Analyse und Simulation von Netzlasten im Bankbereich

Alexander Weimer
Matr. Nr.: 3487055

1. März 2002

Erklärung

Ich versichere, dass ich die beiliegende Diplomarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

(Alexander Weimer)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

In den letzten zwei Jahrzehnten hat die rasante Entwicklung der Computerindustrie dazu geführt, dass immer leistungsfähigere Rechner und Netzwerkkomponenten den Menschen zur Verfügung stehen. Am Anfang dieser Entwicklung führte dieser Fortschritt zum Einsatz von leistungsfähigeren Workstations anstelle der alten Mainframe-Terminal-Systeme. Im letzten Jahrzehnt jedoch fand ein umgekehrter Prozess statt.

Durch die hohen Wartungskosten einzelner Workstations, durch fehlerbehaftete Software und durch die erhöhte Leistungsfähigkeit der Netzwerkkomponenten haben alte Server-Client-Architekturen eine Wiedergeburt erfahren. Durch diese Technologie spart man sich außerdem die komplexen Wartungsarbeiten an einzelnen Clients, da die Funktionalität der Systeme zum grossen Teil vom Server übernommen wird. Die Clients werden demnach immer einfacher, während der Server immer komplexer wird. Durch die Vereinfachung der Clients gewinnen diese an Ausfallsicherheit. Außerdem können zur Wartung der vielen einfachen Clients Arbeitskräfte eingesetzt werden, die keine umfassende Ausbildung benötigen, und damit kostengünstiger sind.

Dieser Ansatz verursacht allerdings Kosten an einer anderen Stelle. Durch das höhere Volumen an Daten die zwischen Server und Client ausgetauscht werden wird das Netz stärker belastet. Außerdem kommen mit der Verlagerung der Funktionalität auf die Server höhere Anforderungen an die Leistungsfähigkeit und Ausfallsicherheit des Servers.

Durch die schnellen Veränderungen ergeben sich Planungsprobleme. Die existierenden Netzwerke von Workstations, Clients, Terminals, Server, Router und anderen Netzwerkkomponenten sind sehr heterogen. Die Verantwortlichen stehen vor einem komplexen Konfigurationsproblem, wenn sie ihr Netzwerk verändern müssen oder wollen. Die Integration neuer Komponenten in ein bestehendes und funktionierendes System ist meist komplexer als die Planung eines neuen Systems: Der Planer muss stets ein Optimierungsproblem lösen.

Es gilt die neuen Komponenten mit einem Minimum an Zusatz-kosten zu integrieren. Wie lässt sich dieses Ziel am einfachsten erreichen? Um diese Frage beantworten zu können müssen wir uns die Frage stellen: Wo entstehen diese Zusatz-kosten?

Probleme können in solch einem System an drei verschiedenen Stellen auftreten: Client, Server, Netzwerk. Wie in vielen anderen Fällen auch, sind die spät entdeckten Planungsfehler die teuersten. Betrachten wir jede der Fehlerquellen genauer:

- **Client.** Hier sind für den Anwender keine Probleme zu erwarten. Jegliche Probleme, die der Client verursachen sollte fällt bei Neuanschaffungen in die Zuständigkeit des Vertreibers und des Herstellers. Durch Tests lässt sich die Funktionsfähigkeit eines Gerätes vor der Anschaffung überprüfen. Nach dem Kauf eintretende Probleme mit der Maschine können häufig kostengünstig gelöst werden, z.B. durch einen einfachen Austausch der defekten Komponente.

- **Server.** Dieser Punkt ist schwieriger zu erfassen. Wird der Server leistungsfähig genug sein, um eine Aufgabe zu erfüllen? Hier kommt es auf die genaue Situation an. Die Zusatzbelastung ist durch eine Hochrechnung kaum abschätzbar. Wenn der Server skalierbar ist kann er durch die Erhöhung seiner Verarbeitungskapazität die Arbeit bewältigen. Im schlimmsten Fall wird die Installation eines neuen Servers oder die Aufstockung des zuständigen Personals notwendig, was kostspielig werden kann.

- **Netzwerk.** Der letzte Punkt ist der Analyse am schwersten zugänglich, da er von sehr vielen Faktoren abhängt. Wo wird der Client in das Netz gehängt? Ist die Netzbelastung permanent konstant? Gibt es Störungen?

Welche Möglichkeiten bestehen um Probleme dieser Art zu lösen? Uns stehen drei Methoden zur Verfügung um das Problem zu Erkennen: Messungen, Simulationen und mathematische Analysen. Welche Methode sich eignet, hängt von der konkreten Situation ab.

- **Messungen** setzen ein existierendes System voraus, an dem die Messungen durchgeführt werden. Komponenten, die noch nicht gebaut sind, können also auch nicht erfasst werden. Diese Methode ist damit beim Planen der Konfiguration eines Systems nicht einsetzbar, sondern nur bei der Analyse schon existierender Systeme anwendbar.

- **Mathematische Analysen** hingegen beschreiben Abstraktionen vom Verhalten eines Systems. Insofern eignen Sie sich eher zum Planen eines möglichen Systems, als zur Analyse eines bereits bestehenden Systems. Hier gibt es ein Problem mit Komponenten, über deren Verhalten keine Aussagen vorliegen. Diese können damit auch nicht erfasst werden.

- Die Letzte verbleibende Methode, **Simulation**, ist in ihrem Abstraktionsgrad und ihrer Genauigkeit ein Kompromiss zwischen den beiden erstgenannten Methoden. Hier hängt die Genauigkeit der Analyse nur von der Genauigkeit der Modelle ab, die bei der Analyse benutzt werden. Real existierende Komponenten und mathematische Modelle können gleichzeitig benutzt werden, um sich gegenseitig zu ergänzen.

Ziel dieser Diplomarbeit ist es, das Verhalten von Netzlasten im Bankbereich zu untersuchen, sowie das Verhalten von möglichen zukünftigen Netzkomponenten (Bankautomaten) zu modellieren und eine Möglichkeit zu finden dieses Verhalten zu simulieren. Diese Untersuchung wird mit dem Hintergrund eines konkreten Beispiels der Firma WINCOR NIXDORF GmbH & Co. KG durchgeführt.

Im Kapitel 2 wird die Situation analysiert, das Problem definiert und ein Lösungsansatz präsentiert. Im Kapitel 3 wird Simulation als Methode vorgestellt, um das Problem aus Kapitel 2 zu lösen. Dabei werden einzelne Schritte einer Simulationsstudie beschrieben. Im Kapitel 4 wird ein Simulationsmodell entwickelt und vorgestellt, welches in der Lage ist das Verhalten von Client-Server Systemen im Bankbereich nachzuahmen. Im Kapitel 5 wird das Simulationsmodell, dass in Kapitel 4 entwickelt wurde, in ein Softwaremodell übertragen. Hier wird auch die Benutzung der Software erklärt. Im Kapitel A ist der Quellcode dieses Softwaremodells aufgelistet.

2. Problemanalyse und die Formulierung der Ziele

Den Hintergrund für die Diplomarbeit stellt folgendes Szenario dar: Derzeit werden in der Firma Wincor Nixdorf GmbH & Co. KG „intelligente“ Terminals (fat clients) eingesetzt. Diese sollen vermehrt durch billigere „dumme“ Terminals (thin clients) ersetzt werden. Hierbei entstehen neben der Neukonzeption der Terminals auch Fragen in Bezug auf die Modifikation sowohl der Server, als auch des Netzwerkes.

Einige von der Fragen, die in diesem Zusammenhang von Kunden gestellt werden, lauten wie folgt: Wird das Netzwerk den erhöhten Durchsatz an Daten zufriedenstellend bewältigen? Wie viele Terminals kann ich an das bestehende Netz anschließen ohne es modifizieren zu müssen? Gibt es Änderungen in der Auslastung des Netzes, wenn man die Konfiguration der Clients ändert?

2.1. Situation und Präzisierung der Problembeschreibung

Die zu untersuchende Problemstellung entsteht durch eine Veränderung eines bestehenden Systems. Um die Auswirkungen des Wandels betrachten zu können, ist es notwendig das bestehende System unter die Lupe zu nehmen.

Im bestehenden System lassen sich drei Hauptbestandteile identifizieren:

- Server
- Netzwerk
- Clients

Diese Bestandteile lassen sich unabhängig von einander betrachten.

Diese Diplomarbeit befasst sich mit den Auswirkungen des Architekturwandels bei den Clients auf das Netzwerk. Ausgehend von dieser Tatsache ist es möglich einige Vereinfachungen in dem zu untersuchendem System zu treffen.

2.1.1. Server

Als *Server* betrachten wir Rechnersysteme, die anderen Systemkomponenten Dienste anbieten. Diese Dienste können z.B. Authentisierung, individuelle Profilverwal-

tung oder aber auch die Bearbeitung von Aufgaben von Seiten der Arbeitsplätze innerhalb einer Filiale sein. Ein Server kann also in seiner Architektur beliebig variieren. Aus dem Betrachtungswinkel dieser Arbeit ist einzig wichtig, dass ein Server seine Dienste Ausfallsicher und innerhalb festgelegter Zeitrahmen anbietet.

Die Auswirkungen der Neukonzeption der Clients auf die Server sind weitreichend. Da durch die Vereinfachung der Clients stellen-weise Funktionalitäten in die Server verlagert werden, müssen die Server an die neuen Anforderungen angepasst werden. Diese Anpassungen sind allerdings nicht Gegenstand dieser Arbeit. Im weiteren Verlauf der Arbeit wird davon ausgegangen, dass die Serverarchitektur dem jeweiligem Szenario angepasst ist.

2.1.2. Netzwerk

Als *Netzwerk* betrachten wir in dieser Arbeit die Menge von Routern, Leitungen, Rechnernetzen, verteilten Systemen und anderen Medien, die die Kommunikation zwischen zwei Netzwerkkomponenten ermöglichen. Diese Definition eines Netzwerkes unterscheidet sich von der Definition eines Rechnernetzes in [Tan98].

Tanenbaum unterscheidet zwischen einem *Rechnernetzwerk* und einem *verteiltem System*, wobei Rechnernetzwerke aus *miteinander verbundenen autonomen Computern* bestehen, während in einem *verteilten System* die Rechner für den User transparent sind.

Wenn man diese Terminologie benutzt ergeben sich zwei Sichtweisen.

Aus der Sicht des Terminalbenutzers, der vor einem Bankautomaten steht, ist das zu untersuchende System ein verteiltes System. Der einzige Hinweis auf die Existenz anderer Komponenten als der unmittelbar Sichtbaren kann nur vermutet werden.

Aus der Sicht des Entwicklers oder des Wartungstechnikers ist das System ein Rechnernetzwerk. Da viele unabhängige Rechnersysteme schon innerhalb eines Terminals miteinander über ein gemeinsames Medium kommunizieren.

In dem gegebenen Szenario spielt die genaue Topologie des Netzwerkes keine Rolle. Wir können im weiteren Verlauf der Arbeit das Netzwerk als eine Black Box ansehen, da uns hier allein die „Leistungsfähigkeit“ des Netzwerkes interessiert. Aber was ist diese „Leistungsfähigkeit“ ?

