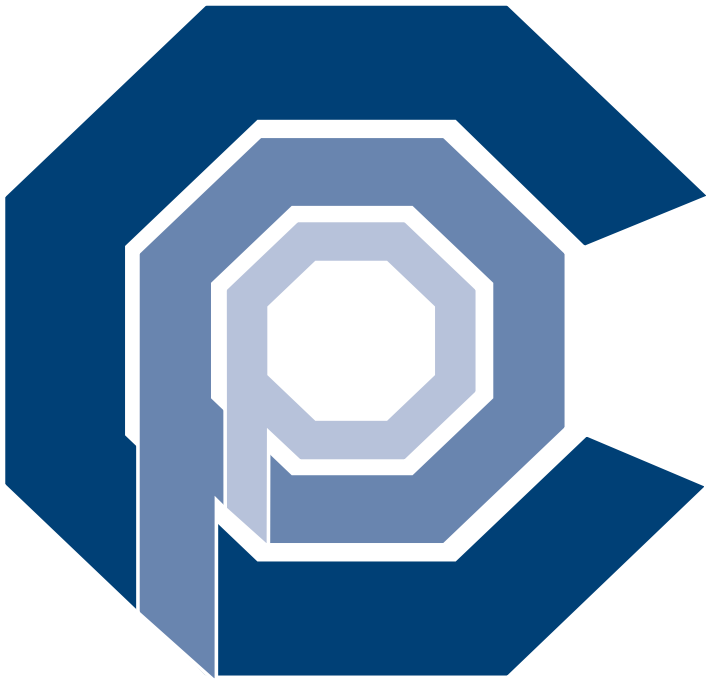
A Reliable, Extensible, and Distributed
Platform for Internet Applications

CPP – Calipso Passix Phenix



Autoren: Markus Egli, Daniel Hottinger, Till Kleisli, Fabio Lanfranchi, Roman Metz, Franziska Meyer, Lukas Oertle, Andreas Pfenninger

Betreuer: Keno Albrecht, Aaron Zollinger, Roger Wattenhofer

Kontext: Labor-Arbeit Sommersemester 2004, Distributed Computing Group ETH Zürich

Bilder: Keynote, OmniGraffle, Sketch

Textsatz: pdfLaTeX

Inhaltsverzeichnis

Kapitel 1

CPP: Calipso Passix Phenix

1.1 Aufgabenstellung

Die Aufgabe war, ein erweiterbares, ausfallsicheres Server-System zu schreiben, welches Load Balancing und Persistenz hat. Damit sollte die Erreichbarkeit und die Antwortzeit von Internet-Services und Webseiten im Speziellen verbessert werden. Konkreter sollte das System folgende Punkte enthalten:

Gateway Eine Gruppe von Gateways soll für Load-Balancing sorgen. Dieses Subsystem heisst Passix (Seite 23).

Persistenz Bei einem Ausfall soll ein einzelner Server-Rechner nicht alle Daten von seinen Peers holen müssen, sondern auf seine eigenen, auf der Festplatte gespeicherten Daten zurückgreifen können. Dies verhindert zugleich Datenverlust bei einem Totalausfall. Die Persistenz wird von Subsystem Phenix (Seite 35) sichergestellt.

Anwendungen Das System soll nicht nur einen Dienst (Web-Server oder Mail-Server oder ...) zu Verfügung stellen, sondern mehrere. Details dazu gibt es bei den Applikationen auf Seite 55.

Erweiterbarkeit Neue Anwendungen sollen möglichst einfach programmiert und hinzugefügt werden können. Diese Funktionalität übernimmt Calipso (Seite 11), welches ausserdem beim Starten Phenix und die Applikationen initialisiert und im laufenden Betrieb die Anfragen von Clients entgegennimmt und an die Applikationen verteilt.

Benchmarks, Marktanalyse Ausserdem waren Benchmarks und eine Marktanalyse gefragt, welche jeweils von den Zuständigen der einzelnen Subsysteme durchgeführt wurden.

1.2 Architektur

CPP steht für Calipso, Passix und Phenix.

Unsere Interpretation der Aufgabenstellung sieht folgendermassen aus:

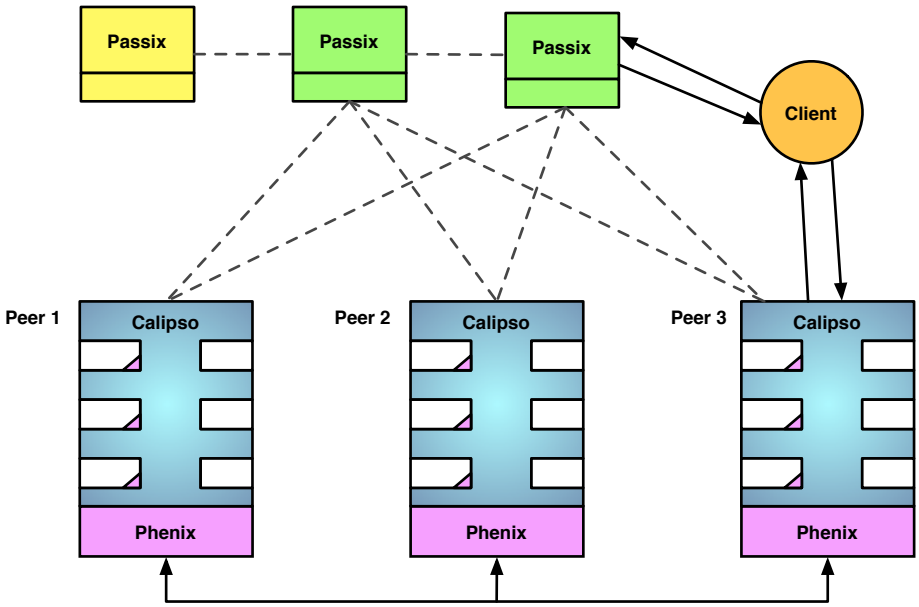


Bild 1: Architektur des CPP-Systems

Abbildung 1 zeigt ein Szenario mit einem Client und sechs Rechnern auf Server-Seite, von denen drei als Gateway konfiguriert sind. Die Calipso-Rechner (Peer 1-3) melden sich regelmässig bei den aktiven Gateways (die beiden grünen Passix-Rechner), sodass diese eine Liste der erreichbaren Server aufbauen können. Diese Liste wird auch an das inaktive (gelbe) Gateway verteilt. Untereinander gleichen die Calipso-Server ihre Daten mittels Phenix ab. Ein Client baut nun zuerst eine Verbindung zu einem der aktiven, ihm bekannten Gateways auf, wo ihm die Adresse eines laufenden Calipso-Rechners mitgeteilt wird. Zu diesem nimmt der Client Verbindung auf und stellt seine Anfrage(n).

Kapitel 2

Calipso

2.1 Etymologie von Calipso

Calipso steht für „Concurrent Application Loading Infrastructure with Persistent Server Operations“. Davon abgesehen hat das homophone Kalypso noch eine weitere Bedeutung:

Kalypso ist in der griechischen Mythologie eine Nymphe, welche in Homers Odyssee vorkommt. Sie ist die Tochter von Atlas, dem Titanen, der den Himmel auf seinen Schultern trägt. Sie liebt Odysseus und hält den Schiffbrüchigen sieben Jahre lang bei sich auf der bewaldeten Insel Ogygia fest. Erst auf Geheiss der olympischen Götter gibt sie ihn frei. Obwohl Kalypso ihm Unsterblichkeit verspricht, wenn er bei ihr bleibt, verlässt er sie, um zu seiner Frau Penelope zurückzukehren. (Beschreibung teilweise aus Wikipedia[1].)

2.2 Architektur

Die geforderte Erweiterbarkeit erreichten wir auf zwei Arten: Einerseits können zusätzliche Applikationen hinzugefügt werden (Applikations-Plugins), andererseits kann Calipso durch Protokoll-Plugins neue Protokolle lernen. Fix zu Calipso gehören je eines dieser Plugins, nämlich die Admin-Applikation und der HTTP-Handler. Somit bietet Calipso die Möglichkeit, über einen normalen Browser neue Plugins zu installieren.

Die Architektur ist trotz ihrer Einfachheit erstaunlich flexibel:

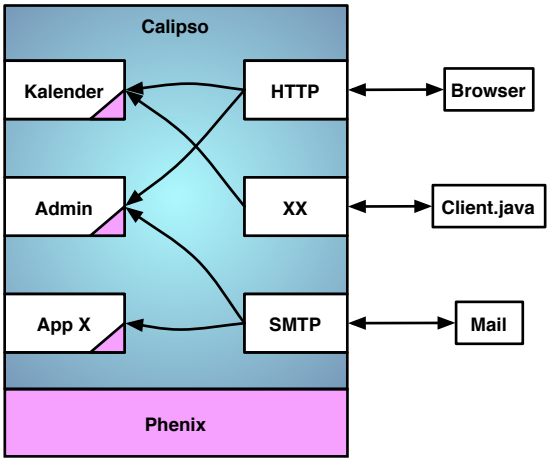


Bild 2: Calipsos Architektur

Das Kernstück bildet der Multiplexer (der grosse Block), der von den Protokoll-Plugins (rechts eingeschoben) die Requests bekommt und diese an das entsprechende Applikations-Plugin (links eingeschoben) weiterleitet. Wie die Pfeile andeuten, ist es dabei kein Problem, über verschiedene Protokolle auf die gleiche Anwendung oder umgekehrt über das gleiche Protokoll auf verschiedene Applikationen zuzugreifen.

[1] http://www.wikipedia.de/

Sowohl Calipso als auch die Applikationen haben Zugang zu Phenix, was durch die kleinen Dreiecke in jedem Applikations-Plugin symbolisiert wird. Calipso nutzt diese Möglichkeit, um Applikationen zu speichern und zu laden.

2.3 Implementation

Das Grundgerüst war recht schnell erstellt. Es besteht aus der Klasse Calipso, welche eine statische `main()`-Methode enthält. In dieser werden die benötigten Klassen instanziert und Subsysteme initialisiert. Dies sind namentlich:

2.3.1 Klassen

Multiplexer

Hat die Aufgabe, die vom Protokoll-`Handler` eintreffenden Anfragen auf die Anwendungen zu verteilen, deren Namen jeweils aus dem Request-Objekt hervorgehen (`getApplication()`). Wenn keine Anwendung gefunden werden kann, so wird versucht, die Admin-Anwendung zu laden, welche dann eine für den Anwender hilfreiche Antwort generiert. Ist diese nicht installiert, so wird eine Exception geworfen.

Später kam noch eine weitere Aufgabe für den Multiplexer hinzu: Da es unter Umständen sehr bequem ist, wenn Dateien wie Bilder oder statischer Text direkt aus der Jar-Datei geladen werden, übernimmt dies der Multiplexer, ohne die Anwendung dafür aufzurufen.

Für die Verwaltung der geladenen Anwendungen ist der `ApplicationManager` zuständig.

ApplicationManager

Führt Buch über die geladenen (instanzierten) Anwendungen. Die wichtigste Methode ist `get(String name)`, welche eine Instanz der gewünschten Anwendung liefert. Diese wird nötigenfalls aus Phenix geladen und initialisiert. Wird eine neue Version eingespielt, so ruft der `ApplicationObserver` `unload()` auf, was zur Folge hat, dass die neue Version beim nächsten eingehenden Request geladen wird.

ApplicationObserver

Reagiert auf externe Ereignisse, also das Installieren oder Löschen von Anwendungen. Bei neu installierten Anwendungen wird versucht, einen eigenen Protokoll-`Handler` zu installieren, welcher beim Löschen der Anwendung wieder deinstalliert wird. Dies wird auch für bereits installierte Anwendungen bei einem Neustart gemacht.

2.3.2 Subsysteme

Log4j

Wurde von Daniel Hottinger als flexibler Logging-Mechanismus in Calipso eingeführt. Die Vorteile sind vor allem die Angabe der genauen Klasse und Zeilennummer in jeder Log-Meldung, sowie flexible Konfiguration, bei der man in einer externen Datei klassenweise die Stufe der geloggten Meldungen einstellen kann.

Da Log4j nicht standardmässig in Java integriert ist, mussten wir auf die externe Jar-Datei zurückgreifen. Leider fiel die Konfiguration dem Anwendungsteam nicht ganz leicht, sodass Log4j bei ihnen lange Zeit unkonfiguriert blieb und keine Meldungen ausgab; was bei der Fehlersuche nicht sehr hilfreich war.

Phenix

Wurde schrittweise integriert, zuerst gar nicht, dann ein einfaches Dummy-System, welches später auf eines mit Objektverteilung mittels Clippee umgestellt wurde. Zuletzt kam noch die persistente Datenhaltung in Form der Mckoi-Datenbank hinzu. Die Umstellungen verliefen weitgehend problemlos.

Passix

Passix kam am Schluss dazu. Die Integration klappte problemlos, da es nur wenige, kleine Schnittstellen gab, die durch wenige Zeilen im Code hinzugefügt werden konnten.

2.3.3 Plugins

Admin-Anwendung

Ist für die Administration von Anwendungen verantwortlich. Dabei unterscheidet sie sich keineswegs von anderen Anwendungen wie dem Kalender oder dem Webserver. Bei der Entwicklung kam uns zugute, dass wir von Anfang an auf die einfache Entwicklung von Anwendungen Wert gelegt hatten. Im Wesentlichen war für jeden „Bildschirm“ (Zustand, State) eine Methode zu implementieren, die das HTML dafür generiert. Diese kann beliebig viele Parameter für Rückmeldungen an den Anwender haben. Eine weitere Methode für die Verarbeitung (Übergang) der eingegebenen Daten ist ebenfalls notwendig. Diese ruft als letztes wieder eine Methode auf, die einen „Bildschirm“ aufbaut und gibt deren Rückgabewert zurück. Alles in allem eine sehr elegante Methode, um Web-Anwendungen zu schreiben.

HTTPHandler

Für alle TCP-basierten Protokolle übernimmt der `ProtocolAdapterSingleConnection` das Entgegennehmen der Verbindungen. So auch für den `HTTPHandler`, von welchem für jede neue Verbindung eine Instanz in einem Thread gestartet wird.

Der Handler wurde bereits am Anfang geschrieben und ist relativ gross, da er viel Code für die Fehlerbehandlung enthält. Er beherrscht sowohl POST- als auch GET-Requests, sowie HTTP-Keep-Alive, was das Absetzen mehrerer Requests über eine Verbindung ermöglicht.

Virtuelle Pfade wie `/application/path/to/data` werden so umgebaut, dass die für die Anwendung nicht von Requests wie `/application?path=/path/to/data` zu unterscheiden sind.

Exceptions bei der Bearbeitung in der Anwendung oder auch von anderswo können bis hierher propagiert werden, wo sie dem Endanwender als „500 Internal server error“ mit Stacktrace im Browser präsentiert werden.

Für die Bearbeitung von POST-Requests greifen wir auf eine externe Klasse zur Decodierung des MIME-Formats zurück, da Java nicht standardmässig so einen Mechanismus bietet.

Weitere Handler

Franziska implementierte für den Kalender noch den `JavaClientHandler`, welcher serialisierte `HashMaps` hin- und herschicken kann. Ein `MailHandler` oder ein IRC-Bot-Handler waren zwar mal diskutiert worden, wurden aber aus zeitlichen Gründen nicht mehr implementiert.

2.4 Konfiguration

Um plattformunabhängig zu beiben, beschlossen wir, auf die Java-eigenen Preferences zu setzen. Diese sind unter Unix in Form einer einfachen XML-Datei abgelegt, deren Anpassung dem Endanwender gerade noch zugemutet werden kann. Unter Windows werden die Daten jedoch in die Registry geschrieben, was definitiv nicht endanwendertauglich ist. Daher kombinierten wir diese mit den Properties, welche auf der Kommandozeile mittels der Option `-D` übergeben werden können. Es ist allerdings zu beachten, dass die Properties der JVM übergeben werden müssen, also auf der Kommandozeile vor der auszuführenden Klasse/Datei stehen sollten. Wo möglich, werden sinnvolle Default-Werte angenommen. Ist dies nicht möglich, werden unbekannte Werte beim Starten auf der Kommandozeile erfragt.

2.5 Anleitung

2.5.1 JAR-Files für die Serverdienste erstellen

- ant installieren
- `log4j.jar`[2], `mckoidb.jar`[3] herunterladen
- ins Verzeichnis `.../trunk/` wechseln

[2] http://logging.apache.org/log4j/docs/download.html
[3] http://mckoi.com/database/

- in `build.xml` am Anfang bei Bedarf die Pfade anpassen
- `ant admin kalender calipso passix` erstellt die nötigen JAR-Files
  kurze Version: `ant all`

2.5.2 Kalender-Client erstellen und starten

- `jbcl.jar`, `alloy.jar` herunterladen [4]
- ins Verzeichnis `.../trunk/` wechseln
- `ant kalenderClient` erstellt das JAR-File
- Client mit `java -jar KalenderClient.jar <serverIPAddress> <kalenderName> <true: mit Passix, false: ohne Passix>` starten.
  Der letzte Wert gibt an, ob die IP-Adresse auf einen Passix-Server zeigt.

2.5.3 Starten von Calipso aus JAR-Files

Windows, Linux

- `Calipso.jar` und `Admin.jar` in ein Verzeichnis legen
- `java -jar Calipso.jar` (`java -jar Calipso.jar -h` zeigt die möglichen Optionen an)

Linux, bequeme Variante

- `Calipso.jar` und `Admin.jar` in `~/.calipso` oder `/usr/share/java` ablegen
- `calipso` (Shell-Script) aus dem Repository holen [5].
- `./calipso` zum Starten benutzen (`./calipso -h` gibt Auskunft über alle möglichen Einstellungen)
- Optional kann die Umgebungsvariable `JARPATH` auf ein anderes Verzeichnis als `~/.calipso` gesetzt werden

Frage beantworten

- Angabe des Ortes, wo die McKoi-Datenbank angelegt werden soll. Dies kann auch ein relativer Pfad sein. Sollte dabei ein Eingabefehler passieren, muss die Option `-Dphenix.dbMckoiConfig=/pfad/...` beim Starten angegeben werden.
- Pfad zu `Admin.jar` bei der Frage nach `admin.jar` angeben. Dieser Pfad kann auch über `-Dadmin.jar=/pfad/zu/Admin.jar` gesetzt werden.
- Achtung: Die Pfade werden gespeichert, man muss sie also nur beim ersten Mal angeben. Wenn von früheren Tests noch alte Pfade gespeichert sind, so werden diese genommen und nicht nachgefragt!
- mit einem Browser auf `http://serverip:8080/admin` gehen und dort die Datei `Kalender.jar` hochladen

[4] Auf der CD unter /jars enthalten
[5] Eine ausgecheckte Version ist auf der CD enthalten.

