Eine zustandsbasierte Experimentsteuerung über einen fehleranfälligen Kommunikationskanal

Bachelorarbeit

von

Tobias Amft

aus

Tönisvorst

vorgelegt am

Lehrstuhl für Rechnernetze und Kommunikationssysteme
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

Juni 2012

Betreuer:
Norbert Goebel M. Sc.

Abstract

Auf dem Gebiet der Fahrzeug zu Fahrzeug Kommunikation konkurriert 802.11p (WLAN) mit infrastrukturiertem Mobilfunk. Reale Experimente können Aufschluss darüber geben, welche der beiden Technologien bei bestimmten Anwendungen im Vorteil ist. Im Rahmen dieser Bachelorarbeit wurde deshalb eine zustandsbasierte Experimentsteuerung über einen fehleranfälligen Kanal entwickelt und implementiert, die hauptsächlich zur Durchführung von Mobilfunkmessungen mit mehreren Messgeräten eingesetzt werden soll. Dabei toleriert das Steuerprotokoll die typischen Eigenschaften von Mobilfunknetzen wie beispielsweise häufigen Paketverlust.

Das durchzuführende Experiment wird von einem Serverprogramm an die teilnehmenden Clients verteilt, ohne die Beaufsichtigung eines Benutzers zu verlangen. Die Clients bestätigen den Empfang des Experiments und führen es gemeinsam zu einer angegebenen Startzeit aus. Weil die Kommunikation zwischen dem Server und den Clients bei der Verwendung von mobilen Messgeräten in der Regel über den zu vermessenden Kanal erfolgt, wird das Experiment selbst in Abschnitte unterteilt. Nur zwischen diesen Abschnitten ist es den Teilnehmern erlaubt, Statusnachrichten auszutauschen, damit die eigentliche Messung so wenig wie möglich beeinflusst wird. Um den Ablauf eines Experiments zu beschreiben, erstellt der Benutzer eine XML Datei, in der er vorgibt, welcher Befehl zu welchem Zeitpunkt von der Experimentsteuerung ausgeführt werden soll. Dabei kann der Benutzer beliebige Messprogramme, die er verwenden möchte, in das Experiment einbinden und somit die zu sammelnden Informationen selbst festlegen.

Danksagung

Zuerst möchte ich mich bei Prof. Dr. Martin Mauve für interessante Vorlesungen und seinen freundlichen Umgang mit Studenten und Mitarbeitern bedanken.

Ich bedanke mich an dieser Stelle außerdem herzlich bei meinem Betreuer Norbert Goebel für zahlreiche Gespräche, Korrekturen, Tips und Tricks.

Ich danke Angkhan Choeichuen für sein Verständnis und seine Geduld. Ich danke Andrej Henkel für seine wachsende Ungeduld und seine Art andere Menschen zu motivieren.

Meiner Mutter Antje, meinem Vater Dieter und meiner Schwester Maren möchte ich besonders danken, weil ich mich, auch in schwierigen Lebenslagen, immer auf sie verlassen kann.

Ein weiterer großer Dank gehört meiner Frau Vanessa, die mich unterstützt und ergänzt, aber auch zum Lachen bringt. Ihrer Familie danke ich für eine herzliche Aufnahme meiner Person.

Zu guter Letzt möchte ich mich bei allen hier nicht aufgezählten Freunden und Familienmitgliedern bedanken, die mich so akzeptieren, wie ich bin.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1.

Einleitung

1.1. Motivation

Kommunikation und Mobilität sind Errungenschaften, die aus der Geschichte der Menschheit nicht mehr wegzudenken sind. Aus diesem Grund gehört die Automobilindustrie weltweit nach wie vor zu den bedeutendsten Industriezweigen. Das Automobil wurde schnell zum beliebtesten Fortbewegungsmittel im Straßenverkehr, sodass die Produktion und Entwicklung von Personenfahrzeugen in den letzten Jahren massiv vorangetrieben wurde. Neben wachsender Leistung der eigenen Maschinen richtet sich die Aufmerksamkeit der Hersteller zunehmend auf die Bedürfnisse der Kunden. Effizienz, Komfort und Sicherheit gehören inzwischen zu den wichtigsten Qualitätsmerkmalen moderner Fahrzeuge.

Aktuelle Sicherheitskonzepte wie Airbacks, Antiblockiersysteme (ABS) oder elektronische Stabilitätsprogramme (ESP), um nur einige zu nennen, sollen die Verkehrsteilnehmer vor Unfällen und den daraus resultierenden körperlichen Schäden schützen. Auch die Weiterentwicklung der grundlegenden Fahrzeugtechnik (z.B. Bremsen, Reifen, etc.) kann das Maß an Sicherheit im Straßenverkehr erhöhen. Das deutsche Bundesamt für Statistik bestätigt einen tendenziellen Rückgang von verunglückten Personen im Straßenverkehr innerhalb der letzten zwanzig Jahre [Sta12a]. Erstmals im Jahr 2011 ist die Zahl der verunglückten Personen deutlich angestiegen. Während 1970 noch 10 Verkehrstote je 10000 Fahrzeuge gezählt wurden, ist die Zahl der Verkehrstoten 2011, trotz des höheren Kraftfahrzeugbestandes, auf weniger als

eine Person je 10000 Fahrzeuge gesunken [Sta12b]. Diese traditionellen Sicherheitskonzepte haben die Eigenschaft gemeinsam, dass sie erst dann reagieren, wenn eine Gefahr unmittelbar bevorsteht und den Verkehrsteilnehmer bedroht.

An dieser Stelle kann das Stichwort *Kommunikation* neue Wege aufzeigen. Gegenstand der Forschung ist seit einigen Jahren die Ausstattung von Fahrzeugen mit drahtloser Kommunikation, damit Informationen zwischen diesen ausgetauscht werden können. Auf diese Weise können übermittelte Bewegungsdaten verschiedener Fahrzeuge helfen, kritische Situationen im Straßenverkehr zu erkennen und sie zu vermeiden. Nicht nur Gefahren sondern auch Stausituationen können effizient behandelt werden, wodurch wiederum Kraftstoffverbrauch und Schadstoffemissionen gesenkt werden können.

Inzwischen gibt es auf diesem Gebiet der Forschung einige Organisationen und Projekte, die aktiv an der Verbesserung von Fahrzeug zu Fahrzeug (Car to Car) und Fahrzeug zu Umgebung (Car to X) Kommunikation beteiligt sind. Zu diesen gehören beispielsweise das *Car 2 Car Communication Consortium (C2C-CC)* [CC12], das *Sichere Intelligente Mobilität Testfeld Deutschland(SIM-TD)* [ST12] und das vom Bundesministerium für Bildung und Forschung geförderte Projekt *Network on Wheels (NOW)* [NoW12].

Die Fahrzeug zu Fahrzeug Kommunikation wird unter anderem mit Fahrzeug Ad hoc Netzwerken (engl. Vehicular Ad Hoc Network, VANET) realisiert, die eine Form der Mobilen Ad hoc Netzwerke (engl. Mobile Ad Hoc Network, MANET) sind. Diese bestehen aus sich selbst organisierenden und über drahtlose Kommunikationsschnittstellen miteinander verbundenen mobilen Geräten, ohne dabei jegliche Infrastruktur zu benötigen. Bei der eingesetzten Technologie zur drahtlosen Kommunikation zwischen Fahrzeugen stehen die mit IEEE 802.11p (WLAN) ausgestatteten Ad hoc Netzwerke in Konkurrenz mit dem klassischen, infrastrukturbasierten Mobilfunk (z.B. GSM, GPRS oder UMTS).

Der Lehrstuhl für Rechnernetze und Kommunikationssysteme der Universität Düsseldorf untersucht zu diesem Zweck, ob bestimmte Fahrzeug zu Fahrzeug Anwendungen über Mobilfunk (z.B. GSM, GPRS, UMTS, etc.) realisierbar sind. Dazu wird ein Verfahren zur Simulation von Mobilfunknetzwerken mit Hilfe von realen Messungen entwickelt. Aufgrund mangelnder Aussagen vieler Mobilfunkanbieter über die eigenen Netze und deren charakteristischen Merkmale, sind eigenständige Vermessungen notwendig. In einer Vorangegangenen

Arbeit wurden Mobilfunkzellen mit einem einzigen Messgerät vermessen. Für die Forschung ist es jedoch wichtig, kontrollierte und reproduzierbare Messexperimente mit mehreren Messgeräten in einer Mobilfunkzelle durchzuführen. Benötigt wird dazu eine Experimentsteuerung, die ohne Nutzeraktivität auskommt.

1.2. Struktur der Arbeit

Im Rahmen dieser Bachelorarbeit wurde *SEC* (*Stateful Experiment Control*), eine zustandsbasierte Experimentsteuerung über fehlerbehaftete Kommunikationskanäle entworfen und implementiert. Besonders zu beachten war dabei, dass die Steuerung des Experiments die eigentlichen Messungen so wenig wie möglich beeinflusst. Außerdem soll die Experimentsteuerung mit typischen Eigenschaften von Mobilfunk, wie hohen Latenzen oder häufigem Paketverlust, umgehen können.

In Kapitel 2 werden zwei Arbeiten vorgestellt, die sich bereits vor dieser Bachelorarbeit mit dem Entwurf und der Implementierung eigener Experimentsteuerungen beschäftigt haben. Erfolgreiche Konzepte dieser Arbeiten werden in *SEC* aufgegriffen oder erweitert.

Kapitel 3 beschreibt die grundlegenden Begriffe, Module und Funktionen der entwickelten Experimentsteuerung. Weiterhin wird die Durchführung eigener Messungen anhand eines Beispiels erklärt.

Anschließend werden im 4. Kapitel ausgewählte Module der Experimentsteuerung und deren Implementierung ausführlich dargestellt. Dieses Kapitel erklärt unter anderem die Nutzung von Threads und Netzwerkfunktionen in *SEC*.

Im 5. Kapitel wird die Entwicklung von *SEC* zusammengefasst und ein abschließender Ausblick auf mögliche Erweiterungen der Experimentsteuerung gegeben.

Kapitel 2.

Verwandte Arbeiten

Typischerweise ist zum Verwalten eines Experiments mehr als nur der ausführende Teil notwendig. Benötigt wird immer eine Instanz, die ein Experiment durchführen möchte, es verteilt, beobachtet und bewertet. Wichtige Fragen sind hierbei:

- Welche Rollen spielen die einzelnen Teilnehmer eines Experiments?

- Wie werden Aktionen ausgeführt und anderen Teilnehmern mitgeteilt?

2.1. Experiment Control

Ein bewährtes Programm zur Steuerung von realen Experimenten in drahtlosen Multihop Netzwerken ist das am Lehrstuhl für Rechnernetze und Kommunikationssysteme der Universität Düsseldorf entwickelte Toolkit *EXC* [KOM08].

Experimente in *EXC* werden von einem zentralen Knoten, dem *Monitor*, aus gesteuert und überwacht. Das eigentliche Experiment läuft auf einfachen Knoten, die ein vordefiniertes Kontrollprotokoll abarbeiten. Damit ein Experiment während einer Messung nicht vom Toolkit beeinflusst werden kann, zum Beispiel durch das Senden oder Empfangen von Nachrichten, nutzt *EXC* das Konzept der *semiautomatischen Kontrolle*. Ein komplexes Experiment wird dabei in kleinere Abschnitte, den sogenannten *Runs*, portioniert. Zwischen den *Runs* kann

das Experiment beeinflusst werden, ohne den eigentlichen Ablauf zu stören. In den meisten Fällen wird ein Knoten sein Kontrollprotokoll abarbeiten können, ohne auf die Eingabe eines Benutzers warten zu müssen. Falls unerwartete Fehler auftreten, kann der *Monitor* das Experiment weiterhin kontrollieren, ohne vorherige Ergebnisse zu beeinflussen.

EXC ist in erster Linie für Multihop Netzwerke entwickelt worden, die mit Hilfe von IEEE 802.11 (WLAN) kommunizieren. *EXC* setzt deshalb eine vergleichsweise niedrige Distanz zwischen seinen Teilnehmern voraus, damit eine Kommunikation unter diesen erst möglich ist. In einem Mobilfunknetz jedoch müssen Teilnehmer ihre Nachrichten auch über große Distanzen hinweg austauschen können. *EXC* ist auch deshalb nicht als Experimentsteuerung für Mobilfunkmessungen einsetzbar, da die Zeitsynchronisation der Clients WLAN Beacons verwendet. Diese sind aufgrund der kurzen Reichweite von WLAN Geräten nicht über größere Distanzen einsetzbar. Da *EXC* außerdem nicht mit Rücksicht auf Mobilfunkmessungen entwickelt wurde gibt es keine Garantie, wie *EXC* auf typische Eigenschaften von Mobilfunk, wie hohe Latenzen oder häufigen Paketverlust, reagiert. Interessant für eine Experimentsteuerung über einen fehlerbehafteten Kommunikationskanal (und damit auch für Mobilfunkmessungen) ist jedoch das Konzept des *semiautomatischen Experiments*. Daher wird die Einteilung eines Experiments in *Runs* (siehe Kapitel 3.1.5) in *SEC* übernommen, um die Messungen der einzelnen Teilnehmer so wenig wie möglich, durch Kommunikation mit anderen Teilnehmern, zu beeinflussen.

2.2. Eine In-Band-Experimentsteuerung zur Durchführung von Mobilfunkmessungen

Eine vorangegangene Bachelorarbeit beschäftigte sich bereits mit der Planung und Implementierung einer Experimentsteuerung für Mobilfunkmessungen. Ein erfolgreiches Konzept dieser Arbeit ist die Durchführung eines Experiments unter Einsatz eines serverseitigen und eines clientseitigen Zustandsautomaten. Das Clientprogramm bezieht dabei seine Aufgabe von einem Serverprogramm und kommuniziert mit diesem während der Durchführung des Experiments.

Die *In-Band-Experimentsteuerung* [Zag11] zur Durchführung von Mobilfunkmessungen wurde in Java implementiert und verlangt daher die Installation einer Java Laufzeitumgebung auf eingesetzten Messclients. Diese Laufzeitumgebung kann jedoch aus Performancegründen hinderlich sein, wenn Experimente auf Messgeräten mit vergleichsweise schwacher Hardware ausgeführt werden sollen. *SEC* wurde aus diesem Grund in C++ implementiert, um zeitkritische Aufgaben möglichst direkt und hardwarenah auszuführen.

Sowohl in den beiden vorangegangenen Arbeiten, als auch in *SEC*, werden auszuführende Experimente in XML Dateien beschrieben. Somit können Befehle und Abläufe einfach strukturiert werden. Der Benutzer kann zudem auf beliebige XML Editoren zurückgreifen, die ihn auf die Gültigkeit der XML Dokumente hinweisen.

Kapitel 3.

Design

In dieser Bachelorarbeit wird *SEC (Stateful Experiment Control)*, eine zustandsbasierte Experimentsteuerung über einen fehlerbehafteten Kommunikationskanal vorgestellt. *SEC* soll die Durchführung von Mobilfunkmessungen mit mehreren Teilnehmern vereinfachen. Eine zentrale Einheit soll dabei die Verteilung eines Experiments an die übrigen Teilnehmer durchführen und kontrollieren. Obwohl *SEC* hauptsächlich für Mobilfunkmessungen entwickelt wurde, können die Teilnehmer eines Experiments im weitesten Sinne mobile Geräte sein, die über eine beliebige IPv4 basierte Kommunikationsschnittstelle verfügen.

SEC wird auf kleinen mobilen Messgeräten getestet, die mit gerade einmal 256 MB Arbeitsspeicher und einem 500 MHz Prozessor ausgestattet sind. Das Betriebssystem eines mobilen Messgeräts befindet sich auf einer CompactFlash Karte anstelle einer herkömmlichen Festplatte. Der clientseitige Teil von *SEC* wurde absichtlich ressourcenschonend implementiert, damit es auf der vergleichsweise schwachen Messhardware einsatzfähig ist. Aus diesem Grund wurde *SEC* in C++ implementiert, um den Overhead durch eine Laufzeitumgebung wie beim Einsatz von Java, oder durch eine Interpretersprache wie Python oder Ruby, zu verhindern.

3.1. Design im Überblick

3.1.1. Experiment

Im eigentlichen Sinne ist ein *Experiment* eine planmäßig angelegte Sammlung von Informationen zur Stützung einer Hypothese. In dieser Bachelorarbeit versteht man unter einem *Experiment* jedoch eher eine planmäßig durchgeführte Messung oder Beobachtung von Variablen, Daten, oder Zuständen unter dafür angelegten Rahmenbedingungen mit Hilfe von Computern zur Datenerfassung. Die Teilnehmer eines *Experiments* sind demnach Computer, die als Knoten in einem Netzwerk von Computern funktionieren.

3.1.2. Messsoftware / Messprogramm

Die *Messsoftware* oder das *Messprogramm* wird vom Benutzer selbst ausgesucht. Das *Messprogramm* legt fest, welche Art von Experiment durchgeführt werden soll und welche Informationen dabei gewonnen werden sollen. Beispielsweise könnte die *Messsoftware* so implementiert sein, dass sie Informationen (wie Latenz oder Bandbreite) über ein Mobilfunknetz sammelt. Die *Messprogramme*, die während eines Experiments benutzt werden sollen (In *SEC* können mehrere *Messprogramme* zur Verwendung angegeben werden), müssen sich während des Experiments auf dem Client befinden. In Kapitel 3.4.2 wird näher beschrieben, wie ein *Messprogramm* in ein Experiment eingebunden wird.

3.1.3. Aufgabe

Ein komplexes Experiment sollte unter mehreren Teilnehmern aufgeteilt werden können. Zum einen um fehlerhafte Messungen durch redundante Messungen mehrerer Teilnehmer ausgleichen zu können, zum anderen um das Experiment sinnvoll zu gliedern. Derjenige Teil, der aus der Gliederung eines Experiments hervorgeht und für einen Teilnehmer bestimmt ist, wird im Rahmen dieser Arbeit *Aufgabe* genannt. Jeder Client bezieht seine persönliche *Aufgabe* in einem Experiment vom Server (siehe Kapitel 3.5).