Netzwerkmanagement nach ISO 7498-4

Eine mögliche Antwort darauf liefert uns die ISO-Norm 7498-4. Dieser Standard beinhaltet einen Rahmen für Netzwerkmanagement. Hier werden Begriffe definiert, die sich im allgemeinen Gebrauch als nützlich erwiesen haben. In dieser Norm werden Aufgaben für das Netzwerkmanagement mit dem Ziel definiert, dem Benutzer des Netzes den gewünschten Dienst in der gewünschten Qualität zu liefern. Der Standard unterscheidet dabei die folgenden Funktionsbereiche des Netzwerkmanagements:

- Konfigurationsmanagement (configuration management)
- Sicherheitsmanagement (security management)

- Fehlermanagement (fault management)
- Leistungsmanagement (performance management)
- Abrechnungsmanagement (accounting management)

Da wir Begriffe suchen, die die Leistungsfähigkeit des Netzes beschreiben, ist der für uns interessante Teil das Leistungsmanagement.

Leistungsmanagement

Bei funktionierendem Netzwerk ist es die Aufgabe des Leistungsmanagements Sorge dafür zu tragen, dass die anfallenden Aufträge der Benutzer möglichst effizient ausgeführt werden können. Ziel des Netzbetreibers ist es, eine möglichst hohe Auslastung des Netzes zu erreichen um Kosten für eine „Überdimensionierung" des Netzes zu sparen. Der Benutzer ist an einer schnellen Reaktions- und Antwortzeit des Netzwerkes interessiert. Diese beiden Seiten müssen vom Leistungsmanagement berücksichtigt werden. Für die Bestimmung der Leistung müssen *Indikatoren* gefunden werden, anhand derer die Leistung *gemessen* werden kann. Terplan und Stallings unterscheiden die Indikatoren in *Service-orientierte Indikatoren* und in *Effizienz-orientierte Indikatoren* [Ter92, Sta99]. Eine Auflistung findet sich in der Tabelle 2.1.2.

Als Service-orientierte Indikatoren werden diejenigen Variablen (Indikatoren) bezeichnet, die vorwiegend für den Benutzer des Netzwerkes von Interesse sind. An der Optimierung der Effizienz-orientierten Indikatoren sind hingegen insbesondere Netzbetreiber interessiert. Im folgende werden die Indikatoren Verfügbarkeit, Antwortzeit, Exaktheit, Durchsatz und Auslastung definiert und ihre Einsetzbarkeit beschrieben [Lau00].

- **Verfügbarkeit:** Verfügbarkeit ist der Zeitraum, in dem ein Netzwerk, eine Komponente des Netzes oder eine Anwendung nutzbar ist. Dieser Indikator drückt also den Anteil der Ausfälle an der Gesamtzeit aus.
- **Antwortzeit:** Der Benutzer eines Computersystems will mit seinem Rechner zügig (ohne Verzögerungen) seine Arbeit erledigen. Die Bearbeitungsdauer seiner Benutzeraufträge soll ihn nicht in seiner Arbeit behindern. Die Antwortzeit als Indikator für Netzwerkleistung basiert auf dem Client-Server Modell. Der Benutzer (client) fordert eine Leistung bei einem „entfernten" Rechner (server) an und wartet auf die Erledigung. Der Zeitraum zwischen der Anforderung und dem Eintreffen der Erledigungsnachricht wird als Antwortzeit betrachtet.
- **Exaktheit:** Dieser Indikator beschreibt die Anzahl bzw. den Anteil nicht korrekter Datenübertragungen. Durch Korrekturmechanismen auf allen Ebenen des Protokollstapels werden Fehler weitgehend behoben.
- **Durchsatz:** Dieser Indikator wird von [Ter92, Sta99] als anwendungsorientiertes Maß bezeichnet, d.h. dass die angegebenen Werte von der verwendeten

Service-orientierte Indikatoren (service-oriented indicators)	
Verfügbarkeit (availability)	Der Prozentsatz der Zeit, für den ein Netzwerk, eine Komponente oder eine Anwendung für den Benutzer verfügbar ist.
Antwortzeit (response time)	Zeitraum, der benötigt wird, um einen Benutzerauftrag zu bearbeiten und das Ergebnis dem Benutzer zu übermitteln.
Exaktheit (accuracy)	Der Prozentsatz der Zeit, für welchen keine Fehler in der Übermittlung von Nachrichten auftreten.
Effizienz-orientierte Indikatoren (efficiency-oriented indicators)	
Durchsatz (throughput)	Die Rate, in welcher anwendungsbezogene Ereignisse auftreten.
Auslastung (utilization)	Der benutzte Prozentsatz der theoretischen Kapazität einer Ressource.

Tabelle 2.1.: Indikatoren der Leistungsmessung

Anwendung abhängig sind. Der Indikator Durchsatz beschreibt, wie viele Aktionen pro Zeiteinheit durchgeführt wurden.

- **Auslastung:** Als Auslastung wird der Prozentsatz der (theoretischen) Gesamtkapazität einer Netzwerkkomponente, der in einer Zeiteinheit genutzt wurde, bezeichnet. Mit Hilfe dieses Maßes können Flaschenhälse (bottlenecks) und Staus (congestions) im Netzwerk identifiziert werden.

Das interessante Maß für die Beantwortung unserer Fragen ist die *Antwortzeit*. Die anderen vier Masse können aus der Sicht der Studie aus verschiedenen Gründen vernachlässigt werden: Da wir uns für das Verhalten eines Systems im Normalfall interessieren, können wir annehmen, dass die gewünschte Verfügbarkeit gegeben ist. Die Exaktheit wird durch die Datenübertragungsmethoden sichergestellt, und ist daher ebenfalls gewährleistet.

Die Effizienz-orientierten Indikatoren Durchsatz und Auslastung sind in diesem Zusammenhang von geringem Interesse. Beide Indikatoren sind für Netzwerkbetreiber sehr wichtig, für den Betreiber eines Clients aber nur dann interessant, wenn diese Indikatoren den Indikator Antwortzeit beeinflussen. Aus der Sicht des Clientbetreibers, der Antwortzeit als primären Indikator für die Leistungsfähigkeit des

Netzwerkes wählt, werden beide Indikatoren daher ausreichend berücksichtigt.

2.1.3. Clients

Die Modifikation der Architektur der Clients ist der Grund für diese Studie. Daher ist es unbedingt notwendig sich Gedanken über mögliche Veränderungen auf diesem Gebiet zu machen.

Grundkomponenten eines Terminals

Ein Client besteht aus verschiedenen Peripheriegeräten und evtl. einem Rechner. Die Peripherie besteht im einzelnen aus:

- Kartenleser (CHD)
- Sicherheitstastatur (EPP - Encrypted Pin Pad)
- Tastatur
- Bildschirm
- Drucker
- Geldausgabe
- Sonderelektronik
- Möglich:
  - Münzeinzahlung
  - Münzauszahlung
  - Briefmarkenausgabe
  - Ticketausgabe
  - usw, ...

Jedes der Geräte (devices) ist eigenständig lauffähig und besitzt verschiedene Grundelemente:

- Ethernet-Controler
- Prozessor (skalierbar)
- RAM (skalierbar)
- ROM

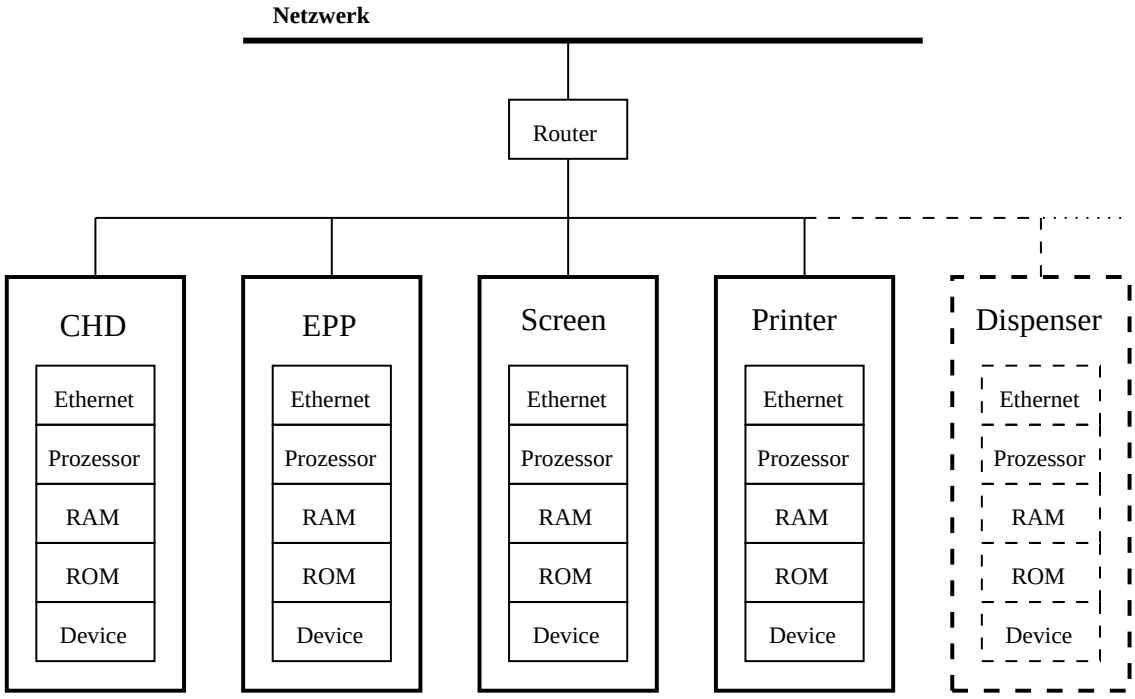

Abbildung 2.1.: Terminal mit verschiedenen Einschüben.

Terminalarchitektur

Wie im Diagramm 2.1 beschrieben, ist ein Client ein verteiltes System, bestehend aus einer Menge von Geräten, die miteinander durch ein lokales Netz verbunden sind. Dieses System kann einen Computer mit Festplatte beinhalten. Das Vorhandensein dieses Rechners macht den Unterschied zwischen einem Fat-Client und einem Thin-Client aus. Ist ein Computer mit Speicher- und Verarbeitungskapazität im lokalen System vorhanden, ist es ein *Fat-Client*. Ist diese Speicher- und Verarbeitungskapazität auf einen entfernten Server ausgelagert, nennt man das System *Thin-Client*.

Mögliche Veränderungen und ihre Konsequenzen.

Da wir in dieser Studie die Auswirkungen der Clientarchitekturen auf das Netzwerk untersuchen, interessieren uns vor allem die Änderungen im Verhalten des Clients, die das Netz betreffen. Unter diesem Aspekt ist der Wandel von Fat- zu Thin-Client mit einer gravierenden Änderung des Verhaltens aus der Sicht des Netzwerkes verbunden. Da lokale Verarbeitungskapazitäten auf einen fernen Server ausgelagert werden, ist die Inanspruchnahme dieser Dienste zwangsläufig mit der Verlagerung der damit einhergehen Kommunikationsaktivitäten, vom lokalen Netz in das umgebende Netzwerk verbunden. Beim Austausch einer Komponente zur Münzeinzahlung gegen eine zur Noteneinzahlung, kann hier keine solche Verhaltensänderung erwartet werden.

Wie bereits angedeutet, bedeuten Architekturänderungen der Hardware nicht

immer auch eine Änderung des Kommunikationsverhaltens des Clients. Es kann auch zu Änderungen des Kommunikationsverhaltens Aufgrund von Softwareänderungen kommen. Hier ein Beispiel: Um die laufenden Betriebskosten eines Automaten zu senken kann der Inhaber sich entschließen das Display des Automaten, in den Leerlaufzeiten in denen der Automat von den eigentlichen Nutzern nicht benutzt wird, als Werbefläche zu nutzen. Die Hardware des Automaten muss für diesen Zweck nicht prinzipiell verändert werden. Durch die Übermittlung der Werbedaten (Bilder, Filme, ...) an den Automaten wird das Netzwerk allerdings zusätzlich belastet.