2.6 Benchmarks

Um die Benchmarks zu erstellen, beschlossen wir, nicht auf eine Java-Lösung zu setzen, um mögliche Verfälschungen der Resultate durch begrenzende Faktoren in Java auszuschliessen. Da Calipso bereits ein HTTP-Plugin hatte, bot sich das Apache-Benchchmarktool aus dem Paket `apache-utils` (Debian) an. Nebst vieler weiterer, mehr oder weniger nützlicher Informationen, misst das Programm die Anzahl der Anfragen, die pro Sekunde gemacht werden können. Dabei kann man die Benutzung von Keep-Alive sowie die Anzahl der parallelen Clients auf der Kommandozeile angeben.

Mit `sed` waren die relevanten Zahlen schnell extrahiert – blieb noch die grafische Aufbereitung. Dafür schien Gnuplot[6] das geeignete Tool zu sein. Gnuplot legt weniger Wert auf eine „fancy“ Darstellung, ist dafür aber sehr flexibel bei der Auswahl und Darstellung der Daten. Das Erstellen der Plots erinnert sehr an Matlab, ist allerdings anfänglich weniger intuitiv zu verstehen. Hat man die Bedienung aber erst einmal begriffen, will man Gnuplot nicht mehr missen.

Das Benchmarkscript wurde schliesslich erfreulicherweise auch noch von Passix verwendet, um die Leistungsfähigkeit der Redirects herauszufinden. Für die Präsentation konnten wir die Daten im selben Format wie für Gnuplot/Matlab üblich auch an Fabio schicken, welcher sie in Keynote grafisch ansprechend visualisierte:

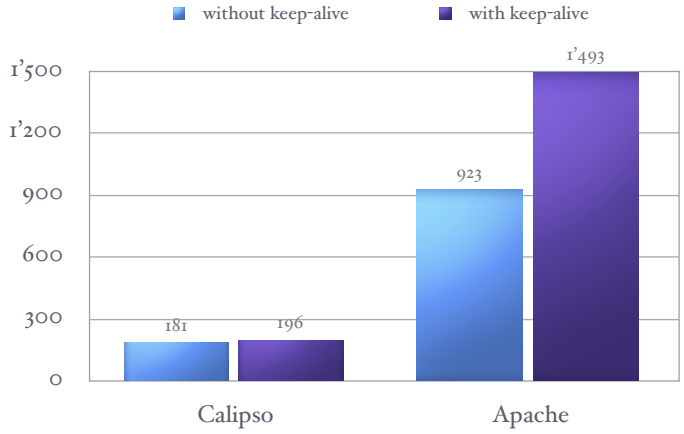


Bild 3: Anzahl Anfragen pro Sekunde bei einem einzelnen Klienten

[6] http://www.gnuplot.info/

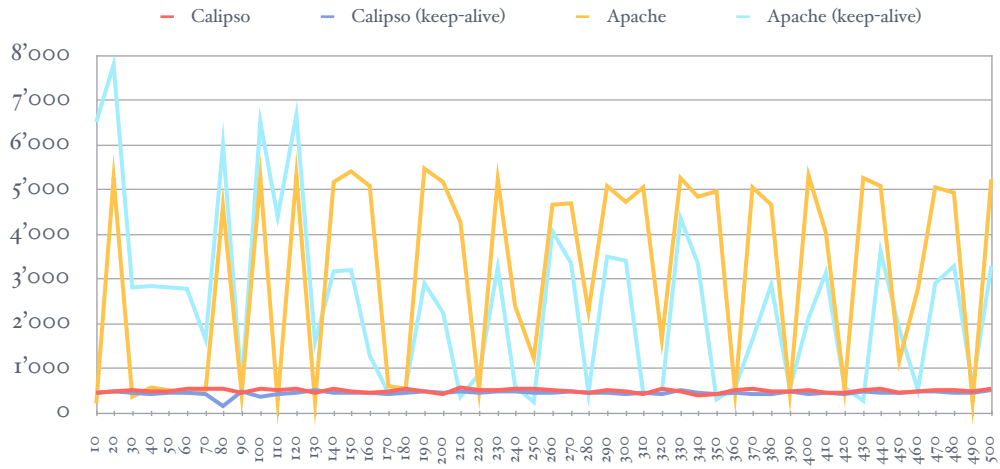


Bild 4: Anzahl Anfragen pro Sekunde (Ordinate) bei parallel zugreifenden Klienten (Abszisse).

Bei den sequentiellen Zugriffen (Bild 3) zeigt sich, dass Calipso deutlich langsamer ist als Apache, doch zu unserer Verteidigung ist anzumerken, dass Apache im Test im Gegensatz zu Calipso eine statische Webseite auslieferte und natürlich sehr viel mehr Zeit in die Optimierung von Apache gesteckt wurde.

Bei den parallelen Zugriffen (Bild 4) zeigt sich, dass Calipso (wie erwartet) langsamer ist, doch scheinen 500 parallele Requests noch kein so grosses Problem zu sein, dass es zu Performance-Einbussen im Vergleich zu einem einzelnen Request käme.

2.7 Probleme und Lösungen

Bis auf das Problem mit dem Classloader gab es eigentlich keine grösseren Probleme. Kleinere Probleme konnten meist innerhalb eines Nachmittags gelöst werden.

2.7.1 Classloader

Dies war das grösste Problem und wurde erst erkennbar, als Phenix für die Objektverteilung auf Clippee umgestellt hatte. Es stellte sich heraus, dass die über einen `URLClassLoader` geladenen Anwendungen auf ihre eigenen, in Phenix gespeicherten Klassen nicht mehr zugreifen konnten, da bei der Deserialisierung der System-Classloader verwendet wurde, welcher die zu deserialisierenden Klassen nicht kannte.

Alle unsere Bemühungen, dem System-Classloader nachträglich die neuen Klassen bekannt zu machen, scheiterten. Auch das Ausweichen auf einen `URLClassLoader` zum Laden von Phenix half nichts, es wurde weiterhin der System-Classloader verwendet.

Glücklicherweise boten Fabio und Roman an, das Problem in Phenix zu lösen.

2.7.2 Socket-Thread schliessen

Wenn eine neue Version einer Anwendung mit eigenem Protokoll-Handler installiert wurde, musste der Handler-Thread beendet werden. Da sich dieser im `accept()`-Systemaufruf befand, war das einfache Beenden durch `interrupt()` nicht möglich. Aber auf die nächste eingehende Verbindung zu warten war auch keine Option.

Die Lösung zeichnete sich erst ab, als wir herausfanden, dass `accept()` eine Exception wirft, wenn der Socket von einem anderen Thread geschlossen wird. Somit war das Problem gelöst.

2.8 Mögliche Erweiterungen

Zwei Dinge wurden beim Entwurf von Calipso nicht beachtet:

- Grundsätzlich kann über jedes Protokoll-Plugin auf jedes Applikations-Plugin zugegriffen werden. Das heisst, dass jede Applikation grundsätzlich für jeden möglichen Client sinnvolle Antworten generieren müsste. Das ist natürlich nur bedingt möglich, da zusätzliche Protokoll-Plugins ohne Probleme nach bereits installierten Applikations-Plugins entwickelt werden können. Dazu gibt es zwei Lösungen:
  - mit dem Problem leben und jedem Client beibringen, dass er jederzeit unabhängig von seinen Vorlieben eine Antwort mit dem Mime-Type text/plain zurückbekommen kann
  - Calipso beibringen, was für Requests von welcher Applikation bearbeitet werden können – dafür müssen die Protokoll-Plugins wissen, wie sie eine adäquate Fehlermeldung generieren, und Calipso muss wissen, welche Applikationen welche Mime-Typen unterstützen
- Kurz vor der Schlusspräsentation tauchte der Wunsch auf, mehrere Kalender parallel zu betreiben. Da das grössere Änderungen im Kalender selbst bedeutet hätte, beschlossen wir, die Kalender-Applikation einfach unter mehreren Namen zu installieren. Da aber jede ihr eigenes Protokoll-Plugin mitbringt, welches natürlich identisch ist, führt das zu Problemen. Leider kamen wir nicht mehr dazu, eine entsprechende Plugin-Verwaltung zu schreiben, sodass es beim Laden einer zweiten Instanz einer Anwendung mit Plugin zu einer Fehlermeldung kommt (Socket schon benutzt) und beim Entladen dieser das Plugin ganz entfernt wird, sodass ein Neustart des Calipso-Servers notwendig wird. Das Problem könnte gelöst werden
  - auf Anwendungsebene, was den Kalender komplexer machen würde. Andererseits könnte so auch beim Java-Klienten eine Liste aller laufenden Kalender angezeigt werden.
  - durch zwei Kalender, die ihr Protokoll-Plugin auf separaten Ports installieren. Dies ist allerdings unflexibel, da für jeden zusätzlichen Kalender die Jar-Datei kopiert und angepasst werden müsste.
  - durch eine Datenverwaltung in Calipso. Dabei würden von Calipso mehrere

Instanzen derselben Anwendung mit verschiedenen Daten gestartet. Diese Lösung behebt die Mehrfachspeicherung in Phenix, bringt aber für Calipso unverhältnismässig viel Komplexität mit sich. Zudem ist es so nicht mehr möglich, mehrere Anwendungen in verschiedenen Versionen gleichzeitig zu betreiben.

– durch die oben erwähnte Plugin-Verwaltung.

2.9 Fazit

Die vorgestellte Lösung für ein erweiterbares System zeigt eine flexible Lösung, um über verschiedene Protokolle auf die gleiche Applikation beziehungsweise über das gleiche Protokoll auf verschiedene Applikationen zuzugreifen. Für den produktiven Einsatz müssten noch die in „Mögliche Erweiterungen“ (Seite 19) beschriebenen Punkte ergänzt werden.

2.10 Persönlicher Bericht Daniel Hottinger

Anfänglich hatte ich grössere Bedenken, da wir bei der Infrastruktur auf von der ISG eingerichtete Rechner angewiesen waren. Es war mir nicht klar, wie lange es dauern würde, bis die von uns gewünschten Entwicklungswerkzeuge (Subversion, Wiki, ...) auf dem uns zugeteilten Server (benutzbar) laufen würden. Nach hartnäckigem Nachfragen erklärte sich Keno bereit, uns einen seiner Privat-Rechner zu geben, auf dem ich ein aktuelles Debian (Sarge) installieren konnte. Ich möchte ihm an dieser Stelle dafür nochmals herzlich danken! Damit war es problemlos möglich, alle nötigen Dienste zum Laufen zu bringen.

Nach einer anfänglich stabilen Phase legte der Server aber immer mehr spontane Reboots ein. Prof. Wattenhofer stellte uns daraufhin (ebenfalls) spontan ein neues Thinkpad T40p zur Verfügung. Auch ihm sei hier nochmals herzlich gedankt. Der Umzug der Daten klappte mit Hilfe des System-Administrators problemlos an einem Nachmittag (das grösste Problem war es, einen Rechner zu finden, bei dem die 3.5"-Platte zur Image-Erstellung eingebaut werden konnte). Danach lief der Server zuverlässig und stabil bis ans Ende des Labors durch.

Im Nachhinein hat sich der Einsatz von freier Software als sehr gute Entscheidung herausgestellt, die ich jederzeit wieder treffen würde. Vor allem das Wiki[7] war für die dezentrale Entwicklung sehr hilfreich, da es eine einfach zu bedienende Oberfläche bietet, die auf allen Plattformen ohne zusätzliche Software benutzt werden kann und für das Schreiben von Dokumentation genauso geeignet ist wie für das Sammeln von Vorschlägen, die Publikation von Interfaces oder das Bugtracking (mit dem von Franziska schnell geschriebenen Plugin).

Ärgerlich waren die Verzögerungen von Seiten der Informatikdienste, welche mehrere Wochen brauchten, um den Port 25 freizuschalten, sodass die Mailingliste für Diskus-

[7] http://moinmoin.wikiwikiweb.de/

sionen in der ersten Hälfte des Semesters praktisch unbenutzbar blieb. Glücklicherweise hatte ich auf Anraten von Keno einen A-Record für den Entwicklungsserver unter meiner eigenen Domain eingerichtet, sodass dieser von Anfang an (und auch nach den anfänglichen IP-Wechseln durch den verkonfigurierten DHCP-Server im Labor) unter einem leicht zu merkenden DNS-Namen erreichbar war.

Auch aus der Sicht des Softwareentwicklers war das Labor lehrreich. So klappte die Arbeit in einem so grossen Team, das sich weitgehend selbst organisierte, meist problemlos. Ein Grund dafür dürfte sicher sein, dass wir uns in Zweierteams aufgeteilt hatten, die klar abgetrennte Aufgaben hatten und untereinander lediglich die Schnittstellen absprechen mussten. Ab und zu gab es Probleme, da nicht alle die gleichen Fähigkeiten hatten. Das liess sich aber meist über IRC schnell lösen. Die Abhängigkeiten unter den Teilprojekten konnten durch schnell lauffähige, aber noch nicht voll funktionale (Hilfs-)Klassen weitgehend entschärft werden.

Witzig war auch die Bestimmung des Namens sowie die Auswahl des Logos. Alles in allem blicke ich auf das Labor als eine lehrreiche und gute Erfahrung zurück.

2.11 Persönlicher Bericht Franziska Meyer

Ich war am Anfang etwas skeptisch, da die Ausschreibung doch recht schwammig war, und ich nicht wusste, was ich mir darunter vorstellen musste. Inzwischen bin ich aber froh, dieses Labor gewählt zu haben, wenn ich mich so an die Schlusspräsentation erinnere, so war es wahrscheinlich schon das, was mich am meisten interessiert. Auch innerhalb des Labors konnte ich den Teil bearbeiten, der mich am meisten interessierte – ich wollte etwas Neues lernen, das ich später wieder benutzen kann. Das ist mit einer Plugin-Schnittstelle sicher der Fall, unsere Marktanalyse zeigte ohne grossen Aufwand knapp zehn grössere und kleinere Systeme mit Plugin-Möglichkeiten.

Wirklich begeistert bin ich von den eingesetzten Hilfsmitteln wie Wiki, IRC, Subversion, Mailingliste, ant und eclipse. Die letzten drei hatten sich bei mir lokal schon bewährt, auch mit Wikis hatte ich etwas Erfahrung. Doch die enge Zusammenarbeit am bevorzugten Arbeitsort (zu Hause) war nur dank dem IRC möglich. Diese Erfahrungen haben dazu geführt, dass ich die von mir geschriebenen Wiki-Plugins noch ein bisschen erweiterte und für meine Semesterarbeit einsetzen werde – zusammen mit Subversion, ant und eclipse. Etwas zu kurz kam Javadoc, anscheinend ist die zusätzliche Arbeit wie das Schreiben von sinnvollen Kommentaren ziemlich unbeliebt. Da die Schnittstellen zwischen den Teil-Projekten entweder dokumentiert waren oder mit mündlicher Hilfe benutzt wurden, spielte das aber keine grosse Rolle.

Natürlich gibt es bei einer so grossen Gruppe ohne klare Führung (wir hatten uns gegen eine Projektleitung entschieden) hin und wieder Probleme bei der Koordination, vor allem, wenn sich niemand zuständig fühlte. Doch schlussendlich haben diese Probleme dazu geführt, dass ich etwas über mich selbst gelernt habe. Und die Zusammenarbeit funktionierte mit wenigen Ausnahmen sehr gut, so dass diese Probleme die Ausnahme waren.

Für mich war das Labor sowohl im Bereich Gruppenarbeit als auch in Bezug auf die

Informatik-Kenntnisse eine gute Erfahrung, die durchaus auch Spass gemacht hat.

Kapitel 3

Passix

3.1 Aufgabenstellung

Damit ein aus mehreren Calipso-Servern bestehendes System nach aussen wie eine Einheit wirkt, sollte ein zentraler Einstiegspunkt für das System geschaffen werden, welcher unter einer öffentlichen Adresse Anfragen entgegennimmt und an die vorhandenen Calipso-Server verteilt. Da dadurch ein potentieller Flaschenhals entsteht, muss das System hohe Anforderungen punkto Ausfallsicherheit und Performance erfüllen. Des Weiteren soll es möglich sein, einfach weitere Passix-Rechner zum System hinzuzufügen.

Die Lösung sollte auch untersuchen, ob mittels dynamischen Wechsels der Server-IP erreicht werden kann, dass die Konfiguration des umgebenden Netzes möglichst nicht angepasst werden muss.

3.2 Architektur

Für den Gateway haben wir zwei Implementierungsmöglichkeiten in Erwägung gezogen:

3.2.1 Variante 1: Network Address Translation (NAT)

Der Client wendet sich mit einem Request (zum Beispiel via HTTP-Protokoll an die URL `http://cpp.ethz.ch/application`) an einen aktiven Gateway, der einfach alle ankommenden Verbindungspakete an den Multiplexer eines Calipso-Servers weiterleitet. Dort wird der Request von der betreffenden Applikation bearbeitet und wieder zurück zum Gateway geschickt, der eine erneute Adressübersetzung durchführt. Der Gateway müsste jedem Paket noch einen Identifier beifügen, der vom Calipso-Server auch der Antwort mitgegeben wird, so dass der Gateway die Pakete an den richtigen Client zurückschicken kann. Eine andere Möglichkeit wäre, dass eine Verbindungsmanagementstruktur für die richtige Zuordnung der Pakete zuständig ist. Wir haben zur genaueren Analyse dieses Ansatzes einen Prototyp erstellt, der auf zwei C-Bibliotheken (libpcap, libnet) zurückgreift. Mit libpcap werden die Pakete auf IP-Ebene wie bei einem Sniffer abgefangen und mit libnet an die Calipsoadresse weitergeschickt. Die Integration in das Gesamtsystem funktioniert über das Java Native Interface (JNI).