3.1.4. Befehl

Die kleinste Einheit einer Aufgabe ist ein *Befehl*. Dieser ist eine Anweisung an den Client, ein Stück Programm auszuführen. Dabei kann es sich um eine Anweisung an ein Messprogramm, das Messungen auf dem Client durchführt, oder um Befehle an das System des Clients handeln. In Kapitel 3.4.2 wird beschrieben, wie *Befehle* im XML Format dargestellt werden.

3.1.5. Run

Das Konzept der *Runs* in dieser Bachelorarbeit ist stark an das Konzept der *semiautomatischen Kontrolle* in *EXC* angelehnt (siehe Kapitel 2.1). Dabei werden Experimente in *EXC* in kleine Abschnitte (*Runs*) aufgeteilt und mittels eines Kommandos durch einen *Operator* (in der Regel der Beobachter des Experiments) gestartet. In *SEC* werden Experimente zwar auch in *Runs* unterteilt (siehe Abbildung 3.1), diese müssen aber nicht wie in *EXC* von einem *Operator* gestartet werden.

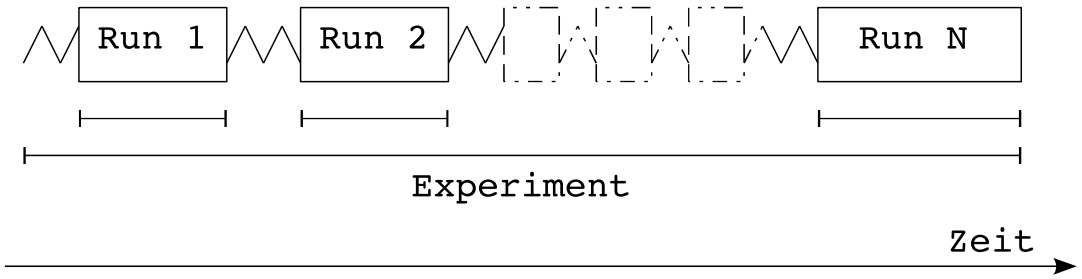

Abbildung 3.1.: Ein in *Runs* unterteiltes Experiment.

Die Startzeit jedes *Runs* wird vor dem Beginn eines Experiments festgelegt. Die Verwendung von *Runs* soll sowohl in *EXC*, als auch in *SEC* die Kommunikation zwischen dem Server und einem beliebigen Client auf den Zeitraum zwischen den *Runs* beschränken, damit das eigentliche Experiment so wenig wie möglich gestört wird. In *SEC* werden zwischen den *Runs* Statusnachrichten von den Clients an den Server gesendet, um diesem die eigene Präsenz mitzuteilen.

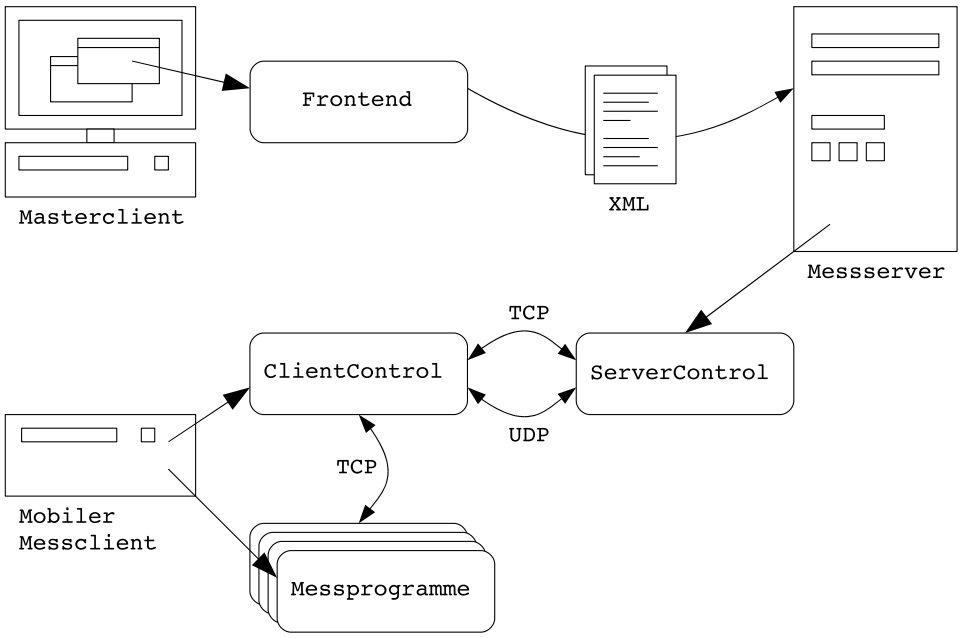

Abbildung 3.2.: *Stateful Experiment Control (SEC).*

3.1.6. Timefile

Die Aufgaben werden in einfachen Textdokumenten, den *Timefiles*, an die Clients gesendet. Dabei enthält jedes *Timefile* nur die Befehle, die ein bestimmter Client während des Experiments auszuführen hat. Jeder Client entnimmt seinem *Timefile* während des Experiments den nächsten Befehl und dessen Ausführungszeit. Der Server selbst benutzt ein eigenes *Timefile*, um die *Runs* (siehe Kapitel 3.1.5) während des Experiments überwachen zu können.

3.1.7. Masterclient

Derjenige Computer, auf dem das Programm *SEC-Frontend* benutzt wird, um mit dem Server zu kommunizieren, wird als *Masterclient* oder kurz *Master* bezeichnet. In der Regel wird der *Masterclient* vom Benutzer, der ein Experiment durchführen möchte, genutzt, um ein Experiment an den Server zu senden (siehe Kapitel 4.4). Selbstverständlich kann der *Masterclient* ebenso als Client in einem Experiment dienen, indem er das Programm *ClientControl* ausführt.

3.1.8. ClientControl

ClientControl ist das clientseitige Programm hinter *SEC*, welches hauptsächlich für die Ausführung einer Aufgabe während eines Experiments zuständig ist (siehe Abbildungen 3.2 und 3.3). Der grundlegende Aufbau des Programms wird in Kapitel 3.2 näher erläutert. *ClientControl* meldet sich beim Server an und wartet zunächst, bis ihm eine Aufgabe mitgeteilt wird. Wenn *ClientControl* die nötigen Messprogramme für eine Aufgabe starten kann und die tatsächliche Startzeit des Experiments vom Server emfangen hat, interpretiert *ClientControl* die Aufgabe des Clients. *ClientControl* reicht dabei unter anderem Anweisungen an die eigentlichen Messprogramme weiter.

3.1.9. ServerControl

Das serverseitige Programm *ServerControl* (siehe Abbildungen 3.2 und 3.8) nimmt Experimente vom Benutzer entgegen und unterteilt diese in Aufgaben an die Clients. *ServerControl* speichert die Verbindungen zu Clients, die sich beim Server anmelden, um auf diese für ein Experiment zurückgreifen zu können. *ServerControl* verteilt die Aufgaben eines Experiments an die Clients und teilt ihnen die entgültige Startzeit des Experiments mit. Während die Clients die Aufgaben des Experiments bearbeiten, kontrolliert *ServerControl* die Präsenz der Clients, um auf Ausfälle der Clients reagieren zu können. Die genaue Funktionsweise von *ServerControl* wird in Kapitel 3.3 beschrieben.

3.1.10. SEC-Frontend

Das Programm *SEC-Frontend* wird dem Benutzer in erster Linie bereitgestellt, damit dieser Dateien mit dem Server austauschen kann. Für eine beliebige Experimentdatei im XML Format können mit *SEC-Frontend* Konfigurationen durchgeführt werden. Diese werden mit der zugehörigen Experimentdatei an den Server gesendet. Ein Beispiel zur Benutzung des Programms wird in Kapitel 4.4 gegeben.

3.2. ClientControl

Die clientseitige Software hinter *SEC* ist ein endlicher Zustandsautomat mit sieben Zuständen (StateCC0 bis StateCC6), welcher in der Abbildung 3.3 dargestellt wird. Obwohl sich *ClientControl* weitestgehend selbst organisiert, werden einige Entscheidungen in diesem Automaten durch den Server beeinflusst. Die Einteilung des Zustandsautomaten wurde in dieser Arbeit stark an die Einteilung der *In-Band-Experimentsteuerung* aus [Zag11] angelehnt. Die Aufgaben der jeweiligen Zustände variieren jedoch leicht. In *SEC* kommunizieren Server und Client während der gesamten Vorbereitung auf ein Experiment über eine TCP Verbindung. Deshalb werden alle Nachrichten in der Reihenfolge von einem Client oder dem Server empfangen, wie sie vom Server oder einem Client gesendet wurden. Außerdem garantiert TCP, dass keine Nachricht bei der Übertragung verloren geht oder beschädigt wird. In *SEC* weiß der Server somit zu jeder Zeit (wenn eine TCP Verbindung besteht), in welchem Zustand sich der Client befindet. In [Zag11] wird nur die Experimentdatei per TCP an den Client übertragen. Die restliche Kommunikation während der Vorbereitungen auf ein Experiment findet bei der *In-Band-Experimentsteuerung* mit UDP statt. Ein großer Nachteil ist hierbei, dass der Client spezielle Statusnachrichten über seinen Zustand an den Server senden muss. Dabei ist allerdings nicht gewährleistet, dass diese Statusnachrichten in der richtigen Reihenfolge beim Server eintreffen.

3.2.1. Zustand StateCC0: Anmeldung

ClientControl versucht zu Beginn eine TCP Verbindung zum Server aufzubauen. Dazu wird dem Programm die IP Adresse des Servers und der Port, auf dem *ServerControl* lauscht, übergeben. Wenn dieser Versuch misslingt, wechselt *ClientControl* in den Zustand StateCC6 und versucht sich erneut beim Server anzumelden. Wenn der Client ordentlich beim Server angemeldet wurde, wechselt der Zustandsautomat selbstständig in den Zustand StateCC1. Anders als in [Zag11] wird der Client nicht vom Server aufgefordert in den nächsten Zustand zu wechseln. Nach der Anmeldung beim Server geht der Client in den Standbyzustand über, bis er vom Server für ein Experiment eingeteilt wird.

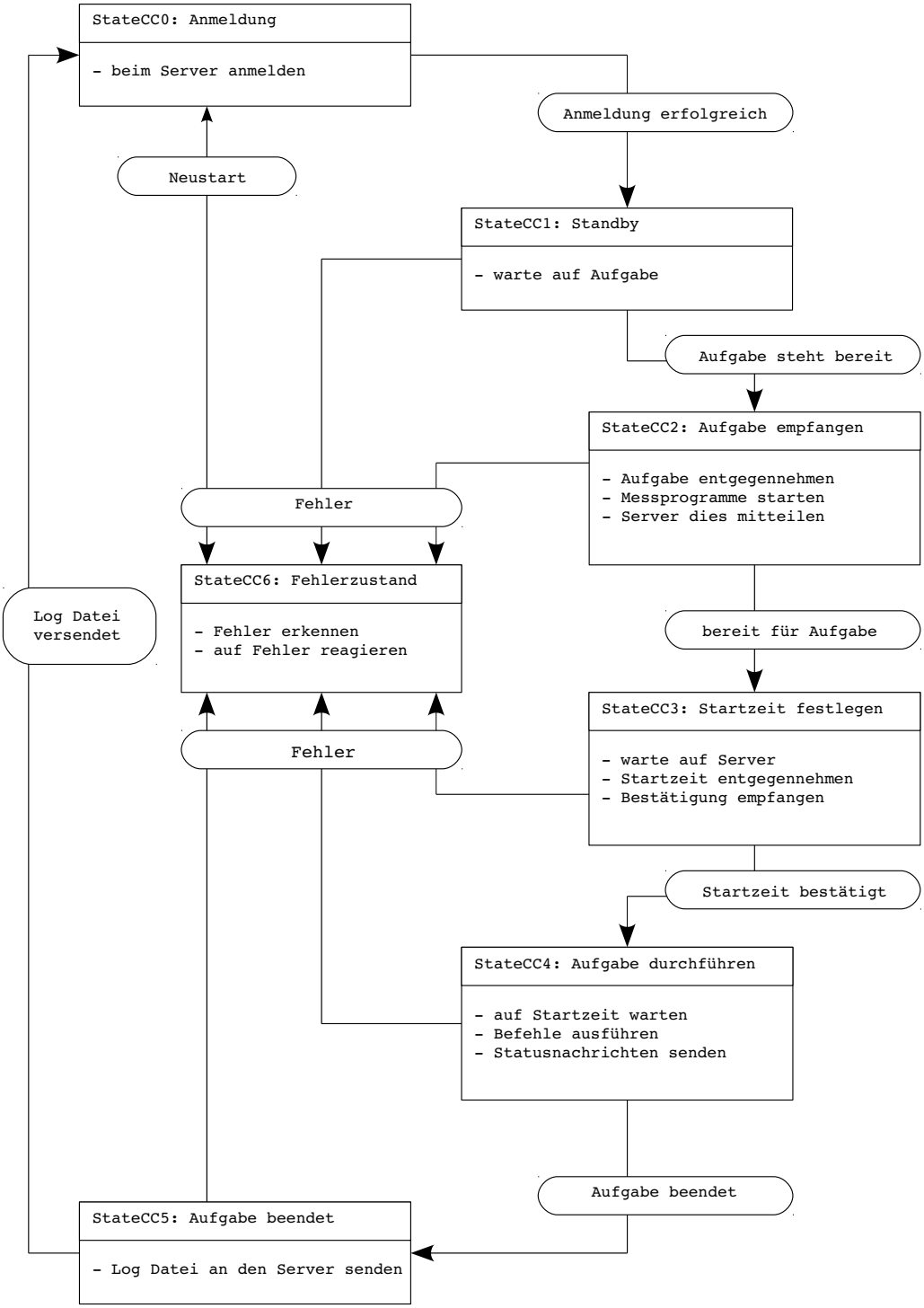

Abbildung 3.3.: *ClientControl* Zustandsautomat.

3.2.2. Zustand StateCC1: Standby

Im Standbyzustand wartet *ClientControl*, bis der Server ihm eine Aufgabe zusendet. Bevor der Client dazu in den Zustand StateCC2 wechselt, übermittelt der Server die ID (Identifikationsnummer) des Clients für das geplante Experiment an diesen. Zusätzlich wird dem Client der UDP Port mitgeteilt, auf dem der Server während des Experiments auf Statusnachrichten wartet. *ClientControl* wechselt bei fehlerhaften Nachrichten in den Fehlerzustand StateCC6 und beendet die Verbindung zum Server. Ist dies nicht der Fall, wechselt *ClientControl* in den Zustand StateCC2, um seine Aufgabe zu empfangen.

3.2.3. Zustand StateCC2: Aufgabe empfangen

ClientControl ist im Zustand StateCC2 bereit, seine Aufgabe für das Experiment entgegen zunehmen. Die Aufgabe besteht zum einen aus einer Konfigurationsdatei, in der Informationen zum Experiment gespeichert sind (siehe Kapitel 3.4.1). Zum anderen besteht die Aufgabe an den Client aus einem vordefinierten Ablauf von Befehlen, dem *Timefile* (siehe Kapitel 3.5). *ClientControl* startet in diesem Zustand alle benötigten Programme, die im *Timefile* als *Includes* angegeben sind. Um mit den externen Messprogrammen kommunizieren zu können, müssen diese ein Serverprogramm starten, sodass sich *ClientControl* per TCP über die Loopbackschnittstelle mit der benötigten Messsoftware verbinden kann. Wenn die Verbindung zu mindestens einem Messprogramm fehlschlägt, teilt *ClientControl* dies dem Server mit und geht in den Fehlerzustand StateCC6 über. Nach einiger Zeit wechselt *ClientControl* in den Zustand StateCC0, sodass sich der Client für andere Experimente zur Verfügung stellen kann. Wenn die Verbindungen zu allen Messprogrammen aufgebaut sind, teilt *ClientControl* dem Server mit, dass die Aufgabe erfolgreich entgegen genommen wurde und wechselt in den Zustand StateCC3. Die Abbildung 3.4 zeigt die Kommunikation, die zwischen dem Server und einem Client stattfindet, bis der Client eine Aufgabe erhalten hat.

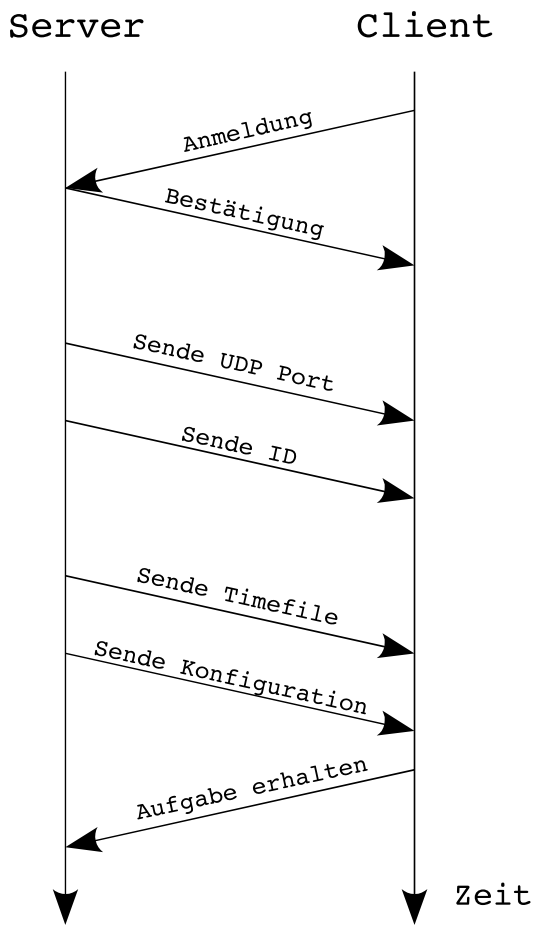

Abbildung 3.4.: Anmeldung eines Clients beim Server.