Anforderungen an das Terminal

Da nun feststeht, dass sich das Verhalten des Terminals mit der Änderung seiner Hardware- und/oder Softwarekomponenten auf das Verhalten des Gesamtsystems auswirken kann, ist es notwendig Anforderungen an das Gesamtsystem zu definieren. Diese Anforderungen müssen Kriterien beinhalten, anhand derer man entscheiden kann, ob ein System in der vorgeschlagenen Konfiguration akzeptabel funktioniert oder nicht.

Entscheidungskriterien für die Benutzbarkeit von Systemen findet man in den Disziplinen der *Ergonomie* und der *Usibility*. Hier gibt es zahlreiche Quellen für geeignete Kriterien. Als Richtlinie wird in dieser Arbeit die Europäische Norm **EN ISO 9241-10** verwendet. Diese Norm definiert die Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten. Der Teil 10 beschäftigt sich mit den Grundsätzen der Dialoggestaltung. Hier werden sieben Grundsätze für die Gestaltung und die Bewertung eines Dialoges als wichtig erkannt:

- **Aufgabenangemessenheit:** Ein Dialog ist aufgabenangemessen, wenn er den Benutzer unterstützt, seine Arbeitsaufgabe effektiv und effizient zu erledigen.

- **Selbstbeschreibungsfähigkeit:** Ein Dialog ist selbstbeschreibungsfähig, wenn jeder einzelne Dialogschritt durch Rückmeldung des Dialogsystems unmittelbar verständlich ist oder dem Benutzer die Anfrage erklärt wird.

- **Steuerbarkeit:** Ein Dialog ist steuerbar, wenn der Benutzer in der Lage ist den Dialogablauf zu starten, sowie seine Richtung und Geschwindigkeit zu beeinflussen, bis das Ziel erreicht ist.

- **Erwartungskonformität:** Ein Dialog ist erwartungskonform, wenn er konsistent ist und den Merkmalen des Benutzers entspricht, z.B. seinen Kenntnissen aus dem Arbeitsgebiet, seiner Ausbildung und seiner Erfahrung sowie den allgemein anerkannten Konventionen.

- **Fehlertoleranz:** Ein Dialog ist fehlertolerant, wenn das beabsichtigte Arbeitsergebnis trotz erkennbar fehlerhafter Eingaben entweder mit keinem oder mit minimalem Korrekturaufwand seitens des Benutzers erreicht werden kann.

- **Individualisierbarkeit:** Ein Dialog ist individualisierbar, wenn das Dialogsystem Anpassungen an die Erfordernisse der Arbeitsaufgabe, sowie an die individuellen Fähigkeiten und Vorlieben des Benutzers zulässt.

- **Lernförderlichkeit:** Ein Dialog ist lernförderlich, wenn er den Benutzer beim erlernen des Dialogsystems unterstützt und anleitet.

Nicht alle der genannten Grundsätze haben einen Bezug zum Gegenstand dieser Arbeit:

- Aufgabenangemessenheit, Selbstbeschreibungsfähigkeit, Individualisierbarkeit und Lernförderlichkeit sind Anforderungen an die Gestaltung der Benutzungsschnittstelle. Aus diesem Grund sind sie als Kriterien für den Client im Bezug auf die Leistungsfähigkeit des Netzwerkes nicht anwendbar.

- Fehlertoleranz stellt Forderungen nicht nur an die Benutzungsschnittstelle, sondern auch an die restliche Software des Clients. Dieses Kriterium betrifft ebenfalls nicht die Leistungsfähigkeit des Netzwerkes.

- Steuerbarkeit ist ein Kriterium, dass Anforderungen an die Geschwindigkeit des Dialoges stellt, und ist daher eventuell interessant. Dieses Kriterium ist aber nur zum Teil anwendbar. Es gibt nur einen Fall in dem das Netzwerk die Steuerbarkeit des Clients beeinflusst: Dies passiert immer nur dann wenn die Antwortzeiten des Systems auf Anfragen des Benutzers zu lang sind. In solchen Fällen ist aber auch das Kriterium Erwartungskonformität anwendbar.

Der für diese Arbeit interessante Grundsatz ist *Erwartungskonformität*. Dieses Konzept beinhaltet unter anderem die Forderung, dass auf die Angaben des Benutzers eine unmittelbare Rückmeldung des Systems zu erfolgen hat, falls dieser eine solche Rückmeldung erwartet. Dieser Teil der Erwartungskonformität deckt sich mit dem schon beschriebenen Fall des Konzeptes Steuerbarkeit. Erwartungskonformität beinhaltet außer dieser auch noch andere Forderungen. Nicht nur die Geschwindigkeit des Dialoges, sondern auch die Darstellungsart und die Menge der Informationen muss den Erwartungen des Benutzers entsprechen. Dies beinhaltet eine zusätzliche Forderung an das Netzwerk, denn die benötigten Daten müssen in der Regel über das Netzwerk an den Client übertragen werden.

2.2. Problemdefinition

Das Problem der Firma WINCOR NIXDORF GmbH & Co. KG besteht in der Abschätzung der Erwartungskonformität des Verhaltens der Automaten, die durch die Einführung neuer Clientarchitekturen entsteht. Genauer: Es ist von Interesse ob die Client-Server-Systeme erwartungskonform funktionieren werden, wenn Clients in das bereits bestehende Netz integriert werden. Erwartungskonformität bezieht sich im Fall dieser Studie auf die Antwortzeiten des Gesamtsystems.

Ziel der Studie

Das Ziel ist es Abschätzungen von erwarteten Antwortzeiten bei einer gegebenen Systemkonfiguration zu ermöglichen. Mit Hilfe dieser Arbeit sollte ein Mitarbeiter der Firma WINCOR NIXDORF GmbH & Co. KG in der Lage sein eine Systemkonfiguration zu Modellieren und eine Schätzung der erwarteten Systemantwortzeiten zu geben.

Mögliche Szenarien und zu beantwortende Fragen

Typische Szenarien für die Fragestellungen sind:

- **Szenario 1:** Man möchte Clients älterer Architekturen durch neue ersetzen.
- **Szenario 2:** Man hat Interesse an einer Anzahl von Clients einer bestimmten Architektur. Diese sollen an das bestehende Netz angeschlossen werden.
- **Szenario 3:** Man hat den Wunsch Clients verschiedener Architekturen anzuschaffen und sie an verschiedenen Punkten in gegebener Anzahl ans Netz anzuschließen.

Die entstehende Frage in allen Fällen ist:
Wird das Gesamtsystem Erwartungskonform funktionieren?

2.3. Lösungsansatz

Es existieren Prinzipiell mehrere Methoden diese Frage zu beantworten. Alle haben Vor- und Nachteile. Im folgenden werden einige Methoden genauer betrachtet.

Möglichkeiten

Folgende Möglichkeiten gehören zu den naheliegenden:

- **Messungen** setzen ein existierendes System voraus an dem die Messungen durchgeführt werden. Komponenten, die noch nicht gebaut sind können also auch nicht erfasst werden. In der speziellen Situation ist selbst der Einsatz existierender Hardware, wenn auch möglich, sehr teuer.
- **Mathematische Analysen** hingegen beschreiben Abstraktionen vom Verhalten eines Systems. Insofern eignen Sie sich eher zur Analyse des möglichen Systems. In unserem Fall haben wir ein Problem mit Komponenten über deren Verhalten keine Aussagen vorliegen. Das betrifft die Systemkomponente Netzwerk. Der Versuch die Lösung auf diesem Weg zu erreichen würde die Modellierung des Netzwerkes zwischen Client und Server beinhalten. In vielen Fällen wird es nicht möglich sein diese Systemkomponente korrekt zu modellieren. Folgende Gründe können die Modellierung des Netzwerkes erheblich erschweren:

  - Die genaue Topologie des Netzwerkes zwischen Client und Server ist oft nicht bekannt.
  - Die Topologie des Netzes ändert sich häufig.
  - Die Auslastung des Netzwerkes ändert sich häufig.
  - Durch den Anschluss vieler Clients an einem Netz, wird die Auslastung des Netzes durch die Clients nicht vorhersehbar.
  - Das Verhalten des Clients wird durch das Verhalten der Benutzer bestimmt, welches zum Teil nur schwer modellierbar ist.
  - Die Häufigkeit und Intensität, mit der ein Client benutzt werden wird, ist häufig im voraus nicht abschätzbar.
  - Je umfangreicher die Möglichkeiten eines Benutzers an einem Client sind, desto schwieriger wird die Modellierung des Verhaltens eines Clients.

- **Simulation** ist in ihrem Abstraktionsgrad und ihrer Genauigkeit ein Kompromiss zwischen den beiden erstgenannten Methoden. Hier hängt die Genauigkeit der Analyse nur von der Genauigkeit der Modelle ab, die bei der Analyse benutzt werden. Real existierende Komponenten und mathematische Modelle können gleichzeitig benutzt werden, um sich gegenseitig zu Ergänzen.

Wahl der Simulation als Methode

In unserer Situation können wir Messung als Untersuchungsmethode ausschließen, da hier häufig das Verhalten von Systemkomponenten untersucht werden soll, die als physikalisches Modell noch nicht verfügbar sind.

Mathematische Analyse ist als Methode nur dann zu empfehlen, wenn genügend Informationen vorhanden sind, um das Verhalten des Netzwerkes gut genug zu modellieren.

Da in dem zu untersuchenden System sowohl nicht implementierte als auch schon existierende Bestandteile verwendbar sein sollen, ist die Wahl der Simulation als Methode, die bestmögliche Wahl. Bei genauer Kenntnis des Verhaltens der real existierenden Bestandteile, können diese durch Simulationsmodelle ersetzt werden. Auf diesem Weg kann der Aufwand für Simulationsexperimente erheblich gesenkt werden.

Ausblick

Für diejenigen Leser, die sich in ihrer bisherigen Laufbahn noch nicht mit der Simulation als Untersuchungsmethode befasst haben, ist im Kapitel 3 eine kurze Einleitung zu diesem Thema gegeben. Im Verlauf der Arbeit ab Kapitel 4 wird die Anwendung der Methode der Simulation auf die gegebene Situation behandelt.

3. Einführung in die Simulation

An dieser Stelle wird eine Einführung in das Thema Simulation gegeben, um den Lesern, die mit diesem Bereich nicht vertraut sind, einen Einblick in diese Methode zu geben. Es sei gesagt, dass viele Wissenschaftler die auf diesem Gebiet tätig sind die Simulation sowohl als Wissenschaft als auch als Kunst sehen [Sha98].

3.1. Definition von Simulation

Nach [Ban00] ist *Simulation* die Imitation eines Prozesses oder Systems in einem Zeitablauf. Die Simulation beinhaltet die Generierung einer künstlichen Geschichte, eines Systems und die Beobachtung dieser künstlichen Geschichte um Schlussfolgerungen auf die Laufzeiteigenschaften des zugrunde liegenden realen Systems zu ziehen.