Der Vorteil dieser Architektur wäre, dass der Gateway für den Client transparent ist, auch für die Anwendungen wäre die Kommunikation einfach. Auch liesse sich konzeptionell ein System, welches mehr Garantien hinsichtlich Reaktionszeit etc. bietet, realisieren. Aktives FTP z.B. ist aber ohne Spezialmodul im Gateway nicht möglich, da dafür der Aufbau von Rückkanälen durch den Server möglich sein muss.

Ein grosser Nachteil ist, dass alle Verbindungen und der gesamte Datenverkehr über den Gateway laufen, was diesen zum Flaschenhals macht. Wir kamen zum Schluss, dass eine ausreichende Performance des Gateways für die Verarbeitung der Pakete auf IP-Ebene mit Java sehr schwierig zu erreichen wäre.

3.2.2 Variante 2: Yellow Pages

Der Client wendet sich zuerst über einen bekannten Port (80) an den Gateway (`cpp.ethz.ch`) und erkundigt sich mit einem normalen Calipso-Request nach einer bestimmten Anwendung. Der Gateway liefert dann die Adresse eines zuständigen Servers in Form eines HTTP Redirects (Code `302 Found`), worauf sich der Client an diese Adresse wendet und ab diesem Zeitpunkt die Kommunikation ausschliesslich mit dem Calipso-Server stattfindet. Wenn der Server ausfällt, muss der Client nochmals von vorne beginnen und die Verbindung neu aufbauen.

Ein Browser-Client folgt somit direkt dem Redirect und die Weiterleitung erfolgt transparent. Ein Java-Client muss zuerst den Redirect parsen (die Adresse steht in der locationLine der Antwort) und dann separat die Verbindung zu dem zugeteilten Calipso-Server aufbauen. Diese Verbindung kann dann auch mit einem anderen, von Calipso unterstützten Protokoll geschehen. Diese Lösung entspricht zwar nicht dem ursprünglich von uns geplanten, möglichst transparenten Server, jedoch ist protokollunabhängige Transparenz im *Yellow Pages* Modell nicht realisierbar. Das Performanceproblem der *NAT*-Variante tritt nicht auf, da nur der erste Request des Clients an den Gateway geht und der Rest der Kommunikation direkt zwischen Client und Calipso stattfindet.

3.2.3 Passix: Yellow Pages

Da eine funktionierende Lösung am Ende des Semesters oberste Priorität hatte, haben wir uns für die zweite Variante entschieden. Die weiteren Hauptgründe waren, wie oben diskutiert, die Performance und die Einfachheit.

3.3 Ausfallsicherheit

Die Voraussetzung ist, dass der Gateway zu jeder Zeit, auch bei Ausfall von mehreren Rechnern, immer von aussen erreichbar sein muss. Um Ausfallsicherheit zu garantieren, müssen ein oder mehrere Stand-by-Gateways die aktiven Gateways überwachen. Falls ein Gateway (aktiv oder stand-by) ausfällt, muss er ersetzt werden. Hierfür haben wir zwei Varianten in Betracht gezogen:

3.3.1 Variante 1: IP-Wechsel

Der einspringende Gateway übernimmt die IP-Adresse des ausgefallenen Gateways. Jeder Gateway verfügt über eine global vordefinierte Liste von IP-Adressen, wovon eine bestimmte Anzahl (aktive Gateways) statisch ist. Im laufenden System überwacht jeder Gateway die in der Liste nach ihm platzierte IP-Adresse, der letzte stand-by vor den aktiven Gateways überwacht alle diese (statische IPs).

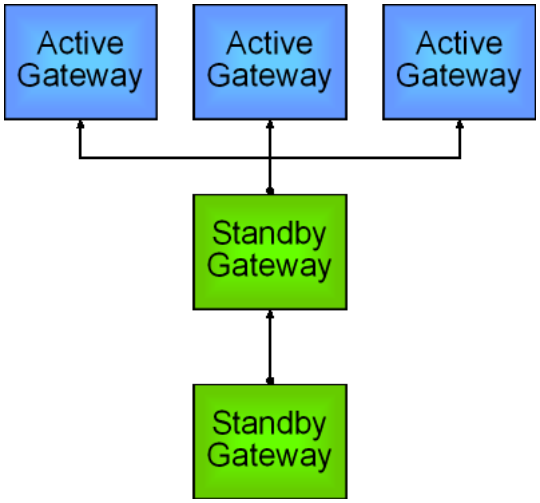


Bild 5:

Die aktiven Gateways haben keine Überwachungsaufgabe. Nach einem Absturz rückt der überwachende Gateway eine Position in der Liste vor, seine nun frei gewordene Adresse wird als Absturz interpretiert und auch übernommen usw. Der abgestürzte Computer wird dann neu gestartet, was in einem IP-Konflikt resultieren würde. Darum wird das Aufstarten das Netzwerks verzögert, und die IP-Adresse zuerst auf die erste Adresse in der Liste (Einstiegsadresse = entry point) geändert.

Technisch wird der IP-Wechsel durch einen Zugriff auf die Systemnetzwerkeinstellungen durchgeführt. Bei der Linux-Variante geschieht dies mit `ifconfig` und bei der Windows-Variante mit `netsh` jeweils mit einem Konsolenaufruf. Die Linux- und Windows-Gateways können so gemischt eingesetzt werden.

3.3.2 Variante 2: DynamicDNS

Ein eigener DNS-Server auf `lab.hotti.ch` erlaubt dynamisches DNS. Alle Gateways könnten gleichzeitig aktiv sein und sich ähnlich wie im obigen Modell überwachen. Bei einem Ausfall müsste bloss der DNS-Eintrag geändert werden. Auch das Aufstarten und erneute Anmelden des Computers wäre kein Problem. Allerdings ist dann der DNS-Server der grosse Schwachpunkt.

3.3.3 Passix: IP-Wechsel

Wir haben uns für den IP-Wechsel entschieden, die Erreichbarkeit des DNS-Servers zu garantieren erscheint uns als sehr schwierig, da dann natürlich wieder dieselben Reliability-Überlegungen auf den DNS-Server angewandt werden müssten. Ausserdem wird das Problem mit dem IP-Wechsel innerhalb des Passix-Systems gelöst und nicht nach aussen getragen.

3.4 Load-Balancing

Für eine effiziente Lastverteilung muss der Gateway erstens alle verfügbaren Calipso-Server und zweitens deren jeweilige Auslastung kennen, um danach die Requests auf eine sinnvolle Art und Weise delegieren zu können. Die Verteilung der Passix-Gateways bedingt eine Synchronisierung der Informationen untereinander.

3.4.1 Leases

Damit ein Calipso-Server bei der Verteilung der Requests berücksichtigt wird, muss er sich periodisch bei einem der aktiven Gateways mit einem Lease anmelden. Leases stellen eine Art Vertrag zwischen einem Calipso-Server und dem Passix-System dar. Der Calipso-Server teilt darin mit, welche Applikationen er bis zum Ablaufzeitpunkt des Leases anbietet. Passix leitet daraufhin während dieser Zeit entsprechende Anfragen an den Calipso-Server weiter. Möchte der Calipso-Server auch noch nach Ablauf der entsprechenden Zeit Anfragen erhalten, muss er das Lease rechtzeitig erneuern.

Leases werden von den Passixservern untereinander mittels *reliable (flooding) broadcast* verteilt. Siehe Konfiguration (Seite 28).

- Ein `lease` enthält folgende Daten:
  - `expiry`: Ablaufdatum des Leases
  - `calipsoAddress`: IP-Adresse des Calipso-Servers
  - `serviceEntries`: Auflistung der angebotenen Applikationen

Damit Leases nicht doppelt verschickt werden, prüft der Empfänger jeweils, ob er von diesem Calipso-Server bereits ein Lease mit gleichem oder späteren Ablaufdatum erhalten hat. In diesem Fall wird das Lease nicht weitergeleitet.

3.4.2 Loadfeedback

Damit Passix in der Lage ist, Anfragen optimal auf mehrere Calipso-Server zu verteilen, muss es Informationen über die Auslastung der Server besitzen.

Es bieten sich zwei Möglichkeiten an:

Variante 1: Aktives Polling

Jeder Gateway fragt periodisch bei den Calipso-Servern nach deren Auslastung. Dafür implementieren wir ein eigenes Protokoll.

Variante 2: Clippee

Zurückgreifen auf die Funktionalität von Clippee (siehe Phenix, Seite 36). Jeder Calipso-Server trägt seine Auslastung selber periodisch in eine Clippee-Tabelle ein, auf welche die Gateways als Clippee-Peers zugreifen.

Passix: Aktives Polling

Der Einsatz von Clippee für Calipso-Server und Gateways würde viele Komplikationen mit sich bringen. So würden die Passix Server nicht stärker mit den Calipsoservern verbunden und der Netzwerkoverhead würde steigen. Wir wählen die erste Variante, die uns einfacher und effizienter erscheint. Die Auslastung wird von den Calipso-Servern selber ermittelt. Die massgeblichen Werte dafür sind die Anzahl Requests, die ein Server in einem bestimmten Zeitintervall beantwortet hat und die durchschnittliche Bearbeitungszeit für einen Request. Zusätzlich wird noch die Varianz der durchschnittlichen Bearbeitungszeit berechnet, was allerdings im Moment von keinem implementierten Load-Balancing-Algorithmus verwendet wird.

- Ein `load feeback` enthält folgende Daten:
  - `measureTime`: Dauer der Messung, entpricht dem Zeitintervall zwischen zwei Abfragen
  - `numOfServedClients`: Anzahl der Requests, die in diesem Zeitintervall bearbeitet wurden
  - `avgServiceTime`: die durchschnittliche Bearbeitungszeit für einen Request
  - `devServiceTime`: die durchschnittliche Abweichung von der durchschnittlichen Bearbeitungszeit

Der Load-Balancing-Algorithmus kann dank einem Interface geändert werden. Der aktuelle Algorithmus berücksichtigt bei einem Request jeweils denjenigen Calipso-Server, bei dem das Verhältnis zwischen Zeit zur Bearbeitung der Requests und Messintervall am kleinsten ist. Durch dieses Verfahren werden entweder Server bevorzugt, die nicht viel Arbeit haben, oder solche, die wegen ihrer Leistungsfähigkeit mehr Requests bearbeiten können.

3.5 Konfiguration

Die Konfiguration des Passix-Systems erfolgt mittels eines XML-Files, welches eine Aufstellung der für Passix zur Verfügung stehenden IP-Adressen enthält. Pro IP ist ausserdem angegeben, ob es sich um eine aktive Gateway-IP handelt und welche anderen Passix-Rechner zu überwachen bzw. zu informieren sind.

Dieses File muss im Voraus manuell verteilt werden.

Beispielkonfiguration:

```
<?xml version="1.0" encoding="UTF-8"?>
<gatewaylist>
    <gateway ip="129.132.177.95" active="false" entrypoint="true">
        <supervises>129.132.177.104</supervises>
        <supervises>129.132.177.105</supervises>
    </gateway>
    <gateway ip="129.132.177.104" active="true" entrypoint="false">
        <informs>129.132.177.95</informs>
        <informs>129.132.177.105</informs>
    </gateway>
    <gateway ip="129.132.177.105" active="true" entrypoint="false">
```

```
        <informs>129.132.177.95</informs>
        <informs>129.132.177.104</informs>
    </gateway>
</gatewaylist>
```

entrypoint gibt dabei an, ob die entsprechende IP als Einstiegspunkt für neu gestartete Passix-Rechner, welche eine nicht aufgeführte IP besitzen, verwendet werden soll.

active gibt an, ob der Passix-Gateway für Clients sichtbar ist und das Weiterleiten von Anfragen zu übernehmen hat.

supervises spezifiziert die von einem Gateway überwachten anderen Rechner

informs definiert die Verbindungen, welche bei der Lease-Verteilung verwendet werden.

Das System muss dabei so konfiguriert werden, dass jeweils ein Passix-Gateway nur durch einen anderen Passix-Gateway überwacht wird. Dadurch können zeitaufwändige Konsensusprotokolle zwischen den überwachenden Passix-Rechnern vermieden werden und ausgefallene Gateways schneller ersetzt werden. Die (wichtigen) aktiven Passix-Rechner werden somit schnell ersetzt, während für das Nachrücken der anderen Gateways mehr Zeit zur Verfügung steht.

3.6 Anleitung

Nachdem die Sourcen heruntergeladen wurden, kann mittels `ant passix` ein Passix.jar erstellt werden (ev. müssen zuerst die Pfadangaben in `build.xml` angepasst werden).

Danach kann Passix wie folgt gestartet werden:

- `java -jar Passix.jar <config.xml> <net device> <passix Port> <redirect port>`

  <config.xml> XML-File, welches die Konfigurationsinformationen für Passix enthält. Mehr dazu unter Konfiguration (Seite 28).

  <net device> Netzwerkdevice, für welches bei Bedarf die IP gewechselt werden soll. Hat die Form ethX (auch unter Windows!).

  <passix port> Port, welcher für die Kommunikation zwischen Passix und Calipso verwendet wird. „Quasi-Standard“ ist 11223 (muss trotzdem angegeben werden!).

  <redirect port> Port, der Redirects versendet. Eine gute Wahl ist `80`.

Passix muss mit root-Rechten ausgeführt werden, damit es die IP des Computers ändern kann. Damit sich Calipso bei einem Passix-Gateway registriert, müssen beim Start auch hier zusätzliche Parameter übergeben werden:

- `java -Dpassix.init=true -Dpassix.port=<passix port> -Dpassix.ownIp=<own ip> -Dpassix.defaultGw=<ip default gw> -jar Calipso.jar`

  <passix init> Option, damit Calipso sich beim angegebenen Passix-Server registriert.

<passix port> Der Port, über den mit Passix kommuniziert wird. Muss dem entsprechen, mit dem Passix gestartet wurde.

<own ip> IP-Adresse, die als Absender in die Leases eingetragen wird.

<ip default gw> IP-Adresse des Passix-Gateways, an den die Leases geschickt werden sollen.

Der Remote Log Viewer kann wie folgt gestartet werden:

- `java -cp Passix.jar ch.ethz.dcg.passix.remoteLogTerminal.LogViewer`

3.7 Benchmark

Anzahl Requests pro Sekunde:

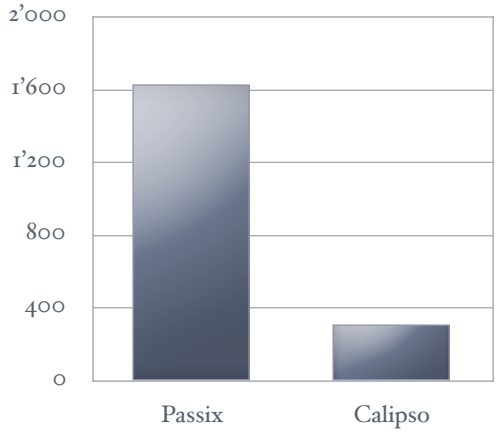


Bild 6:

Auf dem Diagramm sieht man, dass Passix mehr als sechsmal soviele Requests pro Sekunde bearbeiten kann wie Calipso. Da zusätzlich pro Session zwischen Client und Calipso nur der erste Request über Passix läuft, ist auch für eine grosse Anzahl Calipso-Server garantiert, dass das Passix-System keinen Flaschenhals darstellt.

3.8 Probleme und Lösungen

3.8.1 Wahl der Architektur

Vor dieser Entscheidung (siehe *Architektur*) mussten wir uns zuerst einen guten Überblick über die genaue Problemstellung verschaffen. Die von uns favorisierte *NAT*-Methode haben wir zuerst mit einem Prototyp getestet. Dadurch nahm die Planung einige Zeit in Anspruch, was jedoch gerechtfertigt war, denn diese gute Vorbereitung ermöglichte uns ein effizientes Programmieren.

3.8.2 Netzkonsistenz vs. Reaktionsgeschwindigkeit

Auf der einen Seite muss im Falle eines Ausfalls ein Gateway möglichst schnell ersetzt werden, auf der anderen Seite muss vermieden werden, dass zwei Rechner wegen eines Fehlers dieselbe IP zugewiesen bekommen. Als Lösung wurden die Stand-by-Gateways „schlangenförmig“ angeordnet und ein two-level Pingprotokoll zur Überprüfung von anderen Gateways implementiert. Damit kann auf zeit- und nachrichtenintensive Konsensusprotokolle verzichtet werden

3.8.3 Exceptions

Anfänglich wurde bei der Spezifizierung der internen Schnittstellen das Exceptionhandling vernachlässigt, was zu Problemen führte. Daraufhin wurden eigene Exceptiontypen definiert und die entsprechenden Interfaces überarbeitet.

3.8.4 Native Libraries

Während der Evaluation der *NAT* Lösung zeigte sich, dass die plattformabhängige Bereitstellung der native Libraries ein Problem wäre. Da *NAT* nicht implementiert wurde, wurde dies jedoch hinfällig.

3.9 Mögliche Erweiterungen

Bei einer Weiterentwicklung des Systems würden sich folgende Erweiterungen / Verbesserungen anbieten:

Überwachung durch mehrere Stand-by-Gateways In der aktuell implementierten Version ist die Struktur des Netzes, welches durch die sich gegenseitig überwachenden Passix-Server gebildet wird, auf eine „Schlange” beschränkt, eine Baumstruktur mit einer aktiven Wurzel wäre nicht möglich. Möchte man dies ändern, so müsste das Protokoll zur Übernahme von IP-Adressen modifiziert werden.
Es müsste der bisherige Fall, wo ein Gateway genau einen Stand-by-Gateway besitzt, und der Fall, wo mehrere Stand-by-Gateways vorhanden sind, unterschieden werden. Im ersten Fall könnte die bisherige, effiziente Lösung weiterverwendet werden, im zweiten Fall müssten die überwachenden Passix-Gateways ein *leader election* Protokoll ausführen. Der Leader ersetzt dann den ausgefallenen Rechner.

Erkennen von teilweise abgestürzten Passix-Systemen Falls bei einem System nur das Java-Programm abstürzt, aber der Rechner weiterhin lauffähig bleibt, erkennt dies der überwachende Gateway zwar, kann den Rechner aber nicht übernehmen, da dessen IP-Adresse im Netzwerk natürlich nicht frei ist. Dies merkt der überwachende Gateway mit einem 'ping'-Befehl. Der überwachende Gateway müsste nun die Möglichkeit haben, den abgestürzten Passix-Gateway herunterzufahren, was man mit einem remote login versuchen könnte.