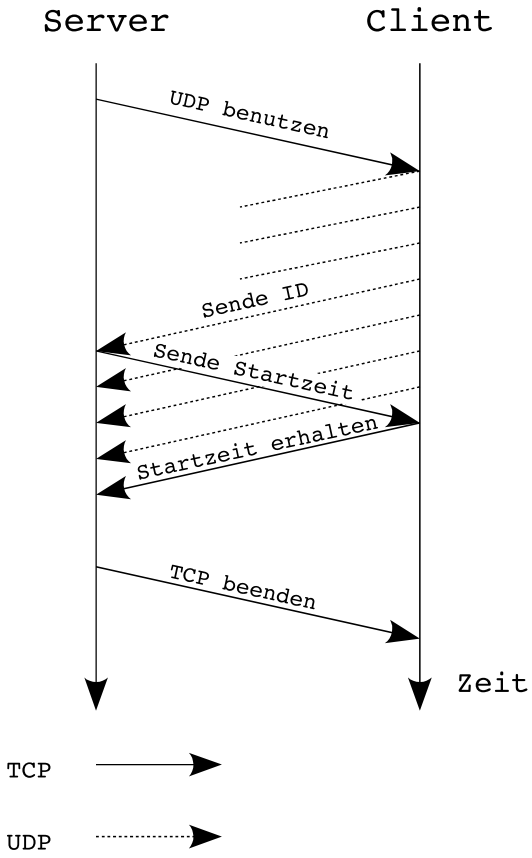

Abbildung 3.5.: Verteilung der Startzeit an die Clients.

3.2.4. Zustand StateCC3: Startzeit festlegen

Während *ClientControl* in den Zustand StateCC3 wechselt, kann der Server noch mit der Verteilung des Experiments beschäftigt sein. Zunächst fordert *ClientControl* einen UDP Socket beim Betriebssystem an, der zur Kommunikation während des Experiments verwendet werden soll. Um die UDP Kommunikation zu testen, wird der Client vom Server aufgefordert, seine ID periodisch an den Server zu senden (siehe Abbildung 3.5). Sobald *ClientControl* die voraussichtliche Startzeit über TCP empfängt, stellt er die periodische Übertragung der UDP Testpakete ein. Anschließend wartet *ClientControl* auf die Anweisung des Servers, seine TCP Verbindung zu schließen. Diese ist bei der Durchführung des Experiments nicht erwünscht, da sie (bei einer drahtlosen Kommunikation) die eigentliche Messung stören kann.

Sollte die TCP Verbindung zum Server schon vorher abbrechen oder geschlossen werden, geht *ClientControl* in den Fehlerzustand StateCC6 über. Wenn der Client seine TCP Verbindung selbst schließen kann, wechselt er in den Zustand StateCC4, um auf den Beginn des Experiments zu warten.

3.2.5. Zustand StateCC4: Durchführung der Aufgabe

In diesem Zustand wartet *ClientControl* bis zum Startzeitpunkt des Experiments. Wenn die Startzeit erreicht ist, beginnt *ClientControl* mit der Ausführung der Befehle aus dem *Timefile*. Die Befehle sind, relativ zur Startzeit, mit einem Zeitstempel versehen, der den Ausführungszeitpunkt bestimmt. Die Abbildung 3.6 verdeutlicht, dass *ClientControl* zwischen den einzelnen *Runs* Nachrichten per UDP an den Server schickt, um diesem seine Präsenz mitzuteilen. Wenn alle Befehle der Aufgabe durchgeführt wurden, wird die Kommunikation über UDP zum Server beendet. Anschließend wechselt *ClientControl* in den Zustand StateCC5. Das Experiment ist damit für den Client beendet.

3.2.6. Zustand StateCC5: Experiment erfolgreich beendet

ClientControl versucht nach dem Experiment eine TCP Verbindung zum Server aufzubauen, um diesem den erfolgreichen Ablauf der Aufgabe mitzuteilen (siehe Abbildung 3.7). Wenn *ClientControl* eine Verbindung zum Server aufbauen konnte, sendet *ClientControl* eine Logdatei, in welcher der Ablauf der Aufgabe dokumentiert wird, an den Server. Im Anschluss beendet *ClientControl* die TCP Verbindung und wechselt in den Zustand StateCC0. Falls die Verbindung zum Server vorher unerwartet abbricht, wechselt *ClientControl* in den Fehlerzustand StateCC6.

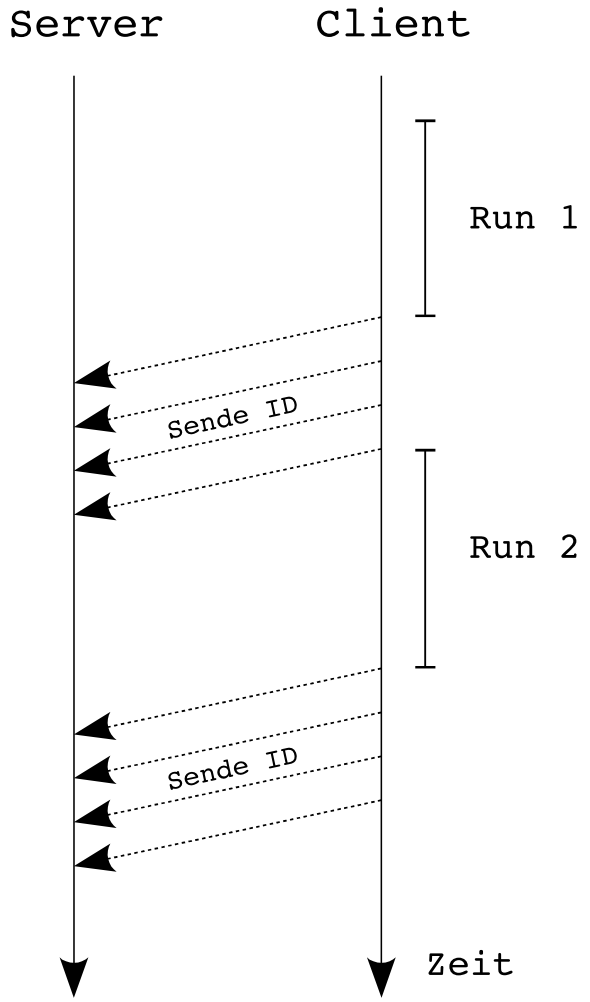

Abbildung 3.6.: Kommunikationsablauf während eines Experiments.

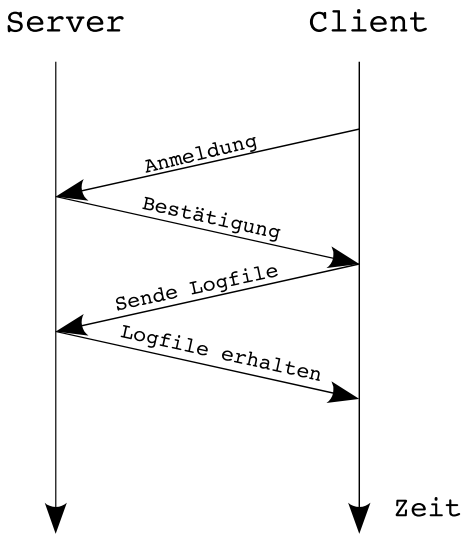

Abbildung 3.7.: Abschluss eines Experiments.

3.2.7. Zustand StateCC6: Fehlerzustand

ClientControl wechselt in den Fehlerzustand, wenn ein ungewolltes Ereignis den Client an der Ausführung seiner Aufgabe hindert. Der Zustandsautomat übergibt dem Fehlerzustand eine Nachricht, weshalb in diesen Zustand gewechselt wurde. Gründe für einen Wechsel in den Fehlerzustand sind:

- fehlerhafte Nachrichten vom Server
- fehlerhafte TCP Verbindung
- zu früher Abbruch der TCP Verbindung
- nicht vorhandene Messsoftware
- Probleme beim Öffnen des *Timefiles*
- Abbruch des Experiments durch den Server

In jedem dieser Fälle ist der Client nicht mehr in der Lage, seine Aufgabe erfolgreich und ohne Einschränkungen auszuführen. Aus diesem Grund setzt der Fehlerzustand den Zustandsautomaten zurück und wechselt in den Zustand StateCC0. In diesem Zustand versucht sich *ClientControl* erneut beim Server anzumelden, damit der Client weiteren Experimenten zur Verfügung gestellt werden kann. Wenn der Server nicht mehr erreichbar ist, versucht sich der Client periodisch bei der angegeben IP Adresse und Portnummer des Servers anzumelden, bis das Programm manuell geschlossen wird.

3.3. ServerControl

Im Folgenden werden die wichtigsten Module des serverseitigen Kontrollprogramms (*ServerControl*) erläutert. Der Server besteht im Wesentlichen aus zwei Hauptteilen. Der erste Teil nimmt eingehende TCP Verbindungen entgegen und verwaltet diese (siehe Abbildung 3.8). Masterclients können diesem Teil ein Experiment zusenden, welches anschließend in Aufgaben unterteilt wird. Der zweite Teil des Servers wird als Thread ausgeführt und kümmert sich um die Ausführung des Experiments (siehe Abbildung 3.9).

Wie auch das Serverprogramm der *In-Band-Experimentsteuerung* aus [Zag11] ist der zweite Teil von *SEC* ein endlicher Zustandsautomat. Dieser wird aber unabhängig vom ersten Teil ausgeführt, sodass jeder Zeit mehr als nur ein Experiment ausgeführt werden kann, sofern sich vorher genügend Clients beim Server zur Teilnahme angemeldet haben.

3.3.1. ServerControl

Beim Start wird die Konfigurationsdatei (siehe Listing 3.1) des Servers mit Hilfe des einfachen XML Parsers im Modul ServerConfigReader eingelesen. *ServerControl* erhält dadurch:

- den Port (*masterport*), über dem das Programm mit *SEC-Frontend* kommuniziert
- den Port (*clientport*), für die TCP Verbindungen zu den Clients
- die niedrigste (*minudpport*) und die höchste (*maxudpport*) Portnummer, aus deren Intervall *ServerControl* jedem Experiment einen Port zur Kommunikation während der Ausführung des Experiments vergibt
- die Angabe, wie viele Minuten vor der eigentlichen Startzeit eines Experiments mit der Verteilung der jeweiligen Aufgaben an die Clients begonnen werden soll (*closebefore*)

Im Anschluss wird jeweils ein Server zur Entgegennahme der Clients und ein Server zur Kommunikation mit Masterclients eingerichtet. Die Verbindungen zwischen dem Server und den Clients werden dem Modul *ClientStorage* übergeben, um diese dort zu speichern.

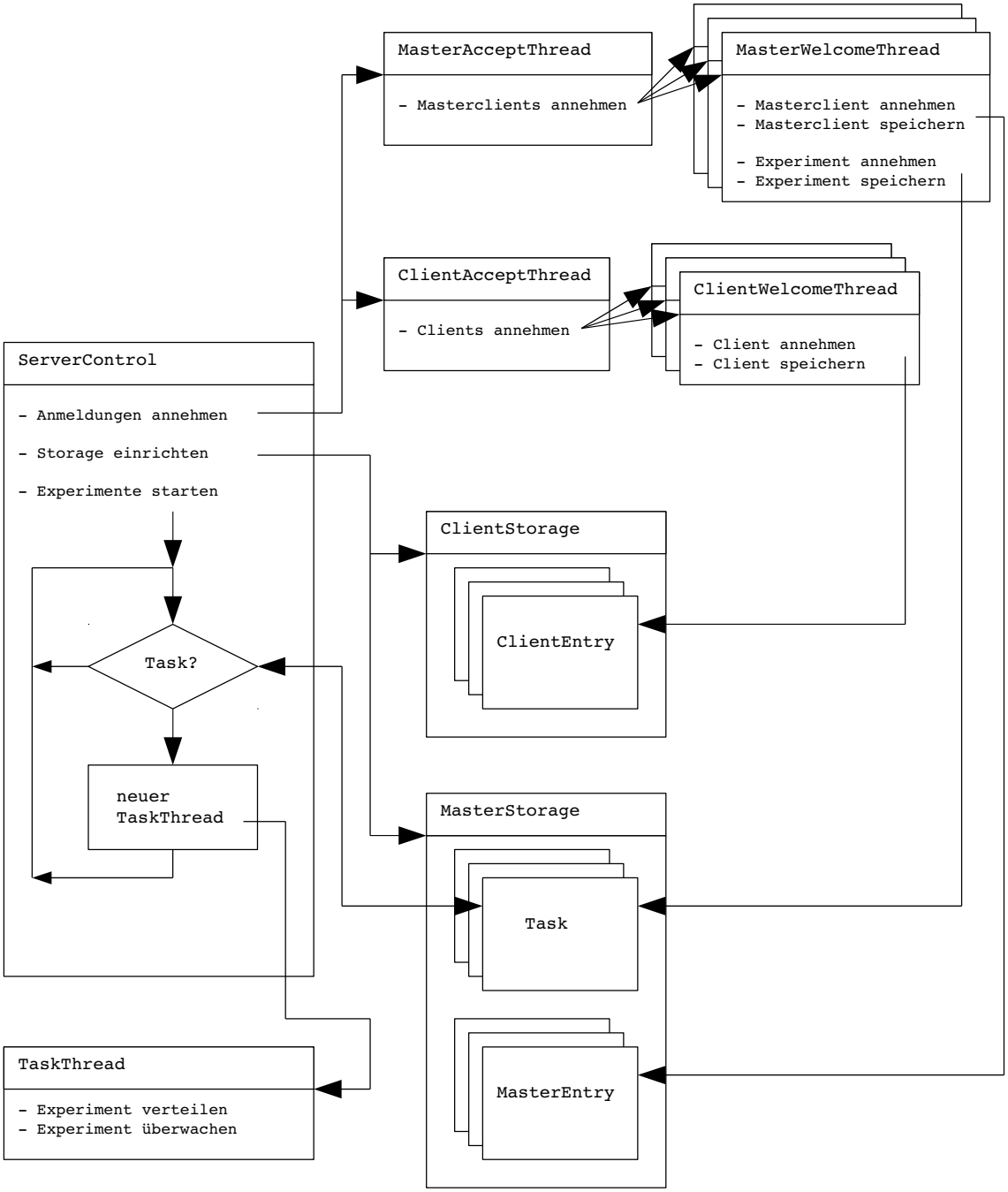

Abbildung 3.8.: *ServerControl*.

Solange *ServerControl* nicht von außen beendet wird, informiert sich das Modul regelmäßig beim *MasterStorage*, ob ein Experiment zur Bearbeitung bereit steht. Wenn dies der Fall ist, wird ein neuer *TaskThread* erstellt, der für dieses Experiment verantwortlich ist. Auf diese Weise ist die parallele Durchführung mehrerer Experimente möglich. Dem *TaskThread* wird zusätzlich zum Experiment noch ein eigener Port mitgeteilt, auf dem er Statusnachrichten der Clients entgegen nehmen kann. Nach erfolgreichem Abschluss eines Experiments wird dieser Port genutzt, damit jeder Client seine Logdatei an den Server senden kann.

Sollte *ServerControl* von Außen beendet werden, so werden zuerst die *TaskThreads* unterbrochen. Die Aufnahme weiterer Clients und Masterclients wird beendet, die Verbindungen zu gespeicherten Clients und Masterclients werden abgebaut und der Server wird geschlossen.

Listing 3.1: Server Konfiguration (*config.xml*).

```
<serverconfig>
    <param masterport="5005" />
    <param clientport="5000" />
    <param minudpport="5010" />
    <param maxudpport="6000" />
    <param closebefore="5" />
</serverconfig>
```

3.3.2. AcceptThread

Ein *AcceptThread* kann entweder zur Annahme von Clients oder zur Entgegennahme von MasterClients instanziiert werden. In beiden Fällen wird ein Thread erstellt, der ständig eingehende TCP Verbindungen zum Server entgegen nimmt. Wenn den Server eine neue Anfrage zum Aufbau einer Verbindung erreicht, speichert der *AcceptThread* die Verbindung und startet je nach Instanz einen *MasterWelcomeThread* oder einen *ClientWelcomeThread.*

Jedes Mal, wenn ein neuer Client oder Masterclient von einem *AcceptThread* angenommen wird, prüft das Modul, ob es alte *WelcomeThreads* entfernen kann, die ihren Client oder Masterclient bereits erfolgreich entgegen genommen haben.

3.3.3. WelcomeThreads

Jeder *WelcomeThread* begrüßt entweder einen Client oder einen Masterclient. Der Vorteil der Begrüßung der Teilnehmer durch einen eigenen Thread liegt darin, dass sich jeder Teilnehmer ausführlich vorstellen kann, parallel dazu aber weitere, beliebige Teilnehmer entgegen genommen werden können.

Der *ClientWelcomeThread* verarbeitet Teilnehmer, die sich über den Port für einfache Clients mit dem Server verbinden. Dieser Thread vergewissert sich, dass es sich tatsächlich um einen Client handelt und fügt dessen Verbindung, IP Adresse und Port zum *ClientStorage* hinzu. In der Regel dauert dieser Vorgang nicht mehr als ein paar Sekunden.

Der *MasterWelcomeThread* begrüßt hingegen Masterclients. Auch dieser vergewissert sich, dass sich ein Masterclient angemeldet hat und fügt dessen IP Adresse, Port und Verbindung zum *MasterStorage* hinzu. Anschließend bleibt der *MasterWelcomeThread* mit dem Masterclient verbunden, um Experimente von diesem erhalten zu können. Jedes empfangene Experiment gibt der *MasterWelcomeThread* an den *MasterStorage* weiter, wo es gespeichert wird.

3.3.4. ClientStorage

Dieses Modul nimmt Informationen von Clients entgegen und speichert:

- die TCP Verbindung

- die IP Adresse und den Port des Clientprogramms

Zusätzlich kann für jeden Client eine ID gespeichert werden, die innerhalb eines Experiments gültig ist.