Simulation ist eine unentbehrliche Problemlösungsmethode für viele reale Probleme. Sie wird benutzt um das Verhalten eines Systems zu beschreiben und zu analysieren, um „Was wäre wenn?“ Fragen über ein reales System zu beantworten und um beim Design realer Systeme zu helfen. Sowohl reale als auch konzeptionelle Systeme können per Simulation modelliert werden.

3.2. Das Simulationsteam

Obwohl manche kleine Simulationsstudien von einem einzigen Analytiker durchgeführt werden, werden die meisten von einem Team durchgeführt. Das ist die Folge des Bedarfs einer Vielzahl von Fähigkeiten, die notwendig sind um ein komplexes System zu untersuchen. Als erstes werden Menschen gebraucht, die das zu untersuchende System kennen und verstehen. Dies sind meistens Designer, System-, Herstellungs- und Prozessingenieur. Aber das können auch Manager, Projektleiter und/oder Betriebs-personal, welche die Resultate benutzen wollen, sein. Als zweites braucht man Leute, die wissen wie man das System sowohl formalisiert und modelliert, als auch implementiert (Simulationsspezialisten). Diese Teammitglieder brauchen auch Fähigkeiten auf dem Gebiet der Datensammlung und der Statistik.

Die erste Personalkategorie muss notwendigerweise intern sein, das heißt Mitglieder der Organisation, für die die Studie durchgeführt wird. Wenn man kein Personal der zweiten Kategorie zur Verfügung hat, hat man mehrere Wahlmöglichkeiten. Man

kann: (a) Personen, die entsprechende Fähigkeiten haben, einstellen, (b) bei der Modellierung auf externe Berater zurückgreifen, (c) einige der eigenen Leute ausbilden, oder (d) eine beliebige Kombination der erstgenannten. Wenn man sich entscheidet eigene Leute auszubilden ist es wichtig darauf zu achten, dass die Fähigkeiten der Datensammlung und Statistik wesentlich wichtiger sind, als Programmierfähigkeiten. Die neuen Simulationspakete haben die benötigten Computerfähigkeiten weniger wichtig gemacht, als diese es mal waren.

Es ist wichtig zu erkennen, dass die Kenntnis eines Simulationssoftwarepakets einen nicht zu einem Simulationsspezialisten macht, genauso-wenig wie FORTRAN Kenntnisse einen zu einem Mathematiker machen. Wie schon erwähnt, *Simulation ist sowohl Wissenschaft als auch Kunst.* Die Statistischen Fähigkeiten und Programmierfähigkeiten ist der wissenschaftliche Teil. Aber die Modellierungs- und Analysekomponenten sind Kunst. Zum Beispiel sind Fragen nach der Detailliertheit des Modells oder nach der Repräsentation einiger Phänomene oder nach Alternativen zur Evaluierung des Modells, alle ein Teil dieser Kunst.

Wie lernt man eine Kunst? Stellen Sie sich vor, Sie wollen lernen ein Ölportrait zu mahlen. Man kann ihnen die Wissenschaft in der Ölmalerei beibringen wie Perspektive, Schattierung, Farbmischung usw.(Programmierung, Statistik und Softwarepakete). Aber das würde sie nicht in die Lage versetzen Ölportraits zu malen. Man könnte Sie in Museen bringen und ihnen Malereien von Meistern zeigen und auf die Techniken, die sie benutzt haben, hinweisen (Modelle anderer Leute untersuchen). Selbst hiernach würden Sie nur minimale Fähigkeiten haben akzeptable Porträts zu mahlen.

Wenn Sie in einer Kunst bewandert werden wollen, müssen Sie das Werkzeug (Palette, Pinsel, Farbe) in die Hand nehmen und anfangen zu malen. Sobald Sie das tun, werden Sie feststellen, was funktioniert und was nicht. Dasselbe gilt für die Simulation. Man lernt die Kunst der Simulation beim Simulieren. Das Vorhandensein eines Mentors kann die Zeit und den Aufwand verringern. Das ist der Grund, warum viele Firmen, die erste Simulationen durchführen, auf eine Mischung von externen Beratern und eigenen Trainees zurückgreifen.

3.3. Schritte einer Simulationsstudie

Die Essenz oder der Zweck einer Simulationsstudie ist es einem Entscheidungsträger zu helfen ein Problem zu lösen [Sha98]. Der Erfolg eines Simulationsprojektes ist allerdings von vielen Kleinigkeiten abhängig. Um Simulationsteams zu helfen sind daher eine Menge Leitfäden zur Durchführung solcher Projekte geschrieben worden.

Hier wird eine mögliche Vorgehensweise beschrieben [Ban00]. So oder ähnlich taucht der Leitfaden auch in anderen Quellen auf, wie z.B. in [Sha98], [LK00], [PSS95]. Die Abbildung 3.1 veranschaulicht die Reihenfolge der Schritte einer Simulationsstudie.

Innerhalb dieser Diplomarbeit werden nur einige Schritte einer Simulationsstudie durchgeführt. Es sind die ersten Schritte einer Simulationsstudie. Die Durchführung

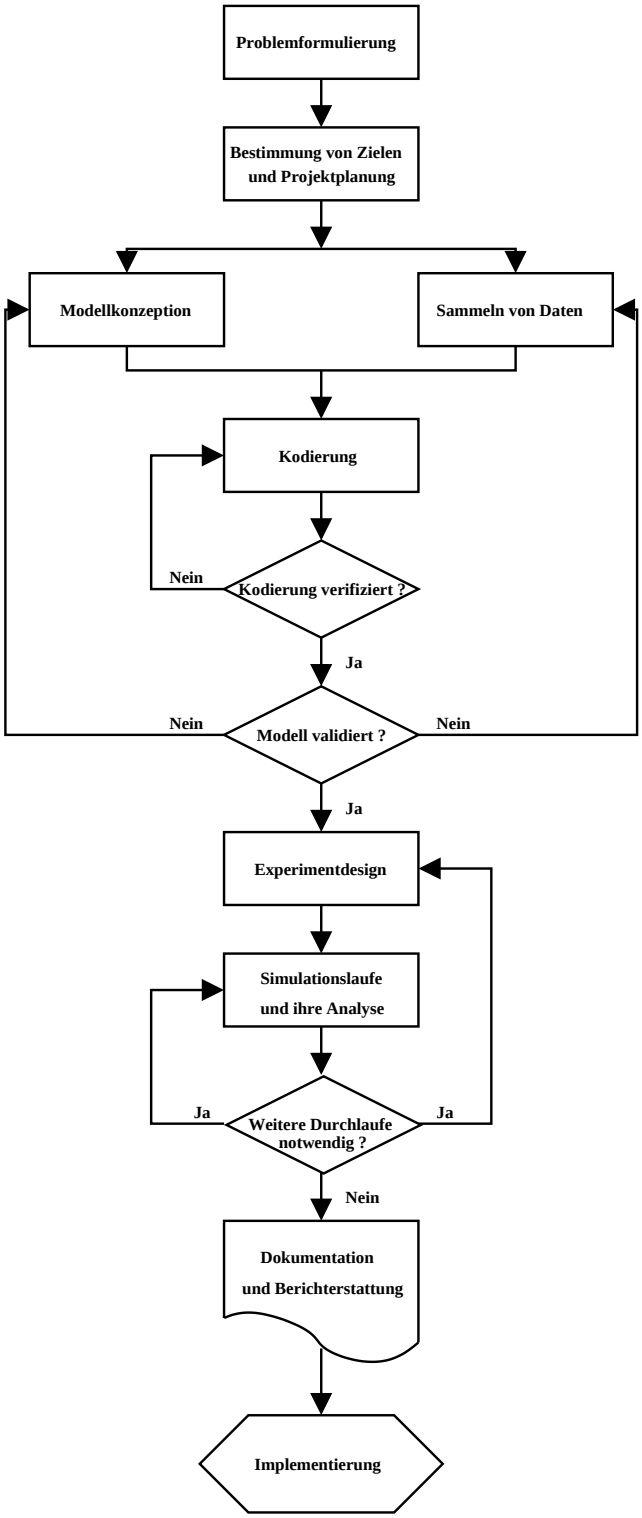

Abbildung 3.1.: Schritte einer Simulationsstudie

kompletter Simulationsstudien bleibt in diesem Fall den Mitarbeitern der Firma WINCOR NIXDORF GmbH & Co. KG vorbehalten.

3.3.1. Durchgeführte Schritte

Hier werden Schritte einer Simulationsstudie beschrieben, die im Lauf dieser Arbeit behandelt wurden. Da diese Diplomarbeit nur eine Vorarbeit zu einer Simulationsstudie ist, ist es nicht notwendig alle Schritte einer Simulationsstudie auszuführen.

Problemformulierung

Jede Simulationsstudie beginnt mit der Formulierung des Problems. Wenn die Formulierung von denen vorgenommen wird, die das Problem haben (Kunden), sollte der Simulationsspezialist extreme Vorsicht walten lassen, um sicherzugehen, dass das Problem vollständig verstanden wurde. Wenn die Problemformulierung von dem Simulationsspezialisten vorgenommen wurde, ist es sehr wichtig, dass der Kunde diese Formulierung versteht und mit ihr einverstanden ist. Es wird vorgeschlagen, dass eine vom Simulationsspezialisten vorbereitete Menge von Annahmen vom Kunden überprüft und bewilligt wird. Selbst mit all diesen Vorkehrungen ist es möglich, dass eine Notwendigkeit entsteht das Problem während einer schon laufenden Simulationsstudie umzuformulieren.

Die Fragen, die am Anfang einer solchen Studie zu beantworten sind, sind unter anderem folgende:

- Was ist das Ziel der Studie?
- Was ist die Frage, die beantwortet werden sollte?
- Welche Entscheidung soll getroffen werden?
- Welche Informationen werden gebraucht um diese Entscheidung zu treffen?
- Welche Kriterien werden angewendet um die Entscheidung zu treffen?
- Wer soll diese Entscheidung treffen?
- Wer wird von dieser Entscheidung betroffen sein?
- Wie viel Zeit steht zur Verfügung um diese Frage zu beantworten?

In dem Fall dieser Arbeit ist dieser Schritt im Kapitel 2 durchgeführt worden.

Bestimmung von Zielen und Projektplanung

In diesem Schritt geht es darum, die durch die Simulationsstudie zu beantwortende Fragen aus dem ersten Schritt in konkrete Ziele zu verwandeln. Der Projektplan sollte die Auflistung der verschiedenen Szenarien enthalten, die untersucht werden sollten. Die Pläne für die Studie sollten folgendes beinhalten:

- Die Zeit für die Durchführung der Studie.
- Das gebrauchte Personal.
- Die Hardware- und Softwareanforderungen, falls der Kunde selbst Simulationen und deren Auswertung vornehmen möchte.
- Die Etappen der Untersuchung.
- Den Output jeder Etappe.
- Die Kosten der Studie.
- Wenn notwendig die Zahlungsmodalitäten.

Nicht zuletzt sollten Kommunikationswege festgelegt werden um sicherzustellen, dass alle am Projekt beteiligten (Kunden, Teammitglieder, Management, ...) über den genauen Status, die Ziele des Projektes, Änderungen im Bedarf der Kunden oder der Benutzer, immer informiert sind.

Dieser Schritt einer Simulationsstudie ist im Fall dieser Arbeit nur partiell im Kapitel 2 dokumentiert worden. Die meisten Absprachen wurden mündlich getroffen und nicht schriftlich festgehalten worden. Im Fall einer größeren Studie ist jedoch auch dieser Schritt sorgfältig zu dokumentieren.