Unterstützung beliebiger IPs Momentan kennt jeder Passix-Gateway alle möglichen IP-Adressen, die Passix-Gateways zugeteilt werden könnten. Möchte man erreichen, dass Passix-Gateways beliebige, den anderen nicht bekannte IP-Adressen verwenden können, so muss die oben unter *Überwachung durch mehrere Stand-by-Gateways* skizzierte Lösung ergänzt werden. Stand-by-Clients müssen dynamisch erkennen können, ob noch andere (ihnen unbekannte Gateways) denselben Passix-Rechner überwachen. Ein möglicher Ansatz wäre, Passix-Gateways eine Statistik der ihnen zugeteilten Stand-by-Gateways führen zu lassen. Dies müsste jedoch noch genauer untersucht und detaillierter ausgearbeitet werden.

Schutz vor Network Partitions Die aktuell implementierte Fassung kann im Falle einer Netzpartition zu massiven Problemen führen. Bricht die Verbindung zwischen einem Stand-by-Gateway und dem von ihm überwachten Gateway zusammen, hält der Stand-by-Gateway den Gateway für ausgefallen und übernimmt dessen IP. Wird die Netzverbindung später wieder reaktiviert, so besitzen zwei Rechner die selbe IP-Adresse und das Netz ist inkonsistent. Eine mögliche Lösung könnte hier sein, dass Passix-Gateways die Doppelbelegung von IP-Adressen erkennen können und in diesem Fall auf eine zufällige andere IP wechseln und von dort aus wieder das normale *Redundancy*-Protokoll ausführen. Unter Umständen müsste diese Erkennung allerdings mit C auf Systemebene geschehen.

Erhöhung der Sicherheit Momentan ist das Passix-System sehr anfällig gegenüber Crackern. Um das System sicher in einer feindseligen Umgebung betreiben zu können, müssten zusätzliche Konsistenzprüfungen der erhaltenen Pakete eingefügt werden. Ausserdem müssten die Leases um eine Art Zertifizierungsmechanismus ergänzt werden, so dass keine gefälschten Leases in das System eingeschleust werden können.

3.10 Fazit

Die im Rahmen dieses Labors fertiggestellte Fassung eines ausfallsicheren Gatewaysystems zeigt eine effiziente Möglichkeit für die Verwirklichung eines Hot-stand-by-Systems auf, welche ohne grosse Modifikation des Netzes verwendet werden kann. Für den produktiven Einsatz im Internet müssten noch Erweiterungen vorgenommen werden, welche jedoch den Rahmen dieses Labors gesprengt hätten (siehe *nächste Schritte*, speziell bei *Sicherheit*). Insbesondere müsste man das Augenmerk auf ein Testen unter grosser Last mit spezifischen Programmen richten.

3.11 Persönlicher Bericht Andreas Pfenninger

Ich habe mich für dieses Lab entschieden, weil ich in meinem bisherigen Studium das Gefühl hatte, das praktische Programmieren sei etwas zu kurz gekommen. Jetzt wollte ich auch einmal bei einem grösseren Projekt mitwirken und meine Programmierfertigkeiten verbessern. Die Durchführung des Labs in einem Achterteam war auch in

Bezug auf das Projektmanagement eine Herausforderung. Wir haben uns meiner Meinung nach sinnvoll in vier Zweiergruppen aufgeteilt, die jeweils mehr oder weniger unabhängig arbeiten konnten, und dann an den zwei wöchentlichen Sitzungen ihre Fortschritte und Probleme besprechen konnten. Für eine weitere Erleichterung der Zusammenarbeit sorgten das Lab-interne WikiWeb und ein eigener IRC-Channel.

Am Anfang des Projekts konnte ich vor allem meine Netzwerkkenntnisse über Linux und Windows erweitern. Für die Low-Level-Netzwerkprogrammierung musste ich mich auch mit C und dem Java Native Interface (JNI) intensiv beschäftigen. Auch über Linux habe ich viel gelernt, setze ich doch sonst eher Windows als Betriebssystem ein. Eine wertvolle Erfahrung war für mich dann auch der Testprozess, der gegen Ende des Labs sehr viel Zeit verschlang. Es wurde mir bald klar, dass insbesondere bei einem verteilten System, wie das bei uns der Fall war, jeder weitere Test viel Zeit braucht und deshalb gut vorbereitet sein sollte. Vor allem deshalb war auch mein Zeitaufwand für das Lab relativ hoch.

Die Zusammenarbeit im Passix-Team mit Markus war sehr gut, wir haben uns jeweils gut ergänzt und die Aufteilung der Arbeit funktionierte reibungsfrei. Bei den meisten Komponenten von Passix haben wir denn auch zusammengearbeitet, was auch den Vorteil hatte, dass wir stets den Überblick über das gesamte Passix-System hatten. Auch mit meinen anderen Teammitgliedern war ich sehr zufrieden. Unsere Aufgabe war relativ unabhängig von der Arbeit der anderen Teams, erst gegen Ende des Projekts musste ich mit dem Calipso-Team für eine Integration von Passix zusammenarbeiten, was sehr gut klappte. Mit unseren beiden Betreuern Keno und Aaron verstanden wir uns sehr gut, und in ihrer Rolle als kritische Supervisors trugen sie dazu bei, dass wir am Ende des Semesters ein lauffähiges Programm hatten.

Das Arbeiten im HRS Wireless Lab machte mir Spass, und auch die Infrastruktur war gut, meistens hatte ich vier Laptops zur Verfügung, was auch zum Testen ausreichte. Abschliessend möchte ich festhalten, dass das Lab für mich eine sehr wertvolle Erfahrung war, die ich gern gemacht habe.

3.12 Persönlicher Bericht Markus Egli

Rückblickend war das Labor für mich sicher eine Bereicherung. Besonders die Bearbeitung eines komplexeren Problems in einem doch grösseren Team war eine Bereicherung. Innerhalb der Passix-Teilgruppe fand ich das Arbeitsklima sehr gut. Man konnte sich darauf verlassen, dass der andere seinen Teil termingerecht erledigte und dadurch mit seiner eigenen Arbeit darauf aufbauen. Dort, wo Arbeit mit anderen Gruppen koordiniert werden musste (was selten der Fall war, da Passix sehr unabhängig von Calipso ist), kam mir die Zusammenarbeit ebenfalls unkompliziert und effizient vor.

Probleme bereitete anfänglich das Testen eines so verteilten Systems. Die Übertragung der auf den eigenen Rechnern entwickelten Versionen auf die Laborrechner war zeitaufwändig und beeinträchtigte die Testeffizienz. Deshalb werde ich zukünftig von Anfang an dem Deployment mehr Beachtung schenken. Während der Evaluation habe ich das Java Native Interface kennen gelernt, ausserdem konnte ich Erfahrungen mit

der Entwicklung für mehrere Plattformen sammeln (IP-Change Windows / Linux)

Motivation war für mich nie ein Problem, gegen Schluss nahm sie sogar noch zu, als sich das fertige System abzuzeichnen begann.

Alles in allem ein gelungenes Labor aus meiner Sicht.

Kapitel 4

Phenix

4.1 Aufgabenstellung

Die an uns gestellten Aufgaben lauten wie folgt:

- Sorgen Sie dafür, dass die im System vorhandenen Daten zu jedem Zeitpunkt (evtl. in einer Datenbank) gespeichert sind und gegebenenfalls vollständig rekonstruiert werden können.
- Das System soll ausfallsicher laufen, d.h. es muss möglich sein, dass ein beliebiger Rechner ausfällt, ohne dass das übrige System in Mitleidenschaft gezogen wird.
- Sie müssen nicht bei Null mit der Programmierung beginnen. Sie können den Code des aktuellen Client/Server-Forschungsprototypen Clippee verwenden.
- Clippee soll die Daten nicht mehr im Hauptspeicher halten (doppelte Datenhaltung).
- Es muss möglich sein, nicht nur Bytearrays zu versenden.

Wir erfüllen diese Aufgaben und erweitern die Funktionalität mit folgenden Komponenten:

- Persistenz & Ausfallsicherheit: wird mit einer Datenbank gewährleistet
- Verteilung, Konsistenz & Ausfallsicherheit: Clippee wurde an unsere Bedürfnisse angepasst
- Komplexe Typen: es ist nun möglich, beliebige Objekte zu versenden
- Doppelte Datenhaltung: wird durch ein eigenes Cachingsystem verhindert
- NameSpace & ClassLoader: Laden/Entladen von Applikationen zur Laufzeit

4.2 Wie wird Phenix gestartet

Die Hauptklasse von Phenix ist der `DataStore` und die Startmethode in ihr ist `startup(ObjectDistributor, DatabaseSystem)`. Wenn diese Methode aufgerufen wird, läuft Phenix mit dem übergebenen `DatebaseSystem` und `ObjectDistributor`.

Diese beiden Komponenten können durch beliebige andere Komponenten ersetzt werden, welche die vorgegebenen Interfaces implementieren. Somit ist es möglich, auch andere Datenbanken oder einen anderen Verteilungsmechanismus zu wählen.

In unserem System verwenden wir für die Verteilung Clippee.

`ClippeeObjectDistributor(Hashtable)`

Die Hashtable muss 3 Strings beinhalten:

1. eine statische Liste von kommagetrennten IPs der vorhandenen Peers (Key = „RemoteIPs“)
2. den Remote Port (Key = „RemotePort“), für alle Peers gleich
3. den Local Port (Key = „LocalPort“), entspricht meistens dem Remote Port, ausser wenn man 2 Peers auf dem selben Computer starten will

Beim Starten wird die IP-Liste durchlaufen. Der erste laufende Peer in dieser Liste dient als Bootstrapping-Peer. Wenn in der Liste kein laufender Peer gefunden wird, wartet die Startroutine einen Moment und versucht es erneut.

Damit das System auch hochfährt, wenn noch kein Peer am Laufen ist, gibt es einen besonderen Peer, welcher selbständig starten kann. Dieser Peer wird durch die Reihen-

folge der IPs bestimmt. Ist die eigene IP gleich der ersten IP in der Liste, wird nach einmaligem erfolglosem Durchlauf der Liste, der eigene Peer als „First Peer“ gestartet. Das heisst, der Peer kontaktiert keinen Bootstrapping-Peer, sondern liesst alle Daten aus der Datenbank.

Durch dieses Vorgehen wird garantiert, dass während einer laufenden Session nur ein aktiver Peer als „First Peer“ gestartet wurde und alle anderen Peers sich dann bei irgend einem laufenden anmelden. Auf dem Local Port wird lokal die Verbindung geöffnet, und der Remote Port sagt, wo auf den „Remote Peers“ die Verbindungen stehen.

Als Datenbank kommt Mckoi zum Einsatz.

```
MckoiDatabaseSystem(String)
```

Als Parameter übergibt man den Pfad, wo die Datenbank angelegt werden soll.

4.3 Kern von Phenix

Zu Beginn des Labs wurden wir uns einig, dass es sinnvoll ist, eine neue Schnittstelle zur verteilten Speicherung von Datenobjekten zu definieren. Sie sollte sich zwar konzeptuell am bestehenden Clippee-System anlehnen, aber dessen Komplexität so weit wie möglich verbergen. Erwartet wurde also eine einfacher und bequemer zu benutzende Funktionalität, welche weitgehend dem Update-Konzept von Clippee entspricht, die aber gleichzeitig auch noch die Persistenz aller Datenobjekte garantiert.

Die Neugestaltung einer Schnittstelle vom Objektspeichersystem zu Calipso und dessen Applikationen brachte uns bei der Implementation grosse Vorteile gegenüber tiefgreifenden Anpassungen innerhalb des Clippee-Systems. Da Phenix von Grund auf modular gestaltet wurde, konnte die Entwicklung an den einzelnen Komponenten weitgehend unabhängig erfolgen. Die strikt getrennten Bereiche zur Objektverteilung und Datenbankanbindung lassen sich ausserdem jederzeit durch Module ersetzen, die andere Teilsysteme als Clippee und Mckoi verwenden.

4.3.1 Architektur

Die wesentlichen Kompenenten von Phenix und die dazwischen ausgetauschten Meldungen:

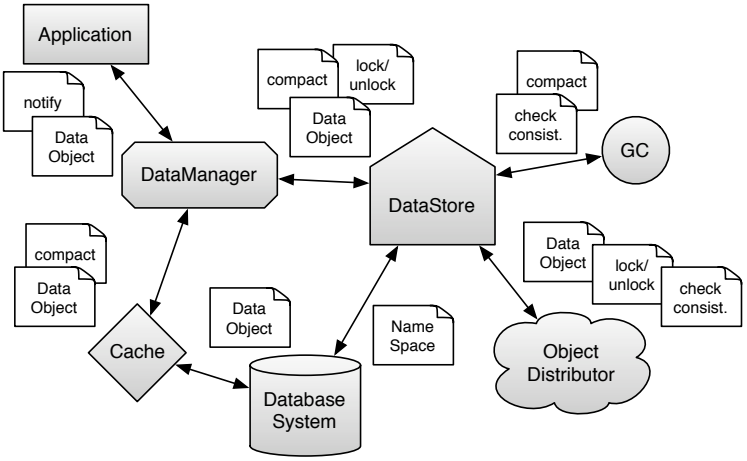


Bild 7:

DataStore

DataStore steuert zentral das Zusammenspiel aller Komponenten in Phenix. Die gesamte Funktionalität wird in Form einer statischen Schnittstelle bereitgestellt. Gestartet wird Phenix über die Methode `startup(DatabaseSystem, ObjectDistributor)`, wobei das zu verwendende Datenbanksystem und der einzusetzende Objektverteiler angegeben werden. DataStore initialisiert diese beiden Teile, lädt die persistent gespeicherten NameSpaces aus der Datenbank, instanziert die dazugehörenden DataManagers und veranlasst den ObjectDistributor, einen vollständigen Datenabgleich mit anderen Peers durchzuführen. Wenn `startup()` beendet ist, kann Phenix benutzt werden. Gestoppt wird Phenix über die Methode `shutdown()`, die u.a. dafür sorgt, dass die Datenbank korrekt heruntergefahren wird.

DataManager

Jede Applikation erhält einen eigenen `DataManager`, welcher sämtliche Datenobjekte (`DataObject`) einer Applikation verwaltet. So ist sichergestellt, dass zwischen den Applikationen kein Konflikt bei der Benennung von Objekten entstehen kann. Phenix erzeugt und löscht DataManagers, wenn sich Änderungen am RootManager ergeben haben (siehe Namensraumkonzept), das heisst, wenn Calipso neue Namensräume (`NameSpace`) hinzufügt oder entfernt. Der zu einem bestimmten NameSpace gehörende DataManager kann über `getDataManager()` von DataStore bezogen werden. Den RootManager liefert die Methode `getRootManager()`.

ObjectDistributor

`ObjectDistributor` ist für die gemeinsame Verwendung derselben Objekte auf verschiedenen Peers zuständig. In unserem Fall kapselt er vollständig das Clippee-System. Phenix übergibt dem ObjectDistributor bei einem Update ein DataObject in Form einer

Kombination von globalem Key (String mit Präfix des Namensraums) und serialisierten Objektdaten (Bytearray). Um gleichzeitige Updates auf verschiedenen Peers zu vermeiden, wird auch ein globaler Locking-Mechanismus vorausgesetzt (Lock, Unlock). Lesezugriffe auf Objekte geschehen ohne Locking, da die Datenobjekte auf einem Peer lediglich Kopien der aktuellsten gültigen Version sind und deshalb von gleichzeitigen Änderungen nicht beeinflusst werden.

Die vom DataStore erhaltenen Nachrichten versendet der ObjectDistributor beispielsweise übers lokale Netz an andere Rechner. Phenix wurde im Hinblick auf die eher schwachen Garantien von Clippee entwickelt, weshalb Updates jederzeit fehlschlagen oder auch nicht auf allen aktiven Peers ankommen können. Ein Hintergrund-Thread wurde deshalb eingeführt, der von Zeit zu Zeit Abgleichungen der Datenbestände mit anderen zufälligen Peers auslöst. ObjectDistributor erhält von fremden Peers die Nachrichten und leitet sie über die Schnittstelle des DataStores ins lokale Phenix-System.

Garbage Collector und Consistency Checker

Dem DataStore ist ein Thread (`GC`) zugeordnet, der zu konfigurierbaren Zeitpunkten die Caches leert und für überprüfende Abgleichungen der Datenbanken auf den verschiedenen Peers sorgt. Er ruft dazu `compact()` auf den vorhandenen DataManagers und `checkConsistency()` auf dem ObjectDistributor auf. Um die abzugleichenden Datenbestände einzuschränken, werden die Konsistenzchecks DataManager-weise durchgeführt.

DatabaseSystem

Das `DatabaseSystem` kapselt den Zugriff auf eine SQL-Datenbank. Intern wird dazu JDBC und das frei verfügbare RDBMS Mckoi in der Embedded-Fassung eingesetzt. Die Abfragen und Manipulationen in SQL arbeiten alle auf einer einzigen Tabelle, welche die DataObjects in der aktuellsten committeten Version persistent speichert. Eine mögliche Erweiterung des DatabaseSystems könnte aber sein, auch eine SQL-Schnittstelle für Applikationen anzubieten.

Cache

Zur Performance-Steigerung beim lesenden Zugriff auf DataObjects trägt der `Cache` zwischen DataManager und DatabaseSystem bei. Er verwaltet seinen Inhalt nach dem LRU-Prinzip, indem bei jedem Zugriff ein Timestamp gesetzt wird. Ausserdem stellt er einen simplen Locking-Mechanismus bereit, um gleichzeitige Update-Vorgänge auf demselben DataObject zu verhindern. Dank dieser Locks ist auch sichergestellt, dass Update-Nachrichten nicht in einer falschen Reihenfolge abgearbeitet werden. Denn ein Update eines DataObjects wird nur verarbeitet, wenn seine Versionsnummer grösser ist, als die bisher grösste bekannte Versionsnummer.