3.3.5. MasterStorage

Ähnlich wie das Modul *ClientStorage* speichert *MasterStorage* Daten über angemeldete Masterclients. Gespeichert wird für jeden Masterclient:

- die TCP Verbindung zum Masterclient
- die IP Adresse und der Port des Masterclients
- Experimente, die dem Server zugesendet werden

Auf Anfragen des Moduls *ServerControl* übergibt *MasterStorage* diesem ein anstehendes, noch nicht erfolgreich durchgeführtes Experiment (*Task*), zur Durchführung. Wurde ein *Task* an *ServerControl* übergeben, wird er als aktiv markiert. Sollte dass Experiment aufgrund fehlender Clients nicht durchgeführt werden können, wird die Markierung entfernt sodass der *Task* nach einer kurzen Wartezeit erneut von *ServerControl* angefordert werden kann. *Tasks*, die erfolgreich beendet wurden, werden gelöscht.

3.3.6. Task / TaskDivider

Informationen über ein Experiment werden im Modul *Task* gespeichert. Wenn ein *MasterWelcomeThread* ein Experiment in Form zweier XML Dateien enthält (siehe Konfigurationsdatei und Experimentdatei in Kapitel 3.4) und diese zum *MasterStorage* hinzufügt, erstellt *MasterStorage* eine neue *Task* Instanz und initialisiert diese. Dabei unterteilt das Modul *TaskDivider* das Experiment in Aufgaben für die beteiligten Clients. Die Aufgaben der Clients werden in gewöhnlichen Textdateien, den *Timefiles* gespeichert. Das *Timefile* erspart dem Client zusätzliche Arbeit, da dieser nun keine verschachtelten XML Strukturen und Tags, sondern nur einfachen Text parsen muss (siehe Kapitel 3.5).

3.3.7. TaskThread

TaskThreads sind für die Durchführung eines Experiments zuständig. Ähnlich wie das gesammte Serverprogramm in [Zag11] ist dieser Thread ein endlicher Zustandsautomat (siehe Abbildung 3.9), dessen Zustände und Aufgaben jedoch an einigen Stellen abweichen. Wie schon in Kapitel 3.2 beschrieben, liegt der größte Unterschied in der Art der Kommunikation. In *SEC* kommunizieren Server und Clients während der gesamten Vorbereitung auf ein Experiment über eine TCP Verbindung. Deshalb weiß der Server zu jeder Zeit (außer während des Experiments), in welchen Zuständen sich die Clients befinden.

Zustand StateSC0: Voraussetzung prüfen

Anders als in [Zag11] muss dieser Zustand nicht auf ein Experiment warten, da es dem Modul *TaskThread* vom Modul *ServerControl* übergeben wurde. In diesem Zustand prüft der Thread, ob die aktuelle Zeit bereits über der angegebenen Startzeit inklusive einer einstellbaren Zeit (siehe Kapitel 3.4.1) liegt. Wenn dies der Fall ist, wird das Experiment beendet und aus dem *MasterStorage* entfernt. Sollte die aktuelle Zeit innerhalb des festgelegten Zeitrahmens für das Experiment liegen, wird geprüft, ob sich genug Clients beim Server angemeldet haben, um das Experiment durchführen zu können. Entweder stehen genug Clients zu Verfügung, sodass *TaskThread* in den Zustand StateSC1 wechseln kann, oder *TaskThread* muss wegen eines Clientmangels beendet werden. Im letzten Fall wird das Experiment im *MasterStorage* als inaktiv markiert, damit es nach kurzer Wartezeit erneut gestartet werden kann. Ist die Prüfung erfolgreich beendet, wechselt *TaskThread* in den Status StateSC1, um die Aufgaben an die Clients zu verteilen.

Zustand StateSC1: Aufgaben verteilen

Die eigentliche Verteilung der Aufgaben übernimmt die Klasse *ClientPool*, welche zusammen mit jedem *TaskThread* instanziiert wird. Im Zustand StateSC1 wird dem *ClientPool* mitgeteilt, dass dieser mit der Verteilung der Aufgaben beginnen soll.

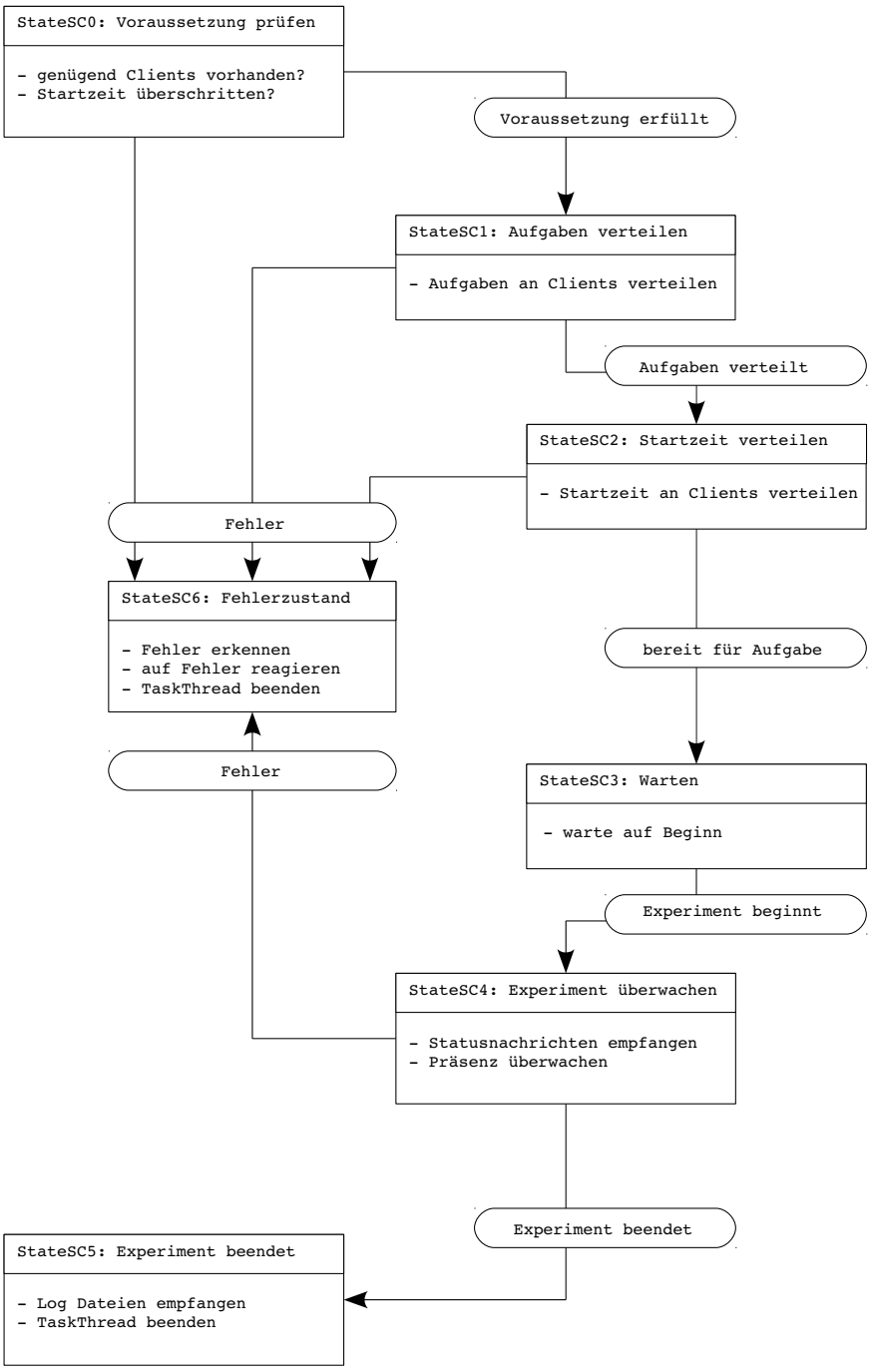

Abbildung 3.9.: Zustandsautomat des *TaskThreads*.

Ist die Verteilung nicht erfolgreich, weil benötigte Clients fehlen, so geht der Zustandsautomat in den Fehlerzustand StateSC6 über. Bei erfolgreicher Verteilung wird der *ClientPool* vom *TaskThread* angewiesen, Verbindungen zu nicht benötigten Clients zu beenden (siehe Abschnitt 3.3.9), damit diese sich für andere Experimente beim Server anmelden können. Daraufhin wechselt *TaskThread* in den Zustand StateSC2, um die Startzeit an die teilnehmenden Clients zu verteilen.

Zustand StateSC2: Startzeit verteilen

TaskThread teilt dem *ClientPool* in diesem Zustand mit, dass die Startzeit an alle teilnehmenden Clients verteilt werden soll. Vorher jedoch startet *TaskThread* einen UDP Server für die weitere Kommunikation während des Experiments. *TaskThread* fordert die Clients auf, UDP Testpakete mit ihren IDs an den Server zu senden. Auf diese Weise erfährt *TaskThread* die IP Adressen und die Ports der UDP Clients, damit er diesen auch während des Experiments Nachrichten zusenden kann. Wenn alle UDP Testpakete beim Server eingetroffen sind, verteilt der *ClientPool* die Startzeit des Experiments per TCP an alle Clients (siehe Abbildung 3.5). Wenn die Startzeit rechtzeitig an alle Clients übergeben wurde, werden die TCP Verbindungen zu den Clients geschlossen und *TaskThread* wechselt in den Zustand StateSC3.

Zustand StateSC3: Auf Beginn warten

Solange die Startzeit nicht überschritten ist, wartet der *TaskThread* im Zustand StateSC3. Nach erreichen der Startzeit springt der Zustandsautomat sofort in den Zustand StateSC4.

Zustand StateSC4: Experiment überwachen

Im Zustand StateSC2 hat *TaskThread* einen UDP Server erstellt, damit das Modul Statusnachrichten von den Clients empfangen kann. Falls ein Client sich nach mehreren *Runs* nicht beim Server melden sollte, kann das Experiment abgebrochen werden. *TaskThread* verfolgt mittels eines eigenen *Timefile*, welche *Runs* des Experiments durchgeführt worden sein müssten. Wenn *TaskThread* aufgrund des *Timefiles* annehmen kann, dass alle *Runs* bearbeitet wur-

den (*TaskThread* weiß nicht, ob alle Clients erfolgreich waren), schließt das Modul den UDP Server und wechselt in den Zustand StateSC5, um das Experiment ordentlich zu beenden.

Zustand StateSC5: Experiment erfolgreich beendet

Die Clients versuchen nach dem Experiment eine TCP Verbindung zum Server aufzubauen, um diesem ihre Logdateien zu zuschicken (siehe Abbildung 3.7). Aus diesem Grund erstellt der *TaskThread* einen eigenen TCP Server, der den nicht mehr benutzten Port des inzwischen geschlossenen UDP Servers verwendet. Nach einer kurzen Zeit wird der TCP Server beendet. Bis dahin sollten die Clients dem Server ihre Logdateien zugesendet haben. Wenn der TCP Server geschlossen wurde, wird der Zustandsautomat beendet und das Experiment als beendet markiert.

Zustand StateSC6: Fehlerzustand

TaskThread wechselt in den Fehlerzustand, wenn:

- keine oder zu wenig Clients für das Experiment zur Verfügung stehen
- die Startzeit inklusive eines konfigurierbaren Zeitfensters verstrichen ist
- ein Abbruch des Experiments gefordert wird

Wenn keine Clients für das Experiment zur Verfügung stehen, kann das Experiment nach einer kurzen Wartezeit erneut vom Modul *ServerControl* gestartet werden. Ansonsten wird das Experiment als beendet markiert, damit es im *MasterStorage* gelöscht wird. Wenn der aufgetretene Fehler vom Fehlerzustand bearbeitet wurde, wird der Zustandsautomat beendet und der *TaskThread* ebenfalls als beendet markiert, damit dieser gelöscht werden kann.

3.3.8. DistributionThread

Diese Threads werden instanziiert, um eine parallele Übertragung von Aufgaben an die Clients eines Experiments zu ermöglichen. Die Aufgabe an einen Client besteht aus einem *Timefile* und einer Konfigurationsdatei für das Experiment. Außerden teilt *DistributionThread* dem Client eine ID für das Experiment, sowie den UDP Port mit, auf dem *TaskThread* während der Durchführung des Experiments lauscht. Dieser Port wird nach dem Experiment benutzt, um die Logdateien der Clients zu empfangen. Wenn der Client den Erhalt der Aufgabe bestätigt (entweder mit Erfolg oder der Antwort, dass mindestens ein Messprogramm nicht antwortet), wird der *DistributionThread* als beendet markiert.

3.3.9. ClientPool

Wie die Klasse *ClientStorage* speichert *ClientPool* Informationen über Verbindungen zu den Clients, die sich beim Server angemeldet haben. Während *ClientStorage* nur solche Clients enthält, die sich beim Server angemeldet haben und daraufhin in den Standbyzustand gewechselt sind, werden im *ClientPool* Verbindungen zu Clients gespeichert, die einem Experiment zugeordnet wurden. Clients, deren Verbindung im *ClientPool* gespeichert sind, können sich also in jedem Zustand (außer StateSC0: Anmeldung) befinden. Der *ClientPool* kann bei Bedarf Clients aus dem *ClientStorage* entfernen und für seine eigenen Zwecke benutzen. Neben der Aufbewahrung der Clients erfüllt der *ClientPool* zwei wichtige Aufgaben:

- Verteilung der Aufgaben an die Clients

- Verteilung der Startzeit an die Clients

Verteilung der Aufgaben

Zunächst entnimmt der *ClientPool* dem Modul *ClientStorage* die Anzahl an ClientEntries, die er für die Durchführung des Experiments benötigt. Für jeden Client wird dann ein *DistributionThread* erstellt und gestartet. *ClientPool* wartet, bis alle *DistributionThreads* beendet

wurden und prüft, ob jeder Client seine Aufgabe erhalten hat und diese auch ausführen kann. Jeder Client, der die Aufgabe nicht ausführen kann (weil ein Messprogramm nicht gestartet werden konnte), wird durch einen neuen Client aus dem *ClientStorage* ersetzt, solange bis genügend Clients ihre Aufgabe bestätigt haben (siehe Abbildung 3.10).

ClientPool trennt im Anschluss die Verbindung zu allen Clients, die aus dem *ClientStorage* entnommen wurden, aber nicht am Experiment teilnehmen werden. Diese Clients können sich beim Server für andere Experimente anmelden.

Verteilung der Startzeit

ClientPool erstellt für jeden teilnehmenden Client einen *SyncThread* und startet diesen. Dieser übernimmt die Kommunikation mit dem Client und ermöglicht dem Server die parallele Verteilung der Startzeit an alle Clients.

Als Startzeit wird die aktuelle Zeit gewählt, auf die ein benutzerdefinierter Wert addiert wird, der sicherstellen soll, dass die Startzeit rechtzeitig von allen Clients empfangen wird. Die so erhaltene Startzeit wird zusätzlich auf eine volle Minute aufgerundet und den Clients mitgeteilt. Wenn der Wert unter der vom Benutzer hinterlegten Startzeit liegt, wird diese stattdessen verwendet. Wenn alle Clients rechtzeitig über die Startzeit informiert wurden, gibt der Server den Clients die Anweisung, ihre TCP Verbindungen zu schließen.

Wenn die Startzeit nicht rechtzeitig von allen Clients bestätigt wird, erhöht der Server den Wert auf die gleiche Weise, wie zuvor beschrieben. Dieser Schritt wird wiederholt bis alle Clients den Erhalt rechtzeitig bestätigen, die Verbindung zu mindestens einem Client abbricht, oder die vorgegebene Zeitspanne überschritten ist, in der das Experiment starten darf.

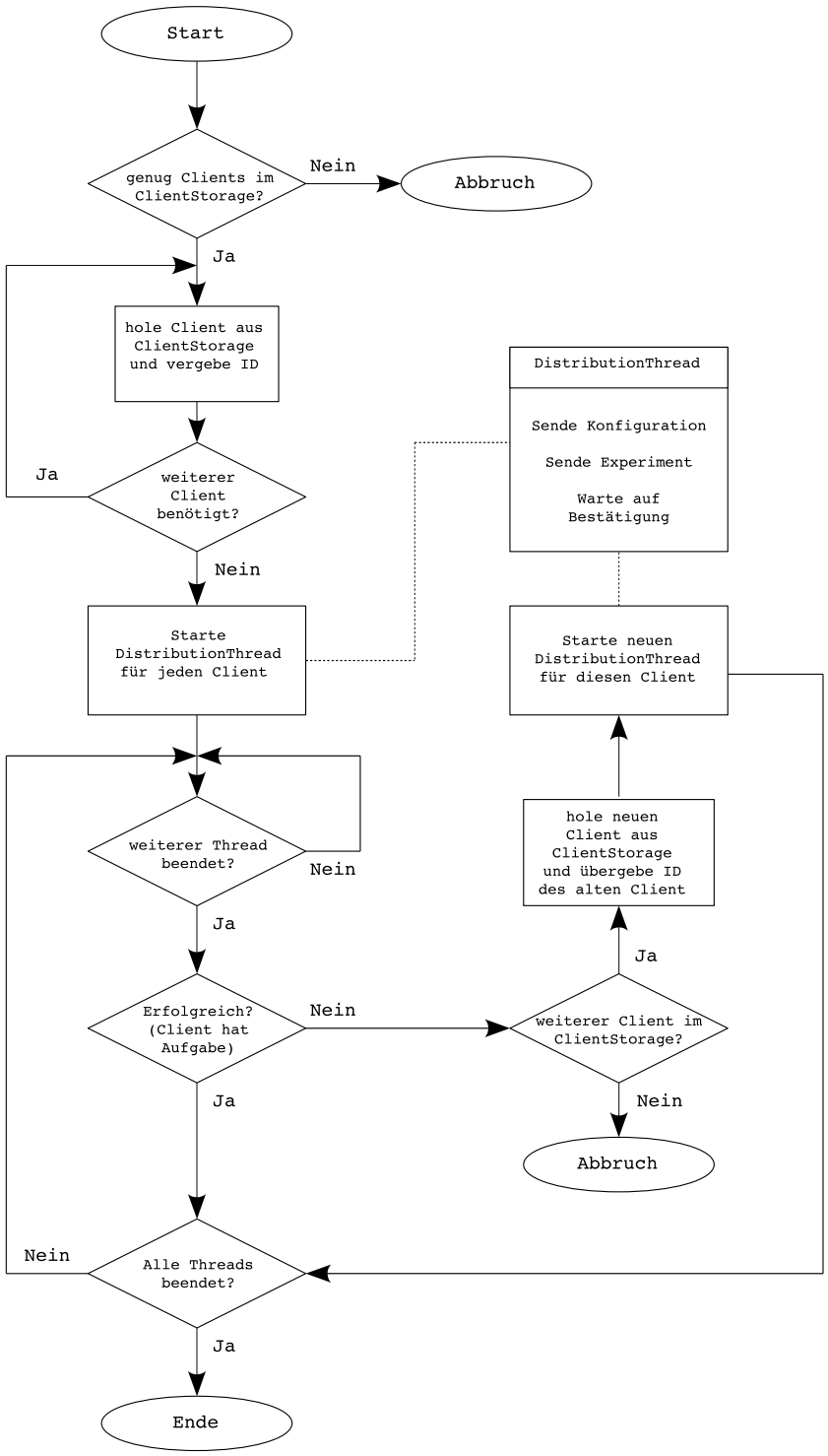

Abbildung 3.10.: Verteilung eines Experiments im *ClientPool*.