Modellkonzeption

Die Essenz der Modellierungskunst ist die Abstraktion und die Vereinfachung. Ein „Real-World“ System wird von einem konzeptuellen Modell abstrahiert. Eine Reihe mathematischer und logischer Beziehungen betreffen die Komponenten und die Struktur des Systems. Es wird empfohlen die Modellierung einfach zu beginnen und das Modell zu erweitern, bis die notwendige Komplexität erreicht wird. Es wird versucht einfache Untermengen von Charakteristiken oder Eigenschaften des Systems zu finden, die den speziellen Zielen der Studie dienlich sind. Man versucht ein Modell des realen Systems zu Entwerfen, dass weder zu einfach ist, um die an die Studie gerichteten Fragen korrekt zu beantworten, noch zu Komplex. Die Konstruktion unnötig komplexer Modelle erhöht die Kosten und den Zeitaufwand der Studie, ohne die Qualität der Resultate der Studie zu verbessern. Die Beteiligung der Kunden an der Modellbildung erhöht die Qualität des resultierenden Modells und erhöht ihre Akzeptanz für dieses Modell und seinen Einsatz.

Während der Definition des Modells sind einige Aufgaben zu erledigen. Darunter sind folgende:

- Unterteilung des Systems in logische Untersysteme.
- Definition von Entitäten, die sich innerhalb des Systems bewegen(fliessen).
- Definition von Stationen (Stellen an denen mit oder für die Entitäten etwas getan wird) für jedes Subsystem.

- Definition der grundlegenden Flußmuster der Entitäten durch das System.
- Definition alternativer Entwürfe für das System.

Dieser Schritt einer Simulationsstudie wird in dem Kapitel 4 ausführlich behandelt.

Sammeln von Daten

Kurz nachdem der Kunde das „Angebot“ akzeptiert hat, sollte ihm eine Liste der für die Studie erforderlichen Daten mitgeteilt werden. In dem besten aller Fälle wird der Kunde die gebrauchte Art von Daten in dem benötigten Format gesammelt haben und sie dann dem Simulationsanalytiker im elektronischen Format zur Verfügung stellen. In vielen Fällen allerdings wird der Kunde angeben, dass die benötigten Daten nicht verfügbar sind. Wie auch immer, sobald die Daten geliefert wurden, werden sie meist anders empfunden als erwartet.

Im Fall dieser Diplomarbeit war dieser Schritt einer Simulationsstudie sehr problematisch. Aus Datenschutz-gründen konnten keine Messungen zur Datengewinnung an real existierenden Systemen vorgenommen werden. Deshalb waren auf Schätzungen der Entwickler von Client-Server Systemen der Firma WINCOR NIXDORF GmbH & Co. KG die einzige Datenquelle.

Der Mangel von Daten ist der Grund für die Entwicklung des im Kapitel 4 vorgestellten Verhaltensmodells. Mit Hilfe des Modells ist es möglich das Verhalten eines Simulationsmodells schnell an neu gewonnene Daten anzupassen, um dann die Validierung des Simulationsmodells durchführen zu können.

Kodierung

In diesem Schritt wird das konzeptuelle Modell in ein vom Computer zu verarbeitendes operationales Modell übersetzt. Hierbei werden meist Verfeinerungen des Simulationsmodells vorgenommen, um die Bestandteile des Modells zu integrieren, die zur „Realen Welt“ gehören.

Die Beschreibung der Implementierung des im Kapitel 4 entwickelten Simulationsmodells wird im Kapitel 5 vorgestellt. Den Quellcode der Implementierung kann man im Kapitel A finden.

Verifiziert?

Die Verifikation betrifft das operationale Modell. Funktioniert das Modell richtig? Vor allem komplexe Modelle von dem Umfang kleiner Bücher neigen dazu Verifikationsprobleme zu haben. Diese Modelle sind um Größenordnungen kleiner als reale Systeme (sagen wir 50 Zeilen Programmcode zu 2000 Zeilen). Es ist sehr zweckmäßig die Verifikation als einen kontinuierlichen Prozess zu sehen. Dabei ist es für den Simulationsspezialisten nicht ratsam mit dem Verifikationsprozess erst dann zu beginnen, wenn das Modell schon vollständig ist. Auch die Benutzung von Debuggern kann den Verifikationsprozess erleichtern.

Bei der Implementierung des Simulationsmodells wurde hier die Klasse *Debug* zur Fehlersuche verwendet. Diese Klasse ist im Kapitel A zu finden.

Validiert?

Validation ist die Überprüfung ob das konzeptuelle Modell eine akkurate Repräsentation des realen Systems ist. Kann das Modell das reale System für die Zwecke der Untersuchung ersetzen? Existiert ein reales System, auch Basissystem genannt, ist der ideale Weg zur Validierung des Modells dieses Systems, der Vergleich der Outputs der beiden Systeme. Bedauerlicherweise existiert nicht immer solch ein Basissystem. Es gibt viele Methoden die Validation durchzuführen.

Die Validierung des Simulationsmodells konnte aus Mangel an Daten nicht durchgeführt werden. Näheres dazu finden Sie in der Beschreibung des Schritts *Sammeln von Daten*.

3.3.2. Nicht durchgeführte Schritte

Hier werden Schritte einer Simulationsstudie beschrieben, die im Lauf dieser Arbeit nicht behandelt wurden. Die Durchführung dieser Schritte macht nur innerhalb eines tatsächlich vorliegenden Szenarios Sinn. Da in dieser Diplomarbeit nur Vorarbeit zu einer Simulationsstudie zu Leisten war, wurden diese Schritte nicht beachtet.

Experimentdesign

Für jedes zu simulierende Szenario müssen Entscheidungen getroffen werden, die die Länge des Simulationsdurchlaufs, die Anzahl der Durchläufe (*replications*), und die Art der Initialisierung betreffen.

Simulationsläufe und ihre Analyse

Simulationsdurchläufe und die ihnen folgende Analysen werden benutzt, um Maße für die Performance der simulierten Szenarios zu schätzen.

Weitere Durchläufe notwendig?

Basierend auf der Analyse der Simulationsdurchläufe die schon abgeschlossen wurden, hat der Simulationsspezialist zu entscheiden, ob zusätzliche Durchläufe notwendig sind und ob weitere Szenarien simuliert werden sollen.

Dokumentation und Berichterstattung

Dokumentation ist aus zahlreichen Gründen notwendig. Wenn das Simulationsmodell noch einmal vom Selben oder einem anderen Simulationsspezialisten benutzt wird, ist es unumgänglich zu verstehen wie das Simulationsmodell funktioniert. Dies

ermöglicht Vertrauen in das Simulationsmodell, so dass der Kunde Entscheidungen auf der Basis der Analysen treffen kann. Wenn das Modell modifiziert werden soll, kann dieser Prozess durch eine adäquate Dokumentation erheblich beschleunigt werden.

Die Resultate aller Analysen sollten klar und prägnant berichtet werden. Das wird dem Kunden ermöglichen, sich über den Abschlussbericht eine Übersicht über die angegebenen Alternativen, über die Kriterien nach denen die Alternativen Systeme verglichen wurden, die Resultate der Experimente, und die Empfehlungen des Analytikers (wenn welche vorhanden) zu verschaffen.

Umsetzung

Der Simulationsanalytiker agiert eher als Reporter denn als Anwalt. Der Bericht der im vorhergehenden Schritt angefertigt wurde, erleichtert dem Kunden seine Entscheidung bei der Umsetzung seines Vorhabens. Wenn der Kunde in die Studie einbezogen wurde und der Simulationsspezialist keine groben Fehler gemacht hat, dann wird die Wahrscheinlichkeit für die erfolgreiche Implementierung des Vorhabens erhöht.

4. Modellierung

In diesem Kapitel wird die Methode der Simulation auf das vorliegende Problem angewandt. Es wird ein Simulationsmodell entwickelt, um eine Abschätzung der Erwartungskonformität eines geplanten Client-Server Systems im Bankbereich zu ermöglichen.

4.1. Ziel der Modellierung

Ziel der Modellierung ist es ein Simulationsmodell zu entwickeln, dessen Verhalten durch Parametrisierung des Modells existierende Client-Server Systeme im Bankbereich imitieren soll. Das Modell soll sich ebenfalls eignen Entwürfe von geplanten Systemen auf erwartungskonformes Verhalten der Clients zu testen. Bei dem Test soll nur der Aspekt der Leistungsfähigkeit des Netzwerkes in Form von Antwortzeiten gemessen werden. Anhand der so erhaltenen Messwerte soll abgeschätzt werden ob die Leistungsfähigkeit des Netzwerkes erwartungskonformes Verhalten des Clients zulässt.

Bei der Modellierung müssen verschiedene Clientarchitekturen berücksichtigt werden. Insbesondere soll der Unterschied zwischen Fat-Client- und Thin-Client-Systemen untersucht werden.

4.2. Beschreibung der Situation

Um die Modellierungssituation klar zu sehen, ist es hier ratsam zuerst ein möglichst einfaches real existierendes System zu beschreiben. Ausgehend von diesem einfachen System kann man später ein einfaches Simulationsmodell erstellen, welches dann nach Bedarf komplexer gestaltet werden kann.

4.2.1. Beschreibung eines einfachen Geldautomaten

Eines der einfachsten Systeme im Bankbereich ist ein Cashautomat. Das Verhalten des Cashautomaten wird im Activity Diagramm 4.1 verdeutlicht. Dieser Automat bietet nur eine Funktion an: Geld vom Konto abheben. In dem im Activity Diagramm 4.1 gezeigtem Verhalten wird davon ausgegangen das der Automat über eine Datenbank verfügt, die ihm die Authentifizierung der Operation ermöglicht. Im Regelfall ist das aber nicht der Fall.

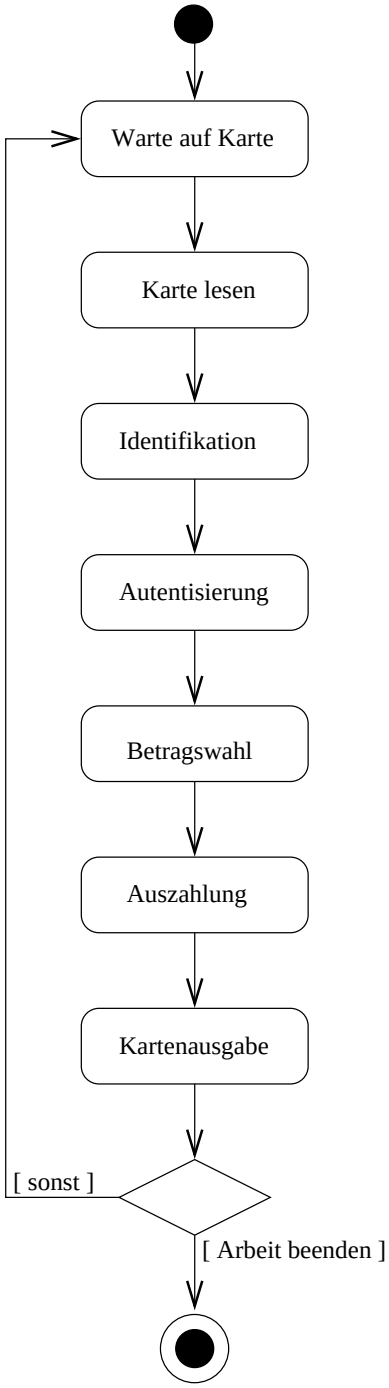

Abbildung 4.1.: Activity Diagramm eines Cashautomaten.

