusätzliche Optionen angeben, nach welchen Kriterien der Flow berechnet werden soll. Zum Beispiel kann sie einen gewünschten Routing-Algorithmus oder Eigenschaften angeben, die bei der Erzeugung des Flows eingehalten werden sollen. Falls die Anwendung die Route selber berechnen will, muss sie vorher die aktuelle Topologie abrufen und gibt dann die notwendigen Updates an CoCoMi weiter.

3.2 Problembeschreibung

Jetzt werden die auftretende Probleme formuliert, die durch CoCoMi verhindert werden sollen. Durch Updates von unterschiedlichen Instanzen, die bei den Switchs im Netzwerk in einer unterschiedlichen Reihenfolge verarbeitet werden, können Inkonsistenzen in den Weiterleitungstabellen entstehen. Durch diese Inkonsistenzen können Kreisläufe im Netzwerk entstehen, die zur Überlastung führen und verhindern, dass Pakete am gewünschten Ziel ankommen.

3.2.1. Konkurrierende Updates und Flows

Falls die Middleware innerhalb kurzer Zeit nebenläufig zwei unterschiedliche Flows mit identischem Matching-Kriterium im Netzwerk installieren will, kann der Fall auftreten, dass ein Switch zwei Updates $\mathcal{U}_{\text{Switch1}}$ und $\mathcal{U}_{\text{Switch2}}$ empfängt, die dem gleichen Matching-Kriterium m_1 unterschiedliche Ausgangsports p_1 und p_2 zuweisen. Der Switch würde beide Updates seriell ausführen und der später

erhaltene würde den früheren überschreiben und dauerhaft gespeichert werden. Entscheidend dabei ist, wann die Switch-Updates empfangen wurden und nicht wann sie versendet wurden. Die Dauer, die eine Nachricht im Netzwerk unterwegs ist, und der Zeitpunkt, an dem sie ankommt, ist bei zwei in kurzem Abstand versendeten Updates dafür verantwortlich, welches Update dauerhaft implementiert wird. Falls nun die Reihenfolge von Switch-Updates, die das gleicher Matching-Kriterium besitzen, aber einen unterschiedliche Route schreiben wollen, bei zwei Switchs variiert, kann dies dazu führen, dass die Flows nach der abgeschlossen Aktualisierung nur teilweise installiert wurden und Problemen auftreten.

Für die Formulierung des Problems werden zuerst konkurrierende Switch-Updates definiert:

> Definition: Konkurrierende Switch-Updates:
> Zwei Switch-Updates $\mathcal{U}_{Switch1}=(S_1, m_1, p_1)$ und $\mathcal{U}_{Switch2}=(S_2, m_1, p_2)$ und sind konkurrierende Updates, falls $S_1=S_2$, $m_1=m_2$ und $p_1 \neq p_2$.

Um die Definition von Konkurrenz auf Flow-Updates zu erweitern, wird eine Menge definiert, die alle konkurrierenden Updates der beiden Flows enthält:

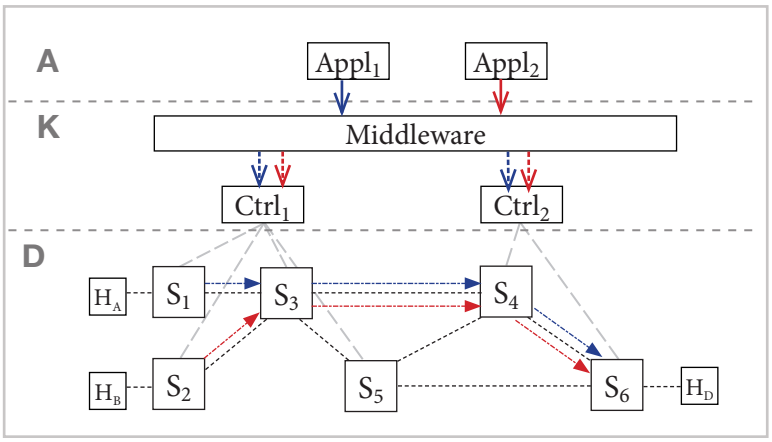
> Definition: Menge aller konkurrierenden Switch-Updates zweier Flow-Updates:
> In der Menge $\mathcal{K}(\mathcal{U}_{Flow1}, \mathcal{U}_{Flow2})$ sind alle Switch-Update-Paare $(\mathcal{U}_{Switch\,k}, \mathcal{U}_{Switch\,m})$ mit $\mathcal{U}_{Switch\,k} \in \mathcal{U}_{Flow1}$ und $\mathcal{U}_{Switch\,m} \in \mathcal{U}_{Flow2}$ enthalten, die miteinander konkurrieren.

Mit Hilfe Menge der konkurrierenden Updates wird die Konkurrenz für Flows in einem Netzwerk definiert:

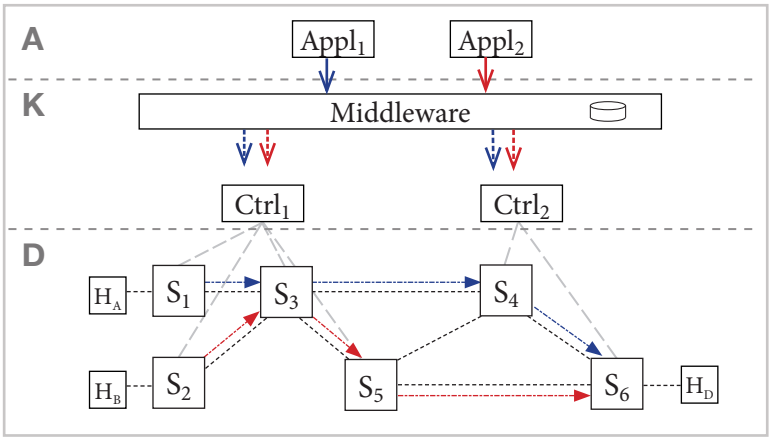
> Definition: Konkurrierende Flows
> Zwei Flows $\mathcal{F}_1$ und $\mathcal{F}_2$ konkurrieren miteinander falls $\mathcal{K}(\mathcal{F}_1, \mathcal{F}_2) \neq \{\}$.

In einem Beispiel werden nun Flows in einem Netzwerk auf Konkurrenz untersucht. In Abb. 5.1 sieht man in einem Netzwerk mit sechs Switchs und drei Hosts, wie durch Anwendungen unterschiedliche Routen beauftragt werden. Die Kontrollanwendung $Appl_1$ will eine Route von H_A nach H_D schreiben und $Appl_2$ eine Route von H_B nach H_D. CoCoMi empfängt die dazugehörigen Requests und berechnet anhand der Topologie des Netzwerks das dazugehörige Flow-Update. Der mit blauen Pfeilen im Netzwerk dargestellte Flow entspricht der Route von $Appl_1$ und der mit roten Pfeilen der Route von $Appl_2$. Die beiden Flow-Updates wurden durch zwei CoCoMi-Instanzen unabhängig voneinander berechnet und besitzen das identische Matching-Kriterium für Pakete, die an den Adressaten H_D geschickt werden. Bei beiden Flows sorgt die Weiterleitung, die in gestrichelten Pfeilen im Netzwerk angedeutet ist, dafür, dass die gefilterten Pakete später auch H_D erreichen.

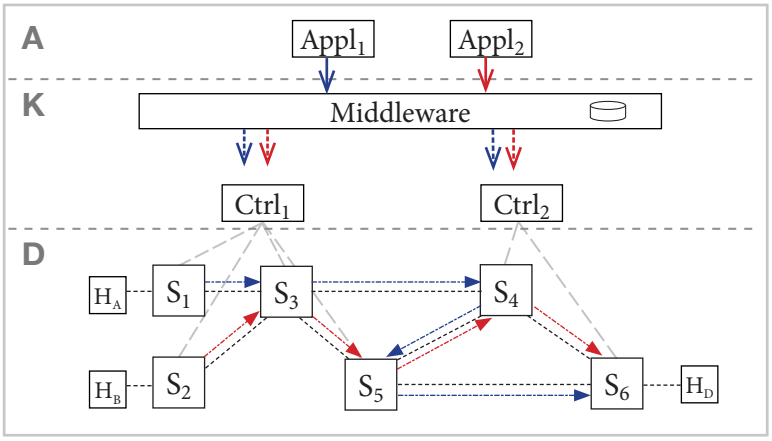
Die Switchs in dem Netzwerk werden von zwei Controllern gesteuert. $Ctrl_1$ verwaltet die Anfragen für S_1, S_2, S_3 und S_5, $Ctrl_2$ für S_4 und S_6, CoCoMi berechnet die nötigen Switch-Updates und versendet



(b): zwei konkurrierende Flows mit $|\mathcal{K}|=1$

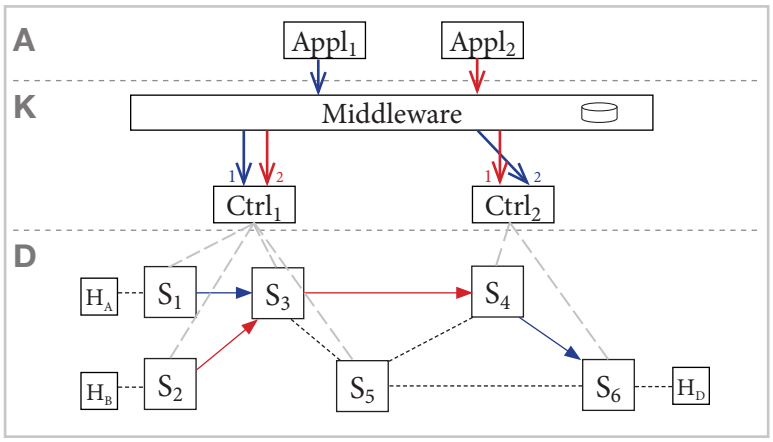


(b): zwei konkurrierende Flows mit $|\mathcal{K}|=1$

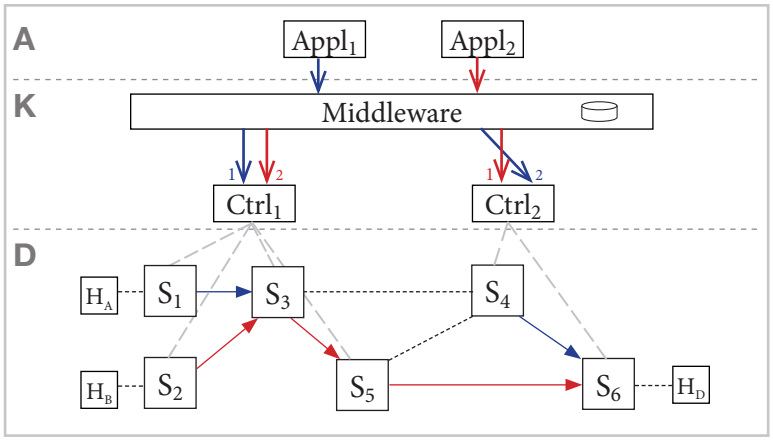


(c): zwei konkurrierende Flows mit $|\mathcal{K}|=3$

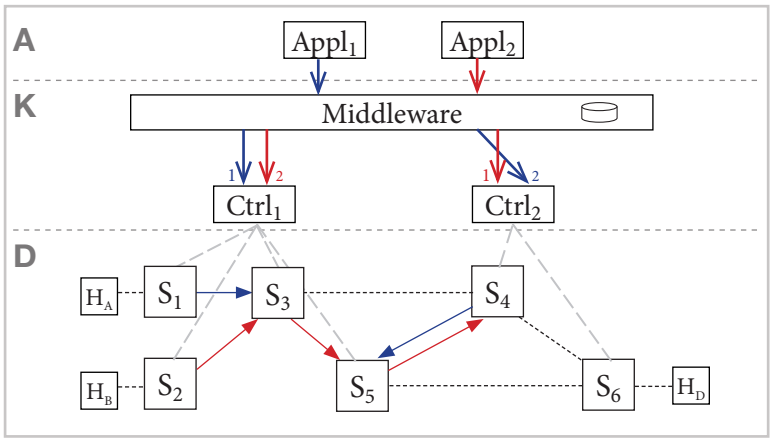
Abb.3.2: Beispiele für Konkurrenz bei Flows



(a) zwei nicht konkurrierende Flows werden Pakete wie gewünscht weitergeleitet



(b) zwei konkurrierende Flows: Pakete kommen am Ziel-Host an



(c): zwei konkurrierende Flows: Pakete kommen nicht an

Abb. 3.3: Beispiele für Inkonsistenzen in Weiterleitungstabellen

diese an die zuständigen Controller. In Abb 3.2(a) existiert bei den zwei berechneten Flows keine Konkurrenz, da in beiden Varianten bei allen Updates, die an einen Switch geschickt werden, ein Paket auch immer über den gleichen Ausgangsport weitergeleitet werden soll. Man sieht es z.B. am Switch S_3, der Pakete an H_D in beiden Flows an Switch S_4 weiterleiten muss. Dadurch macht es keinen Unterschied, welches Switch-Update als erstes oder zweites bei einem Controller ankommt.