DataObject

Calipso und die Applikationen benutzen zur verteilten Speicherung ihrer Daten die Schnittstelle des DataManagers und als Dateneinheiten Subklassen von `DataObject`. DataManager stellt im Wesentlichen zwei Methoden bereit, um ein bestehendes Objekt zu erhalten: `get()` und `getLocked()`. `get` liefert ein lesbares DataObject in seiner aktuellsten Version. `getLocked` liefert eine Kopie des DataObjects in seiner aktuellsten Version, das aber zusätzlich global gelockt und somit zur Veränderung vorgesehen ist. Weil `getLocked` immer eine Kopie des Cache-Inhalts zurückliefert, sollten Manipulationen auf einem DataObject andere lokale Threads nicht beeinflussen (Snapshot-Prinzip).

Ist ein Objekt verändert, kann die Applikation das Objekt an Phenix zum Commit übergeben. Dafür ist die Methode `commit()` in DataObject vorgesehen, welche die eigene Versionsnummer inkrementiert (und bei Misserfolg des Commits wieder zurücksetzt). Phenix serialisiert das DataObject, leitet es dann an den ObjectDistributor weiter und blockiert die Ausführung des Threads, bis im Erfolgsfall das Objekt anschliessend an die Verteilung auch im lokalen Speicher persistent abgelegt wurde. Soll ein Datenobjekt neu erstellt werden, erzeugt die Applikation eine neue Instanz einer Subklasse von DataObject und committet sie. Der Vorgang schliesst erfolgreich ab, wenn der Key des Objekts nicht an ein existierendes DataObject vergeben ist.

Damit DataObjects auch gelöscht werden können, was Clippee von sich aus nicht unterstützt, fügten wir den DataObjects einen Expiry-Eintrag hinzu. Er gibt an, ab welchem Zeitpunkt ein Objekt als abgelaufen (d.h. nicht mehr existent) angesehen werden soll. Als Hilfsmethode steht deshalb in DataObject die Methode `delete()` bereit, die das Expiry-Datum auf einen Zeitpunkt in der Vergangenheit setzt und das Objekt committet.

Der eigentliche Löschvorgang wird durchgeführt, wenn bei einem Durchlauf des GC-Threads Objekte in der Datenbank gefunden werden, die ein Ablaufdatum in der Vergangenheit haben, aber noch nicht gelöscht wurden. Die Versionsnummer zum Key muss allerdings weiterhin gespeichert bleiben, damit zukünftige DataObjects mit demselben Key eine höhere Versionsnummer erhalten (siehe Methode `getLastVersion()`) und keine Konflikte mit veralteten Datenbeständen anderer Peers entstehen. Kapitel Verteilung und Konsistenz beschreibt den Aufbau der Keys näher.

DataObjectObserver

Applikationen können sich über stattfindende Updates informieren lassen (notify). Dazu registrieren sie sich mit einem `DataObjectObserver` bei ihrem DataManager und erhalten fortan Benachrichtigungen (`created`, `updated`, `expired`), wenn sich eines der beobachteten Objekte verändert hat.

4.3.2 Namensraumkonzept

Calipso muss zur Laufzeit Applikationen laden und entladen können. Phenix unterstützt diese Funktionalität durch das `NameSpace`-Konzept. Dazu gehört in erster Linie,

dass Objekte verwaltet, instanziert, serialisiert und deserialisiert werden können, deren Klassen beim Start des Calipso-Servers noch nicht bekannt sind. Diese Anforderung kann nur durch eigene ClassLoaders erfüllt werden, denn über sie lädt die JVM bei Bedarf die benötigten Klassen und Ressourcen.

`NameSpace` ist eine Unterklasse von `DataObject`. Der Konstruktor erhält einen `JarStream`, der sämtliche zu einer Applikation gehörenden Ressourcen enthält. Calipso muss somit bloss einen solchen NameSpace committen, damit eine Applikation mit ihren Ressourcen auf sämtlichen Peers verfügbar und geladen bzw. entladen wird. Alle NameSpace-Objekte werden in einem zentralen DataManager – genannt RootManager – innerhalb des DataStores abgelegt.

Kommt ein neuer NameSpace hinzu, wird ein entsprechender DataManager erzeugt, der wiederum einen `ClassLoader` bereithält, welcher ausschliesslich die Klassen und Ressourcen genau dieser Applikation kennt. Packages, die zum System gehören und somit vom Standard-ClassLoader geladen werden sollen, können mit der Methode `setSystemPackages()` von DataStore definiert werden. Will sich Calipso ein applikationsabhängiges Objekt erzeugen, verwendet es `loadClass()` in DataManager. Darauf folgende Aufrufe innerhalb der Applikation geschehen ohne zusätzliche Vorkehrungen, weil Applikationsklassen standardmässig mit dem ClassLoader ihres NameSpaces ausgerüstet werden.

Wichtig ist das Namensraumkonzept auch bei der Deserialisierung von DataObjects, die vom ObjectDistributor oder der Datenbank geliefert werden. Für die Serialisierung und Deserialisierung von Objekten sind die jeweiligen DataManagers mit den Methoden `serialize()` und `deserialize()` zuständig.

Um ein Datenobjekt zu serialisieren, wird Javas Standard-Serialisierung mit dem `Serializable`-Interface und `ObjectOutputStream` verwendet. Problematischer ist hingegen die Deserialisierung von Applikations-Objekten, weil über den `ObjectInputStream` Objekte ankommen können, deren Klassen der JVM nicht bekannt sind. Aus diesem Grund haben wir die Methode `resolveClass()` in `ObjectInputStream` überschrieben, damit an dieser Stelle nicht versucht wird, die Klassen aus dem Standard-ClassLoader zu laden, sondern dass der Ladevorgang an den ClassLoader übertragen wird, der für den NameSpace des DataManagers zuständig ist.

4.4 Persistenz

Für die persistente Speicherung der Objektdaten benutzen wir ein relationales Datenbanksystem mit JDBC-Schnittstelle. SQL ist für unsere Zwecke nicht unbedingt die effizienteste Implementationsvariante, jedoch erlaubt die Verwendung von JDBC für die Zukunft eine einfache Anpassung des bestehenden `DatabaseSystem`-Moduls an andere SQL-Systeme oder die Erweiterung des Moduls mit mehr Funktionalität wie z.B. einer SQL-Schnittstelle für Applikationen.

Wir haben uns nach gründlicher Untersuchung verschiedenster relationaler Datenbanksysteme für die vollständig in Java implementierte Lösung names Mckoi entschieden. Sie erfüllt alle unsere Anforderungen optimal und hat sich als sehr gut geeignetes

Testsystem bewährt. Sie ist kostenlos als Open Source verfügbar, was sich insbesondere bei Abklärungen zur Implementation des SQLObjects auszahlte.

Die relationale Tabelle, welche die DataObjects speichert, besteht aus fünf Spalten:

Domain	Bezeichnung des NameSpaces
Name	Key des Objects
Version	vollständige Versionsnummer
Expiry	Verfallsdatum, 0 für unbeschränkte Gültigkeit
Data	serialisierte Objektdaten, `NULL` bei abgelaufenen Objekten

Bei der Erzeugung des MckoiDatabaseSystems muss dem Konstruktor lediglich der Pfad zu einem leeren Verzeichnis übergeben werden, in welchem Mckoi alle notwendigen Dateien erzeugt. Die Methode `getConfig()` erlaubt die Anpassung weiterer Konfigurationsparameter. Erwähnenswert ist io_safety_level. Der Wert 10 steht für die grösste Datensicherheit, geht aber auf Kosten der Performanz, weil jeder Commit eine Synchronisation der Dateipuffer auf die Festplatte verursacht. Üblicherweise sollte auch der Wert 2 ausreichen, bei dem die Dateipuffer asynchron durch einen Hintergrundthread auf die Festplatte geschrieben werden.

4.4.1 Anforderungen ans DBMS

- Einbindbarkeit der Datenbank in die JVM von Calipso. So entstehen keine Abhängigkeiten von externen Prozessen und Betriebssystemeigenschaften. Die Datenbank ist vollständig unter der Kontrolle von Phenix.
- Performantes Locking-Verfahren, möglichst Row-Level-Locking und Concurrency Control mit Snapshots. Lesende und schreibende Transaktionen sollen sich nur selten blockieren, dank Snapshots würden sie sich nie blockieren.
- Im Hinblick auf zukünftige Entwicklungen eine möglichst weitgehende Implementation von JDBC und eine umfangreiche Unterstützung des SQL-Standards. Ebenfalls für künftige Erweiterungen ein integriertes User- und Rechte-Management. Dadurch liessen sich die Tabellen und Schemas nach NameSpaces auftrennen, sowie sicherstellen, dass gewisse Transaktionen nur lesenden Zugriff erhalten.

Mckoi erfüllt alle Anforderungen und liess sich auf unseren Testsystemen auch ohne zusätzliche Konfiguration bequem ausführen. MySQL unterstützt erst mit InnoDB Row-Level-Locking, ist aus einer JVM heraus nur sehr beschränkt steuer- und konfigurierbar und muss als plattform-abhängiges Paket installiert werden. HSQLDB und QED (ebenfalls reine Java-Pakete) sind allgemein weniger mächtig als Mckoi.

4.4.2 SQLObject

Auch bei JDBC trafen wir das Problem der Deserialisierung von DataObjects an. Wir verwenden zur Objektspeicherung die standardisierten JDBC-Methoden `setObject()` und `getObject()`. Die Data-Tabellenspalte deklarierten wir entsprechend als `JAVA_OBJECT`. Die Serialisierung und Deserialisierung der übergebenen Objekte geschieht automatisch innerhalb des DBMS bzw. JDBC-Treibers.

Da Phenix eigene ClassLoaders benutzt, hat die Datenbank von sich aus keinen Zugriff auf Applikationsklassen. Um die Deserialisierung der DataObjects trotzdem gemäss JDBC zu ermöglichen, übergibt DatabaseSystem die Objekte immer in Form eines SQLObjects. Die Klasse `SQLObject` ist in `ch.ethz.dcg.phenix` vorhanden und darum innerhalb JDBC immer ladbar. Sie enthält ein Domain-Feld, das die Bezeichnung des NameSpace beinhaltet und ein Bytearray mit der serialisierten Form des ursprünglichen DataObjects.

Lässt JDBC nun das SQLObject serialisieren, wird ein Objekt abgespeichert, für das die Deserialisierung keine Applikationsklasse mehr benötigt. Veranlasst JDBC die Deserialisierung eines SQLObjects, ruft der ObjectInputStream die Methode `readResolve()` auf (siehe Javas Serializable-Spezifikation), welche das Bytearray mit Angabe des Domains zur Deserialisierung an Phenix weiterleitet und dann das korrekt instanzierte DataObject zurückliefert. `getObject` in `ResultSet` gibt somit jeweils ein echtes DataObject zurück.

4.5 SQL-Schnittstelle

Mit „SQL-Schnittstelle“ ist hier die Funktionalität gemeint, welche es den Applikationen erlauben würde, mittels SQL-Befehlen auf selbstdefinierte Datenbanktabellen zuzugreifen. Eine solche SQL-Schnittstelle wurde während unseres Labs zwar in den Grundsätzen durchdacht aber nicht implementiert, weil sie nicht hätte fertiggestellt werden können, bis sie die Applikationsentwickler benötigt hätten. Deshalb seien hier bloss stichwortartig Punkte für mögliche Implementationsansätze aufgelistet.

4.5.1 Schwierigkeiten

Die grösste Schwierigkeit für ein derartiges SQL-System ergibt sich aus der fehlenden Garantie, dass versendete Updates tatsächlich umgehend auf allen fremden Peers ankommen. Der sendende Peer erhält nur ungenügende Rückmeldungen über den Erfolg eines Updates. Updates können jederzeit fehlschlagen und auf Remote Peers gar nicht, nur teilweise oder überall ankommen, ohne dass der sendende Peer diese Ergebnisse mit Sicherheit unterscheiden könnte.

Da eine Datenbank besonders kritisch ist hinsichtlich verpasster Updates, müssen spezielle Vorkehrungen getroffen werden, damit die Tabellen wenigstens auf jedem Peer für sich jederzeit konsistent sind. Ausserdem ist es natürlich wichtig, dass niemals gleichzeitige Veränderungen an derselben Datenbank (des NameSpaces) bzw. an denselben Tabellen stattfinden. Es muss also pro NameSpace oder pro Tabelle einen globalen Lockingmechanismus geben. Mit SQL-Statement ist im folgenden eine atomare Ausführungseinheit gemeint, die nicht nur ein einzelner Befehl, sondern auch eine in sich geschlossene Transaktion sein könnte.

4.5.2 SQL-Statements versenden

Idee: Jedes manipulierende SQL-Statement wird an die übrigen Peers verteilt. Probleme: Wohin wendet sich ein Peer, wenn er feststellt, dass er ein Statement verpasst hat? Wird in einem solchen Fall der Zugriff auf die lokale Datenbank solange verweigert, bis die fehlenden Statements angekommen sind? Vorteil: Jedes Statement wird genau einmal auf jedem Peer ausgeführt.

4.5.3 SQL-History-Objekt

Idee: Das Protokoll der ausgeführten SQL-Statements wird in einem DataObject gespeichert. Somit kann bei Erhalt einer entsprechenden Update-Nachricht, die lokale Datenbank gelöscht und durch Abarbeitung des Protokolls neu aufgebaut werden. Vorteil: Ein einziges Objekt, das einen konsistenten Zustand darstellt und weitgehend mit der bestehenden Phenix-Infrastruktur verarbeitet werden kann. Nachteile: Das Protokoll wächst mit jedem manipulierenden SQL-Statement. Das Protokoll muss immer wieder durchlaufen werden.

4.5.4 SQL-Snapshot-Objekt

Idee: Wie das History-Objekt enthält das Snapshot-Objekt den aktuellen Zustand der Datenbank. Allerdings nicht in Form eines Statement-Protokolls, sondern als Dump der in den Tabellen enthaltenen Daten und deren Metadaten (Tabellendefinitionen). Vorteil: Der Snapshot wächst und schrumpft mit den tatsächlich gespeicherten Daten. Nachteil: Daten müssen immer wieder gedumpt und eingelesen werden.

4.5.5 Kombination

Wahrscheinlich wird sich eine Kombination obiger Ansätze als beste Lösung erweisen: z.B. Snapshot, der zusätzlich die letzten beiden ausgeführten Befehle und deren eindeutige Identifikationen beinhaltet. (Als Identifikation wird die Versionsnummer des DataObjects nicht ausreichen, da sie den Zeitpunkt der SQL-Ausführung und den urhebenden Peer nicht angibt.) Nun kann ein Peer feststellen, ob er den vorletzen Befehl schon ausgeführt hat. Wenn dies der Fall ist, muss er nur den neuesten nachführen. Im anderen Fall kann er anhand des Snapshots die gesamte Datenbank neu aufbauen. Vorteile: Datenbank muss selten neu aufgebaut werden. Snapshot wächst und schrumpft mit tatsächlicher Datenmenge. Nachteil: Gesamter Snapshot muss immer erzeugt und versendet werden.

4.6 Verteilung und Konsistenz

In diesem Teil wird beschrieben, wie Clippee verändert wurde, damit es unseren Bedürfnissen entspricht. Alle nicht erwähnten Dinge wurden direkt von Clippee übernommen und werden deshalb auch nicht weiter beschrieben.

4.6.1 Bootstrapping

Beim Startvorgang werden als erstes alle Key-Version-Paare aus der Datenbank gelesen und in einer Liste gespeichert. Diese Liste wird zum einen benötigt, um die Locking-Hashtable (`lockedObjects`) im DataManager zu initialisieren, und zum andern wird sie dem Bootstrapping-Peer übermittelt.

Da man eine Liste nicht einfach so übermitteln kann, musste eine Message mit entsprechendem Handler geschrieben werden. Dies sind die `GetNewObjectsMessage` und der `GetNewObjectsHandler`, welche die beiden Klassen `GetLockedObjectsMessage` und `GetLockedObjectsHandler` in Clippee ersetzen. Der Handler, welcher im Bootstrapping-Peer ausgeführt wird, sendet als Antwort nur noch diejenigen Objekte zurück, welche eine höhere Version haben, oder jene, die auf dem startenden Peer noch nicht in der Datenbank vorhanden sind.

Ein Peer kann erst dann als Bootstrapping-Peer dienen, wenn der eigene Startvorgang beendet ist. Dies ist dann der Fall, wenn man alle Daten von seinem eigenen Bootstrapping-Peer erhalten hat oder wenn man als „First Peer“ startet und `initFirstPeer` in der PeerMain-Klasse aufruft. In der ursprünglichen Version von Clippee war es möglich, dass gewisse Peers als Bootstrapper erreichbar waren, obwohl sie selber noch am Starten waren. Dies lag daran, dass beim Initialisieren des DataManagers, `bootstrapper = null` war und `isBootstrapping` dies als Indiz dafür nahm, dass der Peer schon vollständig gestartet wurde. Dieses Fehlverhalten wurde durch weitere Flags beseitigt.

4.6.2 LazyDataChecker

Das Problem mit dem `LazyDataChecker` ist, dass er eine vollständige Kopie von `lockedObjects` im DataManager anlegt. Bei diesem Vorgang muss er fast alle Daten aus der Datenbank auslesen, da in der Locking-Hashtable keine Daten gehalten werden (doppelte Datenhaltung, siehe Lock). Einige der Daten könnten etwas effizienter aus dem Cache gelesen werden, aber das verbessert die Situation nicht gross. Um dies zu vermeiden haben wir den LazyDataChecker deaktiviert und ein eigenes Verfahren für diesen Dienst implementiert.

Unser Verfahren heisst ConsistencyCheck und funktioniert nach dem gleichen Prinzip wie unser Bootstrapping. Wir senden einem zufälligen Peer eine Liste mit allen Key-Version-Paaren aus der Datenbank. Nun könnte man meinen, dass dies die Datenbank im gleichen Mass wie der LazyDataChecker blockieren würde. Das tut es aber nicht, da im Gegensatz zum LazyDataChecker, wo alle DataObjects verlangt sind, was ein Deserialisieren mit sich bringt, hier nur der Key und die Version verlangt sind. Diese Information liegt in der Datenbank in separaten Kolonnen und muss nicht aus den DataObjects extrahiert werden.