In der Regel bestehen die einfachen Cashautomatsysteme aus Client (Geldautomat) und Server (Authentisierungsserver). Diese Teile des Systems sind über das Netz miteinander verbunden. Im Activity Diagramm 4.2 wird das Verhalten eines solchen Systems gezeigt.

Die Authentisierung nimmt bei einer *Fat-Client* Architektur ein Authentisierungsserver vor, während alle anderen Aktivitäten von dem Client gemacht werden. An diesem Punkt ist Kommunikation zwischen Client und Server über das Medium Netzwerk notwendig. Im Activity Diagramm 4.3 sind die dazu notwendigen Schritte gezeigt.

Bei einem Cashautomaten der *Thin-Client* Architektur ist das Verhalten ein wenig komplexer. Da hier meist lokal nur sehr wenig Speicherkapazität zur Verfügung steht, müssen die zum Betreiben des *User Interfaces* notwendige Daten auf einen entfernten Server ausgelagert werden. Das hat zur Folge, dass im schlimmsten Fall bei jeder Änderung des User Interfaces die zur Modifikation des Zustandes notwendigen Daten von diesem Server bezogen werden müssen. Zusätzlich müssen Daten, die auch schon zum Betreiben des Fat-Client Automaten notwendig waren, über das Netz bezogen werden.

Bei dem Verschicken der Nachrichten über das Netzwerk, wird eine Nachricht üblicherweise vom Absender in eine Menge Datenpackete unterteilt. Diese Datenpackete werden von dem Netzwerk an den Empfänger übertragen. Der Empfänger setzt aus den empfangenen Datenpacketen die Nachricht wider zusammen. Dieser Vorgang wird im Activity Diagramm 4.4 verdeutlicht.

4.3. Art des benötigten Modells

Durch die Beschreibung des Automaten in 4.2.1 können wir die Eigenschaften und die Art des benötigten Simulationsmodells bestimmen, die in der gegebenen Situation benötigt werden:

- Das Modell muss *dynamisch* sein, d.h. das Modell muss sich durch die Zeit verändern können. Da wir an den Antwortzeiten eines dynamischen Systems interessiert sind, darf das Simulationsmodell nicht *statisch* sein.
- Das Modell kann *diskret* sein, d.h. das Modell muss seinen Zustand nur an abzählbar vielen Punkten in der Zeit ändern. Solche Zeitpunkte sind zum Beispiel:
  - Ankunft einer Nachricht an einem Client.
  - Ankunft einer Nachricht an einem Server.
  - Ankunft eines Benutzers an einem Client.
  - Eine durch den Benutzer ausgelöste Aktion.
  - Anfang der Simulation.
  - Ende der Simulation.

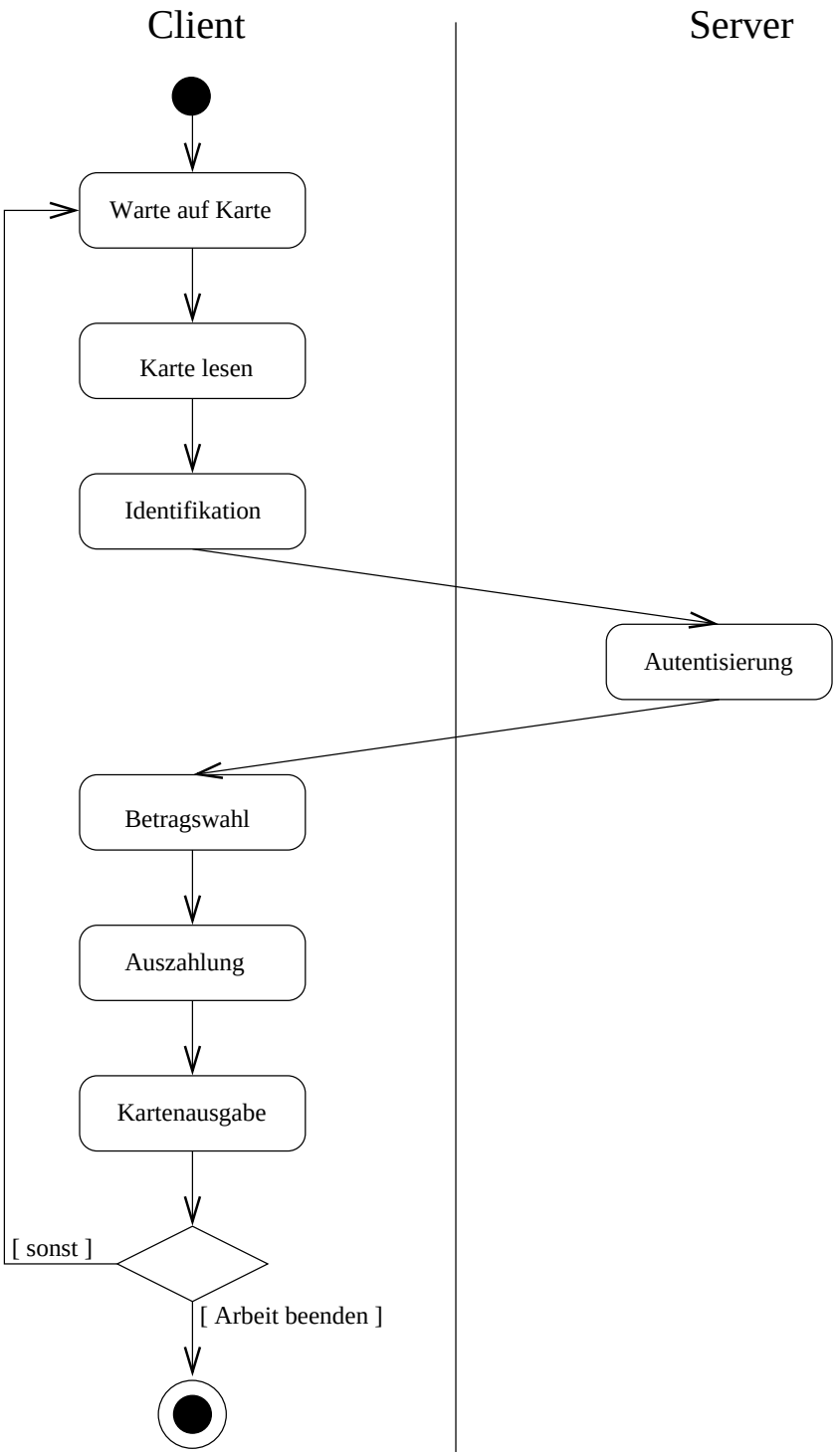

Abbildung 4.2.: Activity Diagramm eines Fat-Client Cashautomaten.

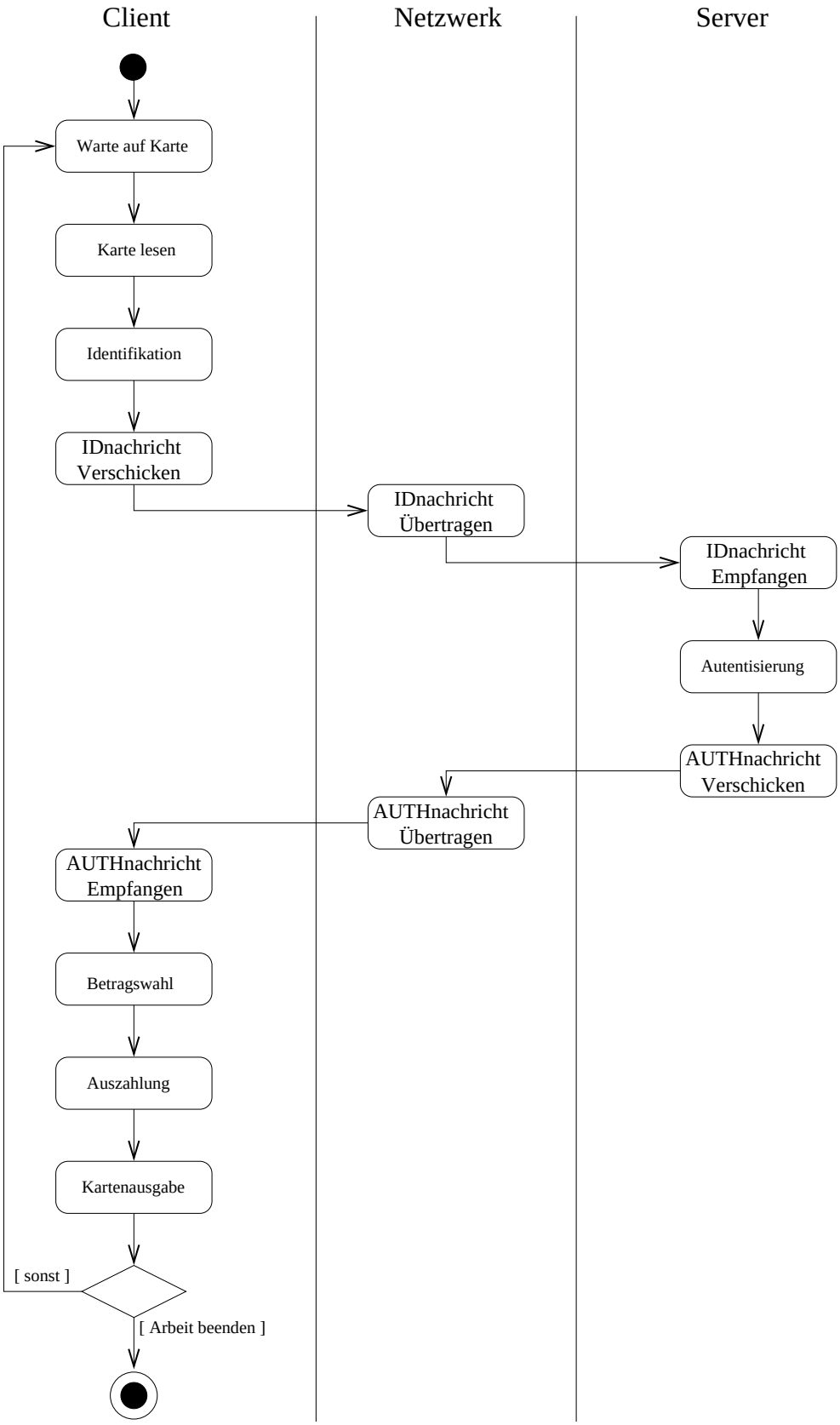

Abbildung 4.3.: Activity Diagramm eines Fat-Client Cashautomaten mit einer Darstellung der Netzwerkaktivitäten.