3.4. XML Dateien

Damit ein Experiment durchgeführt werden kann, muss ein zeitlicher Ablauf vorliegen, der die einzelnen Ausführungsschritte des Experiments beschreibt. *SEC* benötigt deshalb zur Durchführung eines Experiments, dessen Beschreibung im XML Format. Dieses Format wurde für die Beschreibung des Experiments aus folgenden Gründen gewählt:

- Dateien im XML Format lassen sich sehr gut strukturieren
- Das XML Format ist menschenlesbar
- Es gibt viele Editoren, welche die erstellten XML Dokumente direkt auf Gültigkeit prüfen

Ein Experiment in *SEC* wird von zwei XML Dateien beschrieben. Eine Datei enthält einen Zeitplan des eigentlichen Experiments (*Experimentdatei*), die andere Datei beschreibt Konfigurationen zum Experiment (*Konfigurationsdatei*). XML Strukturen und Befehle werden in beiden Dateien grundsätzlich klein geschrieben. Die Verwendung der separaten Konfigurationsdatei (siehe Listing 3.2) hat den Vorteil, dass eine Experimentdatei einfach wiederzuverwenden ist. Wenn beispielsweise zwei identische Experimente zu unterschiedlichen Zeiten ausgeführt werden sollen, braucht nur die Startzeit in der Konfigurationsdatei verändert werden. Die Experimentdatei bleibt für beide Experimente gleich.

3.4.1. Konfiguration

Die Konfigurationsdatei beginnt mit dem XML Start Tag <*expconfig*> und endet dementsprechend mit dem XML End Tag </*expconfig*> (siehe Listing 3.2). Zwischen diesen Tags können Einstellungen für das Experiment vorgenommen werden. Die Zuweisung erfolgt über den XML Short Tag <*param key*=“*value*“ /> wobei *key* eine einstellbare Variable und *value* deren Wert ist. Der Tabelle 3.1 können dazu die gültigen Variablen und deren Format entnommen werden.

Listing 3.2: Konfigurationsdatei eines Experiments.

```
<expconfig>

    <param starttime="20:26" />
    <param date="17.5.2012" />
    <param experiment="experiment.xml" />
    <param maxdelay="1:00" />
    <param name="testexp" />
    <param maxtrmtime = "60" />
    <param abortifnoresponse="true" />
    <param maxnoresponse="2" />

</expconfig>
```

Variable	Bedeutung
starttime	gewünschte Startzeit des Experiments im Format h:mm oder hh:mm
date	Datum an dem das Experiment stattfinden soll im Format d.m.yyyy, d.mm.yyyy, dd.m.yyyy oder dd.mm.yyyy
experiment	Name der XML Datei, die das Experiment beschreibt
maxdelay	Zeitfenster (Format hh:mm oder h:mm), in dem das Experiment auch nach Verstreichen der gewünschten Startzeit (*starttime*) noch gestartet werden darf
name	Name des Experiments als ASCII Zeichenkette
maxtrmtime	Angabe der größten geduldeten Übertragungsdauer von Nachrichten zwischen dem Server und den Clients in Sekunden
abortifnoresponse	Angabe, ob ein Experiment abgebrochen werden soll (abortifnoresponse="true"), wenn der Server während der Durchführung des Experiments von mindestens einem Client keine Statusnachrichten empfängt, oder nicht (abortifnoresponse="false")
maxnoresponse	Wenn abortifnoresponse="true" ist, gibt maxnoresponse an, wieviele *Runs* die Statusnachricht jedes Clients maximal ausbleiben darf, bevor das Experiment geschlossen werden muss. Wenn abortifnoresponse="false" ist, wird maxnoresponse ignoriert

Tabelle 3.1.: Werte der XML Konfigurationsdatei.

3.4.2. Experiment

Der zeitliche Ablauf des Experiments wird in einer eigenen XML Datei beschrieben, die mit dem XML Start Tag <*experiment*> beginnt und dementsprechend mit dem XML End Tag </*experiment*> endet. Im Folgenden wird die Syntax der in *SEC* verwendeten XML Befehle beschrieben und durch ein anschließendes Beispielexperiment ergänzt. Dazu kann das XML Dokument im Listing 3.3 mit der Abbildung 3.11 verglichen werden, das den Ablauf eines Beispielexperiments zeigt.

<include>

Direkt nach dem XML Start Tag <*experiment*> können Messprogramme angegeben werden, denen *SEC* während des Experiments Befehle über ein TCP/IP Loopback Interface gibt (siehe Kapitel 4.2.5). Dazu wird in der Experimentdatei folgender Befehl angegeben:

<*include port=“p_port“ name=“p_name“*> *p_command* </*include*>

Die Tabelle 3.2 beschreibt die Parameter des Befehls. Der Text zwischen den *Include* Tags wird wie ein Befehl an das System behandelt. Programme im Binärcode (C/C++) werden beispielsweise mit *./programm* gestartet, wobei *programm* der Name der auszuführenden Binärdatei ist. Mit dem Attribut *name* kann der Software ein Name zur weiteren Verwendung innerhalb der XML Datei vergeben werden, der dann als gültiger Name für Tags zur Verfügung steht. *SEC* nimmt an, dass die Messsoftware auf dem angegebenen Port Verbindungen über das Loopback Interface entgegen nimmt. Dem Messprogramm muss der Port daher zusätzlich, wie bei einem Kommandozeileninterpreter, übergeben werden (Beispiel: *./programm 6789*).

Value	Bedeutung
p_port	gibt den Port an, auf dem das Messprogramm lauscht
p_name	gibt der Software zur weiteren Verwendung innerhalb der XML Datei einen Namen, der nun als Tag Name zur Verfügung steht
p_command	gibt an, wie das Messprogramm geöffnet wird

Tabelle 3.2.: Beschreibung des *Include* Tags.

<run>

Der Ablauf des Experiments wird in *Runs* unterteilt, zwischen denen die Clients mit dem Server kommunizieren können. Der Anfang eines *Runs* wird mit dem XML Tag <*run*> beschrieben. Grundsätzlich folgt auf jeden *Run* direkt der nächste *Run*, wenn es einen gibt. Mit dem Attribut *offset* kann eine Zeitspanne angegeben werden, die zwischen zwei *Runs* geschoben wird. Beispielsweise verzögert <*run offset="8s"*> den Beginn des *Runs* um acht Sekunden relativ zum letzten *Run* (oder zum Anfang des Experiments, wenn dies der erste *Run* ist). *Runs* werden mit dem XML End Tag </*run*> beendet. Derjenige Client, der die längste Zeitspanne für die Ausführung seiner Befehle in einem *Run* benötigt, bestimmt die zeitliche Länge des *Runs*. Ein *Run* dauert dann so lange, wie die Zeitspanne, die dieser Client in Anspruch nimmt.

<client>

Innerhalb eines *Runs* (definiert durch die Tags <*run*> und </*run*>) werden Befehle an die einzelnen Clients vergeben. Zur Beschreibung welcher Befehl von welchem Client durchgeführt werden soll, wird das XML Tag Paar <*client*> und </*client*> benutzt. Befehle innerhalb dieser Tags werden nur von ausgewählten Clients ausgeführt. Die Auswahl der Clients wird mit Hilfe des Attributes *id* vorgenommen. Die Zuweisung kann auf verschiedene Arten erfolgen:

- *id="nr"*: nur der Client mit der ID nr wird ausgewählt
- *id="nr_1-nr_2"*: alle Clients mit den IDs von nr_1 bis nr_2 werden ausgewählt
- *id="nr_1 nr_2 ... nr_n"*: alle angegebenen Clients werden ausgewählt
- *id="all"*: alle Clients, die noch nicht benutzt wurden, werden ausgewählt
- *keine* Angabe einer *id*: der nächste noch nicht benutzte Client wird ausgewählt

Jeder Client, kann innerhalb eines *Runs* nur einmal ausgewählt werden. Wurde die ID eines Clients schon vorher im selben *Run* benutzt, werden die Befehle zwischen den Tags <*client*>

und *</client>* für diesen Client ignoriert. Alle Clients, die nicht im *Run* ausgewählt wurden, führen während dieses *Runs* keine Befehle aus.

<system>

Zur Durchführung eines Experiments wird ein Client in der Regel ein Messprogramm benutzen, um Daten zu erfassen. Mit dem *Include* Tag wird bekannt gemacht, welche Messsoftware im Experiment benutzt werden soll. Mit dem Attribute *name* wurde der Messsoftware einen Namen zu weiteren Verwendung innerhalb der XML Datei vergeben. Befehle an ein Messprogramm können mit dem User Block gemacht werden:

<name offset="p_offset" duration="p_duration">command</name>

Dabei ist *name* der im *Include* Tag vergebene Name für die Messsoftware. Mit dem Wert *p_offset* wird ein Offset zu vorangegangenen Befehlen (auch Start eines *Runs*) angegeben. Nach einer beliebigen Dauer, durch *p_duration* angegeben, wird der Messsoftware mitgeteilt, die Ausführung des Befehls zu beenden. Der eigentliche Befehl an die Messsoftware (*command*) steht zwischen Start Tag und End Tag. Auf diese Weise ist der Benutzer sehr flexibel, was die Verwendung von Befehlen betrifft. Es ist zum Beispiel möglich, dass der Benutzer ein Kommando zum Starten eines Befehls und ein Kommando zum Beenden eines Befehls definiert. Auf diese Weise kann er komplexe und verschachtelte Anweisungen an ein Messprogramm oder gleich an mehrere Messprogramme übergeben.

Befehle an das System des Clients können mit dem *System* Tag gemacht werden:

<system offset="p_offset">command</system>

Der *System* Tag ermöglicht es dem Benutzer jegliche Art von Software zu benutzen, die keine Anweisungen von SEC entgegen nehmen muss. Für den Befehl kann wieder ein Offset zu vorangegangenen Befehlen angegeben werden. Der Befehl (hier *command*) an das System des Clients steht zwischen den Tags *<system>* und *</system>*.

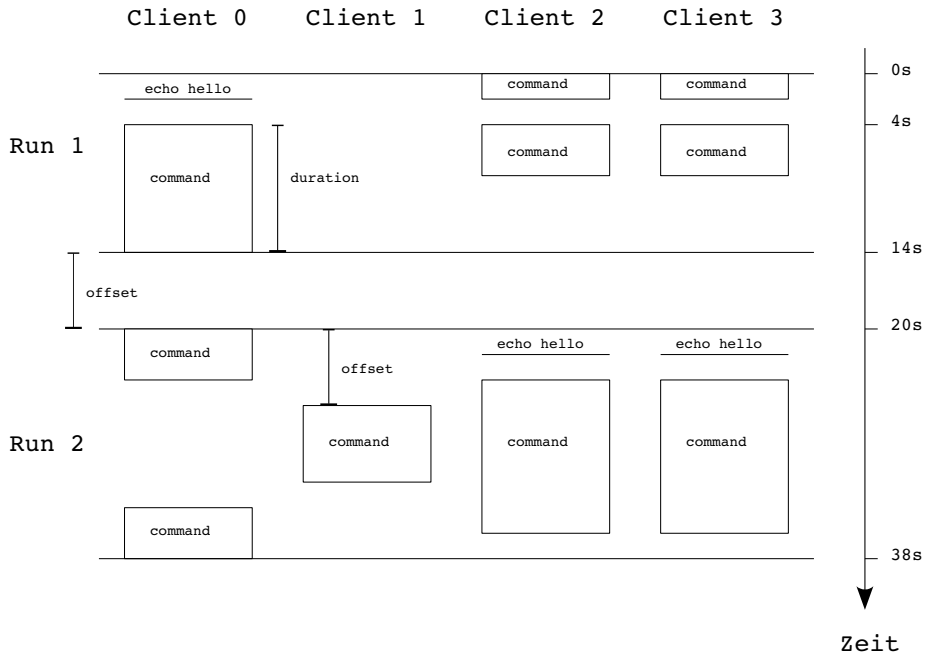

Abbildung 3.11.: Ablauf eines Beispielexperiments.

Listing 3.3: Experimentdatei zum Beispielexperiment.

```
<experiment>
    <include port="6789" name="soft" >./testsoftware</include>
    <include port="9876" name="ware" >java -jar software.jar</include>

    <run>
        <client id="0">
            <system offset = "2s">echo hello</system>
            <soft offset="2s" duration="10s">command</soft>
        </client>

        <client />

        <client id="all">
            <soft duration="2s">command</soft>
            <soft offset="2s" duration="4s">command</soft>
        </client>

    </run>

    <run offset = "6s">
        <client id="0">
            <soft duration="4s">command</soft>
            <soft offset="10s" duration="4s">command</soft>
        </client>

        <client id="1">
            <soft offset="6s" duration="6s">command</soft>
        </client>

        <client id="2␣3">
            <system offset = "2s">echo hello</system>
            <soft offset="2s" duration="12s">command</soft>
        </client>

    </run>

</experiment>
```

3.4.3. Offset und Duration

Zeitangaben für das Experiment werden in der XML Datei immer relativ zu vorangegangenen Zeitangaben gemacht. Die erste Zeitangabe wird immer relativ zur Startzeit gemacht. Bei Befehlen und *Run* Tags kann deshalb ein *Offset* eingestellt werden, der eine Zeitspanne zwischen zwei *Runs*, zwischen zwei Befehlen oder zwischen Beginn eines *Runs* und einem Befehl angibt. Bei Befehlen an die Messsoftware kann eine Dauer (*Duration*) für den Befehl angegeben werden. Da dieser eine gewisse Zeit benötigt, bis er bei der Messsoftware angekommen ist und von dieser interpretiert wurde, kann die Angabe einer Dauer als Zeitpuffer zwischen mehreren Befehlen dienen. Die Angabe einer Dauer kann beispielsweise genutzt werden, um Messungen in einem bestimmten Zeitintervall von der Messsoftware durchführen zu lassen. Für *offset* und *duration* ist dabei zu beachten:

- wird offset oder duration nicht benutzt, wird auch kein relativer Zeitsprung gemacht
- einzelne Zahlen ohne Einheiten werden als Sekunden interpretiert
- die Einheiten d für Tage, h für Stunden, m für Minuten, s für Sekunden und ms für Millisekunden können vergeben werden, um genaue Zeitangaben zu machen

Soll ein Befehl beispielsweise acht Minuten, zehn Sekunden und einhundert Millisekunden dauern, wird dem Start Tag des Befehls *duration="8m 10s 100ms"* hinzugefügt. Dabei spielt es keine Rolle, in welcher Reihenfolge die Einheiten angegeben werden. Werden Einheiten mehrfach benutzt, so addieren sich Zahlen mit gleichen Einheiten zusammen (*duration="2m 12m 20s"* entspricht *duration="14m 20s"*). Die Addition gleicher Einheiten soll dem Benutzer das Rechnen mit relativen Zeitangaben vereinfachen. Dazu könnte man sich folgendes Szenario vorstellen:

Ein Client soll hintereinander drei Befehle ausführen, die jeweils einen bestimmten Offset und eine bestimmte Dauer haben. Der erste Befehl habe keinen Offset und die Dauer *duration="8m 50s 200ms"*. Der zweite Befehl habe den Offset *offset="4s"* und die Dauer *duration="5m 20s 900ms"*. Der dritte Befehl habe den Offset *offset="10s"* und die Dauer *duration="9m 30s 700ms"*. Ein zweiter Client soll nun einen Befehl ausführen, der so lange dauert, wie die Ausführung der drei Befehle des ersten Clients zusammen. Der Benutzer kann nun

alle Millisekunden, Sekunden und Minuten zusammenrechnen, oder einfach *duration="8m 50s 200ms 4s 5m 20s 900ms 10s 9m 30s 700ms"* als Dauer angeben.

3.5. Das Timefile

In *SEC* wird ein Experiment in mehrere Aufgaben unterteilt, die als gewöhnliche Textdatei (*Timefile*) im ASCII Format an die teilnehmenden Clients gesendet werden. Während eines Experiments wird jeweils der nächste Befehl und dessen Ausführungszeit aus dem *Timefile* gelesen. Die Messgeräte, auf denen *SEC* eingesetzt werden soll, verwendet Compact-Flash Karten statt herkömmlichen Festplatten. Demzufolge fällt die Zugriffszeit für das Lesen des *Timefiles* auf diesen Geräten kaum ins Gewicht. Das *Timefile* beginnt mit einer Liste der Messprogramme, die für das Experiment benutzt werden. Ein Listeneintrag besteht aus dem Port der Messsoftware, gefolgt von der Anzahl an Parametern, mit denen das Messprogramm geöffnet wird. Die danach angegebenen Parameter bestimmen, wie das Messprogramm geöffnet wird. Ein Messprogramm, das in Java geschrieben wurde, könnte zum Beispiel mit *java -jar software.jar* geöffnet werden.

Die XML Tags aus der Experimentdatei werden vom *TaskDivider* beim erstellen des *Timefiles* in eine einfachere Struktur umgewandelt (siehe Kapitel 3.3.6). So kann *ClientControl* die einzelnen Anweisungen direkt lesen, ohne in der Datei zurück springen zu müssen oder vorherige Strukturen speichern zu müssen. Im Folgenden wird beschrieben, nach welchen Regeln die XML Befehle vom *TaskDivider* umgewandelt werden. Das Listing 3.4 zeigt das *Timefile* für den Client 2 des in der Abbildung 3.11 beschriebenen Experiments.