Die Übermittlung der Liste funktioniert hier aber etwas anders als beim Bootstrapping. Sie wird direkt als Bytearray in der `ConsistencyCheckMessage` versendet und dann mit dem `ConsistencyCheckHandler` dem DataStore des Remote Peers überreicht. Dieser vergleicht dann die Liste mit seinen eigenen Key-Version-Paaren und macht auf je-

dem lokal neueren Objekt ein normales Update. Somit werden alle Peers nochmals die aktuelle Version dieses Objektes erhalten. Dies scheint auch vernünftig, da die Wahrscheinlichkeit, dass bei einem Fehler in der Konsistenz nicht nur ein Peer eine veraltete Version hat, ziemlich gross ist.

Der ConsistencyCheck wird alle paar Minuten ausgeführt.

In einer überarbeiteten Version von ConsistencyCheck wäre es gut, wenn man in der `ConsistencyCheckMessage` nur eine begrenzte Anzahl von Key-Version-Paaren versenden würde, damit die Nachrichten nicht zu gross werden. Man könnte die einzelnen Nachrichten dann auch auf verschiedene Peers verteilen und somit die Belastung auf mehrere Peers verteilen.

4.6.3 Lock

Hier musste das Problem mit der doppelten Datenhaltung gelöst werden. Die doppelte Datenhaltung entstand da durch, dass alle über Clippee versendeten Daten zum einen in den Locks, genauer im DataObject des Locks, und zum anderen in den jeweiligen Applikationen gehalten wurden. Unser Ziel war es nun, die Locks so abzuändern, dass sie nur noch die für Clippee relevanten Daten beinhalten. Damit man die nachfolgenden Erklärungen versteht, darf man den Unterscheidung zwischen den Locks, welche in `lockedObjects` gespeichert sind und jenen Locks, welche ausserhalb von `lockedObjects` verwendet werden, nie vergessen.

Ein erster Schritt zur Verhinderung der doppelten Datenhaltung ist, dass man die Instanzen innerhalb und ausserhalb von `lockedObjects` trennt. Somit kann man auf einem Lock ausserhalb von `lockedObjects` alle Methoden ausführen, ohne dass die Daten im Lock unkontrolliert die Locks in `lockedObjects` überschreiben. Leider gibt es aber auch Daten, die man an beiden Orten haben muss, sprich in `lockedObjects` und im äktuellen"Lock. Dies ist zum Beispiel bei der Version der Fall.

Damit solche wichtigen Informationen nicht verloren gehen, ruft jede Methode im Lock, welche eine Veränderung relevanter Informationen mit sich bringt, `updateLock` im DataManager auf. `updateLock` kopiert dann alle neuen Informationen nach `lockedObjects`.

Bei einem `getLock` im DataManager, dies ist eine Methode, welche zu einem Key den entsprechende Lock zurückliefert, wird nur eine Kopie des Locks in `lockedObjects` erstellt. Diese Massnahme muss sein, da so die Locks innerhalb und ausserhalb von `lockedObjects` getrennt sind. Das DataObject im Lock wird erst dann von Phenix nachgeladen, wenn es gebraucht wird, sprich, wenn im Lock die Methode `getDataObject` aufgerufen wird. Somit wird der Datenbankzugriff auf ein Minimum reduziert. Bei einem erneuten Aufruf von getDataObject wird das zuvor geladene Bytearray verwendet, ohne dass die Datenbank abgefragt wird. Dies ist auch sinnvoll, da die aktuellen Daten schon vorhanden sind.

4.6.4 DataObject

Die `DataObjects` von Clippee dienen nur noch zur Datenübermittlung zwischen den einzelnen Peers. In Phenix gibt es ein eigenes DataObject, welches Superklasse aller Objekte ist, die eine Applikation über Phenix updaten will.

Bei einem Update wird ein Applikationsobjekt zu einem Bytearray serialisiert und in das entsprechende DataObject von Clippee geschrieben, d.h. dass beide Objekte den gleichen Key und die gleiche Version haben. Dieses Clippee-DataObject entspricht demjenigen, welches im Lock gespeichert wird (ohne das Bytearray wegen der doppelten Datenhaltung).

Die Version wurde von Integer auf Long erweitert, so dass die vorderen 32 Bit die Version des Namespaces und die hinteren 32 Bit die Version des Objektes selbst angeben. Bei jedem Update auf einem Objekt wird seine Version inkrementiert. Wenn ein Namespace gelöscht und wieder neu erstellt wird, wird seine Version ebenso inkrementiert. Auf diese Weise ist es einfach möglich gelöschte Objekte eines alten Namespaces von denen eines neuen Namespaces zu unterscheiden auch wenn sie den gleichen Key haben, da die Version monoton steigend ist. (siehe "Kern von Phenix: DataObject")

Im DataObject von Clippee gibt es eine neue Methode `rebuildData`, welche von `getDataObject` im Lock aufgerufen wird. Sie veranlasst, dass, wenn das Bytearray im DataObject noch nicht aus der Datenbank rekonstruiert wurde, dies nun geschieht. Des weiteren bekam jedes DataObject in Clippee einen Type, welcher angibt, ob Updates an Phenix weitergeleitet werden sollen (Type = 1), oder ob es sich um Clippee-interne Updates handelt, welche Phenix nicht kümmern müssen (Type = 0). Dies ist zum Beispiel beim Erstellen von Locks der Fall.

4.6.5 ClippeeListener

Der ClippeeListener ist ein ChangeListener, welcher in Clippee bei jedem Notify aufgerufen wird, aber nur solche Nachrichten an den DataStore weiterleitet, welche ein DataObject mit dem Type = 1 enthalten. Er nimmt somit alle Updates auf Clippee-Ebene entgegen und leitet die für den DataStore relevanten weiter.

4.6.6 Updates

Die Updates werden über die von Clippee zur Verfügung gestellten Methoden `updateObject`, `acquireLockGlobally`, `releaseLockGlobally` und weitere Hilfsmethoden ausgeführt. Die Anbindung an den DataStore erfolgt über den ClippeeObjectDistributor, welcher neben der Anbindung auch noch das Aufstarten von Clippee erledigt (siehe „Wie wird Phenix gestartet“, Seite 36).

4.7 Benchmarks

Alle Benchmarks wurden auf den Laptops im HRS F9 durchgeführt.

Es gab sowohl bei den Updates als auch bei den Gets immer 3 verschiedene Szenarien mit je 10 Wiederholungen. Diese Szenarien wurden jeweils einmal mit Datenbank und einmal ohne Datenbank ausgeführt. Die Anzahl der Updates bzw. Gets pro Durchlauf wurde so hoch gewählt, dass die Abweichung der einzelnen Messungen vom Durchschnitt unter 5% lag. In der Hälfte aller Fälle lag die Abweichung sogar unter 1%.

Als Updates werden nur jene gezählt, die erfolgreich auf allen Peers durchgeführt wurden, d. h. bei denen `updateObject` im DataManager von Clippee nicht fehl schlug. Grund für diese Fehlschläge sind abgelaufene Timeouts, welche etwa pro hundert Updates einmal auftritt. Dies geschieht aber nur, wenn zu viele Threads auf den DataManager von Clippee zugreifen. Mit aktiver Datenbank werden die Objekte im Cache und in der Datenbank aktualisiert und alle Observer, welche in diesem Namespace auf dieses Objekt horchen werden benachrichtigt. In den Testszenarien gab es jeweils immer einen Observer pro DataManager. Ohne Datenbank operiert das System nur auf dem Cache, der Rest verhält sich aber gleich wie mit Datenbank.

4.7.1 Szenarien

- 1 DataManager with 1 thread: simuliert das Verhalten einer einzelnen Applikation, welche sequenzielle Zugriffe auf ein DataObject macht. Dazu verwendet die Applikation einen DataManager. Damit die Zugriffe sequentiell ablaufen, gibt es nur einen Thread in der Applikation.
- 1 DataManager with 24 threads: simuliert das Verhalten einer einzelnen Applikation, welche 24 parallele Zugriffe auf 24 verschiedene DataObject macht. Dazu verwendet die Applikation einen DataManager. Damit die Zugriffe parallel ablaufen, gibt es 24 Threads in der Applikation, wobei alle Threads den gleichen DataManager verwenden aber auf ein eigenes DataObject zugreifen.
- 24 DataManagers with 1 thread each: simuliert das Verhalten von 24 Applikation, welche jeweils sequentielle Zugriffe auf ein eigenes DataObject machen. Dazu verwendet jede Applikation einen eigenen DataManager. Damit die Zugriffe sequentiell ablaufen, gibt es nur einen Thread pro Applikation.

Es gibt immer drei Säulen nebeneinander, wobei die linke Säule den maximal gemessenen Wert, die mittlere den Durchschnitt und die rechte den minimal gemessenen Wert darstellt.

 Messungen ohne Datenbank

 Messungen mit Datenbank

4.7.2 Updates

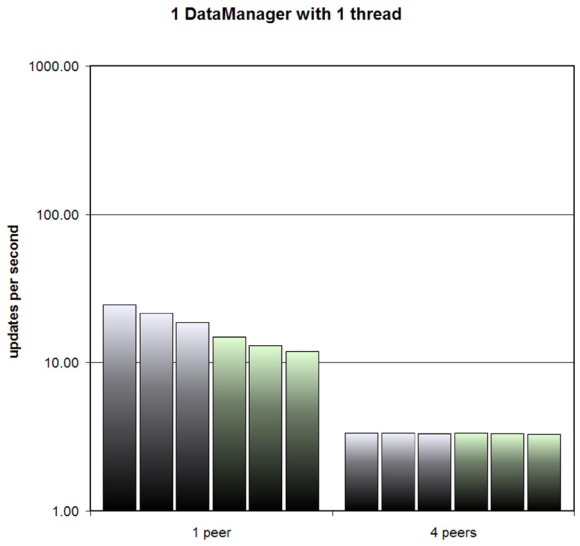


Bild 8:

Durchschnittswerte von links nach rechts 21, 13, 3, 3. Man sieht gut, wie die Performance bei 4 Peers auf Grund der Netzdelays stark zusammenfällt und die Geschwindigkeitseinbusse durch die Datenbank verschwindet.

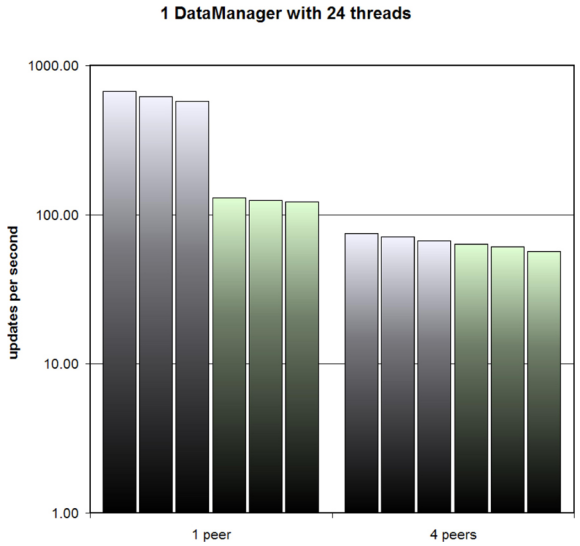


Bild 9:

Durchschnittswerte von links nach rechts 610, 125, 70, 60. Mit dieser Messung kann man zeigen, wie das System mit 4 Peers gut skaliert.

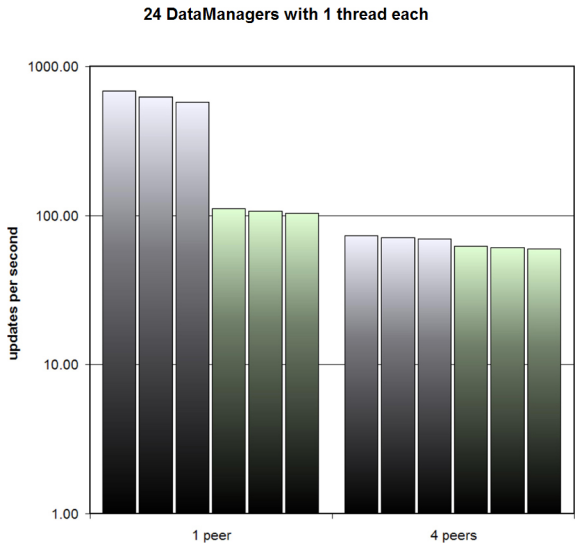


Bild 10:

Durchschnittswerte von links nach rechts 622, 106, 70, 60. Da die Durchschnittswerte fast identisch mit der vorherigen Messung sind, kann man getrost sagen, dass der DataManager und die Synchronisation darin keinen Flaschenhals des Systems darstellen, denn im Gegensatz zur oberen Messung fällt hier kein Synchronisationsaufwand an.

4.7.3 Gets

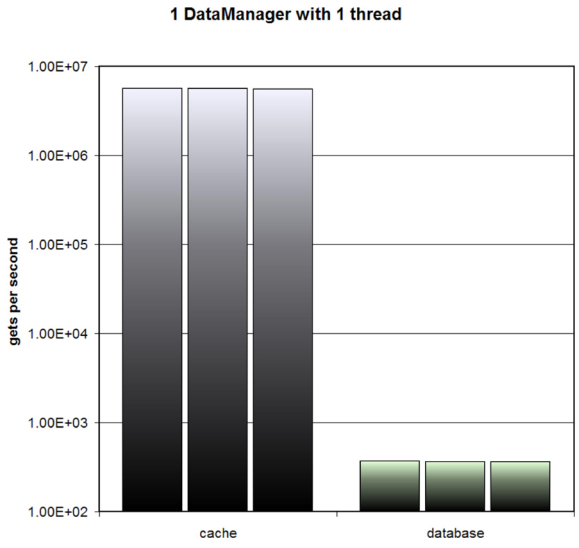


Bild 11:

Durchschnittswerte von links nach rechts 5.5 Mio. und 360. Das typische Verhalten im laufenden Betrieb entspricht jenem mit überwiegendem Cachezugriff. Nur beim

Aufstarten des Systems ist es notwendig, dass alle Objekte aus der Datenbank gelesen werden.

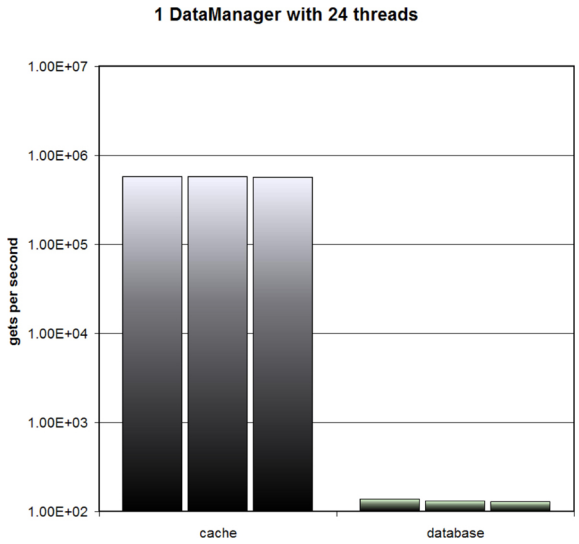


Bild 12:

Durchschnittswerte von links nach rechts 570000 und 130. Im Gegensatz zu den Updates, wo die Performance mit parallelem Zugriff gesteigert werden konnte, sieht man hier, dass die Synchronisation im DataManager den Cachezugriff um den Faktor 10 ausbremst. Der Datenbankzugriff aber wird nicht durch den DataManager, sondern durch die Datenbank selber ausgebremst, denn auch in ihr muss synchronisiert werden, da nun 24 verschiedene Objekte verlangt werden und nicht nur immer dasselbe wie in der ersten Messung.

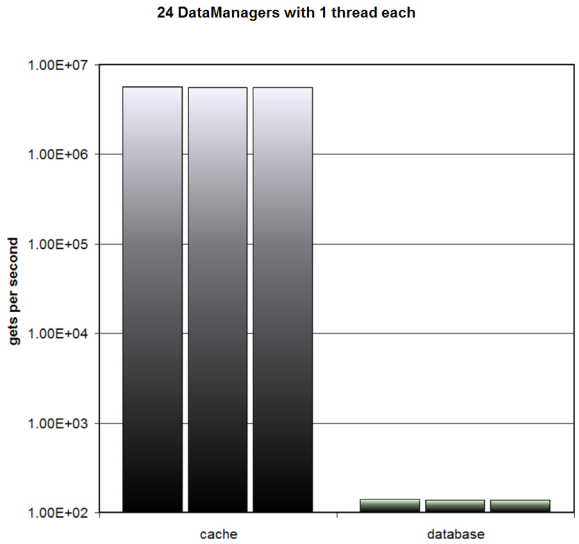


Bild 13:

Durchschnittswerte von links nach rechts 5.5 Mio. und 136. Diese letzte Messung bestätigt die vorherige Vermutung, da der Cachezugriff nicht mehr ausgebremst wird, wenn man anstelle von 1 nun 24 DataManagers verwendet.

4.8 Persönlicher Bericht Fabio Lanfranchi

Ich blicke mit sehr positiven Erinnerungen auf das Lab zurück. Die gemeinsame Arbeit an unserem recht umfangreichen Projekt war lehrreich. So konnte ich mich zum einen intensiv mit sehr praktischen Aspekten der Informatik befassen. Selten hat man während des Studiums eine so gute Gelegenheit, die theoretischen Kenntnisse direkt in eigener Programmierarbeit anzuwenden. Der Bereich der Objektverwaltung und Datenbanken, mit dem ich mich hauptsächlich befasste, hat mir thematisch sehr zugesagt und stark zu meiner Motivation beigetragen.

Zum andern war natürlich die Zusammenarbeit der Teilgruppen ein wesentlicher Aspekt unseres Labs. So konnten wir wichtige Erfahrungen sammeln und sehen, wie sich die vielen Beteiligten am Projekt organisieren können und sollten, wie grundlegend es bei einem derartigen Projekt ist, dass möglichst bald stabile Schnittstellen zwischen den Teilprojekten und Komponenten definiert sind.

Zu Beginn war ich etwas skeptisch, ob ein Lab mit 8 Personen überhaupt klappen kann. Aber wir organisierten uns rasch in Zweiergruppen, die fast von Beginn weg klare Zuständigkeiten sowie Aufgaben hatten und ihre Teilprojekte recht unabhängig verfolgen konnten. Der Einsatz von Mailing-Listen, IRC, Subversion und Wiki machte sich rasch bezahlt für die direkte Kommunikation und den Austausch von Sourcen und Dokumenten.