Bei den zwei Flows in Abb. 3.2(b) unterscheidet sich bei Switch S_3 der von den Flows gewünschte Ausgangsport, wodurch es ein konkurrierendes Switch-Update-Paar gibt. In dem blauen Flow sollen die Pakete über den Ausgangsport an S_4, in dem roten Flow an S_5 weitergeleitet werden. Die Menge der konkurrierenden Updates der beiden Flows enthält dann genau dieses Update-Paar und das Kriterium für die Konkurrenz bei Flow-Updates ist erfüllt. In Abb. 3.2(c) kommen zusätzlich zu diesem Paar noch zwei weitere Paare dazu, nämlich die Updates bei Switch S_4 und S_5. Hier können abhängig von der Reihenfolge, in der die Updates bei den Switchs ankommen, zahlreiche unterschiedliche Kombination von Einträgen in den Weiterleitungstabellen entstehen.

Eine Ursache für konkurrierende Updates kann sein, dass die Flows durch zwei unterschiedliche CoCoMi-Instanzen berechnet werden, ohne dass diese voneinander wissen. Falls sie zu dem Zeitpunkt eine unterschiedliche Sicht auf das Netzwerk haben, da es kurzfristig Veränderungen gab, werden unterschiedliche Flow-Updates berechnet. Eine andere Möglichkeit ist, dass die Instanzen unterschiedliche Routing-Algorithmen verwenden, die aufgrund des Wunschs der Anwendung gewählt wurden. Zwei konkurrierende Flow-Updates können nicht nebenläufig installiert werden, das sie immer Updates für mindestens einen Switch enthalten, der dem gleichen Matching-Kriterium einen unterschiedlichen Ausgangsport zuweist. Deswegen wird bei konkurrierenden Flow-Updates immer nur höchstens ein Flow-Update komplett installiert, aber wie man im nächsten Abschnitt sehen wird, kann es auch vorkommen, dass keiner von beiden vollständig realisiert wird.

3.2.2 Inkonsistenzen in Weiterleitungstabellen

Von Inkonsistenzen in Weiterleitungstabellen spricht man, wenn die Einträge in den Weiterleitungstabellen nach der vollständigen Installation zweier konkurrierender Flow-Updates unterschiedliche konkurrierende Updates von beiden Flows enthalten. Inkonsistenz wird wie folgt definiert:

> Definition: Inkonsistente Einträge in Weiterleitungstabellen
> Zwei Weiterleitungstabellen-Einträge $\mathcal{E}_i = (m_i, p_i)$ auf Switch S_i und $\mathcal{E}_j = (S_j, m_j, p_j)$ auf Switch S_j sind inkonsistent, falls die beiden dazugehörigen Updates $\mathcal{U}_{\text{Switch i}}$ und $\mathcal{U}_{\text{Switch j}}$ nicht von einem Flow-Update stammen und miteinander konkurrieren.

Durch die Definition kann man Rückschlüsse auf die Anzahl der konkurrierenden Updates zweier Flows ziehen, die benötigt werden, um inkonsistente Einträge zu verursachen:

Beobachtung:
Um Inkonsistenzen in den Weiterleitungstabellen zu verursachen muss bei zwei Flow-Updates $\mathcal{U}_{\text{Flow1}}$ und $\mathcal{U}_{\text{Flow2}}$ die zugehörige Menge an konkurrierenden Updates $|\mathcal{K}(\mathcal{U}_{\text{Flow1}}, \mathcal{U}_{\text{Flow2}})| > 1$ sein.

In Abb. 3.3 sieht man drei Möglichkeiten, wie die Switch-Updates aus Abb. 3.2 verarbeitet werden. Hierbei wurde die Reihenfolge der Switch-Updates so bestimmt, dass bei Controller Ctrl_1 zuerst die Updates für den blauen Flow ankommen und weitergeleitet werden, bei Ctrl_2 hingegen die für den roten Flow. Die unterschiedliche Reihenfolge der Switch-Updates entsteht hier zwischen CoCoMi und Controllern, wobei es genauso möglich ist, dass sich die Reihenfolge erst beim Versand zwischen Controllern und Switchs ändert. Es wird nun untersucht, ob durch die unterschiedliche Reihenfolge inkonsistente Einträge entstanden sind. In Abb. 3.3(a) sieht man, dass es bei Updates, die nicht konkurrieren, zu keinen inkonsistenten Einträgen und zu keinen Problemen kommt. Hier macht es für die Controller keinen Unterschied, welches Switch-Update als erstes und welches als zweites beim Controller ankommt. Auch wenn bei Switchs des Controller Ctrl_1 dauerhaft der rote und bei Switchs von Ctrl_2 der blaue Flow installiert wurden, führt dies zu keinen Problemen, da beide Flow-Updates den Paketen die gleichen Ausgangsports zuweisen.

In Abb. 3.3(b) hat man bei Switch S_3 ein konkurrierendes Update-Paar und sieht, dass in dessen Weiterleitungstabelle der Update des roten Flows als zweites installiert wurde. Die Pakete werden aber hier wie gewünscht an H_D weitergeleitet und es entstehen keine Inkonsistenzen, da nur ein konkurrierendes Update vorliegt. Der Schreibvorgang ist identisch mit einem seriellen Ablauf der Aktualisierungen, bei dem beide Flows nacheinander geschrieben werden und der rote den blauen Flow-Update überschreibt.

Eine Inkonsistenz in den Weiterleitungstabellen entsteht erst in Abb. 3.3 (c), wo nach der Verarbeitung der Updates eine Schleife entsteht, die verhindert, dass Pakete ihr Ziel erreichen. Diese Schleife ist nicht nur vorübergehend während einer Installation vorhanden, sondern bleibt so lange erhalten, bis ein neuer Flow installiert wird. Es gibt in dem Beispiel drei konkurrierende Update-Paare, wobei für die Entstehung der Schleife entscheidend ist, dass S_4 den blauen Flow und S_5 den roten Flow dauerhaft installiert. Dadurch schicken die beiden Switchs die Pakete jeweils zurück, während zusätzliche Pakete dazukommen und keines die Schleife verlässt. Solche Schleifen müssen in einem Netzwerk verhindert werden, da sie schnell zu Überlastungen führen und verhindern, dass Pakete ihr Ziel erreichen.

Die Middleware erfährt nichts von diesen Inkonsistenzen, da sie eine Bestätigung für die erfolgreiche Aktualisierung der Weiterleitungstabellen von den Controllern erhält und nicht wissen kann, dass diese teilweise gleich wieder überschrieben wurden. Sie müsste erst Statusnachrichten von den Switchs einfordern und die erfolgreiche Installation des Flows überwachen, was einen großen Aufwand bedeuten würde.

3.3 Ziel dieser Arbeit

Ziel dieser Arbeit ist, Konzepte und Mechanismen für die Middleware zu entwickeln, die verhindern, dass nach einer vollständigen Installation nebenläufiger Flow-Updates Inkonsistenzen in den Weiterleitungstabellen auftreten können. Um Eigenschaften für den Schreibvorgang zu definieren, wird der Ablauf einer Aktualisierung mit Transaktionen in verteilten Datenbanken verglichen, bei denen für eine zuverlässige Ausführung die vier Eigenschaften Atomizität, Konsistenz, Isolation und Dauerhaftigkeit erfüllt sein müssen. Im folgendem wird untersucht, wie diese vier Eigenschaften auf das Schreiben von Routen in unserem SDN-Netzwerk übertragen werden können und welche von ihnen in dieser Arbeit erfüllt sein müssen.

3.3.1 Konsistenz

Jedes SDN-Netzwerk muss durch einen Schreibvorgang einer Route von einem konsistenten Zustand in einen neuen konsistenten Zustand überführt werden. In dieser Arbeit bedeutet ein konsistenter Zustand des Netzwerks, dass keine inkonsistenten Einträge in den Weiterleitungstabellen vorhanden sind und bei einem Schreibvorgang mehrerer Flows am Ende mindestens einer vollständig im Netzwerk installiert wurde. Dabei ist nicht gefordert, dass dies der zeitlich am letzten beauftragte Switch ist. Vorübergehende Inkonsistenzen während einer Aktualisierung sind toleriert und es wird nur gefordert, dass die Konsistenz nach Abschluss der Schreibvorgänge im Netzwerk vorhanden ist (*eventual consistency*). Die Konsistenz ist das grundlegende Ziel dieser Arbeit und muss auf jeden Fall erfüllt sein.

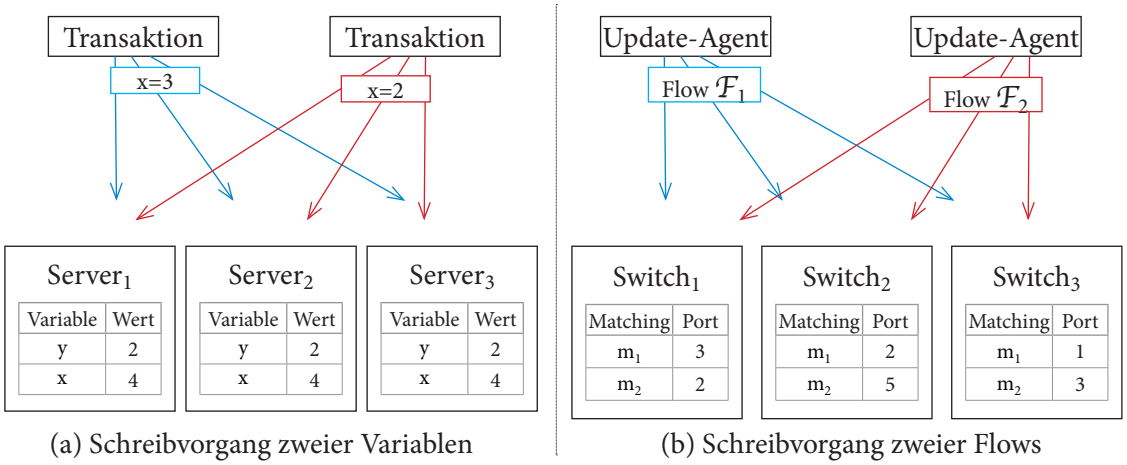
3.3.2 Isolation

Für die Isolation einer Aktualisierung muss diese so ausgeführt werden, als würde der Update-Manager als einziger Teilnehmer im System die Einträge in den Weiterleitungstabellen verändern. Um zu beurteilen, ob die Isolation durch einen Update-Agenten gewährleistet ist, muss man sich analog zu Transaktionen den Schedule der Updates anschauen, der beschreibt, wann sie von den Switchs ausgeführt werden. Der Effekt der Switch-Updates bei mehreren nebenläufigen Aktualisierungen muss dem eines seriellen Schedules entsprechen, bei dem die Schreibvorgänge der Flows nacheinander durch die Update-Agenten ausgeführt werden. Eine Folge einer seriellen Ausführung der Schreibvorgänge ist, dass der zuletzt geschriebene Flow immer komplett im Netzwerk installiert ist. Wenn man nun Inkonsistenzen in Weiterleitungstabellen betrachtet, sieht man, dass von zwei Flows keiner komplett installiert wurde und dies deswegen keiner seriellen Ausführung von Flow-Updates entspricht. Umgekehrt kann man bei der seriellen Ausführung von Aktualisierungen der Weiterleitungstabellen zeigen, dass keine Inkonsistenzen entstehen können, da immer die konkurrierenden Updates der späteren Flows die des früheren überschreiben.

Darüber hinaus wird auch gefordert, dass der Schedule Eine-Kopie-serialisierbar ist. Dazu wird

zuerst der Begriff der Eine-Kopie-Serialisierbarkeit auf ein SDN-System übertragen. Wenn man ein SDN-Netzwerk mit einer transaktionalen Datenbank vergleicht, stellt man fest, dass der Ablauf eines Schreibvorgangs von Flows vergleichbar mit dem Schreibvorgang von Variablen ist, die dupliziert in einer verteilten Umgebung auf mehreren Servern gespeichert wird. In Abb. 6.1(a) sieht man, wie bei Transaktionen Werte von Variablen auf unterschiedlichen Servern als Kopien gespeichert werden. Bei der Eine-Kopie-Serialisierbarkeit von Transaktionen sind die Auswirkungen der Schreib- und Lesevorgänge identisch mit der Variante, in der die Variablen nicht dupliziert gespeichert werden.

In Abb. 6.1(b) wird der Vorgang auf ein SDN-Netzwerk übertragen und zwei Update-Agenten entsprechen den Transaktionen. Sie schreiben Flows mit je einem Matching-Kriterium und weisen diesen unterschiedliche Ausgangsports auf den Switchs zu. Das Aktualisieren des Werts einer Variable durch eine Transaktion entspricht hier dem Aktualisieren eines Eintrags in der Weiterleitungstabelle. Nach Verarbeitung der beiden Updates müssten bei der Eine-Kopie-Serialisierbarkeit nicht wie bei Transaktionen alle Variablen den gleichen Wert haben, sondern ein Matching-Kriterium muss den Ausgangsport immer bezüglich eines Flow-Updates zugewiesen bekommen haben. Die Eine-Kopie-Serialisierbarkeit würde hier bedeuten, dass die Auswirkungen des Schreibvorgangs vergleichbar mit der Variante sind, dass alle Switchs auf einem Switch vereint wären und der Agent nur eine Update-Nachricht pro Flow verschicken müsste. Bei dieser Umgebung mit nur einem Switch ist die Reihenfolge der zu schreibenden Flows klar definiert und man sieht, dass die Eine-Kopie-Serialisierbarkeit durch eine Totalordnung der Flow-Schreibvorgänge realisiert wird.