Die Zeitangaben sind im *Timefile* nicht mehr relativ zum vorherigen Befehl angegeben. Jeder Befehl im *Timefile* ist nun relativ zur Startzeit angegeben. Dazu wird vor jedem Befehl die Sekunde und die Mikrosekunde der Ausführungszeit im *Timefile* gespeichert.

Der <*run*> Start Tag wird durch den Befehl *run start* ersetzt. Äquivalent dazu wird der End Tag <*/run*> durch *run end* ersetzt. Der umgewandelte Befehl <*system*>*command*<*/system*> beginnt im *Timefile* mit dem Bezeichner *system*, der von der Anzahl der zu übergebenden

Argumente gefolgt wird. Dieser Anzahl folgen schließlich die Argumente, die dem Betriebssystem übergeben werden sollen.

Kommandos können in der Experimentdatei mit <*name*> *command* </*name*> an ein Messprogramm übergeben werden, wobei *name* das Messprogramm identifiziert und *command* ein Kommando beschreibt. Dieser Befehl wird im *Timefile* durch den Bezeichner *user_command* dargestellt. Weil für jedes Kommando an die Messsoftware eine Dauer angegeben werden kann, folgt diesem Bezeichner die Anweisung *start*, wenn der Befehl beginnt, oder *end*, wenn der Befehl beendet werden soll. In *ClientControl* werden die Messprogramme in der Reihenfolge durchnummeriert (beginnend mit 0), wie sie in der Experimentdatei mit dem <*include*> Befehl angegeben wurden. Die nächste Zahl in der *Timefile* ist deshalb die Nummer des Messprogramms, das den Befehl ausführen soll. Dieser Nummer folgt die Anzahl der zu übergebenden Zeichenketten und schließlich die Zeichenketten, die der Messsoftware übergeben werden.

Listing 3.4: Timefile eines Clients.

```
includes start
6789 1 ./testsoftware
9876 3 java -jar software.jar
includes end

0 0 run start
0 0 user_command start 0 1 command
2 0 user_command end 0 1 command
4 0 user_command start 0 1 command
8 0 user_command end 0 1 command
14 0 run end
20 0 run start
22 0 system 2 echo hello
24 0 user_command start 0 1 command
36 0 user_command end 0 1 command
38 0 run end
```

Kapitel 4.

Implementierung

Im vorherigen Kapitel wurde das Design, die Komponenten und die allgemeine Funktionsweise von *SEC* erläutert. In diesem Kapitel sollen ausgewählte Themen der Implementierung näher behandelt werden.

SEC wurde unter dem Aspekt entwickelt, dass auch Systeme mit vergleichsweise schwacher Hardware an Experimenten teilnehmen können. *SEC* soll auf mobilen Messgeräten eingesetzt werden, die mit gerade einmal 256 MB Arbeitsspeicher und einem 500 MHz Prozessor ausgestattet sind. Aus diesem Grund wurden die Teilprogramme *ServerControl* und *ClientControl* in C++ und ohne Zuhilfenahme von zusätzlichen Programmbibliotheken (ausgenommen POSIX Threads und Linux Sockets) implementiert. Vielmehr wurden in dieser Arbeit Module entwickelt, die gemeinsam von *ClientControl* und *ServerControl* benutzt werden. Lediglich *SEC-Frontend* wurde aufgrund der grafischen Benutzeroberfläche zwar in C++, aber unter Verwendung des Qt SDKs [QP12] entwickelt.

Ein weiterer wichtiger Gesichtspunkt, unter dem *SEC* entwickelt wurde, ist die fast grenzenlose Möglichkeit an Erweiterungen für eine Experimentsteuerung. Es ist sehr schwer eine Experimentsteuerung zu entwickeln, die jedes beliebige Experiment und dessen Anforderungen ohne Erweiterung des Steuerungsprogramms (sei es nur durch die Einbindung von Bibliotheken oder Plugins) berücksichtigt. Aus diesem Grund wurde viel Wert auf die Erweiterbarkeit von *SEC* gelegt. Fast alle Aufgaben der Experimentsteuerung wurden in separaten Modulen implementiert, um eine leichte Erweiterung dieser Module zu ermöglichen. Aus diesem Grund fiel die Wahl bei der Programmiersprache auf C++, da diese Sprache die Möglichkeit

zur Objektorientierung bietet, aber immer noch weniger Ressourcen als beispielsweise Java benötigt. Zwar ist das Konzept der Objektorientierung mit Einschränkungen auch in C umsetzbar, dies würde jedoch die einfache Erweiterbarkeit der Software einschränken.

In Kapitel 3 wurden die Aufgaben und Funktionsweisen der beiden Programme *ClientControl* und *ServerControl* erläutert. Im Folgenden soll nun auf die Implementierung der wichtigsten Module eingegangen werden, da diese grundlegende Funktionen für alle drei Teilprogramme in *SEC* bereitstellen. Diese Module stellen *SEC* jeweils Multithreading, Netzwerkfunktionen und einen XML Parser zur Verfügung.

4.1. Threading

Der C++11 Standard (ISO/IEC 14332:2011) beinhaltet inzwischen eine benutzerfreundliche Schnittstelle zur Verwendung von Threads. Der C++11 Standard ist jedoch in den wenigsten Compilern vollständig implementiert. Aus diesem Grund wurde *SEC* nach dem alten C++03 Standard (ISO/IEC 14332:2003) implementiert, der jedoch keine Standardbibliotheken zur Verwendung von Threads beinhaltet. *SEC* benötigt jedoch Nebenläufigkeit, um mehrere Teilnehmer parallel bedienen zu können. Zu diesem Zweck wurde für *SEC* eine abstrakte Klasse implementiert, welche die Verwendung von POSIX Threads (*pThreads*) mit C++ vereinfacht. POSIX Threads wurden ursprünglich zur Verwendung in C entwickelt und sehen deshalb keine direkte Möglichkeit zur objektorientierten Programmierung vor, die in *SEC* jedoch erwünscht ist, da sie an vielen Stellen die einfache Erweiterbarkeit und Wiederverwendung eines Softwaremoduls fördern kann.

4.1.1. Die Klasse Thread

Die Klasse Thread soll das Benutzen von nebenläufigen Programmen in Kombination mit Objektorientierung in C++ vereinfachen. Diese Klasse ist absichtlich abstrakt, damit beliebige Klassen als Thread benutzt werden können. Im Folgenden soll der wesentliche Aufbau der Klasse Thread erläutert werden. Im Abschnitt 4.1.2 wird die Nutzung der Klasse anhand eines Beispiels beschrieben.

Thread::start()

Eine Klasse, die von der Klasse Thread abgeleitet ist, soll in der Regel nebenläufig ausgeführt werden. Wann immer eine Instanz solch einer Klasse erstellt wird, kann diese mit der (geerbten!) Methode *start()* gestartet werden (siehe Listing 4.1). Die Klasse Thread erstellt dazu einen *pThread*, dem eine Klassenmethode zur nebenläufigen Ausführung übergeben werden kann. Mit der Klassenvariablen bool *m_isRunning* wird vorher abgefragt, ob die Instanz bereits einen Thread gestartet hat.

Listing 4.1: Thread::start().

```
int Thread::start()
{
    if (m_isRunning == false){
        m_isRunning = true;
        if ( pthread_create(&m_thread, 0, &Thread::startRun, this ) != 0)
        {
            m_isRunning = false;
            return -1;
        }
    }
    return 0;
}
```

Zur weiteren Identifikation des Threads wird seine ID in die Speicheradresse von *m_thread* (Variable vom Typ *pthread_t*) geschrieben. Mit dem zweiten Parameter können dem *pThread* spezielle Eigenschaften übergeben werden. *NULL* wird benutzt, um den *pThread* mit Standardeinstellungen zu starten. Damit der *pThread* weiß, welche Methode er auszuführen hat, wird ihm die Adresse der statischen Methode *startRun()* übergeben. Dieser Funktion wird nur ein Argument vom Typ void* übergeben. Damit der *pThread* weitere Eigenschaften der Klasse Thread benutzen kann (zum Beispiel Zugriff auf die Klassenvariable *m_isRunning*), wird der Methode *startRun()* eine Referenz auf die Instanz der Klasse Thread übergeben, die *pthread_create()* aufgerufen hat.

Thread::startRun()

Die Methode *startRun()* wird dem *pThread* bei seiner Erstellung übergeben, damit sie nebenläufig ausgeführt werden kann. Dazu muss die Methode im Header Thread.h als statisch deklariert werden. Die Methode *startRun()* wird im Listing 4.2 gezeigt.

Dieser Methode kann eine Referenz auf ein Objekt vom Typ void als Parameter übergeben werden. Durch diese Besonderheit wird dem Zeiger *obj* kein Datentyp zugeordnet, sodass *obj* ein Zeiger auf ein beliebiges Objekt im Speicher sein kann. Mit dem C++ Type Cast *reinterpret_cast <Thread *>(obj)* wird *obj* als Zeiger vom Typ *Thread* verwendet. Nun kann auch auf Klassenvariablen und Klassenmethoden von einem *pThread* aus zugegriffen werden.

Listing 4.2: Thread::startRun().

```
static void *startRun(void *obj);

void* Thread::startRun(void* obj)
{
    reinterpret_cast<Thread *>(obj)->run();
    reinterpret_cast<Thread *>(obj)->m_isRunning = false;
    pthread_detach(pthread_self());
    pthread_exit(NULL);
}
```

In der Methode *start()* (siehe Listing 4.1) wird der Thread als laufend markiert (*m_isRunning = true*), damit keine weiteren *pThreads* von diesem Thread erstellt werden können. Im Anschluss wird die Methode *run()* aufgerufen, in der die Anweisungen implementiert sind, die der *pThread* ausführen soll. Nachdem die Methode zurückgesprungen ist, wird der Thread als nicht laufend markiert. Mit *pthread_detach(pthread_self())* wird dem laufenden *pThread* (*pthread_self()*) mitgeteilt, dass er all seine Ressourcen nach seiner Terminierung freigeben soll. Mit *pthread_exit(NULL)* wird der laufende *pThread* beendet.

Thread::run()

Diese Methode wird im Header Thread.h als abstrakt deklariert, wodurch die gesamte Klasse Thread zur abstrakten Klasse wird. Somit muss *run()* von einer erbenden Klasse implementiert werden, die dadurch selbst bestimmen kann, was der *pThread* nebenläufig ausführen soll.

Thread::stop()

Normalerweise wartet ein Elternprozess nicht, bis seine Kindprozesse beendet wurden, wenn er selbst beendet wird. Ein Elternprozess muss jedoch solange angehalten werden können, bis jeder Thread seine Ressourcen freigegeben hat. Zu diesem Zweck sollte die Methode *stop()* vom Destruktor eines Thread Objektes aufgerufen werden:

Listing 4.3: Thread::stop().

```
void Thread::stop()
{
    if (m_isRunning == true) pthread_join(m_thread, 0);
}
```

Wenn der *pThread* des Objekts noch läuft, wird der Destruktor von diesem zum Anhalten gezwungen, bis der Thread von sich aus beendet wird. Die Funktion *pthread_join()* blockiert dazu den Elternprozess solange, bis der *pThread m_thread* mit *pthread_exit(NULL)* beendet wurde.

4.1.2. Beispiel zur Nutzung

Die Nutzung der Klasse Thread wird im Folgenden anhand eines einfachen Beispiels demonstriert (siehe Listing 4.4). Ein Programm (Elternprozess) erstellt drei SimpleThreads (Kindprozesse), die nebenläufig ausgeführt werden und dabei jeweils eine Variable runterzählen und auf der Konsole ausgeben. Die Klasse Thread vereinfacht die Nutzung von *pThreads* soweit, dass die im Listing 4.4 implementierten SimpleThreads fast schon wie Threads in Java benutzt werden.

Listing 4.4: Beispiel zur Nutzung der Klasse Thread.

```
/* Begin File: main.cpp */
#include <iostream>
#include "SimpleThread.h

using namespace std;

int main(int argc, char *argv[])
{
    SimpleThread thread1 = SimpleThread();
    SimpleThread thread2 = SimpleThread();
    SimpleThread thread3 = SimpleThread();

    thread1.start();
    thread2.start();
    thread3.start();

    return 0;
}
/* End File: main.cpp */


/* Begin File: SimpleThread.h */
#include <iostream>
#include <string>
#include "Thread.h"

using namespace std;

class SimpleThread : public Thread
{
    private:
        void run();

    public:
        SimpleThread();
        virtual ~SimpleThread();
};
/* End File: SimpleThread.h */
```

```
/* Begin File: SimpleThread.cpp */
#include "SimpleThread.h"

SimpleThread::SimpleThread() : Thread(0)
{
}

SimpleThread::~SimpleThread()
{
    Thread::stop();

    // Thread::stop() muss hier aufgerufen werden,
    // damit dieser Konstruktor wartet, bis der Thread
    // durchgelaufen ist. Sonst wuerde SimpleThread::run()
    // geloescht werden, obwohl Thread::startRun() diese
    // Methode noch benutzt durch den Aufruf:
    // reinterpret_cast<Thread *>(obj)->run();
}

void SimpleThread::run()
{
    int i=100000;

    while (i>0){
        cout << i << endl;
        i--;
    }
}
/* End File: SimpleThread.cpp */
```

4.2. Netzwerk Programmierung

Ebenso wichtig wie die Nebenläufigkeit durch Threads ist die Kommunikation zwischen Server und Clients über das von ihnen verwendete Netzwerk. Da die Netzwerkprogrammierung für *SEC* essentiell ist, soll hier näher auf das für *SEC* entwickelte Modul *NetHost* eingegangen werden, welches aus den Dateien NetHost.h und NetHost.cpp besteht. Für die Socketprogrammierung bringt der in *SEC* verwendete C++03 Standard (ISO/IEC 14332:2003) keine Standardbibliotheken mit sich. *NetHost* soll deshalb die Verwendung der Linux Socket API in C++ vereinfachen, weil diese ursprünglich für C entwickelt wurde.

4.2.1. NetHost

Dieses Modul stellt im Wesentlichen zwei Klassen zur Verfügung. Die Klasse *Client* wird in *SEC* verwendet, um die clientseitige Kommunikation in einem Netzwerk bereitzustellen. Die Klasse *Server* vereinfacht die Verwendung von serverseitigen Sockets. Beide Klassen lassen sich sowohl für TCP Verbindungen, als auch für UDP Pakete verwenden. Die Verwendung und Implementierung des Moduls soll im Folgenden anhand eines einfachen TCP Echo Servers und einer zugehörigen Clientanwendung erklärt werden. Auf die Bedeutungen einzelner Befehle der Linux Socket API wird hier nur oberflächlich eingegangen. Ein tieferer Einblick in die Linux Socket API wird in [SFR03] gegeben.

4.2.2. TCP Client

Das Listing 4.5 zeigt, wie ein einfacher TCP Echo Client unter der Verwendung des Moduls *NetHost* aussehen könnte. Bei der Instanziierung eines neuen Clients wird diesem mitgeteilt, ob eine Kommunikation über TCP oder über UDP eingerichtet werden soll. Außerdem werden dem Konstruktor IP Adresse und Port der Serveranwendung übergeben. Der Konstruktor initialisiert das neue Clientobjekt, indem er diese Parameter an die Methode *init()* weitergibt. Diese Methode fordert mit dem Befehl *m_socket = socket(AF_INET, SOCK_STREAM, 0)* ein Socket beim Betriebssystem an und speichert den Rückgabewert in *m_socket* vom Typ Integer ab.

Um UDP benutzen zu können, wird *SOCK_STREAM* automatisch durch *SOCK_DGRAM* ersetzt. In einem Struct (hier *sockaddr*) werden IP Adresse und Port der Serveranwendung gespeichert, mit der sich das Clientobjekt verbinden soll. Wenn die Kommunikation über TCP erfolgen soll, versucht die Funktion *connect(m_socket, (struct sockaddr*)&m_serverAddress, sizeof(struct sockaddr))* den Socket *m_socket* mit dem Struct sockaddr zu verknüpfen und eine Verbindung zum Serverprogramm aufzubauen. Wenn die Verbindung erfolgreich aufgebaut ist, kann die Kommunikation zwischen Server und Client stattfinden.

Listing 4.5: Beispiel: TCPClient.

```
#include <string>
#include <iostream>
#include "NetHost.h"

using namespace std;

int main(int argc, char *argv[])
{
    // TCP ECHO CLIENT
    string text;
    Client tcpEchoClient = Client(TCP,"127.0.0.1",6789);
    while(true)
    {
        cout << "Eingabe:" << endl;
        cin >> text;
        if (text.compare("q") == 0) break;
        tcpEchoClient.sendString(text);
        if (tcpEchoClient.recvString(&text) < 1 ) break;
        cout << "Server:" << endl;
        cout << text << endl;
    }
}
```

Die Klasse *Client* stellt zwei Methoden zur Übertragung von Strings bereit. Mit *sendString()* werden Zeichenketten im ASCII Format zum Server gesendet und mit*recvString()* wird das Programm angehalten, bis eine Nachricht vom Client empfangen wird. Beide Methoden basieren auf einfacheren Methoden zum Senden und Empfangen von Bytes. Der Methode *sendBytes()* wird eine Referenz auf eine Bytekette und deren Länge übergeben. Die Bytekette wird mit *send()* über den Socket *m_socket* an den Server gesendet. Bei einer Übertragung

mit UDP wird *send()* durch *sendto()* ersetzt. Dazu äquivalent wird der Methode *recvBytes()* ein Zeiger auf ein Stück Speicher übergeben, das vom Socket Befehl *recv()* überschrieben wird, sobald eine Bytekette empfangen werden kann. Wenn eine Verbindung zu einem Server abgebaut wurde, gibt *recvBytes()* 0 zurück. Sollte ein Fehler in der Verbindung auftreten, geben *recvBytes()* und *sendBytes()* -1 zurück. Die Rückgabewerte werden von *sendString()* und *recvString()* weiter gereicht. Aus diesem Grund wird das Programm hier beendet, wenn *recvString()* einen Wert kleiner als eins zurückgibt. Die Funktion *sendString()* überträgt festgelegte Blöcke von 1400 Bytes, wodurch Zeichenketten auf 1400 Zeichen (inklusive *terminierende Null*) begrenzt sind. Zeichenketten, die zu lang sind, werden nach 1399 Zeichen abgeschnitten, damit die String terminierende Null das letzte Byte belegen kann. Die Funktion *recvString()* erwartet dementsprechend einen 1400 Byte großen Block, dessen Inhalt in einen String kopiert wird.