Die Zusammenarbeit im „Phenix-Team“ mit Roman klappte vorzüglich. Besonders ab dem Zeitpunkt, wo wir den endgültigen modularen Aufbau unseres Systems festgelegt hatten, lief unsere Produktion auf Hochtouren – zwar mit beträchtlichem Zeitaufwand, dafür ohne gröbere Komplikationen.

Gewisse Reibung mit anderen Teilgruppen entstand, als wir Phenix einem gründlichen Umbau unterziehen mussten, um zusätzliche, im Verlauf des Labs aufgetretene Anforderungen zu erfüllen (NameSpaces und LRU-Cache). Dadurch verzögerte sich die Fertigstellung von Phenix etwas, was nicht besonders optimal war für ein Grundsystem, auf das die Applikationen angewiesen sind.

Stünde ich wieder vor der Wahl, mich an einem Lab dieser Art zu beteiligen, würde ich mich wieder klar dafür entscheiden.

4.9 Persönlicher Bericht Roman Metz

Das Lab hat mir sehr gut gefallen. Es war eine gute Erfahrung, mit anderen Leuten gemeinsam an einem Projekt zu arbeiten. Bei der Grösse unserer Gruppe hatte ich Anfangs meine Bedenken, dass sich Koordinationsprobleme ergeben würden. Die Aufgabenstellung erlaubte es uns aber, vier kleine Gruppen zu bilden, welche in ihrer Aufgabe zum Teil sehr autonom arbeiten konnten. Somit war das Thema der Koordination

etwas entschärft. Des weiteren sind die Sitzungen reibungslos abgelaufen, so dass alle wichtigen Dinge besprochen wurden.

Ein ganz anderer Punkt ist der Zeitaufwand. Hier haben die 15 Stunden pro Woche bei weitem nicht gereicht. Mein Aufwand lag im Schnitt bei ca. 25 Stunden pro Woche, wobei ein grosser Teil dieser Zeit während 4 Wochen investiert wurde, in denen ich 60 Stunden pro Woche arbeitete. In den übrigen Wochen lag der Aufwand bei den prognostizierten 15 Stunden.

Die an mich gestellte Aufgabe habe ich gerne gemacht, auch wenn es noch interessanter gewesen wäre, wenn ich die Funktionalität von Clippee selber implementiert hätte. Als Fazit würde ich sagen, dass ich mit meinem jetzigen Wissen wieder ein Lab absolvieren würde.

Kapitel 5

Applikationen

5.1 Motivation

Die Motivation für diesen Teil des Labs sollte eigentlich klar sein. Denn was nützt eine schöne Architektur, die ausfallsicher, persistent, erweiterbar und lastbalanciert ist, wenn man sie nicht entsprechend nutzen kann? Genau an diesem Punkt setzten die Applikationen an. Sie zeigen mit konkreten Beispielen das gesamte System in Aktion.

5.2 Applikationsentwicklung

5.2.1 public interface Application

Alle Applikationen, die in Calipso geladen werden sollen, müssen das Application-Interface implementieren. Das Interface umfasst vier Methoden, von denen aber noch nicht alle von Calipso auch benutzt werden.

void init(DataManager)

Diese Methode wird beim Laden einer Applikation aufgerufen, bevor ein Request sie erreicht. Im wesentlichen wird hier der empfangene DataManager in dem Objekt gespeichert, um ihn später für das Speichern und Laden eigener DataObjects benutzen zu können. Hier können auch weitere applikationsspezifische Initialisierungen vorgenommen werden.

Response handleRequest(Request)

Diese Methode ist das eigentliche Herzstück einer Applikation. Sie wird vom Calipso-System bei jedem Aufruf der Applikation aufgerufen und mit einem entsprechenden Request versorgt. Die Methode verarbeitet den Request und gibt eine Response mit der gewünschten Antwort zurück.

void upgrade()

Das Upgrade von Applikationen wird im Moment noch nicht unterstützt, falls es aber mal unterstützt werden sollte, wird in diesem Fall diese Methode aufgerufen.

void dispose()

dispose() wird als letzte Methode aufgerufen, bevor eine Applikation entladen wird. Hier können noch gewisse Aufräumarbeiten erledigt oder Ressourcen freigegeben werden.

5.2.2 public interface Handler

Calipso unterstützt prinzipiell beliebige Standard- aber auch eigene Protokolle. Dazu muss einfach ein Protokoll-Handler nach diesem Interface implementiert werden. Standardmässig ist im Moment nur ein HttpHandler vorhanden, der Anfragen von Webbrowsern, aber auch anderen Clients, welche HTTP verstehen, auf Port 8080 verarbeiten kann.

void init(Multiplexer, Socket)

Die Initialisierungsmethode wird vor dem Ausführen von `start()` beim Laden des Handlers ausgeführt und nimmt einen Calipso-Multiplexer und einen Socket entgegen, die in dieser Methode lokal gespeichert werden sollten. Hier können auch noch weitere protokollabhängige Initialisierungen gemacht werden.

void start()

Diese Methode startet den Protokoll-Handler, der dann die Daten vom Socket liest, diese in ein Request verpackt, dem Multiplexer via `handleRequest()` übergibt und die Response wieder geeignet formatiert über den Socket zurückgibt. Falls der Handler die Klasse Thread erweitert, können mehrere Requests von der gleichen Verbindung bearbeitet werden, ohne den Port zu blockieren. Dann muss einfach der vorher beschriebene Vorgang in der `run()`-Methode geschehen, die `start()`-Methode befindet sich dann in der Superklasse Thread.

5.2.3 preferences

Zu einer Applikation gehört auch immer eine Textdatei namens „preferences“, die sich im Hauptverzeichnis der Jar-Datei befindet. Sie besteht aus einer Anzahl Schlüssel/Wert-Paaren, die jeweils durch ein Gleichheitszeichen getrennt und durch einen Zeilenumbruch abgeschlossen werden. Diese Schlüssel sind:

name

Der Name der Applikation.

```
name = kalender
```

description

Eine kurze Beschreibung, was die Applikation leistet.

```
description = ein einfacher Kalender fuer eine Gruppe von Leuten
```

mainClass

Die Klasse, welche geladen werden muss. Diejenige, die das Interface `Application` implementiert.

```
mainClass = ch.ethz.dcg.app.kalender.Kalender
```

handler

Die Klasse, welche einen eigenen Handler zur Verfügung stellt und das Interface `Handler` implementiert.

```
handler = ch.ethz.dcg.app.kalender.JavaClientHandler
```

Falls kein eigener Handler benutzt wird kann einfach „none“ angegeben werden.

```
handler = none
```

handlerPort

Optionaler Schlüssel, falls handler nicht none ist. Er gibt an, auf welchem Port der eigene Handler arbeitet.

```
handlerPort = 3425
```

publicFilePath

Falls Dateien über das Http-Protokoll aus der Jar-Datei direkt an den Client zurückgesendet werden sollen, müssen sie mit einem speziellen Präfix versehen werden, der durch diesen Schlüssel bestimmt wird.

```
publicFilePath = public_
```

Falls dann eine Anfrage der Form http://server/app/file.ext den HttpHandler erreicht, wird direkt die statische Datei public_file.ext aus der Jar-Datei zurückgeschickt. So können einfach Logos oder andere Bilder in Web-Applikationen eingebunden werden.

5.2.4 public class Utils

static Response fileResponse(String, Class, String)

Dateien aus der Jar-Datei der Applikation laden und als Response zurücksenden kann man auch aus den Applikationen. Dazu ist kein spezieller Präfix nötig. Die Klasse Utils stellt eine Methode zur Verfügung, die eine Datei liest und gleich auch noch in eine Response verpackt. Als erster Parameter muss der Mime-Type der Datei angegeben werden, als zweiter die eigene Klasse (am einfachsten mit `getClass()`) und als dritter der Dateiname inklusive Pfad in der Jar-Datei.

```
Utils.fileResponse("text/html", getClass(), "admin/admin.html");
```

static String escapeForHTML(String)

Diese Methode nimmt einen String entgegen und ersetzt alle kritischen Zeichen (>, <, ö, ä, usw.) durch Html-konforme Zeichenketten (`>`, `<`, `&ouml;`, `&auml;` usw.).

static byte[] loadFile(InputStream)

Lädt ein File von einem gegebenen InputStream und gibt es als Byte-Array zurück.

5.2.5 Praktische Hinweise

Parameter an die Applikation übergeben

Im Fall einer Webapplikation können Parameter mittels eines HTML-Formulars übergeben werden oder direkt in einem Link.

```
<a href="?key=value&blobb=zonk">klick mich</a>
```

Vor dem Fragezeichen muss in den meisten Fällen nichts angegeben werden, weil der Browser selber den Teil vor dem Fragezeichen mit der aktuellen URL ergänzt. Das action-Attribut bei Formularen kann meistens auch leer gelassen werden, weil dann automatisch das Dokument verwendet wird, in welchem sich das Formular befindet.

```
<form action='' method='post' enctype='multipart/form-data'>
```

Alternativ kann in beiden Fällen der Rückgabewert vom

```
Request.getServerAttributes().get("INVOKE_INFO");
```

angegeben werden, was die Verweise absolut macht, indem die URL der Applikation angegeben wird.

Bei anderen Protokollen als HTTP wird im ProtokollHandler bestimmt, wie ein Request des Protokolls in die clientAttributes-HashMap „übersetzt“ wird.

Dateien hochladen

Dateien hochladen mit einen Webbrowser funktioniert mit einem normalen Upload-Formular.

```
<form action='' method='post' enctype='multipart/form-data'>
  File: <input type='file' name='uploadFile'>
  <input type='submit' value='OK'>
</form>
```

Der Name des Files befindet sich dann in den clientAttributes unter dem Namen des file-Felds

```
String fileName = (String)clientAttributes.get("uploadFile");
```

die Datei an sich bekommt man über

```
byte[] file = Request.getPostData();
```

und ihren Mime-Type über

```
String mimeType = (String)serverAttributes.get("MIME_TYPE");
```

Objekte speichern

Um Objekte speichern zu können, müssen die entsprechenden Klassen `ch.ethz.dcg.phenix.DataObject` erweitern. Der Konstruktor einer solchen Klasse muss also mindestens einen key in Form eines String und den DataManager der Applikation entgegennehmen und diese an den Konstruktor des DataObject übergeben.

```
super(key, dataManager);
```

Auch müssen diese Klassen das Marker-Interface `java.io.Serializable` implementieren. Achtung, der Konstruktor des DataObject kann eine DataManagerException werfen, die vom eigenen Konstruktor einfach weitergeleitet werden kann und beim Erzeugen eines neuen Objekts abgefangen werden muss.

Das eigentliche Speichern des Objekts und das Verteilen auf alle Calipso-Server funktioniert dann mittels der von DataObject geerbten Methode `commit()`, die auf dem Objekt ausgeführt werden muss.

Objekte laden

Beim Laden von Objekten muss man unterscheiden, ob es sich nur um Lesezugriffe handelt oder ob das Objekt auch verändert werden soll. Im ersten Fall reicht folgender Aufruf auf dem lokalen DataManager

```
datamanager.get(key);
```

was ein Objekt zurückliefert, das dann noch entsprechend gecastet werden muss. Wenn man ein Objekt verändern will, muss man sich einen Lock auf dem Objekt verschaffen

```
datamanager.getLocked(key, timeout);
```

wobei das timeout in Form eines longs die Tausendstelsekunden angibt, die man maximal warten will. Der Lock auf dem Objekt muss natürlich auch wieder freigegeben werden, was einerseits mit einem `commit()` (speichert die Änderungen) oder mit einem `unlock()` (verwirft die Änderungen) auf dem Objekt geschehen kann.

Veranschaulichende Applikation

Im Folgenden wird eine sehr einfache Beispielapplikation vorgestellt.

```
/*
 * Import von einigen Klassen, die immer benötigt werden.
 */
import ch.ethz.dcg.calipso.Request;
import ch.ethz.dcg.calipso.Response;
import ch.ethz.dcg.calipso.exceptions.ApplicationException;
import ch.ethz.dcg.phenix.DataManager;
import ch.ethz.dcg.phenix.DataManagerException;

import java.io.Serializable;

/*
 * Die Klasse implementiert das Application-Interface
 */
```

```
public class Hello implements ch.ethz.dcg.calipso.Application
{
    /*
     * Der DataManager, der von Calipso der init-Methode
     * übergeben wird, muss lokal gespeichert werden
     */
    private DataManager dm = null;

    public void init(DataManager dm) throws ApplicationException
    {
        /*
         * Der DataManager wird lokal gespeichert, und zwei
         * DataStrings werden in Phenix abgelegt
         */
        this.dm = dm;
        DataString html =
          new DataString("<h2><blink><i>Hello</i> <b>Browser</b>.</blink><h2>",
            "html", this.dm);
        DataString plain =
          new DataString("Hello to some other Client than a Browser.",
            "plain", this.dm);

        /*
         * Durch den Aufruf der commit-Methode des DataObjects
         * werden die Objekte in Phenix gespeichert
         */
        html.commit();
        plain.commit();
    }

    /*
     * In der handleRequest-Methode findet die eigentliche Action statt.
     * Das Request-Objekt bekommt sie von Calipso, wenn die
     * Applikation von einem User aufgerufen wird.
     */
    public Response handleRequest(Request request) throws ApplicationException
    {
        DataString response;

        /*
         * Der erwartete MimeType der Antwort kann im Request
         * herausgelesen werden.
         */

        if(request.getContentType()=="text/html")
        {
            /*
             * Über DataManager.get("key") erhält man das Objekt, das
             * mit dem Schlüssel "key" gespeichert wurde.
             * Wenn das Objekt verändert werden soll, muss hier
             * DataManager.getLocked("key") aufgerufen werden.
             */
            response = this.dm.get("html");

            /*
             * Es muss ein Response-Objekt zurückgegeben werden, mit
             * dem MimeType der Antwort.
             */
            return new Response("text/html", response.string);
        }
        else
        {
```

```
            response = this.dm.get("plain");
            return new Response("text/plain", response.string);
        }
    }

    public void upgrade()
    {
        System.out.println("HelloWorld:␣upgrading...");
    }

    public void dispose()
    {
        System.out.println("HelloWorld:␣disposing...");
    }
}

/*
 * In Phenix können nur von DataObject erweiterte Klassen gespeichert
 * werden. DataString ist ein Wrapper für einen String, damit er als
 * DataObject in Phenix gespeichert werden kann.
 * Die zu speichernde Klasse muss auch serialisierbar sein.
 */

public class DataString extends ch.ethz.dcg.phenix.DataObject
  implements Serializable
{
    public String string;

    public DataString(String string, String key, DataManager manager)
      throws DataManagerException
    {
        /*
         * Dem Konstruktor von DataObject wird der zuständige Manager
         * und ein Key als String, mit dem das Objekt später
         * wiedergefunden werden kann übergeben. Die eigentlichen
         * Daten, der String, werden lokal in der Klasse gespeichert.
         */
        super(manager, key);
        this.string = string;
    }
}
```

5.3 Beispielanwendung Kalender

5.3.1 Funktionsweise und Beschreibung

Die Kalenderapplikation stellt einen Terminplan dar, welcher für ganztägige Einträge (=Termine) verwendet werden kann. Pro 'Bildschirmseite' wird immer ein ganzer Monat mit allen Usern angezeigt. Für jeden Tag dieses Monats kann nun pro User genau eine Aktivität und damit auch eine Farbe aus der Aktivitätenliste ausgewählt werden. Die Mindestdauer eines Termins ist ein Tag; er kann aber über mehrere Tage hinweggehen (z.B. für Ferien). Auch kann einem Eintrag eine kurze Beschreibung hinzugefügt werden. Die User sind in zwei Kategorien unterteilt: normale User und Administratoren. Normale User können nebst dem Bearbeiten der eigenen Terminen nur das eigene Passwort ändern und Termine anderer User anschauen. Administratoren können User

und Aktivitäten hinzufügen, löschen oder verändern sowie sämtliche Passwörter ändern. Sie können auch Einträge von allen Usern betrachten und ändern.

5.3.2 Architektur auf der Serverseite

Die Architektur des Kalenders auf der Serverseite ist nicht unbedingt befriedigend und würde wohl bei einer Neuimplementierung anders aussehen. Mehr dazu dann bei den möglichen Verbesserungen und Weiterentwicklungen. Prinzipiell besteht der Kalender aus UserKalender-Objekten, für jeden User eines, wo Termine gespeichert werden.

Activity

In einem Activity-Objekt wird der Name der Aktivität, die zugeordnete Farbe und eine eindeutige ID gespeichert.

ActivityList

ActivityList gibt es pro Kalender jeweils nur eine. Dort werden die Aktivitäten und auch die IDs verwaltet. Man kann neue Aktivitäten anlegen und bestehende löschen oder bearbeiten.

KalenderEntry

Die Termine an sich werden in KalenderEntries verwaltet. Ein Termin besitzt eine Beschreibung, eine via ID zugeordnete Aktivität und eine eigene ID.

User

Die Einführung einer User-Klasse wurde durch das neue Feature des Login notwendig. Innerhalb eines User-Objekts wird sein eindeutiger Name, seine Funktion (user oder administrator) und sein Passwort gespeichert.

UserKalender

Im UserKalender ist dann der eigentliche Kalender zu finden. Da gibt es eine doppelte Datenhaltung, weil es sonst nicht wie in relationalen Datenbanken möglich ist, nach mehr als einer Eigenschaft zu indizieren. Es gibt nun die HashMap userdaten, mit der man über ein Datum das entsprechenden KalenderEntry bekommt. Dann gibt es aber auch noch eine zweite HashMap id_datavector, mit Hilfe derer man über die ID eines KalenderEntrys einen Vector mit allen Daten des entsprechenden Termins bekommt (formatiert als JJJJMMTT). Die erste HashMap braucht man für die visuelle Darstellung des Kalenders und die zweite vor allem zur effizienten Ausführung von Änderungen.

Kalender

Die eigentliche Applikation befindet sich in der Klasse Kalender.

In der `init()`-Methode werden bereits mal einige grundlegende Aktivitäten und ein User Admin mit dem Passwort Admin angelegt, damit der Kalender überhaupt benutzbar ist.