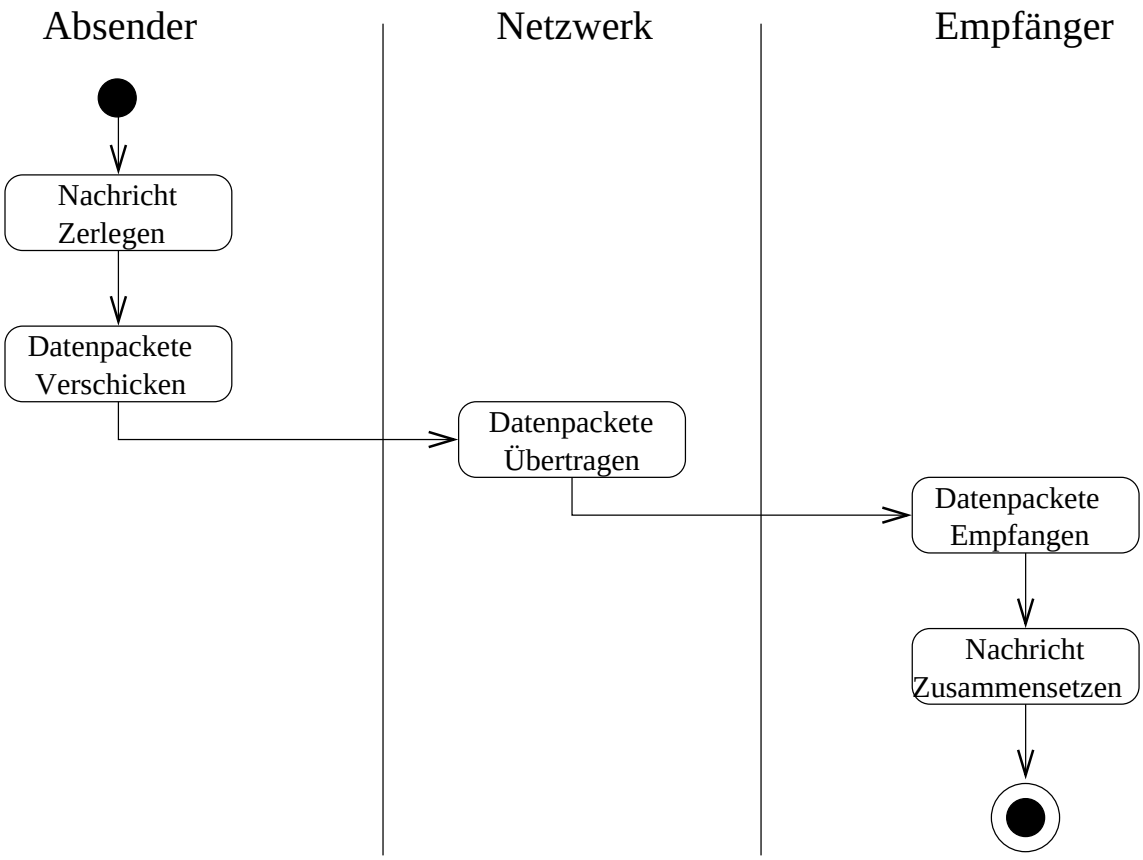

Abbildung 4.4.: Verschicken einer Nachricht über das Netzwerk.

- Das Modell muss *stochastisch* sein, d.h. es muss stochastische Elemente beinhalten. Stochastische Prozesse sind hier notwendig um das Verhalten eines Benutzers an einem Client zu imitieren.

Ein Simulationsmodell das dynamisch, stochastisch und diskret ist nennt man *Diskrete-Event Simulationsmodell*. Was ist also ein Diskrete-Event Simulationsmodell?

4.3.1. Diskrete-Event Modell und der Zeitmechanismus

Nach [LK00] sind *Diskrete-Event Simulationsmodelle* Modelle, die Systeme darstellen, welche sich mit der Zeit entwickeln und sich nur zu bestimmten Zeitpunkten verändern. D.h. das System ändert seinen Zustand an *abzählbar vielen* Zeitpunkten. Diese besonderen Punkte in der Zeit sind Zeitpunkte an welchen *Ereignisse* (events) stattfinden. Dabei werden als *Ereignisse* solche Vorkommnisse definiert, die den Zustand des Systems ändern können[1]. Ereignisse finden als Konsequenzen von *Aktivitäten* und *Verzögerungen* statt. Entitäten können um *Systemressourcen* konkurrieren in dem sie sich Warteschlangen anschließen, während sie auf verfügbare

[1]Ereignisse können den Systemzustand ändern, müssen es aber nicht zwingend tun.

Ressourcen warten. Aktivitäten und Verzögerungen können Entitäten für die Zeit ihrer Dauer „halten“.

Diskrete-Event Simulationen haben zusätzlich eine besondere Eigenschaft: Sie könne prinzipiell auch per Hand durchgeführt werden. Meistens jedoch sind die Simulationsmodelle so komplex dass der Einsatz eines Rechners notwendig wird.

Innerhalb der Diskrete-Event Simulation werden einige Modellarten unterschieden [Ban00], wie zum Beispiel mathematische Modelle, deskriptive Modelle, statistische Modelle und input-output Modelle. Ein „Diskrete-Event“ Modell versucht die Komponenten eines Systems und ihre Interaktionen so zu beschreiben, dass die Ziele der Simulationsstudie erreicht werden können. Die meisten mathematischen, statistischen, und input-output Modelle stellen die Eingaben und Ausgaben eines Systems explizit dar, wobei die Interna des Modells durch mathematische oder statistische Beziehungen repräsentiert werden. Diskrete-Event Simulationsmodelle beinhalten detaillierte Repräsentationen der aktuellen Systemzustände.

Diskrete-Event Modelle sind dynamisch, d.h. die Zeit spielt eine kritische Rolle. Die meisten mathematischen und statistischen Modelle sind statisch, somit stellen sie ein System zu einem Zeitpunkt da.

Ein Diskrete-Event Simulationsmodell wird von einem Mechanismus durch die Zeit geleitet, welcher die simulierte Zeit vorwärts bewegt. Der Systemzustand wird bei jedem Ereignis zusammen mit der Belegung und der Freigabe von Ressourcen aktualisiert, die sich zu diesem Zeitpunkt ereignen können.

Zeitmechanismus

Wegen der dynamischen Natur der Diskrete-Event Simulationen, muss während der Durchführung solcher Simulationen die *simulierte Zeit*, im Zuge des Voranschreitens der Simulation, festgehalten werden. Außerdem wird ein Mechanismus gebraucht, um die simulierte Zeit von einem Wert auf den anderen setzt. Die Variable im Simulationsmodell, die die simulierte Zeit angibt, nennt man *Simulationsuhr* (simulation clock).

Zeiteinheiten werden meist für die Simulationsuhr nicht explizit angegeben, wenn das Modell in einer Universalprogrammiersprache wie C/C++, FORTRAN oder Java implementiert ist. Es wird angenommen, dass hier dieselben Einheiten verwendet werden, die in den Inputparametern verwendet wurden. Im allgemeinen gibt es also keine Beziehung zwischen der simulierten Zeit und der Zeit, die gebraucht wird um das Simulationsmodell auf einem Rechner laufen zu lassen.

Zeitmechanismus des Bankautomatmodells

In dem gegebenen Fall müssen wir die Realzeit als simulierte Zeit verwenden. Diese Festlegung ist hier notwendig, da wir als Komponenten des Simulationsmodells reale Netzwerkkomponenten verwenden. Wir müssen davon ausgehen, dass die Eigenschaften dieser Systemkomponenten uns unbekannt sind und von uns nicht beeinflusst werden können. Da das Verhalten dieser Komponenten also nicht an die künstliche

simulierte Zeit angepasst werden kann, muss das restliche Simulationsmodell mit der Zeit der Netzwerkkomponenten synchronisiert werden.

Falls es gelingt die Systemkomponente Netzwerk zufriedenstellend zu modellieren, ist die beschriebene Beschränkung nicht mehr zwingend notwendig. Solange wir aber über kein Modell dieser Systemkomponente verfügen, müssen wir einige Änderungen in der Organisation des klassischen Diskrete-Event-Simulationsmodells vorgenommen werden.

4.3.2. Komponenten und Organisation eines Diskrete-Event Simulationsmodells

Es gibt etliche Konzepte, die der Simulation zugrunde liegen. Diese beinhalten System und Modell, Ereignisse, Systemzustandsvariablen und Attribute, Listenverarbeitung, Aktivitäten und Verzögerungen. Zusätzliche Informationen zu dem Thema finden Sie in [BCNN00], [LK00]. Die meisten der folgenden Definitionen entstammen [Car93].

System, Modell und Ereignisse

Ein *Modell* (model) ist eine Repräsentation eines wirklichen *Systems*. Wobei es Grenzen in jedem Modell gibt, inwiefern es tatsächlich dem wirklichen System entspricht. Ein Modell sollte Komplex genug sein, um die gestellten Fragen zu beantworten, aber nicht zu komplex.

In dem Fall dieser Arbeit, ist der interessante Aspekt des Systems dessen Netzwerkverhalten. Bei der Modellierung müssen also die Aktivitäten des Systems genau modelliert werden, die dieses Verhalten darstellen. Alle übrigen Aktivitäten können in Form von *Aktivitäten* (activitys) und *Verzögerungen* (delays) modelliert werden.

Als *Ereignis* (event) wird ein Vorfall betrachtet, der den Zustand des Systems verändern kann. Es gibt interne und externe Ereignisse, die entsprechend *endogen* und *exogen* genannt werden. Endogene Ereignisse sind Ereignisse innerhalb des Modells/Systems während exogene Ereignisse das Modell/System von Assen beeinflussen.

Innerhalb des Bankautomat-Simulationsmodells gibt es nur wenige exogene Ereignisse. Die wichtigsten Ereignisse im Normalfall sind: *Start der Simulation* und *Ende der Simulation*. Weitere exogene Ereignisse sind alle Ereignisse, die einen Simulationslauf unerwartet von Assen beeinflussen. Zu solchen Ereignissen gehören: *Ausfall/Fehlfunktion einer Netzwerkkomponente*, *Ausfall/Fehlfunktion einer Simulationsplattform*[2] und *Anweisungen vom Durchführenden des Simulationslaufes*.

Endogene Ereignisse sind in diesem Modell viel zahlreicher. Dazu gehören: *Empfangen und Versenden einer Nachricht*, *Empfangen und Versenden eines Datenpacketes*, *Starten einer Simulationsplattform*, *Eintragungen ins Protokoll des Simulationslaufes*, *Anfang und Ende jeder Verzögerung*, *Anweisungen vom Master des*

[2]Als *Simulationsplattform* ist hier eine Instanz eines verteilten Simulationsexperimentes gemeint.

Simulationslaufes[3].

Systemzustandsvariablen

Die *Systemzustandsvariablen* (system state variables) sind eine Sammlung von Informationen, die gebraucht werden, um hinreichend genau zu definieren, was innerhalb des Systems zu einem gegebenen Zeitpunkt vorgeht. Die Ermittlung der Systemzustandsvariablen ist abhängig von den Zielen der Untersuchung, so dass zwei Modelle ein und desselben Systems sehr unterschiedlich sein können. Die Bestimmung der Systemzustandsvariablen gleicht mehr einer Kunst denn einer Wissenschaft. Während des Modellierungsprozesses sollen die notwendigen Variablen in das Modell integriert und die Überflüssigen beseitigt werden.

Nach der Definition der Systemzustandsvariablen kann man, abhängig von den gewählten Variablen, zwischen Diskreten-Event und Kontinuierlichen Modellen unterscheiden. Die Zustandsvariablen in einem Diskret-Event Modell bleiben über ganze Zeitintervalle konstant und ändern nur zu bestimmten so-genannten *Ereignisszeitpunkten* (event times) ihren Wert. Kontinuierliche Modelle haben Systemzustandsvariablen, die durch Differential- oder Differenzgleichungen definiert werden, was dazu führt, dass diese sich kontinuierlich durch die Zeit verändern.

Manche Modelle sind ein Mix zwischen Diskreten-Event und Kontinuierlichen Modellen. Es existieren auch kontinuierliche Modelle, die nach einer Reinterpretation der Systemzustandsvariablen als „Diskrete-Event“ Modelle behandelt werden und umgekehrt.