Im Destruktor der Klasse Client wird die Verbindung über den Socket *m_socket* geschlossen, wenn diese nicht schon vorher mit *closeConnection()* geschlossen wurde. Dazu wird zuerst *shutdown(m_socket, SHUT_RDWR)* aufgerufen, damit das Senden oder Empfangen von weiteren Nachrichten über den Socket *m_socket* deaktiviert wird. Anschließend wird der Socket *m_socket* ordentlich mit close(m_socket) geschlossen.

4.2.3. TCP Server

Das Listing 4.6 zeigt einen einfachen TCP Echo Server. Das Serverprogramm unterscheidet sich unter Verwendung des *NetHost* Moduls nicht sonderlich vom oben beschriebenen Clientprogramm. Auffällig sind hier die Befehle *acceptClient()* und *closeConnection()*.

Listing 4.6: Beispiel: TCPServer.

```
#include <string>
#include <iostream>
#include "NetHost.h"

using namespace std;

int main(int argc, char *argv[])
{

    // TCP ECHO SERVER

    Server tcpEchoServer = Server(TCP, 6789);

    string text,ip;
    int port;
    int connection = tcpEchoServer.acceptClient(&ip,&port);
    cout << "accept " << ip <<":"<< port <<endl;
    while(true)
    {
        if (tcpEchoServer.recvString(&text,connection) < 1 )
        {
            break;
        }
        tcpEchoServer.sendString(text,connection);
    }
    tcpEchoServer.closeConnection(connection);
}
```

Bei der Instanziierung eines Serverobjekts leitet der Konstruktor zwei Parameter an die Methode *init()* weiter. Die Parameter geben an, ob es sich um eine Kommunikation über TCP oder UDP handelt und auf welchem Port das Serverprogramm auf Clients warten soll. Wie auch in

der Klasse Client wird zunächst ein Socket beim Betriebssystem angefordert und ein Struct mit Informationen über die IP Adresse und den Port des Servers angelegt. Zusätzlich kann dem Struct eine IP Adresse übergeben werden, die an den angeforderten Socket gebunden wird. Die Klasse Server nutzt dafür die Einstellung *INADDR_ANY*, sodass der Socket an alle möglichen IP Adressen des Servers gebunden wird. Anders ist bei einem Serverobjekt, dass die Adressinformationen nun mit *bind()* an den angeforderten Socket gebunden werden. Wenn TCP benutzt wird, muss der Socket mit dem Befehl *listen()* aufgefordert werden, eingehende Verbindungen zu akzeptieren. Weil UDP verbindungslos ist, muss bei dessen Nutzung keine Verbindungen angenommen werden, weshalb *listen()* in diesem Fall nicht aufgerufen wird.

Das Serverobjekt wartet nun bei der Benutzung von TCP auf eingehende Verbindungen, die mit *acceptClient()* angenommen werden. Die Methode gibt einen Integer Wert zurück, der vom Betriebssystem für die Verbindung (*connection*) zwischen Server und Client vergeben wird (Eine TCP Verbindung wird stets durch das Viertupel (Server IP, Server Port, Client IP, Client Port) identifiziert). Per Referenz teilt *acceptClient()* dem Aufrufer die IP Adresse und den Port des Client mit. Die Nummer der Verbindung (*connection*) muss bei einem Aufruf von *sendString()* und *recvString()* zusätzlich angegeben werden. Anders als ein Clientobjekt, kann das Serverobjekt schließlich zwischen mehreren Verbindungen zur Kommunikation auswählen. Weil das Serverobjekt die Verbindungen nicht speichert, sondern nur entgegen nimmt, muss jede Verbindung vor Beendigung des Programms explizit mit *closeConnection()* geschlossen werden. Mit *shutdown()* wird die ausgewählte Verbindung zunächst abgebaut, sodass sie daraufhin mit *close()* geschlossen werden kann.

4.2.4. Nachrichten in SEC

Die Kommunikation in *SEC* findet hauptsächlich über 8 Byte große Nachrichten statt. Dafür stellt das Modul *NetHost* für Clientprogramme die Methoden *sendSecMsg()* und *recvSecMsg()* bereit. Für Serverprogramme stellt *NetHost* die Methoden *sendSecMsg()*, *recvSecMsg()*, *sendSecMsgTo()* und *recvSecMsgFrom()* bereit. Die Methoden senden oder empfangen jeweils eine 4 Byte große Nachricht (*key*) und einen dazugehörigen 4 Byte großen Wert (*value*). Für die Versendung von Dateien stellt das Modul *NetHost* zusätzlich die Methoden *sendFile()* und *recvFile()* bereit, mit deren Hilfe die Experimentdateien verteilt werden.

Eine Übersicht der Nachrichten, die zwischen dem Server und den Clients ausgetauscht werden, geben die Tabellen 4.1 und 4.2.

Tabelle 4.1.: Nachrichten vom Server an den Client.

Key	Value	Bedeutung
0x00000210	0x00000000	Anmeldung des Clients bestätigen
0x00000220	PORT	UDP Port für das Experiment mitteilen
0x00000230	ID	ID für das Experiment mitteilen (danach XML Dateien senden)
0x00000240	0x00000000	Anweisung UDP Kommunikation zu testen
0x00000250	STARTZEIT	Startzeit senden
0x00000260	0x00000000	Anweisung TCP Verbindung zu beenden
0x00000270	ENDZEIT	Experiment abbrechen
0x00000280	0x00000000	Anmeldung des Clients nach dem Experiment bestätigen
0x00000290	0x00000000	Erhalt der Logdatei bestätigen

Tabelle 4.2.: Nachrichten vom Client an den Server.

Key	Value	Bedeutung
0x00000110	0x00000000	Beim Server anmelden
0x00000120	0x00000000	Aufgabe erhalten und Messsoftware ist vorhanden
0x00000130	0x00000000	Messsoftware ist nicht vorhanden
0x00000140	ID	Teste UDP Kommunikation
0x00000150	0x00000000	Startzeit bestätigen
0x00000160	ID	Sende ID während des Experiments
0x00000170	0x00000000	Anmeldung beim Server nach dem Experiment

4.2.5. Kommunikation mit einem Messprogramm

SEC wurde zur Steuerung beliebiger Messprogramme entwickelt, die der Benutzer selbst implementieren kann. *ClientControl* erwartet von einem Messprogramm, dass es eine Serveranwendung zur Kommunikation über TCP bereit stellt. *ClientControl* versucht sich als Client

über das Loopback Interface bei der Messsoftware anzumelden. Deshalb kann jedes Messprogramm, das in einem Experiment verwendet werden soll, in der Experimentdatei mit dem <*include*> Befehl eingebunden werden. Zusätzlich muss der Port, auf dem das Serverprogramm der Messsoftware lauscht, in dieser XML Datei angegeben werden.

Vor dem Beginn eines Experiments teilt *ClientControl* der Messsoftware den Start des Experiments mit. Dazu sendet *ClientControl* die Zeichenkette *Experiment Start* als String an das Messprogramm. Die erfolgreiche Durchführung des Experiments (*Experiment Done*) oder der Abbruch des Experiments (*Experiment Abort*) wird dem Messprogramm ebenfalls mitgeteilt.

In der Experimentdatei können Befehle für jedes benötigte Messprogramm angegeben werden. Das Kapitel 3.4.2 beschreibt, wie Messprogramme mit dem XML *Include* Befehl eingebunden werden und über einen selbst definierten XML Tag angesprochen werden können. Zur Ausführungszeit eines Befehls wird dieser als String mit Hilfe der Methode *sendString()* aus dem Modul *NetHost* an die Messsoftware gesendet. Diese muss selbst entscheiden, wie der Befehl aus der XML Datei interpretiert werden soll. Da die Methode *sendString()* genau 1400 Zeichen vom Typ char sendet, ist der Befehl an die Messsoftware auf 1400 ASCII Zeichen (inklusive terminierende Null) begrenzt.

4.3. Der XML Parser

Wie bereits in Kapitel 3 beschrieben werden Experimente in *SEC* durch XML Dateien konfiguriert und beschrieben. Das XML Format ist für die Beschreibung und Konfiguration deshalb gut geeignet, weil es menschenlesbar ist, gut strukturierbar ist und von vielen Editoren unterstützt wird. Für *SEC* wurde zu diesem Zweck die *xmlread API* entwickelt. Sie ist eine vereinfachte Form einer Simple API for XML (*SAX*), mit deren Hilfe selbst definiert werden kann, wie ein XML Parser auf bestimmte XML Elemente reagieren soll. Im Gegensatz zu DOM Parsern bauen SAX Parser keine Abbildung des XML Dokuments im Speicher auf. SAX Parser lesen XML Dateien sequentiell aus, weshalb sie auch für leistungsschwächere Computer geeignet sind.

4.3.1. Die xmlread API

Die API wird durch die abstrakte Klasse *xmlread* definiert und kann von einer beliebigen Klasse durch Vererbung erweitert werden, um einen XML Parser zu erstellen. Dazu vererbt xmlread der erbenden Klasse die Methode *parse()*, der eine zu parsende XML Datei im *ASCII* Format als Argument übergeben werden muss. Wenn *parse()* aufgerufen wird, führt *xmlread* einen endlichen Zustandsautomaten aus, der je nach eingelesenem Zeichen der geöffneten XML Datei seinen Zustand ändert. Im Anhang A wird der Zustandsautomat der *xmlread API* ausführlich anhand der Darstellung A.1 erklärt. Der *TaskDivider* aus Kapitel 3.3.6 implementiert die *xmlread API* beispielsweise, um ein Experiment in Aufgaben zu unterteilen.

Jedes Mal, wenn *xmlread* eine vorgegebene XML Struktur erkennt, wird eine abstrakte Methode aufgerufen, die von einer erbenden Klasse für genau diese XML Struktur implementiert werden muss. Escape Sequenzen werden von *xmlread* nicht erkannt. Die Tabelle 4.3 beschreibt, welche absrakten Methoden bei welchen XML Strukturen aufgerufen werden.

Tabelle 4.3.: XML Strukturen.

XML Struktur	Aufgerufene Methode
<?xml ?>	onDeclareTag()
<?xml attribut=“wert“ ?>	
<?xml $attribut_1$=“$wert_1$“ ... $attribut_n$=“$wert_n$“ ?>	
<name>	onOpenTag()
<name attribut=“wert“>	
<name $attribut_1$=“$wert_1$“ ... $attribut_n$=“$wert_n$“>	
<name />	onShortTag()
<name attribut=“wert“ />	
<name $attribut_1$=“$wert_1$“ ... $attribut_n$=“$wert_n$“ />	
</name>	onCloseTag()
<!– ... –>	onComment()
<tag> ... TEXT ... </tag>	onText()
XML Datei kann nicht geöffnet werden	onFileError()
Fehler innerhalb XML Datei (nicht erkannte Struktur)	onError()
XML Datei wurde erfolgreich gelesen	onSuccess()

4.3.2. Parameterübergabe

Je nach XML Struktur werden den abstrakten Methoden Parameter übergeben, die von der erbenden Klasse bei der Implementierung dieser Methoden genutzt werden können. Der Tabelle 4.3.2 ist beispielsweise zu entnehmen, dass bei einem XML Start Tag die Methode *onOpenTag()* der erbenden Klasse aufgerufen wird. Der übergebene Parameter *name* enthält den Namen des Tags als String. In *properties* sind alle Attribute des Tags sowie deren Werte gespeichert. Dabei folgt auf jedes Attribut der zugehörige Wert (*Value*). Alle Einträge werden duch Leerzeichen getrennt. Wenn keine Attribute im XML Tag angegeben wurden, bleibt *properties* leer. Sollte ein Wert (*Value*) selbst ein Leerzeichen enthalten, wird dieses durch einen Unterstrich ersetzt. Bei Texten oder Kommentaren wird über den Parameter *text* ein ganzer Text übergeben.

4.4. SEC-Frontend

Das Programm *SEC-Frontend* wird auf dem Masterclient ausgeführt, um dem Benutzer die Versendung eines Experiments zu vereinfachen. Zur vernünftigen Darstellung der GUI benötigt Frontend mindestens Qt3 [QP12].

Im Folgenden soll *SEC-Frontend* anhand eines Beispiels erklärt werden. In diesem Beispiel wird ein Experiment an den Server gesendet, auf dem das Programm *ServerControl* läuft.

4.4.1. Ein Experiment mit SEC-Frontend an den Server senden

Zunächst muss eine Verbindung zu *ServerControl* aufgebaut werden. Durch anklicken des Menüleisteneintrags *Server* → *Connect to Server* öffnet sich ein Dialog (siehe Abbildung 4.1), der die IP Adresse des Servers und den Port, auf dem *ServerControl* wartet, entgegen nimmt.

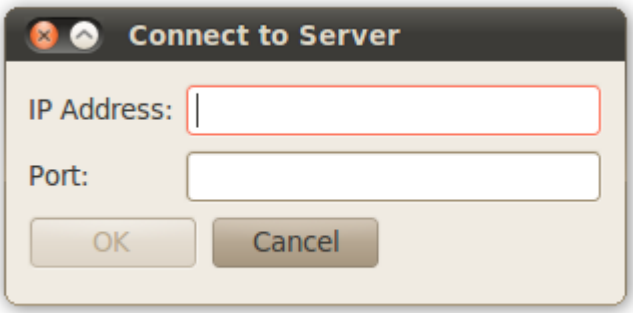

Abbildung 4.1.: Der *SEC-Frontend* Connect Dialog.

Wenn *SEC-Frontend* mit dem Server verbunden ist, wird dies im *Statusmonitor* mit einer Statusnachricht bestätigt. Die Grafik 4.2 zeigt das Hauptfenster der *SEC-Frontend* Benutzeroberfläche.

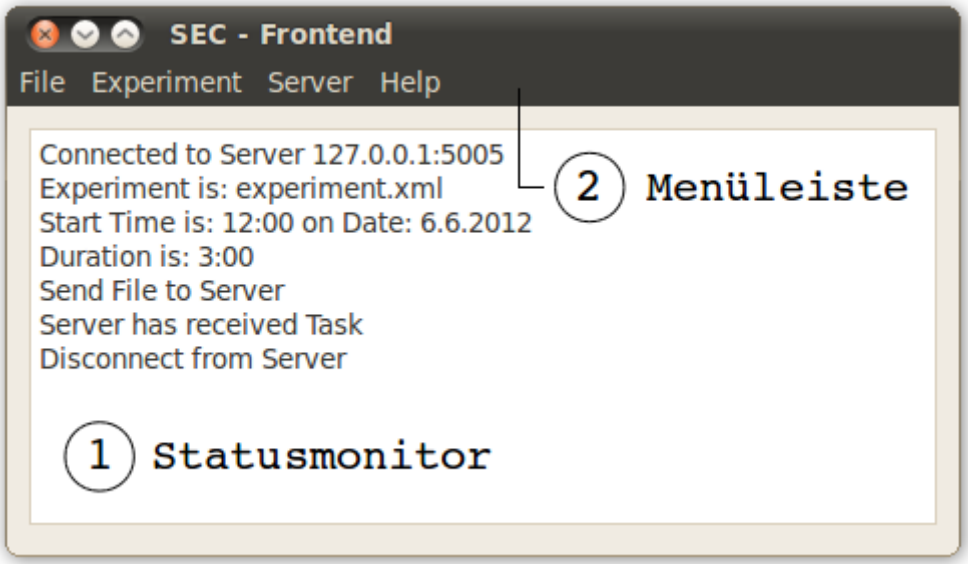

Abbildung 4.2.: Die *SEC-Frontend* Benutzeroberfläche.

Unter dem Menüleisteneintrag *Experiment* → *New Experiment* → *set Experiment* kann die Experimentdatei ausgewählt werden, die den zeitlichen Ablauf des Experiments enthält (siehe Auswahldialog in Abbildung 4.3). Der Abschnitt 3.4 beschreibt den Aufbau der Experimentdatei ausführlich.

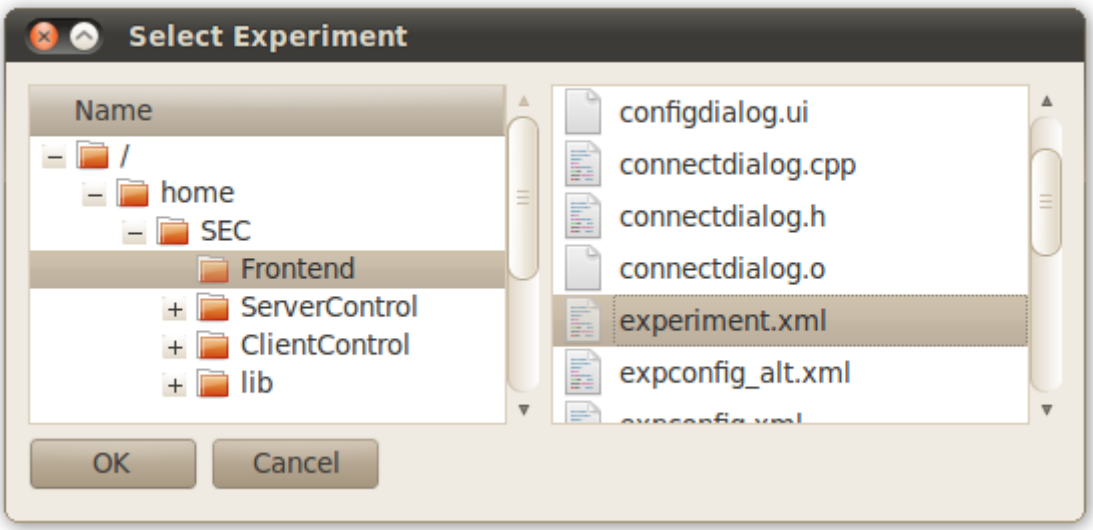

Abbildung 4.3.: Der *Frontend* Select Dialog.