(a) Schreibvorgang zweier Variablen (b) Schreibvorgang zweier Flows

Abb. 3.4 : Vergleich von Eine-Kopie-Serialisierbarkeit bei Datenbanken und Switchs

3.4 Weitere Eigenschaften der Aktualisierung

Zusätzlich wird noch auf Eigenschaften eingegangen, die nicht für die Konsistenz der Weiterleitungstabellen erforderlich sind. Atomizität kann bei Netzwerken gewünscht sein, falls die Anzahl fehlgeleiteter Pakete reduziert werden soll. Dauerhaftigkeit hingegen ist generell sei Software-Defined Networks nicht realisiert.

3.4.1 Atomizität

Falls ein Controller während des Einrichtens eines Flows Weiterleitungstabellen mehrerer Switchs aktualisiert, garantiert die Atomizität, dass entweder alle Updates erfolgreich installiert werden oder keine (Alles-oder-nichts-Eigenschaft). Unvollständige Flow-Installationen entstehen dadurch, dass Updates im Netzwerk verloren gehen oder dass Controller beim Verschicken der Nachrichten abstürzen.

In SDN-Netzwerken muss Atomizität nicht gefordert sein, da davon ausgegangen wird, dass unvollständige Installationen von Flows durch die Switchs entdeckt werden und der zuständige Controller benachrichtigt wird. Falls nämlich bei einem Switch ein Update nicht installiert wurde, bekommt er Pakete, die er in der Weiterleitungstabelle keinem Ausgangsport zuordnen kann. Diese Pakete leitet er dann weiter an den Controller, dem dadurch signalisiert wird, dass ein Flow unvollständig installiert wurde. Der Controller kann dann den kompletten Flow noch einmal installieren oder versuchen herauszufinden, bei welchem Switch das Update nicht durchgeführt wurde. Die Suche nach dem fehlgeschlagenem Update ist oft schwierig, da nicht immer auf dem Switch, der das nicht zu verarbeitende Paket meldet, das Update fehlgeschlagen ist. Es kann auch sein, dass das Paket vorher durch einem nicht überschriebenen Eintrag auf einem veralteten Flow weitergeleitet wurde.

Bei Netzwerken mit niedriger Fehlertoleranz, in denen Verzögerungen verhindert werden sollen, die dadurch entstehen, dass fehlgeleitete Pakete vom Controller erst verarbeitet werden sollen, kann Atomizität aber gewünscht sein. Deswegen wird später eine Variante vorgestellt, in der Atomizität realisiert wird.

3.4.2 Dauerhaftigkeit

Die Dauerhaftigkeit der Aktualisierung der Weiterleitungstabellen ist kein Ziel dieser Arbeit, da Switchs die Tabellen nur auf flüchtigem Speicher ablegen, um Schreibvorgänge und Weiterleitung schnell ausführen zu können. Falls der Switch abstürzt, gehen die Daten verloren und wenn er wieder hochfährt, muss er ankommende Pakete an den Controller weiterleiten, da er keine passenden Einträge in der leeren Weiterleitungstabelle findet. Der Controller wird so über den Absturz des Switchs informiert und muss die Flow-Einträge neu installieren.

4 Entwurf

In diesem Kapitel wird die Architektur von der Controll Coordination Middleware (CoCoMi) vorgestellt und gezeigt wie ein Route-Request verarbeitet wird, so dass keine Inkonsistenzen in den Weiterleitungstabellen durch die erzeugten Switch-Updates entstehen. In CoCoMi wird für jeden Route-Request ein Update-Agent gestartet, der dafür zuständig ist die gewünschten Switch-Updates zu erzeugen und an die Controller zu verschicken. Dieser Update-Agent kann in zwei Varianten gestartet werden. In der Locking-Variante fordert der Agent zuerst eine Schreibsperre über alle Switchs der Route an und kann die Switch-Updates erst nach dessen Bestätigung an die Controller verschicken. In der Logischen-Uhr-Variante werden logische Uhren verwendet, um Zeitstempel für die Updates zu erzeugen, mit deren Hilfe eine Totalordnung unter den Flow-Updates hergestellt wird.

4.1 Architektur der Middleware

CoCoMi wird als verteilte Anwendung realisiert, die auf mehreren Rechnern gestartet werden kann. In Abb. 4.1 sieht man Instanzen, die auf unterschiedlichen Servern laufen und Routen in einem Netzwerk schreiben können. Eine Kontrollanwendung beauftragt CoCoMi über einen Request, eine bestimmte Route im Netzwerk zu schreiben. Die Initiator-Komponente der beauftragten CoCoMi-Instanz verarbeitet den Request und startet darauf einen Update-Agenten, der dafür verantwortlich ist, die notwendigen Flow-Updates zu erzeugen, zu verschicken und zu kontrollieren, dass diese auch im Netzwerk implementiert werden. Falls die Einträge in den Weiterleitungstabellen erfolgreich geschrieben wurden, informiert der Update-Agent den Initiator über die erfolgreiche Installation des Flows und wird geschlossen.

Um zu verhindern, dass abgestürzte Update-Agenten nicht entdeckt werden, besitzt jeder Agent eine Lebenszeit, die er vom Initiator zugewiesen bekommt. Falls ein Agent dem Initiator innerhalb seiner Lebenszeit die erfolgreiche Installierung einer Route nicht bestätigt hat, geht der Initiator davon aus, dass dieser abgestürzt ist und beauftragt einen neuen Agenten mit dem Schreiben der Route. Ein Agent wiederum kennt seine Lebenszeit und beendet sich automatisch, falls diese abgelaufen ist. Dies ist besonders wichtig, wenn der Initiator aufgrund einer Partitionierung des Netzwerks nicht mehr mit dem Agenten kommunizieren kann. So wird verhindert, dass zu einem Schreibvorgang mehrere Agenten existieren.

Alternativ ist es möglich CoCoMi als zusätzliche Komponenten umzusetzen, die in den Kontroll-anwendungen und den Controllern installiert wird. In Abb. 4.3 sieht man, wie die Funktionen

von CoCoMi auf CCM_{App} und CCM_{Ctrl} aufgeteilt wurden, die miteinander kommunizieren. Die Datenbank würde als separater Dienst implementiert werden, der auch dauerhaft besteht, falls Kontrollanwendungen beendet werden.

Die Update-Agenten sind zuständig für die konsistente Aktualisierung der Weiterleitungstabellen. Dabei werden zwei unterschiedliche Agenten vorgestellte, die mit unterschiedlichen Mechanismen für die Konsistenz in den Weiterleitungstabellen garantieren.

4.2 Locking-Agent

Wenn der Initiator einen Locking-Agenten startet, wird mit Hilfe eines Locking-Mechanismus die Isolation des Flow-Updates gesichert. Dazu ist in CoCoMi auf jeder Instanz eine Locking-Komponente installiert, die mit den anderen Instanzen synchronisiert wird. Bevor der Agent die Updates an die Switchs herausschickt, muss er über den Locking-Dienst eine Sperre über das Matching-Kriterium und die Switchs beantragen, deren Weiterleitungstabellen geändert werden sollen. Das Matching-Kriterium identifiziert einen Flow, und kann einzelne oder mehrere Bedingungen an Eigenschaften der Pakete haben wie z.B. IP-Adresse des Ziels, MAC-Adresse der Quelle oder eine Multicast-Gruppen-Adresse. Wenn die Sperre von allen CoCoMi-Instanzen bestätigt

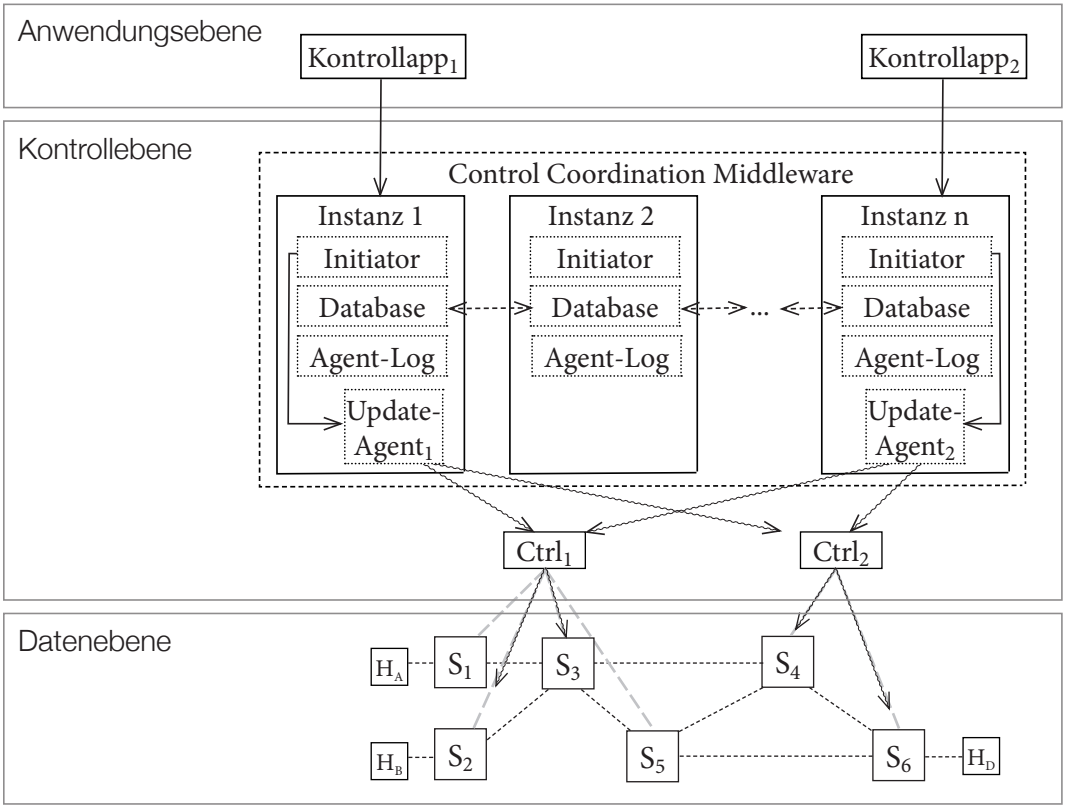


Abb.4.1 : Übersicht der Architektur von CoCoMi

wird, kann der Agent damit anfangen, die nötigen Switch-Updates an die Controller zu verschicken. So kann im Netzwerk zu einem Matching-Kriterium immer nur ein einzelner Update-Agent die Weiterleitungstabellen aktualisieren und es wird verhindert, dass zwei konkurrierende Flow-Updates von Agenten nebenläufig verschickt werden.

Ein weiteres Ziel des Locking-Agenten ist, dass alle Einträge in den Weiterleitungstabellen auf einem aktuellen Stand sind und sich auf ihnen keine veralteten Einträge befinden. Nachdem der Agent eine Sperre über den Flow erhalten hat, fragt er zuerst die Datenbank ab, um alle vorhandenen Einträge zu dem Matching-Kriterium in dem Netzwerk zu erhalten. Dadurch kann er vorhandene Flows umschreiben, indem er gezielt Einträge von bestimmten Switchs überschreibt und veraltete Einträge proaktiv löscht. Dies hat zur Folge, dass veraltete Einträge nicht erst durch den Timeout-Mechanismus gelöscht werden und die Größe der Weiterleitungstabellen schneller reduziert wird, was sinnvoll sein kann, da die Größe des verfügbaren Speichers auf den Switchs beschränkt ist. Zusätzlich kann auch in manchen Fällen die Anzahl der zu versendenden Switch-Updates reduziert werden, wenn alte Flows berücksichtigt werden.

Als zusätzliche Eigenschaft wird durch das Write-Ahead-Log Protokoll die Atomizität des Schreibvorgangs der Route gesichert. Bevor eine Agent ein Switch-Update verschickt, speichert er immer einen Eintrag im Log ab, in dem der Inhalt und Adressat des Updates gespeichert wird. Genauso wird im Log vermerkt, falls er eine Nachricht über die erfolgreiche Aktualisierung eines Eintrags in einer Weiterleitungstabelle erhält. So kann der Initiator bei einem abgestürzten Agenten nachvollziehen, welche Updates dieser schon verschickt hat und der neu erzeugte Agent verschickt nur die Updates, von denen der vorherige Agent noch keine Bestätigungen empfangen hat.