`handleRequest()` besteht im Wesentlichen aus ineinander geschachtelten if-Statements. Es gibt zwei grosse Blöcke, in denen zu Beginn über den Content-Type herausgefunden wird, ob es sich um den Java-Client oder den Web-Client handelt. Innerhalb dieser Blöcke gibt es dann für jede mögliche Aktion wieder ein if-Statement, das über die clientAttributes abfragt, ob diese Aktion ausgeführt werden soll.

Weiter gibt es noch diverse Hilfsmethoden, um User oder Aktivitäten zu erstellen, für die Codierung von Farben der Aktivitäten, zur Serialisierung und Methoden, die gleich ganze HTML-Blöcke generieren.

5.3.3 Architektur auf der Clientseite

Die Kalenderapplikation auf Clientseite stellt nicht viel anderes zur Verfügung als das GUI und die Datendarstellung der in Phenix gespeicherten und auf Serverseite bereitgestellten Daten. Allerdings wird das Datenhandling soweit möglich bereits auf Clientseite gemacht, damit es nicht zu viele Interaktionen mit dem Server gibt und das ganze dadurch langsam wird. Beim Starten der Applikation werden nämlich alle User, Aktivitäten und Termine des aktuellen Monats vom Server geladen und lokal gespeichert. Wird nun beispielsweise ein User oder eine Aktivität gelöscht, so werden diese einerseits auf dem Server gelöscht und andererseits lokal vom Client selbständig entfernt und die Daten neu dargestellt. Nur wenn zum sinnvollen Handling einer Useraktion zwingend Daten vom Server benötigt werden, werden diese zum Client übertragen. Beispielsweise: beim Eintragen eines neuen Termins, welcher über die Monatsgrenze hinweggeht, kann der Client nicht selbst wissen, ob auch das Enddatum noch nicht besetzt ist. In diesem Fall wird zuerst der Server angefragt, ob der Termin eingetragen werden kann. Falls ja, wird der Termin automatisch eingetragen, falls nein, wird der User gefragt, ob er den bestehenden Termin überschreiben möchte. Falls er dies möchte, wird der Termin „getrennt“ auf Server und auf Client eingetragen, so dass keine unnötigen Verbindungen zwischen Client und Server aufgebaut werden müssen.

Swing-/AWT-Komponenten und Standardklassen

- `KalenderClient.java`
- `MainFrame.java`
- `MenuBar.java`
- `Skin.java`
- `StatusPanel.java`

Diese Standardklassen sind grösstenteils Klassen, die zu jedem Projekt gehören. Sie instanzieren einige andere Klassen, stellen die Menu- und Statusleiste bereit und konfi-

gurieren die Look-and-Feel-Komponente.

`MonthNames.java` stellt als einzige Methode eine Konvertierung von einer Monatszahl in den enstprechenden Namen zur Verfügung. `Util.java` stellt einige allgemeine Funktionen zur Verfügung.

`ObjectPool.java` ist eine Klasse, die dazu dient, Instanzierungen von anderen Klassen entgegenzunehmen und aufzubewahren. Grund für diese Klasse war, dass jeweils auf dieselbe Instanz einer Klasse von vielen verschiedenen Klassen zugegriffen werden muss. Zur Lösung dieses Problem gibt es noch die beiden folgenden anderen Varianten, welche aber nicht so sauber erscheinen:

- Die entsprechende Klasse wird in der Hauptklasse instanziert und die Instanz als Referenz durch die ganze Applikation durchgeangelt.
- Die Methode in der entsprechenden Klasse wird als **`static`** deklariert.

Datenhaltung

Dieser Teil folgt einem recht streng hierarchischen Prinzip:

`Month.java`: Diese Klasse hält einen `Vector`, der für jeden User im System eine Instanz der Klasse 'UserMonth' enthält. Mit dem Zugriff auf diese Klasse hat man alle Informationen über den aktuellen Monat.

`UserMonth.java`: Auch diese Klasse enthält einen `Vector`. In diesem wird für jeden einzelnen Tag des aktuellen Monats eine Instanz der Klasse 'UserDay' abgelegt. Diese Klasse beinhaltet somit alle Informationen zum aktuellen Monat eines Users.

`UserDay.java`: In dieser Klasse werden alle relevanten Informationen zu einem entsprechenden Tag, welcher genau einem User, Monat und Jahr zugeordnet ist, abgelegt.

Konkret wird in dieser Klasse folgendes gespeichert:

```
// ID des Users, dem dieser UserDay zugeordnet ist:
private int iUserID;

// Tag im Monat, dem der UserDay zugeordnet ist:
private int iDay;

// Monat im Jahr, dem der UserDay zugeordnet ist:
private int iMonth;

// Jahr, dem der UserDay zugeordnet ist:
private int iYear;

// ID der Aktivitaet. 0, falls freier Termin:
private int iActivityID;

// ID des Termins. 0, falls freier Termin:
private long lID;

// Beschreibung des Termins. Leer, falls freier Termin:
private String strDescription = new String();
```

Action Listener

`ButtonActionListener.java`: In dieser grössten und kompliziertesten Klasse wird das Eintragen von neuen Terminen und das ganze dazugehörige Handling gemacht. Hier werden die Useraktionen abgefangen, welche durch die Klasse `DialogBox.java` ausgelöst werden. Dazu gehört das Handling einer Vielzahl verschiedener Möglichkeiten, welche hier nicht alle besprochen werden können.

Brücke zum Serverteil

`Connection.java`: Diese Klasse behandelt die Kommunikation mit dem Serverteil. Alle Daten von und zum Server werden dabei in eine HashMap verpackt. Die Keys der HashMap wurden von uns definiert. Somit bildet diese Kommunikation ein eigenes Protokoll.

Die Kommunikation mit dem Server läuft folgendermassen ab:

1. Client ruft Methode in Connection.java auf.
2. Methode in Connection baut HashMap zusammen.
3. Die Send-Methode in Connection.java fragt beim Gateway um die IP eines Calipso-Servers.
4. Die Verbindung zum Calipso-Server wird aufgebaut und die HashMap zum Serverteil übertragen.
5. Die Antwort-HashMap wird empfangen.
6. Ein Flag in der Antwort-HashMap gibt an, ob auf der Serverseite eine Exception aufgetreten ist,
7. Falls ja, werfe eine `ConnectionException`.
8. Falls nein, lies die HashMap aus und übergib die Daten dem Client.

PopUp-Fenster

Zu diesem Teil gehören folgende Klassen

- `AddUser.java`
- `AddActivity.java`
- `ChangeActivity.java`
- `ChangeUser.java`
- `ChangeUserPassword.java`
- `DeleteActivity.java`
- `DeleteUser.java`

Diese kleinen und sehr unabhängigen Klassen stellen das GUI zum Hinzufügen, Verändern und Löschen von Usern und Aktivitäten zur Verfügung. Nach der Useraktion werden die entsprechenden Änderungen sofort dem Server mitgeteilt sowie lokal ausgeführt und wenn nötig die Datendarstellung aktualisiert.

Users, Aktivitäten und die Kalendereinträge

`Activity.java`, `User.java` und `KalenderEntry.java` bilden die drei gemeinsamen Klassen mit dem Serverteil der Applikation. Dies sind die Objekte, welche auf Serverseite serialisiert und zum Client geschickt werden, welcher sie deserialisiert und verarbeitet.

Die Klassen `Activities.java` und `Users.java` speichern die Aktivitäten und die User lokal, damit nicht bei jeder Useraktion der Server konsultiert werden muss. Dabei wird bei beiden jeweils eine HashMap mit `User.java` beziehungsweise `Activity.java` als Objekt und der ID als Key verwaltet. Zusätzlich wird auch ein Vector verwaltet, welcher die Namen der Objekte zur Verfügung stellt. Diese werden z.B. beim Auffüllen von `JcomboBox`-Objekten benötigt. Weiter stellen beide Klassen Hilfsmethoden zur Verfügung wie z.B. die Rückgabe der Farbe einer Aktivität mit Namen x.

Externe Komponenten

`jbcl.jar`: Dieses Archiv stellt den XY-LayoutManager zur Verfügung. Dieser ermöglicht das pixelgenaue Platzieren von Swing- oder AWT-Komponenten. Dieses Package wurde von Borland entwickelt und wird mit dem JBuilder vertrieben. Die Lizenz lässt es zu, dieses Package selbstentwickelten Produkten hinzuzufügen und mit ihnen zu vertreiben.

`alloy.jar`: Dieses Archiv stellt die Look-and-Feel Komponente zur Verfügung. Das Package selbst kann zu Testzwecken vom Internet gratis heruntergeladen werden. Für eine ständige Nutzung muss jedoch ein Key gekauft werden, welcher in den eigenen Code eingebettet wird. Dieser Key wurde von mir legal erworben. Da diese Lizenz ausdrücklich eine royalty-free-distribution erlaubt, darf das Package beliebig mit dem Kalender verbreitet werden. Allerdings ist es nicht erlaubt, den Key aus dem Source-Code des Kalenders heraus zu kopieren und für andere Applikationen zu benutzen.

Beide Archive werden bei der Ausführung des KalenderClients benötigt.

5.3.4 Web-Frontend

Das Web-Interface zum Kalender ist mehr eine Demonstration, dass es funktioniert, als eine voll funktionsfähige Alternative zum Java-Client. Dazu fehlen noch wesentliche Sicherheitselemente wie der Login. Daher gibt es im Quellcode des Kalenders die Möglichkeit, mittels der boolean-Variable htmlAllowed festzulegen, ob ein Zugriff über das Web-Interface überhaupt möglich sein soll.

Das Web-Interface bietet folgende Funktionalitäten: User erstellen, Aktivitäten erstellen, Termine erstellen und löschen, Blättern von Monat zu Monat, sowie eine Downloadmöglichkeit des Java-Clients.

Die Details zu den einzelnen Terminen eines Monats befinden sich alle im Quellcode und werden über JavaScript abgerufen. Datentransfer zwischen Browser und Server findet also nur beim Laden eines neuen Monats oder beim Ausführen einer Aktion statt.

5.3.5 Mögliche Weiterentwicklungen und Verbesserungen

Redesign

Die ganze Organisation der Termine und User könnte man bestimmt noch verbessern. Zum Beispiel würde wohl der Einsatz einer relationalen Datenbank vieles vereinfachen, was aber im CPP-System vorläufig nicht möglich ist. Man könnte dann die User in einer Tabelle ablegen und die Termine in einer anderen, verknüpft durch User-Ids. Die Termine über mehrere Tage könnten dann mit Start- und Enddatum eingetragen und einfach wieder abgefragt werden.

Auftrennen von Kalenderlogik, Java-Client-Server und Webkalender

Bis jetzt kann man den Kalender nur mit dem Web-Interface und dem Java-Interface im Paket haben. Es wäre noch schön, wenn man das trennen könnte und dann je nach Bedarf Interface-Module installieren könnte. Man könnte sich da auch noch andere Interfaces wie beispielsweise WAP oder sogar ein Voice-Interface mittels VoiceXML vorstellen.

Ausbau des Web-Interfaces

Das Web-Interface ist noch ausbaufähig, in einem ersten Schritt müsste man wohl mal die Funktionalitäten des Java-Clients implementieren. Dann wären aber auch noch beliebige weitere Features wie zum Beispiel Gruppeneintragungen, Benachrichtigungen oder sich automatisch widerholdende Termine denkbar.

5.4 Beispielanwendung Webserver

Der Webserver bietet einfach die grundlegenden Funktionen, die man von einem Webserver erwartet. Man kann sich mit dem Admin-Tool eine eigene verschachtelte Ordnerstruktur erstellen und in beliebige Ordner beliebige Dateien hochladen, die dann über einen Webbrowser wieder abgerufen werden können.

5.4.1 Architektur

Data

Ein Data-Objekt beinhaltet eine Datei. Die eigentlichen Daten werden als byte-Array gespeichert und daneben wird lediglich noch der Mime-Type der Datei als String abgespeichert.

Filesystem

Das Filesystem verwaltet die gesamte Ordnerstruktur und besitzt auch Verweise zu den jeweiligen Dateien, die unabhängig vom Filesystem in Phenix gespeichert werden.

Die Dateien haben als Key einfach den gesamten Pfad inklusive Dateiname.

5.4.2 WebAdmin

Das Admin-Tool ist nach einer Passwortabfrage über ein Web-Interface bedienbar und bietet Zugang zu allen Funktionalitäten des Webservers. Man kann Ordner erstellen, Dateien hochladen und beides auch wieder löschen oder umbenennen. Natürlich kann man da auch durch die Ordner navigieren und das Admin-Passwort ändern.

5.4.3 Mögliche Weiterentwicklungen

Graphisches

Die Oberfläche des Admin-Tools ist relativ rudimentär, da liesse sich mit etwas Make-Up sicher noch mehr draus machen.

WikiWeb

Man könnte die Möglichkeit anbieten, editierbare Webseiten zur Verfügung zu stellen im Stil eines Wiki.

Dynamische Webseiten

Dynamische Webseiten wären sicher eine aufwändigere Weiterentwicklung. Man könnte vielleicht JSP einbinden, da das System sowieso schon auf Java läuft, oder ein Subset von PHP-Funktionen anbieten.

Multiple Uploads

Im Moment kann man nur einzelne Dateien hinaufladen. Als Weiterentwicklung wäre das Hinaufladen von Zip-Dateien, das automatische Entpacken, Erstellen der Ordnerstruktur des Zips im Filesystem und Speichern aller Dateien in Phenix denkbar.

5.5 Persönlicher Bericht Till Kleisli

Ich habe mich eigentlich primär für das Lab eingeschrieben, weil es ein obligatorischer Teil des Master-Programms Distributed Systems ist. Zusätzlich habe ich mir gedacht, könne es sicher nicht schaden, wenn man als Informatiker etwas Erfahrung im Programmieren allgemein, aber auch in der Arbeit an einem grösseren Projekt hat.

Zu Beginn habe ich mich vor allem auf meine, zugegeben nicht grandiose, Erfahrung gestützt und mir gedacht, dass es bis jetzt ja auch immer irgendwie ging. Erst mit der Zeit habe ich dann realisiert, dass das doch eine Grössenordnung grösser ist als das, was ich bis jetzt so gemacht habe, und dass ich viel weiter komme, wenn ich auch die zum Teil etwas grössere Erfahrung der Anderen miteinbeziehe. Ich bin jemand, der immer versucht mit bereits Bekanntem und Vorhandenem möglichst weit zu kommen.

Daher konnte ich mich erst nach einer gewissen Zeit dazu „überwinden“ den TextPad aufzugeben und Eclipse als Entwicklungsumgebung zu installieren. Ich musste dann aber auch zugeben, dass Eclipse vieles ungemein erleichtert und einem auch einen sehr guten Überblick über eine Klasse oder das ganze Projekt liefert.

Die Entscheidung, mich innerhalb des Projekts um die Anwendungen zu kümmern, würde ich zwar nicht als Fehler bezeichnen, aber ich würde mich im Nachhinein vielleicht doch eher für etwas Anderes entscheiden. Es wäre eine gute Gelegenheit gewesen, mich in einem Gebiet weiter zu vertiefen, in dem ich bis jetzt nicht viel gemacht habe, seien das Netzwerktechnologien oder auch Datenbanken. Vielleicht war meine Entscheidung aber auch gar nicht so schlecht, so konnte ich mich auf das Behandeln oben genannter Probleme konzentrieren.

Das eigentliche Entwickeln der Applikationen, namentlich des Kalenders, war dann nicht ganz einfach. Zum Teil war es, wie wenn man ein Haus auf Sand bauen müsste. Das ganze System veränderte sich kontinuierlich, was trotz nach relativ kurzer Einarbeitungszeit der beteiligten Gruppen festgelegten Interfaces ein Problem darstellte. Beim Testen zum Beispiel konnte man sich nicht immer sicher sein, dass alle Komponenten wunschgemäss funktionieren und zusammenarbeiten. Aber auch sonst war das Testen vor allem zu Beginn ein Problem, weil eine Applikation von Phenix und im Falle einer Web-Applikation auch von Calipso abhängig ist. Nachträglich hätte man die „human resources“ zu Beginn vielleicht auf die Entwicklung lauffähiger Prototypen von Phenix und Calipso konzentrieren sollen und dann mit stabilerem Wissen an die Applikationsentwicklung gehen sollen.

Trotz aller Probleme und negativer Vorzeichen möchte ich ein positives Fazit ziehen. Es war faszinierend, wie sich aus einer Ansammlung von Ideen und später auch Klassen und Interfaces das ganze CPP-System entwickelte und zum Schluss auch noch funktionierte!

5.6 Persönlicher Bericht Lukas Oertle

Eine Arbeit in einer (grossen) Gruppe tut jedem Studenten gut. Ich habe sehr fest von der engen Zusammenarbeit profitiert. In meiner bisherigen Berufstätigkeit arbeitete ich zwar auch in Teams, die Aufgaben waren aber sehr viel klarer voneinander abgegrenzt, so dass kaum zwischenmenschliche Interaktionen nötig waren. Zudem ist das ganze von einem Chef geleitet. Bei unserem Lab mussten wir die Zusammenarbeit selbst organisieren, die Schnittstellen waren nicht von vornherein klar. Dies erforderte eine Zusammenarbeit, welche am Anfang nicht so stark ausgeprägt war, am Schluss des Labs jedoch sehr gut klappte.

Allerdings finde ich, dass durch die beiden Sitzungen manchmal ein gewisser Overhead entstand. Zu Projektbeginn waren zwei Sitzungen sicherlich gut, zum Ende jedoch hätte es wohl mehr genützt, allfällige Probleme bilateral mit den beteiligten Parteien (z.B. via Handy oder IRC) anstatt an einer Sitzung zu diskutieren.

Die Anforderungen an das Produkt des Labs waren ziemlich offen. Dadurch lernte ich, dass eine peinlich genaue Planung des Projekts vor Projektstart enorm wichtig ist.

Teilweise habe ich das beim Kalender nicht gemacht. Dies ergab später beim Ausbau Probleme. Ich habe gelernt, dass die Architektur und der Einsatz von Entwurfsmustern im Voraus geplant werden muss, sonst kann es während der Realisation zu grösseren Umbauten im Code kommen.