Nachdem die Experimentdatei ausgewählt wurde, müssen einige Konfigurationen (wie Startzeitintervall und Datum) vorgenommen werden. Dazu wird unter dem Menüleisteneintrag *Experiment* → *New Experiment* → *set Config* der Konfigurationsdialog für das Experiment geöffnet (siehe Abbildung 4.4). Der Dialog erstellt die Konfigurationsdatei für die zuvor ausgewählte Experimentdatei (siehe Abschnitte 3.4.1 und 3.4.2). Die Konfigurationsdatei wird im selben Verzeichnis angelegt, in dem *SEC-Frontend* ausgeführt wird.

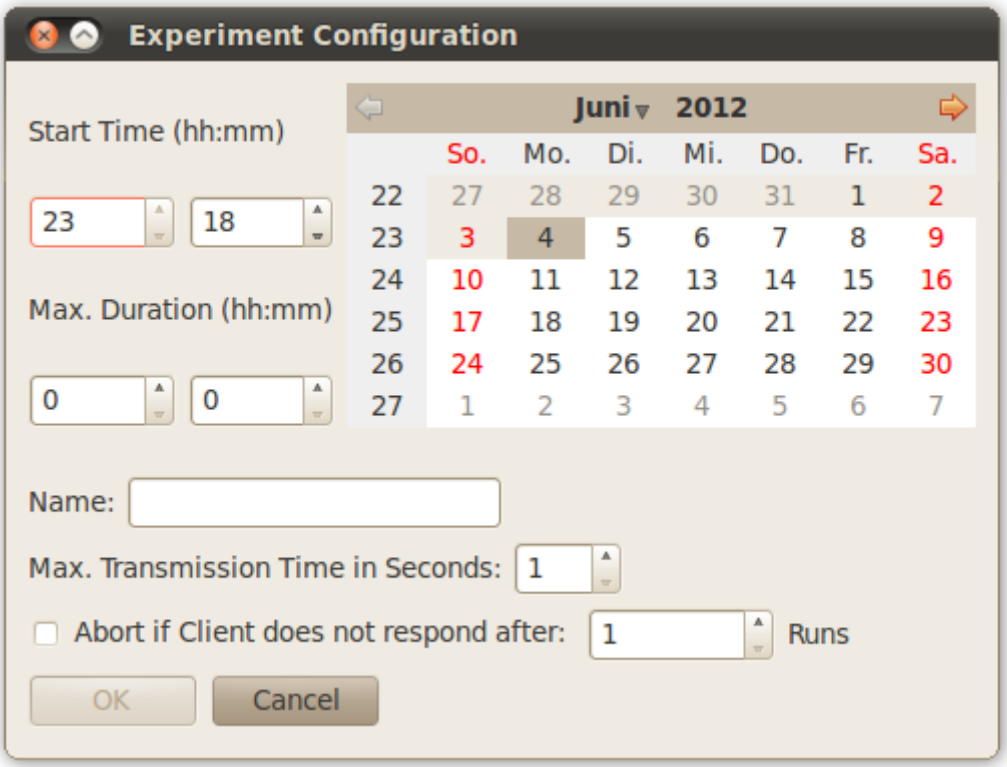

Abbildung 4.4.: Der *Frontend* Config Dialog.

Erst wenn eine XML Datei mit dem Experiment ausgewählt wurde, die Konfiguration für das Experiment durchgeführt wurde und *SEC-Frontend* mit einem Server verbunden ist, kann das Experiment über den Menüleisteneintrag *Experiment* → *Start Experiment* an den Server gesendet werden. Wenn der Server das Experiment erhalten hat, wird dies im *Statusmonitor* durch eine Statusnachricht angezeigt.

Kapitel 5.

Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurde *SEC* (*Stateful Experiment Control*), ein Programm zur automatischen Steuerung von Experimenten, entwickelt und implementiert. Ein Experiment wird von *SEC* selbstständig an die Teilnehmer verteilt, ohne die Überwachung durch einen Benutzer zu erfordern. Die Experimentsteuerung wurde hauptsächlich zur Durchführung von Mobilfunkmessungen mit mehreren mobilen Messgeräten entwickelt. Die Kommunikation zwischen diesen findet dabei über das zu vermessende Mobilfunknetz statt. Aus diesem Grund werden Experimente in *SEC* in Abschnitte unterteilt, zwischen denen die Teilnehmer Statusnachrichten senden können, damit die eigentliche Mobilfunkmessung so wenig wie möglich beeinflusst wird. Das verwendete Protokoll der Experimensteuerung berücksichtigt und toleriert die typischen Eigenschaften eines fehleranfälligen Kanals, wie beispielsweise häufige Paketverluste oder hohe Latenzen.

Die Beschreibung und Konfiguration eines Experiments erfolgt in *SEC* über zwei XML Dateien. In der Experimentdatei wird für jeden Teilnehmer festgelegt, wann dieser welchen Befehl auszuführen hat. Die separate Konfigurationsdatei enthält grundlegende Einstellungen, wie zum Beispiel die Startzeit des Experiments. Eine Serveranwendung (*ServerControl*) teilt das Experiment in einzelne Aufgaben für die teilnehmenden Clients und verteilt die Aufgaben an diese. ServerControl ist in der Lage mehrere Experimente parallel zu steuern, wenn sich vorher genügend Clients beim Server angemeldet haben.

Die clientseitige Software (*ClientControl*) meldet sich beim Server an und wartet, bis ihr eine Aufgabe zur Durchführung übergeben wird. ClientControl selbst führt jedoch keine Messun-

gen im eigentlichen Sinne durch. Der Benutzer wählt die Messprogramme, die während eines Experiments benutzt werden sollen selbst aus. ClientControl ist dafür verantwortlich, das jedes benutzte Messprogramm zur vorgegebenen Zeit den Befehl erhält, der in der Experimentdatei hinterlassen wurde. Auf diese Art bleibt der Benutzer bei der Wahl der Messsoftware flexibel und kann selbst bis ins kleinste Detail vorgeben, welche Informationen er bei einem Experiment gewinnen möchte.

Obwohl *SEC* mit Rücksicht auf Mobilfunkmessungen entwickelt wurde, kann die Experimentsteuerung prinzipiell auch für andere Messungen eingesetzt werden. Dabei muss nicht einmal ein Experiment gesteuert werden. *SEC* kann im weitesten Sinne auch als Ablaufsteuerung betrachtet werden, die zeitlich vorgegebene Aufgaben erfüllt.

Weitere wichtige Aspekte, unter denen *SEC* entwickelt wurde, sind Portabilität und Erweiterbarkeit. Für die Entwicklung von ClientControl und ServerControl wurden neben den C++ Standardbibliotheken, mit Ausnahme der Linux Socket API und POSIX Threads, keine externen Programmbibliotheken verwendet, damit *SEC* auf möglichst vielen Systemen eingesetzt werden kann.

5.1. Ausblick

Die in dieser Bachelorarbeit implementierte Experimentsteuerung *SEC* wurde hauptsächlich zur Durchführung von Mobilfunkmessungen mit mobilen Messgeräten entwickelt. Trotzdem beschränkt sich die Nutzung von *SEC* nicht auf diese Messgeräte. Gerade weil *SEC* vielseitig einsetzbar ist, sind einige Ideen entstanden, die aufgrund zeitlicher Vorgaben nicht in *SEC* implementiert wurden, an dieser Stelle jedoch erwähnt werden sollen.

Derzeit kann in *SEC* nicht vorgegebenen werden, welche der verfügbaren Kommunikationsschnittstellen des Systems verwendet werden soll. Interessant wäre die Erweiterung von *SEC* um die Möglichkeit, verschiedene Netzwerkadapter simultan verwenden zu können. So könnte zum Beispiel die gesammte Kommunikation der Teilnehmer über IEEE 802.3 (Ethernet) stattfinden, während die Messprogramme Informationen über ein Mobilfunknetz mit Hilfe eines UMTS Moduls sammeln.

In *SEC* wird der Ablauf eines Experiments durch eine Experimentdatei im XML Format beschrieben. Dabei wurde darauf geachtet, dass ein Experiment ohne großen Aufwand von Hand strukturiert und erstellt werden kann. *SEC* könnte mit einem Editor zum Erstellen von Experimentdateien erweitert werden. Praktisch wäre die grafische Darstellung und auch Simulation eines Experiments, um dem Nutzer die Planung eines Experiments zu vereinfachen.

An letzter Stelle soll hier die Erweiterung der Experimentkontrolle durch Prioritätenvergabe erwähnt werden. Bisher werden Clients demjenigen Experiment zugeteilt, welches diese zuerst für sich beansprucht hat. Praktisch wäre an dieser Stelle die Fähigkeit, einem Experiment bevorzugt Clients mit bestimmten Eigenschaften (zum Beispiel Standort oder Art des Gerätes) zu zuweisen, damit die Aussagekraft jedes Experiments gesteigert werden kann.

Anhang A.

Der xmlread Zustandsautomat

Im Folgenden werden Methoden beschrieben, die in den Zuständen des *xmlread* Zustandsautomaten aufgerufen werden. (siehe Abbildung A.1) Nach dem Rücksprung aus diesen Methoden entscheidet der Automat anhand der nächsten eingelesenen Zeichen (im Folgenden *Eingabe* genannt), in welchen Zustand er wechseln soll. Der Zustandsautomat wird beendet, wenn die XML Datei vollständig ausgelesen wurde oder ein Fehler in der XML Struktur erkannt wurde. Im ersten Fall wird die abstrakte Methode *onSuccess()* aufgerufen, um den erfolgreichen Lesevorgang zu bestätigen. Im Falle eines Fehlers wird die abstrakte Methode *onError()* aufgerufen und der Zustandsautomat wird beendet.

xmlread::charIsText()

Jedes Mal, wenn der Zustandsautomat einen Text als Eingabe erwartet, ruft er die Methode *charIsText()* auf. Dieser Methode wird das nächste eingelesene Zeichen übergeben, damit entschieden werden kann, ob das Zeichen als Text oder als XML Steuerzeichen gewertet werden soll. Wenn das eingelesene Zeichen kein ASCII Steuerzeichen (ASCII Code 0 bis 31 und ASCII Code 127) oder eines der Zeichen *!*, *“*, *&*, *’*, */*, <, =, > oder *?* ist, wird der boolsche Wert *false* zurückgegeben. Ansonsten wird das eingelesene Zeichen als Textzeichen gewertet und *charIsText()* gibt *true* zurück.

xmlread::parseSpace()

Diese Methode gibt das nächste Zeichen zurück, das kein ASCII Steuerzeichen oder Leerzeichen ist. ASCII Steuerzeichen und Leerzeichen (ASCII Code 32) werden so übersprungen.

xmlread::parseText()

Wenn die Methode *parseText()* aufgerufen wird, werden alle eingelesenen Zeichen mit *charIsText()* überprüft, ob sie als Text gewertet werden können. Das erste Zeichen, das nicht als Text sondern als XML Steuerzeichen erkannt wurde, wird von *parseText()* zur weiteren Verarbeitung zurückgegeben. Die als Text erkannten Zeichen werden zusätzlich per Referenz als String zurückgegeben.

xmlread::parseAll()

Damit alle Textzeichen zwischen einem XML Start Tag und einem XML End Tag zu einem Text zusammengefasst werden, wird *parseAll()* aufgerufen. Diese Methode fasst alle Zeichen (außer ASCII Steuerbefehle) zu einem String zusammen, bis > eingelesen wird. Die entstandene Zeichenkette wird per Referenz zurückgegeben.

xmlread::parseComment()

Ähnlich wie *parseAll()* gibt diese Methode den Inhalt eines Kommentars per Referenz zurück. Ein Kommentar wird durch einlesen eines doppelten Minuszeichens (--) beendet.

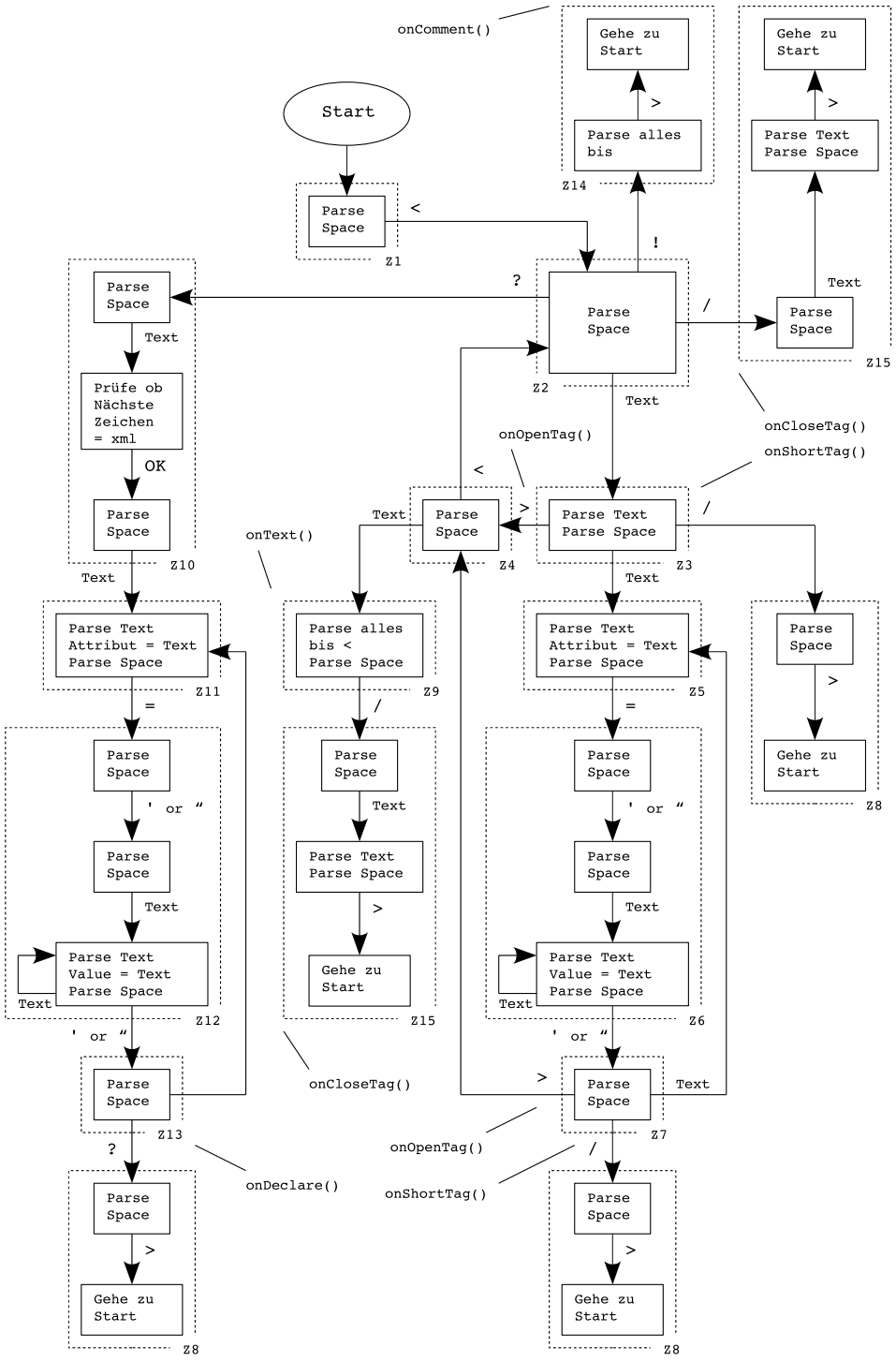

Abbildung A.1.: Zustandsautomat der *xmlread API*.

Anhang B.

Datenträger

Dieser Bachelorarbeit ist ein Datenträger beigefügt, auf dem sich die *SEC* Software und die schriftliche Ausarbeitung der Arbeit im PDF Format befindet.

Literaturverzeichnis

[CC12] C2C-CC: *Car 2 Car Communication Consortium*. http://www.car-to-car.org/. Version: 2012.

[KOM08] KIESS, Wolfgang; OGILVIE, Thomas; MAUVE, Martin: The EXC Toolkit for Real-World Experiments with Wireless Multihop Networks. In: *EXPONWIRELESS 2008: Proceeding of the 3rd Workshop on Advanced Experimental Activities on Wireless Networks and Systems*, Newport Beach, California, USA, 2008.

[NoW12] NOW: *Network on Wheels*. http://www.network-on-wheels.de/. Version: 2012.

[QP12] QT-PROJECT: *Qt*. http://qt.nokia.com/. Version: 2012.

[SFR03] STEVENS, W. R.; FENNER, Bill; RUDOFF, Andrew M.: *Unix Network Programming, Volume 1: The Sockets Networking API, 3rd Edition*. Addison-Wesley Professional, 2003. ISBN 0–13–141155–1

[ST12] SIM-TD: *Sichere Intelligente Mobiliät - Testfeld Deutschland*. http://www.simtd.de/. Version: 2012.

[Sta12a] STATISTISCHES BUNDESAMT DEUTSCHLAND: *Transport und Verkehr*. https://www.destatis.de/DE/ZahlenFakten/Indikatoren/LangeReihen/Verkehr/lrvkr002.html. Version: Mai 2012.

[Sta12b] STATISTISCHES BUNDESAMT DEUTSCHLAND: *Verkehrsunfälle 2011*. https://www.destatis.de/DE/ZahlenFakten/

`Wirtschaftsbereiche/TransportVerkehr/Verkehrsunfaelle/VerkehrsunfaelleAktuell.html`. Version: Mai 2012.

[Zag11] ZAGIDULIN, Stanislav: *Eine In-Band-Experimentsteuerung zur Durchführung von Mobilfunkmessungen*. Bachelor's thesis, September 2011.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 6.Juni 2012 Tobias Amft