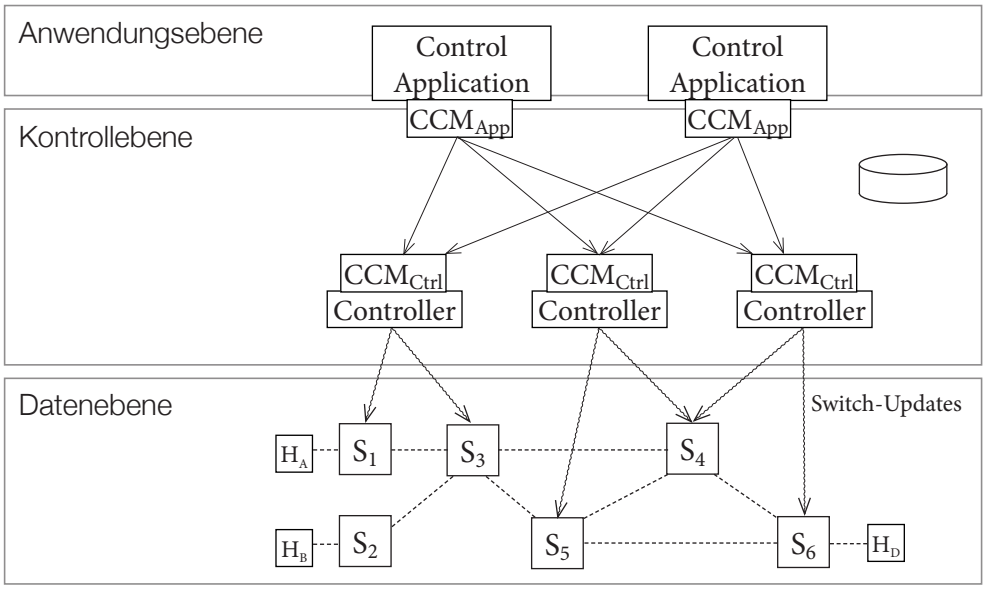
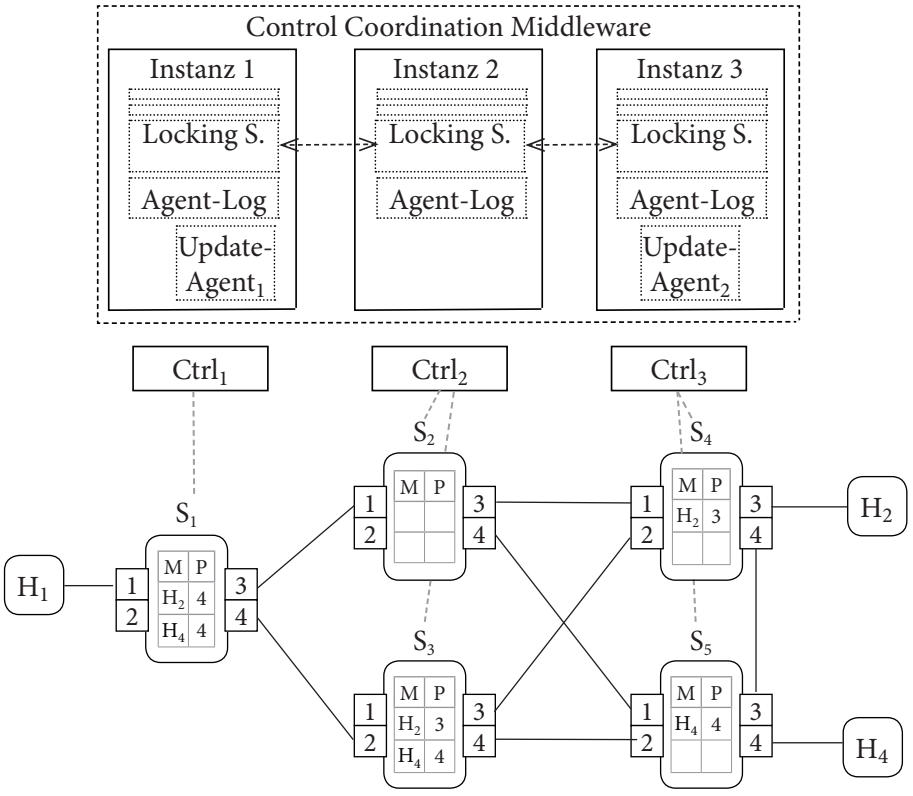


Abb. 4.2 Verteilte Umsetzung von CoCoMi

Den regulären Ablauf eines Locking-Agenten sieht man in Algortihmus 1 und 2. Der Intiator empfängt den Route-Request von der Kontrollapplikation und startet den Agenten. Zuerst beantragt dieser eine Sperre über das Matching-Kriterium und der Switchs und falls er diese erhalten hat, beginnt er den Flow-Update zu berechnen. Er fragt die aktuelle Topologie des Netzes und die zu dem Matching-Kriterium schon installierten Flows ab, und berechnet die benötigten Flow-Updates. Danach versendet er nacheinander an die Controller die Switch-Updates und speichert vor dem Senden immer ein Eintrag im Log ab. Nachdem alle Updates verschickt wurden, wird die Datenbank aktualisiert und die Sperre aufgehoben. Am Ende wird der Initiator informiert, dass der Schreibvorgang erfolgreich abgeschlossen wurde.

In Abb. 4.3 sieht man eine Netzwerk mit drei Controllern, denen insgesamt fünf Switchs zugeordnet sind. Jeder Switch hat vier Ports, über die er mit anderen Switchs und Hosts verbunden werden kann, und eine Weiterleitungstabelle, die zwei Spalten enthält: eine für das Matching-Kriterium und eine für den zugehörigem Ausgangsport. Das Matching-Kriterium entspricht hier dem Adressaten, der in den Paketen angegeben ist. S_1 leitet zum Beispiel ankommende Nachrichten, die an H_2 adressiert sind, zum Ausgangsport 4 weiter, der mit Switch S_3 verbunden ist. Von dort werden Pakete weiter an S_4 geleitet, der sie an H_2 übermittelt. Nachrichten, die an den Host H_4 adressiert sind, werden



(a) CoCoMi mit Locking-Service

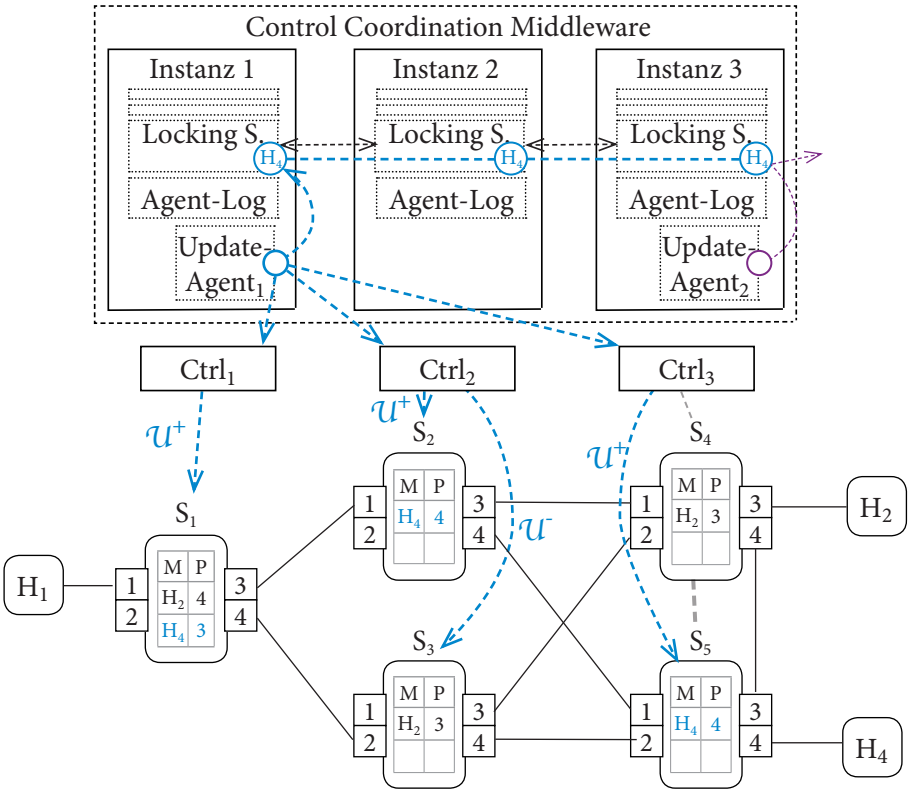
Abb.4.3 : Beispiel für Updates mit Hilfe des Locking-Agenten

Algorithmus 1: Start eines Loking-Agenten durch den Initiator

```
1:  Route r = receive()
2:  startLockingAgent(r)
```

Algorithmus 2: Versand eines Flow-Updates durch Locking-Agenten

```
1:  Matching m = r.getMatching()
2:  if ( waitForLock(m,r) == true) do
3:          Topology T = DB.getTopology()
4:          Flows f = DB.getFlows(m)
5:          U_Flow = calculateUpdates(r, T, f)
6:          for ( U_Switch ∈ U_Flow) do
7:              log.register(U_Switch)
8:              sendUpdate(U_Switch)
9:          od
10:     DB.update(U_Flow)
11:     unlock(m)
12:     informInitiator()
13: od
```



(b) Update-Agent schreibt Flow (H_3 , $H_1 \rightarrow S_1 \rightarrow S_2 \rightarrow S_5 \rightarrow H_4$)

Abb.4.4 : Beispiel für Updates mit Hilfe des Locking-Agenten

hingegen von S_3 über Port 4 an S_5 weitergeleitet. An den Einträgen in den Weiterleitungstabellen erkennt man, dass in dem Netzwerk zwei Flows installiert sind, die Pakete an H_2 und H_4 weiterleiten. CoCoMi ist in diesem Beispiel auf 3 Instanzen verteilt und man sieht den verteilten Locking-Dienst und den Agent-Log, die für den Locking-Agenten benötigt werden.

In Abb. 4.4 führt der Update-Agent eine Aktualisierung bezüglich des Flows durch, der Pakete an H_4 transportiert. Er hat dafür eine Sperre bei dem verteilten Locking-Service beantragt und diese auch von allen Instanzen des CoCoMi erhalten. Durch diese Sperre ist er der einzige Switch in dem Netzwerk, der Einträge in den Weiterleitungstabellen des Netzwerks bezüglich des Matching-Kriteriums schreiben, ändern oder löschen kann. Falls ein anderer Update-Agent von Instanz 3 beim Locking-Service eine Sperre über H_4 beantragt, wird dies abgelehnt, solange der Update-Agent seine Sperre nicht aufgehoben hat. Sperren für andere Matching-Kriterien, die sich mit dem gesperrten Kriterium nicht überschneiden, werden jedoch vergeben, so dass Einträge mit anderen Matching-Kriterien parallel von anderen Controllern verändert werden können. Ein weiterer Update-Agent könnte z.B. eine Sperre für Pakete mit Ziel H_2 beantragen und den zugehörigen Flow überarbeiten.

In der komponenten-basierten Architektur von Abb. 4.2 wäre der Locking-Agent sehr gut umsetzbar. CCM_{App} wäre für das Locking-Verfahren zuständig und müsste über einen verteilten Locking-Service die Sperre beantragen. CCM_{Ctrl} wäre bei dieser Variante gar nicht notwendig, da keine zusätzliche Funktionen beim Controller installiert werden müssten.

Algorithmus 3: Start eines Logischen-Uhr-Agenten durch den Initiator

```
1: Route r = receive()
2: Matching m = r.getMatching()
3: Timestamp ts = getClocktime() + 1
4: for ( Instance i ∈ Middleware) do
5:       sendCocktime(ts, id_i , m )
6: od
7: startLogicalClockAgent(r, ts)
```

Algorithmus 4: Versenden des Flow-Updates durch den Logischen-Uhr-Agenten

```
1: Matching m = r.getMatching()
2: Topology T = DB.getTopology()
3: U_Flow = calculateUpdates(r, T )
4: for ( U_Switch ∈ U_Flow) do
5:       sendUpdate(U_Switch)
6: od
```

4.3 Logische-Uhr-Agent

Beim Logische-Uhr-Agenten wird im Gegensatz zum Locking-Agenten ermöglicht, dass Flows zu einem Matching-Kriterium im Netzwerk nebenläufig durch mehrere Instanzen installiert werden können. Inkonsistenzen in den Weiterleitungstabellen werden hier vermieden, indem mit Hilfe einer logischen Uhr eine Totalordnung unter den Flow-Updates hergestellt wird. Die Switchs sind in diesem Fall dafür zuständig, über die Zeitstempel zu garantieren, dass konkurrierende Switch-Updates immer in der gleichen Reihenfolge ausgeführt werden. Ein verspätetes Update, dass einen Switch erreicht, bei dem schon ein logisch nachfolgendes Update implementiert wurde, wird von diesem verworfen, da er anhand der Totalordnung eindeutig entscheiden kann, ob das Update den Eintrag in seiner Weiterleitungstabelle überschreiben darf oder nicht. Dadurch dass die Totalordnung jedem Switch bekannt ist, trifft jeder Switch im Netzwerk die gleiche Entscheidung und nach dem Schreibvorgang von zwei konkurrierender Flows ist immer einer von beiden komplett im Netzwerk installiert.

Die Totalordnung wird in erster Linie durch eine logische Uhr und in zweiter durch eine Hierarchie unter den CoCoMi-Instanzen realisiert, wie man in Alg. 5 sehen kann. Als logische Uhr dient die Anzahl der Flow-Updates, die insgesamt in dem Netzwerk durchgeführt wurden. Bei jedem neu erzeugten Agenten wird die Uhrzeit erhöht und die anderen Instanzen darüber informiert, um zwischen ihnen die Uhrzeit zu synchronisieren. Jeder Update-Agent erhält einen Zeitstempel, das jedem Switch-Update anhängt wird.

Falls zwei Instanzen innerhalb kurzer Zeit zwei Update-Agenten starten, bevor sie ihre Uhrzeit synchronisieren konnten, kann es vorkommen, dass mehrere Updates den gleichen Zeitstempel besitzen. In diesem Fall ist die Id der beauftragenden Instanz entscheidend, welches Update das andere überschreiben kann. Bei der Synchronisierung der Uhrzeit fügt der Initiator die Id der Instanz und das Matching-Kriterium hinzu, damit die anderen Instanzen beurteilen können, ob der Update eventuell mit einem kürzlich von ihnen versendeten Flow-Update konkurriert, das den gleichen

Algorithmus 5: Empfang eines Switch-Updates mit Zeitstempel auf einem Switch

```
1:  Update u = receive()
2:  TableEntry t = findTableEntry(u.getMatching())
3:  if (t == null) do
4:          t = createEntry(u)
5:  else if (u.getTimestamp() > t.getTimestamp()) do
6:          t = createEntry(u)
7:  else if (u.getTimestamp() == t.getTimestamp())
8:          if (u.getInstanceId > t.getInstanceId()) do
9:                  t = createEntry(u)
10:         od
11: od
```

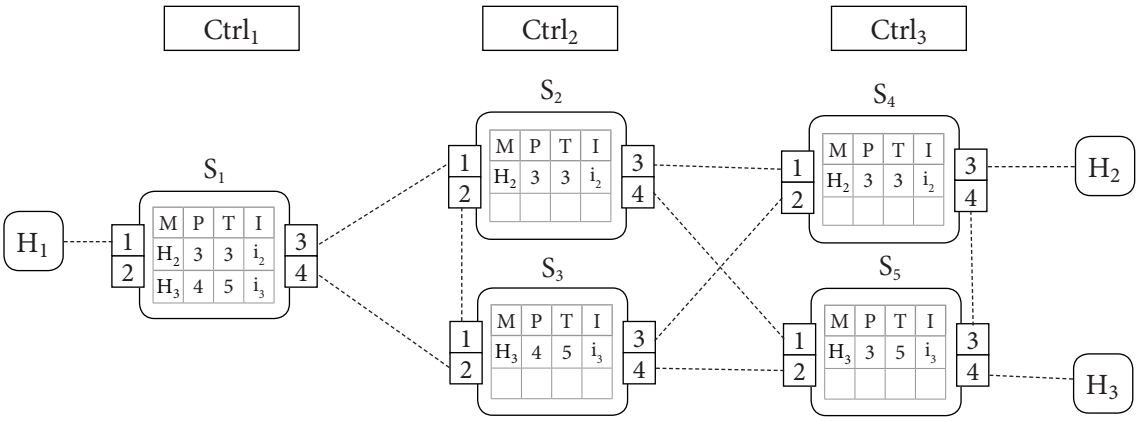


Abb.4.5: Bestehendes Netz mit Zeitstempel in den Weiterleitungstabellen:
t=5: zwei Flows sind installiert (H2 , H1→S1→S2→S4→H2); (H3 , H1→S1→S3→S5→H3)

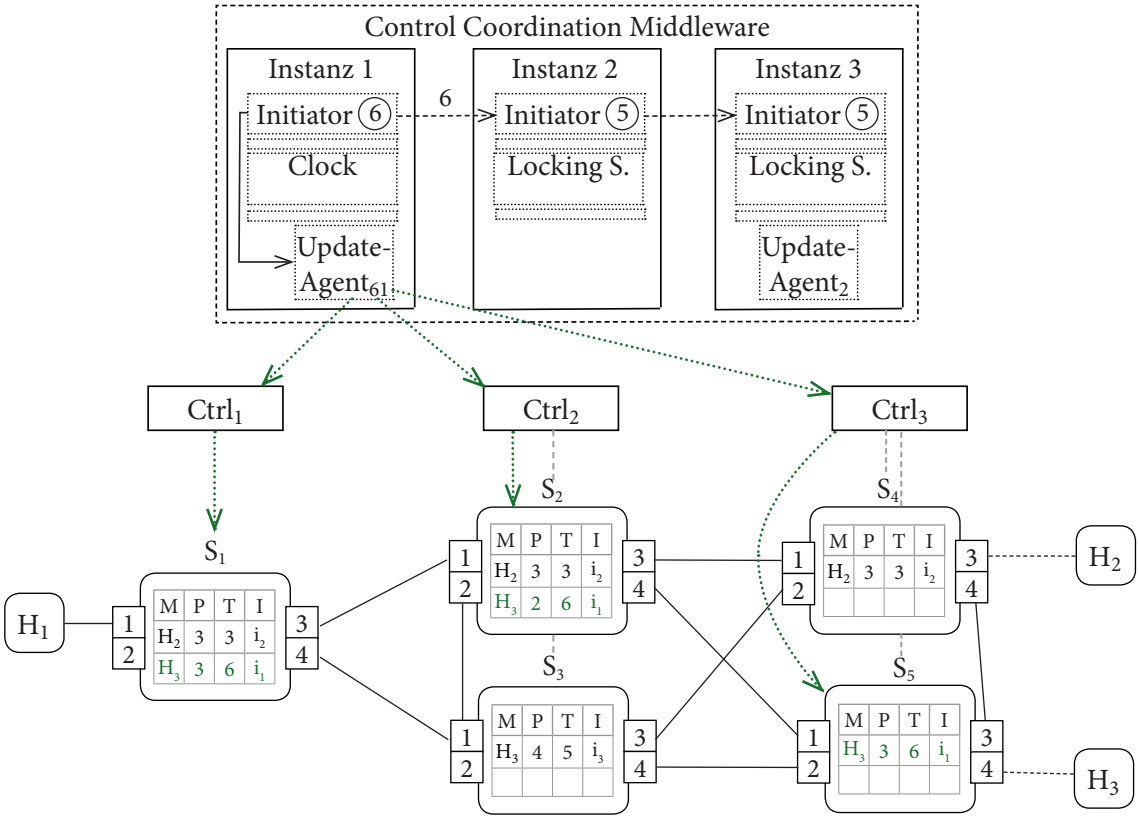


Abb.4.6: Beispiele für ein Update durch einen Logische-Uhr-Agenten:
t=6: Update-Agent$_{61}$ will Flow (H3 , H_1→S_1→S_2→S_5→H_3) einrichten

Zeitstempel besitzt. Die Initiatoren der anderen Instanzen können so anhand der Hierarchie unter den Instanzen informiert werden, ob das von ihnen versendete Update eventuell nicht berücksichtigt wurde und für diesen Fall ihr Update ein weiteres Mal mit einem höheren Zeitstempel verschicken.

In Algorithmus 3 und 4 sieht man den Ablauf, wie ein Logische-Uhr-Agent einen Flow-Update verschickt. Der Initiator empfängt zuerst die Route und berechnet einen aktuellen Zeitstempel. Dann wird der Zeitstempel mit Id der CoCoMi-Instanz und Matching-Kriterium an die anderen Instanzen verschickt, um die Uhrzeit zu synchronisieren. Nun wird der Agent gestartet und die Route und der Zeitstempel übergeben. Der Agent fragt am Anfang die Topologie des Netzes ab und berechnet anhand dieser den Flow-Update. Die Datenbank gibt beim Logischen-Uhr-Agenten nur Auskunft über teilnehmende Switchs und deren Konnektivität, nicht aber über deren Einträge in deren Weiterleitungstabellen. Ab Zeile 4 werden die Switch-Updates an die zugehörigen Controller versendet, die diese an die Switchs weiterleiten.

Um Updates von Logischen-Uhr-Agenten verarbeiten zu können, müssen Switchs in jedem Eintrag der Weiterleitungstabelle den Zeitstempel und die Id der Instanz speichern, um bei einem

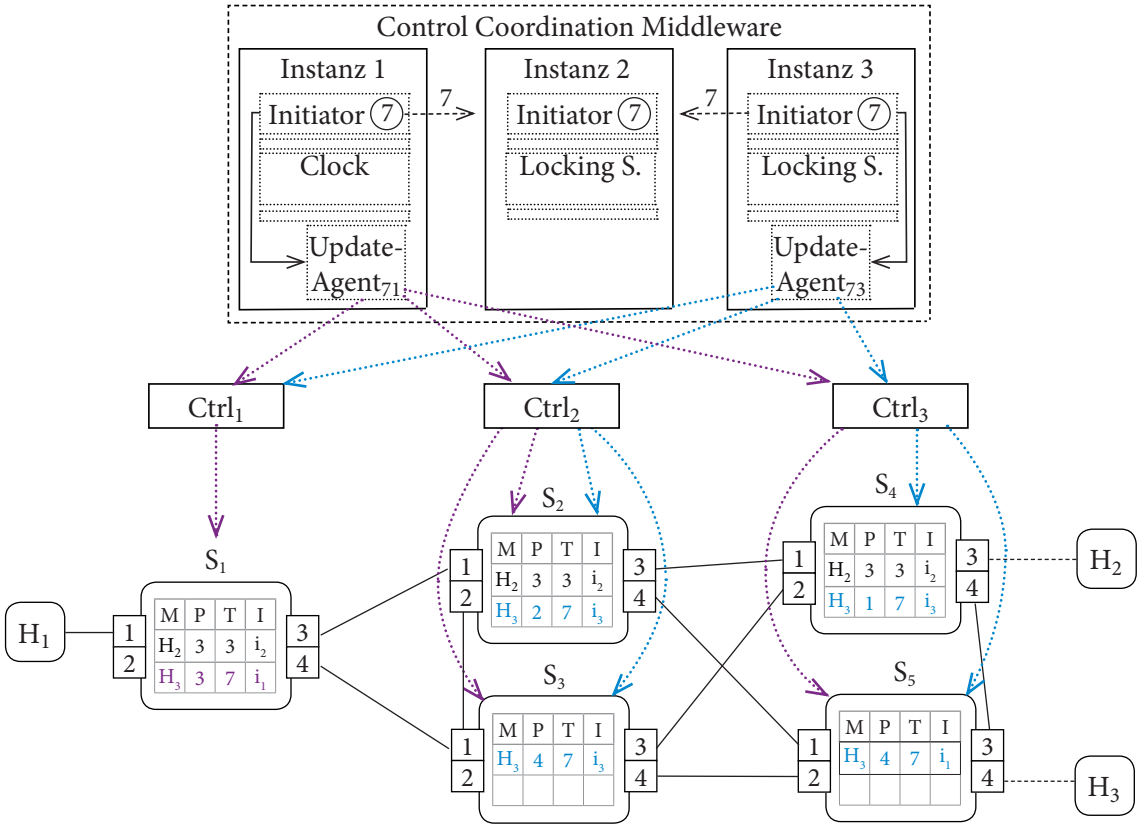


Abb.4.7: Beispiel für ein Update durch zwei konkurrierende Logische-Uhr-Agenten
t=7: Update-Agent$_{71}$ schreibt Flow(H3 , H1→S1→S3→S2→S5→H3);
Update-Agent$_{73}$ schreibt Flow(H3 , H2→S4→S2→S5→H3)

ankommenden Update zu überprüfen, ob es einen bestehenden Eintrag überschreiben darf. Im Algorithmus 5 sieht man, wie ein Switch die Informationen des Updates mit den Einträgen in seiner Weiterleitungstabelle vergleicht. Zuerst sucht er einen Eintrag mit passendem Matching-Kriterium in seiner Weiterleitungstabelle. Falls keines existiert, kann er das Update anwenden, ansonsten muss er Zeitstempel und bei Bedarf auch die Instanz-Id vergleichen, um dies zu entscheiden.

Beim Berechnen von Flows berücksichtigt der Logische-Uhr-Agent keine vorhandenen Flows und er verzichtet auch auf das proaktive Löschen von veralteten Tabellen-Einträgen, was bedeutet, dass er keine $\mathcal{U}^-$-Updates verschickt. Veraltete Tabelleneinträge in den Weiterleitungstabellen werden durch den Timeout-Mechanismus der Switchs gelöscht, bei dem Einträge entfernt werden, über die in einem bestimmten Zeitraum keine Pakete weitergeleitet wurden. Fehlerhaft installierte Flows werden dadurch entdeckt, dass Switchs keinen passenden Eintrag in der Weiterleitungstabellen haben und das Paket an einen Controller weiterleiten.

In Abb. 4.5 sieht man drei Controller und fünf Switchs, auf denen zwei Flows installiert sind. Anhand der Einträge in den Weiterleitungstabellen kann man rekonstruieren, dass der Instanz i_2 zum Zeitpunkt t=3 einen Flow für Pakete mit dem Ziel H_2 geschrieben hat. Die Instanz i_3 hingegen hat ein Flow mit Zeitstempel t=5 installiert, der Pakete mit dem Ziel H_3 weiterleitet. Durch die unterschiedlichen Paket-Ziele unterscheiden sich die Matching-Kriterien der Flows, die deswegen nicht im Konflikt stehen und in unterschiedlichen Tabellenzeilen der Weiterleitungstabellen gespeichert sind.

Auf Abb. 4.5 will CoCoMi-Instanz i_1 einen Flow für Pakete mit dem Ziel H_3 schreiben und startet deswegen ein Agenten mit Zeitstempel t=6. Um die andere Instanzen über die Veränderung der Uhrzeit zu informieren, schickt der Initiator eine Nachricht an die anderen Instanzen und informiert sie über den Zeitstempel und das Matching-Kriterium des Updates. Ein Controller, der solch eine Nachricht empfängt, übernimmt die Uhrzeit des anderen Controllers, falls diese größer als seine eigene ist. Die Updates für Switch S_1 und S_5 überschreiben in dem Beispiel die bestehenden Einträge in den Weiterleitungstabellen, da die Updates einen höheren Zeitstempel besitzen. Weil in der Tabelle von S_2 noch keine Einträge zu dem Matching-Kriterium existieren, wird das Update automatisch übernommen. Auf Switch S_3 bleibt der veraltete Eintrag bestehen, der aber automatisch vom Switch gelöscht werden wird, da an diesen Switch keine Paket mehr mit dem Ziel H_3 ankommen und der Timeout-Mechanismus aktiv wird.

Nachdem der Update mit t=6 durchgeführt wurde, wollen auf Abb. 4.7 zwei Update-Agenten je einen neuen Flow-Update starten, die zueinander im Konflikt stehen. Sie wurden nebenläufig innerhalb eines kurzen Zeitraums erzeugt und die Instanzen wissen zu diesem Zeitpunkt noch nichts von dem anderen Flow, da sie ihre Uhr noch nicht synchronisiert haben. Dadurch sind im Netzwerk zwei konkurrierende Updates mit identischem Zeitstempel und Matching-Kriterium unterwegs. Jetzt wird über die Ids der CoCoMi-Instanzen entschieden, welches Update dem anderen logisch folgt, woraus folgt, dass die Updates von $Agent_{71}$ durch die von $Agent_{73}$ überschrieben werden können,

aber nicht anders herum. Über die Synchronisierung der Uhrzeiten von den Initiatoren bekommen diese mit, dass die Installation der von ihm gestarteten Updates nicht auf allen Switchs gesichert ist, da ein in der Hierarchie über ihm stehender $Agent_{73}$ ein Update mit gleichem Matching-Kriterium gestartet hat. Hier ist es wichtig, dass Agenten bei der Synchronisation auch das Matching-Kriterium des Updates angeben, damit man bei Updates mit gleichem Zeitstempel beurteilen kann, ob diese mit dem eigenen Update im Konflikt stehen. Optional kann man zwischen den Agenten auch noch die Adressen der aktualisierten Switchs austauschen, um genauer zu bestimmen, ob wirklich ein Konflikt besteht. Denn es kann sein, dass trotz identischem Matching-Kriterium und Zeitstempel unterschiedliche Switchs beschrieben werden und deswegen beide Flows komplett installiert werden. Falls ein Konflikt besteht, bemerkt dies der in der Hierarchie tiefer stehende Agent und kann zum Beispiel die Updates noch einmal mit einem größerem Zeitstempel versenden.

Würde man die Logischen-Uhr-Agenten in einer komponenten-basierten Umsetzung aus Abb. 4.2 realisieren, müsste bei den Kontrollanwendungen durch CCM_{App} ein Zeitstempel generiert werden. Denkbar ist, dass auch hier jede Instanz eine Id erhält, wenn sie gestartet wird und zusätzlich auf einer verteilten Datenbank eine logische Uhr zugänglich ist, durch die die Anzahl der Update mitgezählt wird. CCM_{Ctrl} würde Updates herausfiltern, bei denen man anhand des Zeitstempel erkennt, dass sie veraltet sind. Dazu müssten sie in einer Tabelle die Updates mit Zeitstempeln speichern, die schon an den Controller weitergegeben wurden. Wichtig wäre aber bei dieser Variante, dass jeder Switch nur einem Controller zugeordnet ist. Falls ein Switch von zwei Controllern OpenFlow-Nachrichten empfangen würde, könnte dieser nicht nachvollziehen, welche Einträge sich in der Weiterleitungstabelle des Switchs befinden.

4.4 Vergleich von Locking- und Logische-Uhr-Agenten

Nachdem zwei Varianten vorgestellt wurden, mit deren Hilfe inkonsistente Einträge in den Weiterleitungstabellen verhindert werden, wird im folgenden untersucht, welche Vor- und Nachteile sie als Anwendungen in einem verteilten System haben.

Bei Locking-Agenten existiert ein geringerer Grad der Parallelisierung, was bedeutet, dass nicht so viele Routen geschrieben werden können, da immer nur ein Agent zu einem Matching-Kriterium ein Flow-Update verschicken kann und andere Agenten warten müssen, bis die Sperre wieder aufgehoben wurde. Dies hat zur Folge, dass es länger dauern kann, bis ein Route-Request von einer CoCoMi-Instanz ausgeführt wird und die Performanz geringer ist. Beim Logische-Uhr-Agenten existiert ein höherer Grad der Parallelisierung, da mehrere Agenten nebenläufig Routen schreiben können und kein Agent auf einen anderen warten muss. Die Performanz wird durch die Erzeugung der Zeitstempel nur geringfügig eingeschränkt im Vergleich zu einer Middleware, die Inkonsistenzen nicht verhindert. Route-Requests können vom Initiator sofort ausgeführt werden, da man nicht auf andere Instanzen Rücksicht nehmen muss. Durch die feste Hierarchie ist das Verfahren aber

unfair gegenüber Agenten, die von einer Instanz mit niedriger Id erzeugt wurden. Diese werden bei einer hohen Frequenz von Updates immer überschrieben von Updates von anderen Instanzen mit höherwertigen Ids. Um dieses Problem zu lösen, wird vorgeschlagen bei identischen Zeitstempeln anhand eines Hashwerts von Instanz-Id und Matching-Kriterium zu entscheiden, welches Update das andere überschreiben kann. Dadurch unterscheidet sich die Hierarchie bei unterschiedlichen Matching-Kriterien und es gewinnt nicht immer bei allen Matching-Kriterien die gleiche Instanz.

Beim Locking-Agenten wird dafür gesorgt, dass inaktive Einträge in den Weiterleitungstabellen nicht erst durch den Timeout-Mechanismus der Switchs entdeckt werden. Bei der Installation eines Flows werden nicht mehr benötigte Einträge in den Weiterleitungstabellen gelöscht und nur Switch-Updates verschickt, falls sich in den Weiterleitungstabellen durch den neuen Flow wirklich etwas ändert. Bei kleinen Änderungen von langen Flows bedeutet dies, dass manchmal weniger Update-Nachrichten verschickt werden müssen und dadurch schneller ein Flow installiert wird als beim Logische-Uhr-Agent, der an alle Switchs Updates verschickt, unabhängig davon, ob sich in den Weiterleitungstabellen wirklich etwas ändert. Außerdem wird beim Locking-Agent die Atomizität der Flow-Updates, wodurch jeder Route komplett oder gar nicht installiert wird.

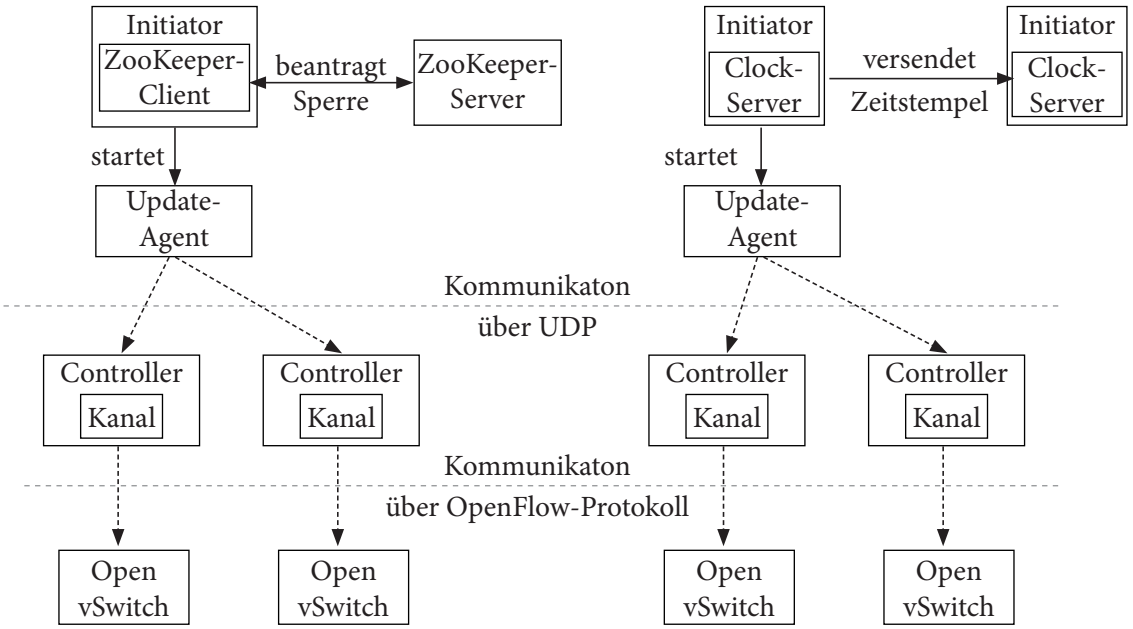
Für die Switchs entsteht in der Logische-Uhr-Variante ein höhere Aufwand, da sie zusätzlich noch die Zeitstempel und die Ids der CoCoMi-Instanzen speichern müssen. Außerdem müssen sie bei jedem ankommenden Update, noch die Zeitstempel vergleichen.

5 Implementierung

Um den unterschiedlichen Durchsatz der beiden Update-Agenten zu testen, wurde an dem SDN-Netzwerk vom IPVS eine verteilte Kontrollogik simuliert, von der aus Flow-Updates versendet wurden. In diesem Kapitel wird beschrieben, wie die unterschiedlichen Komponenten implementiert wurden.

5.1 Übersicht

In der Implementierung wurden die notwendigen Komponenten entwickelt, um die Funktionen der Control Coordination Middleware zu testen. In Abb. 5.1 sieht man die Initiatoren der CoCoMi-Instanzen, die Update-Agenten starten, um eine bestimmte Route im Netzwerk zu schreiben. Als Matching-Kriterium wurde hier bei allen Flows die IP-Ziel-Adresse der Pakete verwendet. Zuerst beantragen die Initiatoren entweder eine Sperre über die betreffenden Swichts und das Matching-Kriterium mit Hilfe des ZooKeeper-Client oder versenden eine Nachricht mit dem aktuellen Zeitstempel an andere Intitiatoren. Die Update-Agenten versenden Switch-Updates über UDP an die Controller, die für die betreffenden Switchs zuständig sind. Die Controller haben eine Kanal zu



(a) Variante mit Locking-Agent (b) Variante mit Logischen-Uhr-Agent

Abb. 5.1: Komponenten der Implementierung

den Open vSwitchs aufgebaut, mit dem über das OpenFlow Protokoll V1.0.0 kommuniziert wird und über den Handshakes ausgetauscht werden. Mit Hilfe je eines Kanals kommunziert der Controller mit den vSwitchs und kann die generierten OpenFlow-Mod-Nachrichten an den vSwitch schicken. Der vSwitch verarbeitet diese nach dem OpenFlow-Protokoll und aktualisiert seine Weiterleitungstabelle.

5.2 Inititiator

Der Initiator ist eine Java-Anwendung, die Locking- und Logische-Uhr-Agenten starten kann, um Routen im Netzwerk zu schreiben. Vor der Erzeugung eines Locking-Agenten verbindet sich der Initiator über einen ZooKeeper-Client mit dem ZooKeeper-Server und beantragt für alle Switchs eine Sperre über das Matching-Kriterium. Um die logische Uhr zu synchronisieren, die für die Zeitstempel des Logischen-Uhr-Agenten notwendig ist, starten die Initiatoren einen ClockServer, über den Nachrichten über versendete Updates von anderen Initiatoren empfangen werden können. Um einen Logische-Uhr-Agenten zu starten, versendet ein Initiator eine Nachricht mit Zeitstempel, Matching-Kriterium und Initiator-Id an die ClockServer anderer Initiatoren, bei denen darauf die interne Uhr aktualisiert wird.

5.3 Locking-Service

Apache ZooKeeper[15] ist ein verteilter Koordinationsservice, der es Anwendungen erlaubt in einem verteilten System Informationen auszutauschen und sich zu synchronisieren. Der ZooKeeper-Service kann auf unterschiedlichen Rechnern installiert werden und speichert Informationen in einem hierarchischen Namensraum. Dabei können Anwendungen über einen Client auf die vorhandenen Informationen zugreifen und neue hinzufügen. Falls in einem verteilten System mehrere ZooKeeper-Server installiert sind, synchronisieren diese sich automatisch und sorgen so für die Verteilungstransparenz der Anwendung.

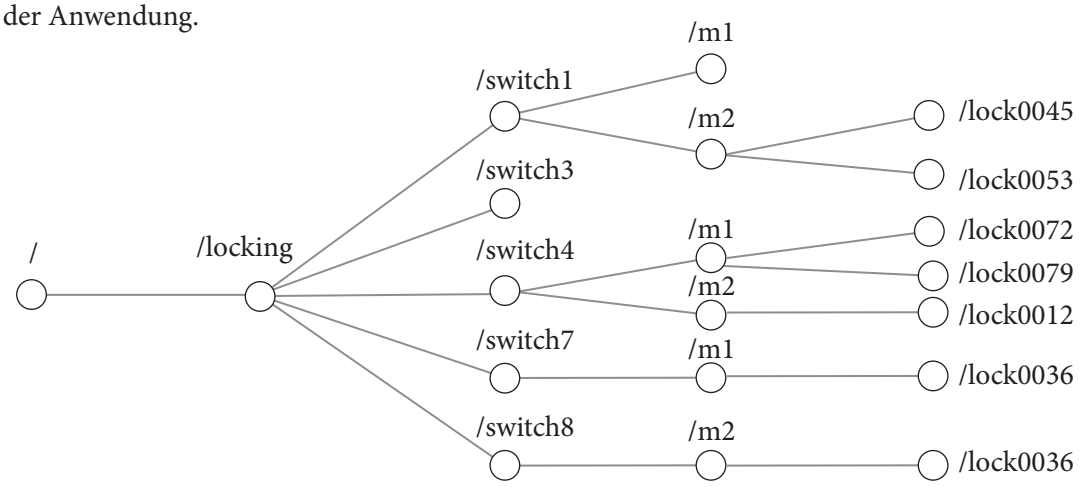


Abb. 5.2: Beispiel für einen Namensraum auf dem ZooKeeper-Server

In Abb. 5.2 ist ein Beispiel für die Ordnerstruktur dargestellte, die für den Locking-Service auf dem ZooKeeper-Server gespeichert sein kann. ZooKeeper verwendet für das Speichern von Informationen eine Baumstruktur, bei dem jedem Eintrag weitere Daten als Kinder angehängt werden können. Im Basisordner wurde hier eine */locking*-Ordner angelegt, in dem für jeden Switch des Netzwerks ein Ordner erzeugt wurde. ZooKeeper-Clients können mit HIlfe vom *create()*-Befehl Sperren für Switchs beantragen, indem sie die Adresse des zugehörigen Ordners und das Matching angeben, wie z.B. */locking/switch1/m1/* für Sperren von Switch1 und dem Machting-Kriterium m_1, und einen */lock*-Ordner erzeugen. Dabei muss das *sequence-* und *ephemeral*-Flag gesetzt sein, wodurch erreicht wird, dass dem erzeugten Ordner durch ZooKeeper eine eindeutige Sequenznumer zugewiesen wird und der Ordner nach dem Schließen des Clients gelöscht wird. Danach muss der Client die vorhandenen Einträge abrufen und überprüfen, ob er den */lock*-Ordner mit der kleinsten Sequenznummer erzeugt hat. In diesem Fall würde der Client eine Sperre über den Switch bezüglich des Matching-Kriteriums besitzen. Falls ein anderer Ordner mit kleinerer Sequenznummer existiert, bedeutet dies, dass keine Sperre möglich ist, da ein anderer Client eine Sperre besitzt. Wenn alle Sperren über die Switchs einer Route bestätigt worden sind, kann der Initiator den Update-Agenten starten. Nach dem Beenden des Update-Agenten wird der ZooKeeper-Client geschlossen und der Ordner automatisch gelöscht.

5.4 Update-Agenten

Den Update-Agenten sind als Java-Anwendungen implementiert, denen beim Aufrufen die Route und Matching-Kriterium übergeben wird, für dessen Schreibvorgang sie verantwortlich sind. Da in der Evaluation bestimmte Routen geschrieben werden sollen, wird hier darauf verzichtet, dass die Agenten die Route berechnen und dadurch auch keine verteilte Datenbank benötigt wird, in der die aktuelle Topologie des Netzwerk gespeichert wird. Die Switch-Updates werden in einer HashMap gespeichert und nacheinander abgearbeitet. Beim Locking-Agenten werden zuerst für alle Switchs Sperren über das Matching-Kriterium beantragt und erst bei der erfolgreichen Sperre über alle Switchs, die Updates verschickt. Der Logische-Uhr-Agent fragt beim Clocktime-Server die aktuelle logische Uhrzeit ab, um den Zeitstempel bei den Switch-Updates zu setzen. Den Aufbau eines Switch-Updates mit Ziel-IP, Ausgangsport, Instanz-Id und Zeitstempel sieht man in Abb. 5.3. Die Update-Agenten der Middleware senden über UDP die Switch-Updates an die Controller.

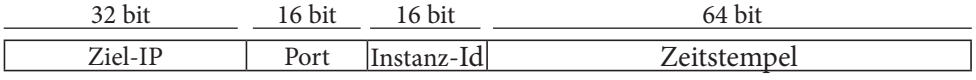


Abb. 5.3: Bit-Verteilung bei Switch-Update für einen Controller

5.5 Controller

Die Controller sind in Java implementierte und vereinfachte Versionen eines OpenFlow-Controllers, die alle für die Evaluation erforderlichen Funktionen erfüllen. Der Controller empfängt die Switch-Updates von den Update-Agenten und wandelt sie in Flow-Mod-Nachrichten des OpenFlow Protokolls um, die über den Kanal an den Open vSwitch weitergeleitet werden. Um die Kommunikation zwischen vSwitchs und Controllern zu überwachen, wurde WireShark[18] verwendet, mit dessen Hilfe man überprüfen konnte, ob die Bits in den OpenFlow-Nachrichten richtig gesetzt waren und Fehlernachrichten von den vSwitchs interpretiert werden konnten.

Der Controller wurde so realisiert, dass er von beiden Varianten der Update-Agenten Nachrichten empfangen kann und diese dann in zwei unterschiedlichen Rollen verarbeitet. Entscheidend dafür ist, ob in den Switch-Updates die Bits für einen Zeitstempel gesetzt sind. Falls kein Zeitstempel gesetzt ist, werden die Updates in der Locking-Variante verarbeitet und direkt an den Switch weitergeleitet, da vorher schon eine Sperre über die Update-Agenten beantragt wurde.

Falls ein Zeitstempel angegeben wird, weiß der Controller, dass die Updates von einem Logischen-Uhr-Agenten stammen und der Zeitstempel erst mit bestehenden Einträgen verglichen werden muss. In dieser Implementierung sorgen nicht die Switchs, sondern die Controller für eine Totalordnung der Updates, was aber für die Ergebnisse der Evaluation keinen Unterschied macht. Der Controller speichert eine Kopie der Weiterleitungstabelle des Switchs und aktualisiert diese, falls er ein Update weiterleitet. Falls nun ein neues Update mit Zeitstempel beim Controller ankommt, wird zuerst überprüft, ob für das Matching-Kriterium schon ein Eintrag vorhanden ist. Falls dieses schon existiert, wird der Zeitstempel und bei Bedarf auch die Agenten-Id des Updates mit den Einträgen in der Tabelle verglichen. Wenn der Zeitstempel größer oder noch kein Eintrag vorhanden ist, wird eine OpenFlow-Mod-Nachricht erstellt und an den Switch verschickt, so dass auch in der Weiterleitungstabelle des Open vSwitchs ein neuer Eintrag erstellt wird.

6 Evaluation

In diesem Kapitel wird untersucht, wie viele Routen von den Agenten geschrieben werden können und dabei die beiden Unterschiede zwischen Locking-Agenten und Logische-Uhr-Agenten ermittelt.

6.1 Versuchsaufbau

In dem IPVS-Netzwerk befinden sich neun Open vSwitchs [16], die acht Hosts verbinden. Die neun vSwitchs laufen auf jeweils einem Intel Xeon E3 4x3.40GHz Rechner mit 16GB RAM. Jeweils vier Hosts befinden sich als virtuelle Maschinen auf zwei Intel Core2 Duo CPU 2.40GHz - Rechnern. Verbunden sind Hosts und Switchs über 4-Port Ethernet-Karten und bilden das Netzwerk, in dem Flows installiert werden können und Pakete transportiert werden. Die Kontrolllogik befindet sich auf dem Management-Rechner, der auf einem Intel Core2 Duo CPU 2.40GHz-Prozessor läuft. Der Management-Rechner besitzt zu jeder Maschine im Netzwerk eine physische Verbindung, so dass die Kontrollnachrichten über ein gesondertes Netzwerk transportiert werden. Mit dem NX-Client[17] kann man sich mit dem Management-Rechner verbinden, und in einer Fedora 17-Umgebung Anwendungen und Controller in der Kontrolllogik starten. Über SSH kann man auf die Hosts zugreifen und überprüfen, ob Routen vollständig installiert wurden.

Um die Funktion der Control Coordination Middleware zu testen, wird sich in dieser Arbeit auf ein Netzwerk mit fünf Switchs und vier Host beschränkt, das in Abb. 6.1 dargestellt ist. Jeder Switch ist mit genau einem Controller verbunden, der auf dem Management-Rechner installiert wird. CoCoMi wird durch zwei Initiatoren repräsentiert, durch die Update-Agenten auf dem Management-Rechner gestartet werden.

Jeder Open vSwitch hat in den Einstellungen einen einzustellenden OpenFlow-Port, über den versucht wird, im Netzwerk eine Verbindung mit einem Controller aufzubauen. Um zu verhindern, dass sich ein Controller mit allen vSwitchs verbindet, muss jedem Paar aus Controller und vSwitch ein eigener OpenFlow-Port zugewiesen werden. Bei den vSwitchs muss dafür der Standard-OpenFlow-Port 6633 verändert werden. Dadurch können sich Controller und vSwitch über einen OpenFlow-Handshake verbinden und der Controller baut einen Kanal zum Switch auf, über den man Nachrichten übertragen kann.

6.2 Durchsatz

Der Durchsatz bezeichnet die Anzahl der Route-Requests, die von CoCoMi erfolgreich durchgeführt werden können. Durch das Locking-Verfahren wird dieser reduziert, da immer nur ein Agent eine Sperre für ein Matching-Kriterium halten kann. Es wird nun untersucht, wie dieser sich ändert abhängig von der Anzahl der Agenten, die eine Route schreiben wollen.

Für die Evaluation werden im dem Netzwerk vier Routen eingeführt, die alle als Matching-Kriterium die IP-Adresse des Ziel-Hosts H_D besitzen und in Abb. 5.2 zu sehen sind. Von den Initiatoren wird jeweils ein Agent gestartet, der innerhalb einer Minute versucht, möglichst oft die Route im Netzwerk zu installieren. In Tab. 5.1 sieht man die Ergebnisse der Messung für die unterschiedlichen Kombinationen von zu installierenden Routen. Zuerst werden die Routen einzeln installiert und man sieht, dass die Agenten bei Routen, bei denen Sperren über fünf Switchs beantragt werden müssen, weniger Aktualisierungen durchführen können als bei Routen, die nur über drei Switchs gehen. Bei zwei nebenläufig zu installierenden Routen sieht man, dass die Anzahl der durchgeführten Updates stark sinkt, da nun auch ein anderer Agent eine Sperre über die Route beantragt hat. Umso mehr Sperren

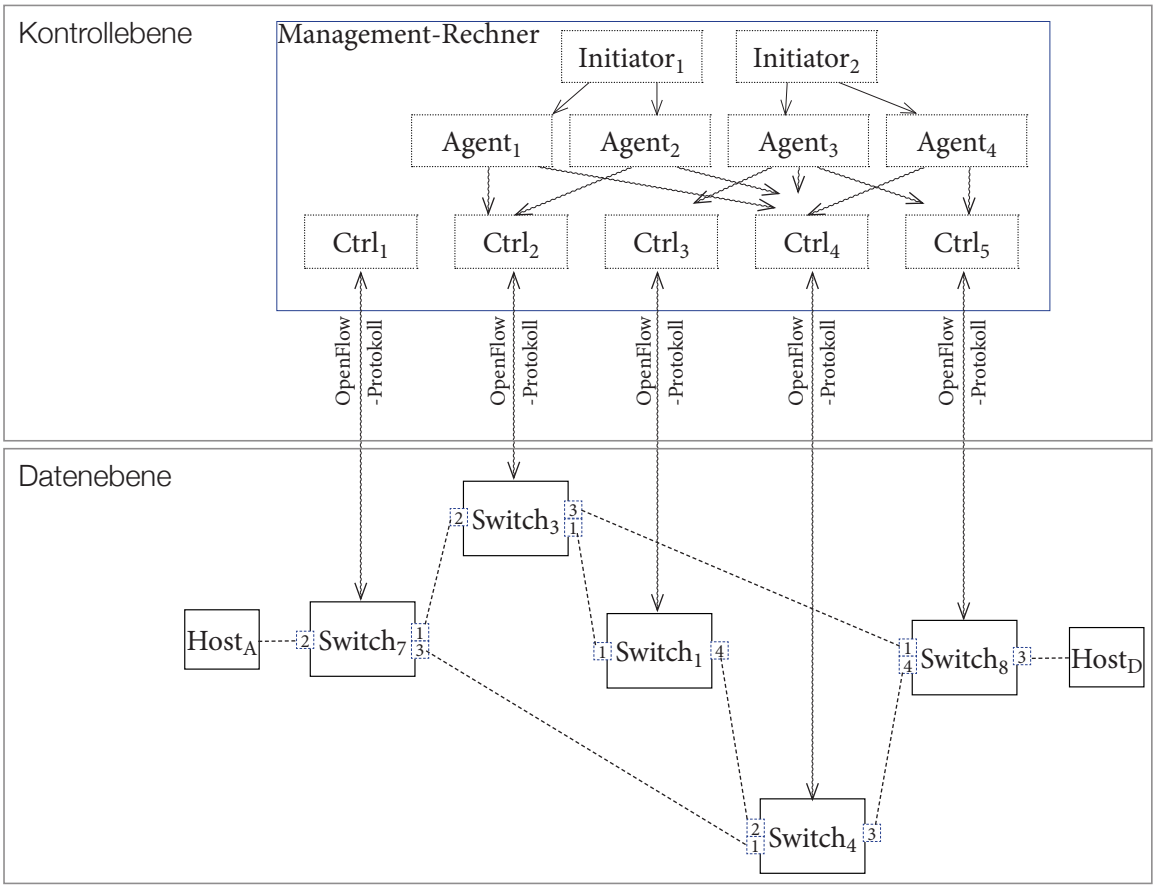


Abb. 5.1 Topologie des verwendeten Netzwerks mit Kontrollogik

beantragt werden müssen, desto stärker wird die Anzahl der durchgeführten Updates reduziert. Falls drei oder vier Agenten eine Sperre beantragen, sinkt die Anzahl der Updates weiter, da es noch schwieriger wird, eine Sperre über alle Switchs zu erhalten.

In den letzten beiden Spalten der Tab. 5.1 sieht man die Gesamtzahl der durchgeführten Aktualisierungen im Netzwerk und der entsprechende Durchschnittswert jedes Agenten. Hier erkennt man, dass der Wert jedes Agenten stark zurückgeht, sobald mehrere Agenten gleichzeitig eine Route mit konkurrierenden Updates schreiben wollen. Die Anzahl aller im Netzwerk durchgeführten Updates halbiert sich ungefähr pro hinzukommenden Agent. Dies liegt daran, dass die Agenten einzeln für

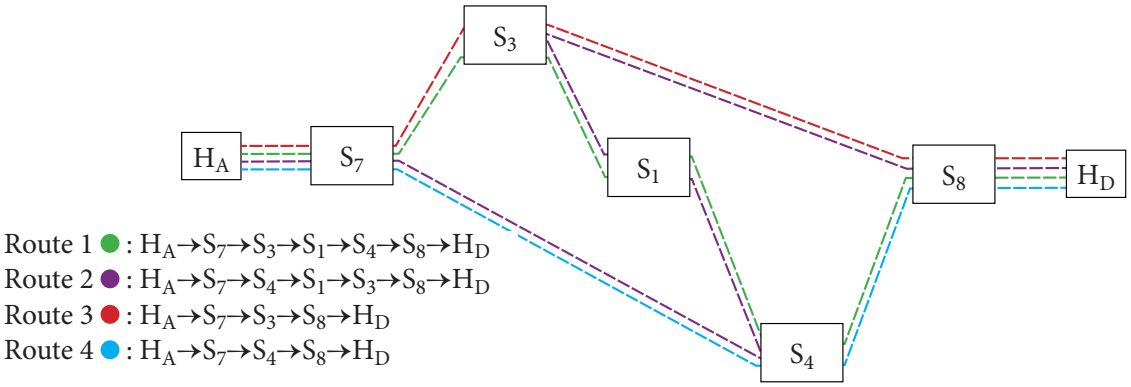


Abb. 5.2 Übersich der zu installierenden Routen

|Routen|	Route 1 ●	Route 2 ●	Route 3 ●	Route 4 ●	Total	ø / Agent
1	1719				1719	1719
		1713			1713	1713
			2150		2150	2150
				2118	2118	2118
2	434	480			914	457
	319		826		1145	573
	270			789	1059	530
		335	780		1115	558
		275		791	1066	533
			718	607	1325	663
3	138	127	185		450	150
	113	125		175	413	138
	124		183	224	531	177
		135	202	209	546	182
4	37	41	55	62	195	49

Tab. 5.1 Anzahl der erfolgreichen Updates pro Minute beim Locking-Angenten

|Routen|	Route 1 ●	Route 2 ●	Route 3 ●	Route 4 ●	Total	ø / Agent
4	17288	17708	29093	30122	94211	23553

Tab. 5.2 Anzahl der erfolgreichen Updates pro Minute beim Logische-Uhr-Angenten

alle Switchs Sperren beantragen müssen und auf andere Agenten warten müssen, bis diese die Sperre freigeben.

In der Tab. 5.2 sieht man die entsprechenden Ergebnisse für den Logische-Uhr-Agenten. Hier wird nur überprüft, wie viele Updates eine Agent verschicken kann, ohne dabei Wechselwirkungen zu untersuchen, da keine Synchronisierung zwischen den Agenten existiert. Hier sieht man, dass schon alleine durch den wegfallenden Komminikation mit ZooKeeper der Durchsatz stark erhöht wird, da ein Locking-Agent, der als einziger im Netzwerk eine Route schreiben will, nur ein Zehntel so viele Updates verschickt wie ein Logische-Uhr-Agent. Der Durchsatz eines Logische-Uhr-Agenten entspricht auch ungefähr dem Durchsatz eines Systems ohne Mechanismen zur Verhinderung von Inkonsistenzen in den Weiterleitungstabellen, da hier nur die Erzeugung eines zusätzlichen Zeitstempels hinzukommt

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

In dieser Arbeit wurde untersucht, welche Probleme in Software-defined Networks auftreten, in denen die Kontrolllogik verteilt realisiert ist und mehrere Controller nebenläufig die Weiterleitungstabellen der Switchs aktualisieren können. Zuerst wurden in den Grundlagen die Vorteile von SDN-Netzwerken dokumentiert und das OpenFlow Protokoll vorgestellt. Da die Aktualisierung der Weiterleitungstabellen durch mehrere Controller dem Prinzip einer verteilten Datenbank ähnelt, die von mehreren Anwendungen beschrieben wird, wurde betrachtet, welche Konzepte und Mechanismen existieren, um bei der Transaktionsverarbeitung Konsistenz zu garantieren. Danach wurden bestehende Ansätze analysiert und gezeigt, dass bei ihnen noch keine konkreten Lösungsvorschläge für eine Synchronisation von Flow-Updates über mehrere Controller vorgeschlagen wurden.

In dem Systemmodell wurden die Komponenten vorgestellt, über die man Routen über eine verteilte Kontrolllogik schreiben kann. Die Controll Coordination Middleware empfängt die Route-Requests von den Kontrollanwendungen und leitet sie als Switch-Updates an die Controller weiter. Indem der Konsistenz-Begriff dieser Arbeit formular definiert wurde, konnte man feststellen, dass diese durch konkurrierende Updates verursacht werden. Falls zwei unterschiedliche Flows in unterschiedlicher Reihenfolge auf Switchs installiert werden, werden Flows nur teilweise installiert und Schleifen in der Paketweiterleitung können entstehen. Anschließend wurde das Ziel dieser Arbeit formuliert, nämlich Mechanismen zu entwickeln, um für konsistente Einträge und eine Serialisierbarkeit der Updates zu garantieren.

In dem Entwurf wird die Controll Coordination Middleware mit ihren Komponenten vorgestellt und der Ablauf aufgezeigt, in dem die Weiterleitungstabellen der Switchs aktualisiert werden. In CoCoMi werden Route-Requests durch die Initiator-Komponente empfangen, die Update-Agenten startet, um die Switch-Updates zu berechnen und an die Controller weiterzuleiten. Es werden zwei Varianten von Update-Agenten vorgestellt, die bestimmen, wie CoCoMi eine konsistente Aktualisierung garantiert. Locking-Agenten verhindern mit einem verteilten Locking-Verfahren, dass andere Agenten nebenläufig konkurrierende Updates starten können. Bei Logische-Uhr-Agenten wird mit Hilfe von Zeitstempeln ermöglicht, dass die Switchs überprüfen können, ob das erhaltene Update nicht logisch einem schon installierten Updates folgt und deswegen verworfen werden muss, um eine Totalordnung unter den Updates zu garantieren.

Danach wurde in der Implementierung gezeigt, wie das bestehende Modell mit Java-Komponenten und einem Netzwerk mit Open vSwitchs realisiert wurde, um zu überprüfen, wie viele Aktualisierungen pro Minute möglich sind. In der Evaluation wurde gezeigt, inwieweit der Locking-Mechanismus die Anzahl der möglichen Updates beschränkt und dass dieses sinkt, umso mehr Agenten eine Sperre über Matching-Kriterium für sich überschneidende Routen beantragen.

7.2 Ausblick

Die nächsten Schritte bei der Entwicklung von CoCoMi sind das Locking-Verfahren bei den Locking-Agenten zu optimieren und zu untersuchen, wie man den Durchsatz an möglichen Updates erhöhen kann. Dabei kann man sich wieder an den Locking-Verfahren der Transaktionsverarbeitung orientieren wie z.B. ein hierarchisches Locking, bei dem Sperren über Bereiche des Netzwerks und nicht über einzelne Switchs beantragt werden.

Bei den Logische-Uhr-Agenten könnte man die Fairness zwischen den Instanzen erhöhen, indem man die Hierarchie unter den CoCoMi-Instanzen vom Matching-Kriterium abhängig macht. Falls man die Hierarchie über einen HashWert von Controller-Id und Matching-Kriterium ermittelt, variiert die Hierarchie zwischen den Instanzen abhängig vom Matching-Kriterium und nicht immer setzt sich die gleiche Instanz durch, falls Updates mit identischen Zeitstempeln verschickt werden.

Als weitere Optimierung kann man untersuchen, wie man auch transiente Schleifen und Inkonsistenzen verhindert werden können, die während der Aktualisierung entstehen. Hier existieren schon Ansätze [7,1,3] für Netzwerke mit einem Controller und es kann überprüft werden, ob eine Pro-Paket-Konsistenz und eine Pro-Flow-Konsistenz auch in eine Netzwerk mit einer verteilten Kontrolllogik übernommen werden kann.

Literaturverzeichnis

[1] S. Ghorbani and M. Caesar. *Walk the line: Consistent network updates with bandwidth guarantees*. In Proceedings of the First Workshop on Hot Topics in Software-defined Networks, pages 67–72, Helsinki, Finland, Aug. 2012.

[2] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. *Onix: A distributed control platform for large-scale production networks*. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010), pages 351–364, Vancouver, Canada, Oct. 2010.

[3] R. McGeer. *A safe, efficient update protocol for OpenFlow networks*. In Proceedings of the First Workshop on Hot Topics in Software-defined Networks, pages 61–66, Helsinki, Finland, Aug. 2012.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. *OpenFlow: Enabling innovation in campus networks*. ACM SIGCOMM Computer Communication Review, 38(2):69–74, Apr. 2008.

[5] Open Networking Foundation. *OpenFlow switch specification, June 2012*.

[6] Open Networking Foundation. *Software-defined Networking: The New Norm for Networks*, Apr. 2012.

[7] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. *Abstractions for network update*. In Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technolo- gies, Architectures, and Protocols for Computer Communication, pages 323–334, Helsinki, Finland, Aug. 2012.

[8] A. Tootoonchian and Y. Ganjali. *HyperFlow: A distributed control plane for OpenFlow networks*. In Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking (INM/WREN 2010), San Jose, CA, Apr. 2010.

[9] S. H. Yeganeh and Y. Ganjali. *Kandoo: A framework for efficient and scalable offloading of control applications*. In Proceedings of the First Workshop on Hot Topics in Software-defined Networks, pages 19–24, Helsinki, Finland, Aug. 2012.

[10] J. Stribling , Y. Sovran,I. Zhang, X. Pretzer, J. Li, M.F. Kaashoek and R. Morris. *Flexible, wide-area storage for distributed systems with wheelfs*. In Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09) (Boston, MA, April 2009).

[11] T. Haerder, A. Reuter. *Principles of Transaction-Oriented Database Recovery*, 1984, Transactions on Database Systems (TODS) , Volume 9 Issue 4

[12] P. A. Bernstein, E. Newcomer; *Principles of Transaction Processing*, 2009, Morgan Kaufmann

[13] https://www.opennetworking.org/

[14] Eric A. Brewer. Towards robust distributed systems. (Invited Talk) Principles of Distributed Computing, Portland, Oregon, July 2000.

[15] http://zookeeper.apache.org/

[16]http://openvswitch.org/

[17] http://www.nomachine.com

[18] http://www.wireshark.org/

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein

---

Stuttgart, 13.05.2013