Die offensichtlichen Systemzustandsvariablen des Bankautomat-Simulationsmodells sind die Zustandsvariablen aller Komponenten dieser verteilten Simulation. Eine genaue Auflistung und Beschreibung aller verwendeten Variablen können sie im Kapitel 5 und in der Dokumentation des Softwaremodells einsehen.

Entitäten und Attribute

Eine *Entität* (entity) repräsentiert ein Objekt, welches einer expliziten Definition bedarf. Eine Entität kann *dynamisch* oder auch *statisch* sein. Dynamische Entitäten bewegen sich durch das System, während statische Entitäten andere Entitäten bedienen. Statische Entitäten werden auch *Stellen* genannt.

Typische Beispiele dynamischer Entitäten im Bankautomatmodell sind *Nachrichten* und *Datenpackete*, die zwischen Client und Server ausgetauscht werden. *Anweisungen an Simulationsplattformen*, *Übertragungen von Konfigurationen* und *Übertragungen von Protokollen* sind dynamische Entitäten, die zur Steuerung einer verteilten Simulation notwendig sind.

Eine *Simulationsplattform* ist die größte im Modell verwendete Stelle. Diese Stelle beinhaltet weitere Stellen, wie z.B. *Serverstellen* und *Clientstellen*, die jeweils das verhalten von Clients oder Servern imitieren.

[3] Als *Master* ist die Instanz eines Simulationslaufes gemeint, die den Simulationslauf der verteilten Simulation steuert.

Eine Entität kann über *Attribute* verfügen, die dieser Entität alleine gehören. Folglich sollten Attribute als lokale Werte betrachtet werden. Attribute, die während einer Untersuchung interessant sind, können in einer anderen Untersuchung nicht von Belang sein. Man kann leicht sehen, dass viele Entitäten dasselbe Attribut aufweisen können (z.B. „Farbe“ bei hergestellten Industrieprodukten).

Beispielattribute der in der Simulation verwendeten Attribute für Entitäten sind: *Absenderadresse der Entität* und *Empfängeradresse der Entität*.

Ressourcen

Eine *Ressource* ist eine Entität, die dynamischen Entitäten Dienste anbietet. Eine Ressource kann eine oder mehrere Entitäten zur gleichen Zeit bedienen. Eine dynamische Entität kann eine oder mehrere Einheiten der Ressource anfordern. Wenn der Dienst von der Ressource abgelehnt wird kann die anfordernde Entität sich in eine Schlange (queue) einordnen oder andere Aktionen ausführen z.B. vom System zu einer anderen Ressource umgeleitet werden. Andere Bezeichnungen für Schlangen (queues) sind unter anderem Karteien (files), Ketten (chains), Zwischenspeicher (buffer), und Warteschlangen (waiting lines). Wenn eine Entität eine Ressource in Anspruch nimmt, verbleibt sie eine Zeit lang dort, und gibt dann diese frei.

Es gibt eine Menge möglicher Zustände für eine Ressource. Die Kleinste dieser Mengen besteht aus zwei Zuständen ungenutzt (idle) und beschäftigt (busy). Aber auch andere Möglichkeiten wie misslungen (failed), blockiert (blocked), oder hungernd (starved) existieren.

Die wichtigste Ressource des Simulationsmodells eines Bankautomatsystems ist das Netzwerk. Diese Ressource wird aus den in Kapitel 2 beschriebenen Gründen nicht modelliert. Das Netzwerk wird als eine Komponente aus der „Realen Welt“ in das Simulationsmodell eingebunden werden.

Listenverarbeitung

Entitäten werden verwaltet, in dem sie Ressourcen zugewiesen werden, welche ihrerseits Dienste anbieten, die an Ereignisnotizen geknüpft sind. Dabei suspendieren sie ihre Aktivitäten bis auf weiteres, oder platzieren sie in einer geordneten Liste (ordered list).

Listen werden meist nach dem FIFO Prinzip (first-in-first-out, Queue, Schlange) verarbeitet, wobei es auch andere Möglichkeiten gibt. Zum Beispiel könnten die Listen nach dem LIFO Prinzip (last-in-first-out, Stack, Keller), abhängig von dem Wert eines Attributes, oder in zufälliger Reihenfolge verarbeitet werden. Ein Beispiel in dem der Wert eines Attributes wichtig sein kann, ist in der SPT (shortest process time) Planung zu finden. In diesem Fall kann die Prozesszeit als Attribut in jeder Entität gespeichert werden. Die Entitäten sind dann abhängig von dem Wert dieses Attributes angeordnet, mit dem kleinsten Wert am Anfang der Warteschlange.

Jede Stelle des Modells verfügt über eigene Listen, in denen dynamische Entitäten verwaltet werden. Nach der Bearbeitung durch die entsprechende Stelle wird die

Entität in eine andere Liste derselben Stelle oder an eine andere Stelle übergeben.

Aktivitäten und Verzögerungen

Eine *Aktivität* (activity) ist ein Zeitintervall dessen Dauer vor dem Beginn der Aktivität bekannt ist, so dass wenn der Zeitabschnitt beginnt, sein Ende geplant werden kann. Dieses Zeitintervall kann konstant, ein Zufallswert aus einer statistischen Verteilung, das Resultat einer Gleichung, Eingabe aus einer Datei, oder aus dem Modellzustand berechnet sein.

Eine *Verzögerung* (delay) ist eine unbestimmte Zeitspanne, die von bestimmten Kombinationen der Systembedingungen verursacht wird. Wenn eine Entität einer Warteschlange für eine Ressource eingeteilt wird, könnte die Zeit, die sie in der Warteschlange verbringt, anfangs unbekannt sein, da die Zeit auch von anderen Ereignissen, die eintreten könnten, abhängen kann.

Diskrete-Event Simulationen beinhalten Aktivitäten, die Zeit brauchen um fortzuschreiten. Die meisten Diskrete-Event Simulationen beinhalten auch Verzögerungen, als Wartezeiten der Entitäten. Der Beginn und das Ende einer Aktivität oder einer Verzögerung ist ein Ereignis.

Beispielaktivitäten im Bankautomatmodell sind: *Simulation der Bearbeitungszeit einer Nachricht* durch eine Serverstelle oder eine Clientstelle, *Simulation der Wartezeit auf einen Kunden* durch eine Clientstelle.

Verzögerungen treten immer dann im Modell auf, wenn eine dynamische Entität in einer Liste auf Bearbeitung wartet. Eine besondere Verzögerung ist die Zeit, die das Netzwerk braucht um ein Datenpacket von einem Absender zu einem Empfänger zu übertragen. Die Längen dieser Verzögerungen in bestimmten Systemkonfigurationen zu erfahren, ist der Grund für die Durchführung der Simulationsexperimente.

4.4. Bankautomat Modell

Wie in dem Abschnitt 4.3 bereits beschrieben, ist der für uns relevante Teil des Systems, das Verhalten das Systems im Bezug auf das Netzwerk. Aus diesem Blickwinkel wird klar, dass wir uns hier auf zwei für uns wichtige Eigenschaften beschränken können:

- Die genaue Beschaffenheit der Nachrichten, die zwischen Client und Server ausgetauscht werden.
- Die Länge der Wartezeiten zwischen Nachrichten.

Damit können wir das Verhaltensmodell unseres Geldautomaten durch Zusammenfassen der Aktivitäten in 4.2 auf das in dem Activity Diagramm 4.5 gezeigte reduzieren.

Da das Versenden einer Nachricht immer an eine Aktivität davor geknüpft ist, kann man das Modell noch weiter auf das in dem Activity Diagramm 4.6 gezeigte reduzieren.

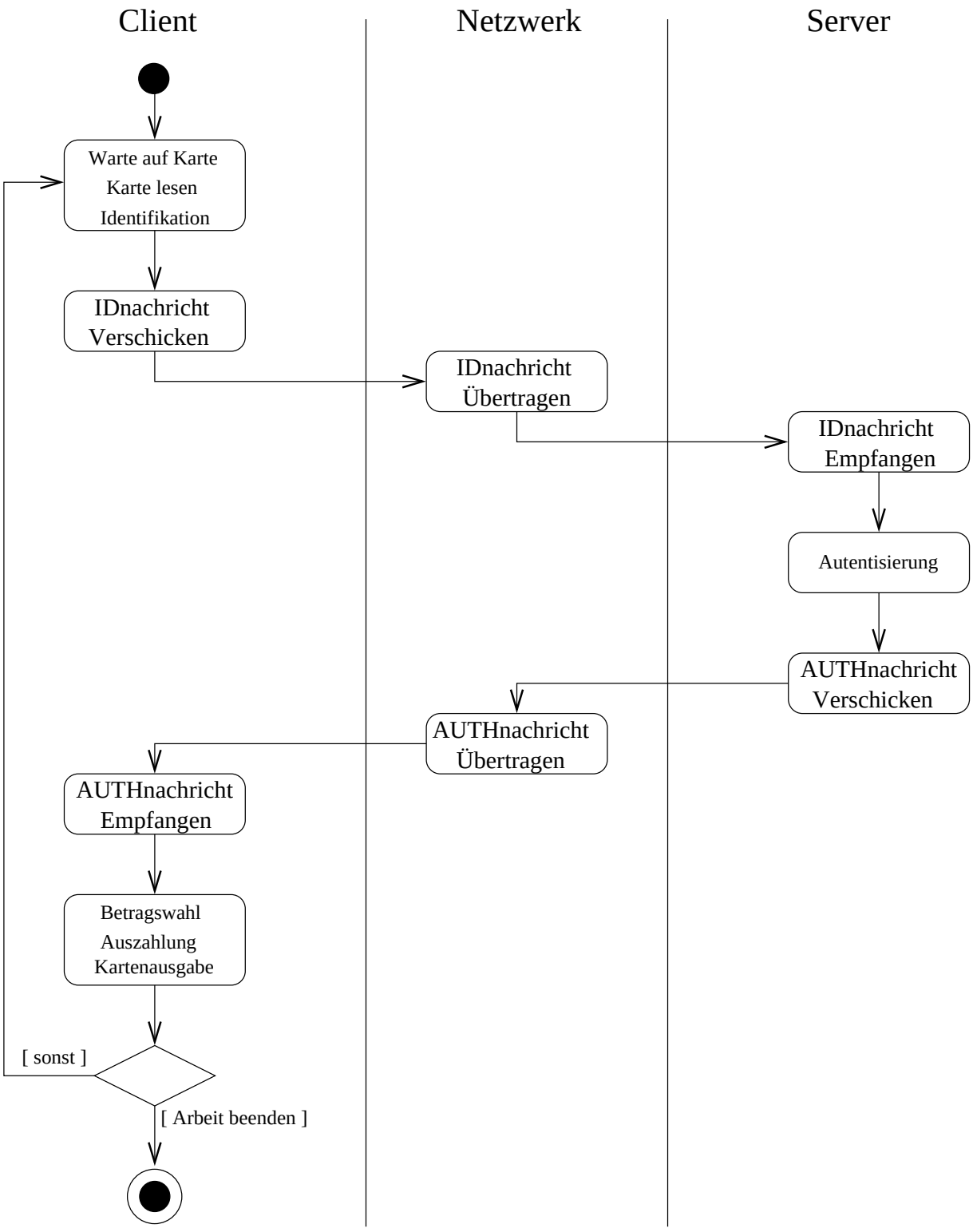

Abbildung 4.5.: Vereinfachtes Verhaltensmodell eines Geldautomaten

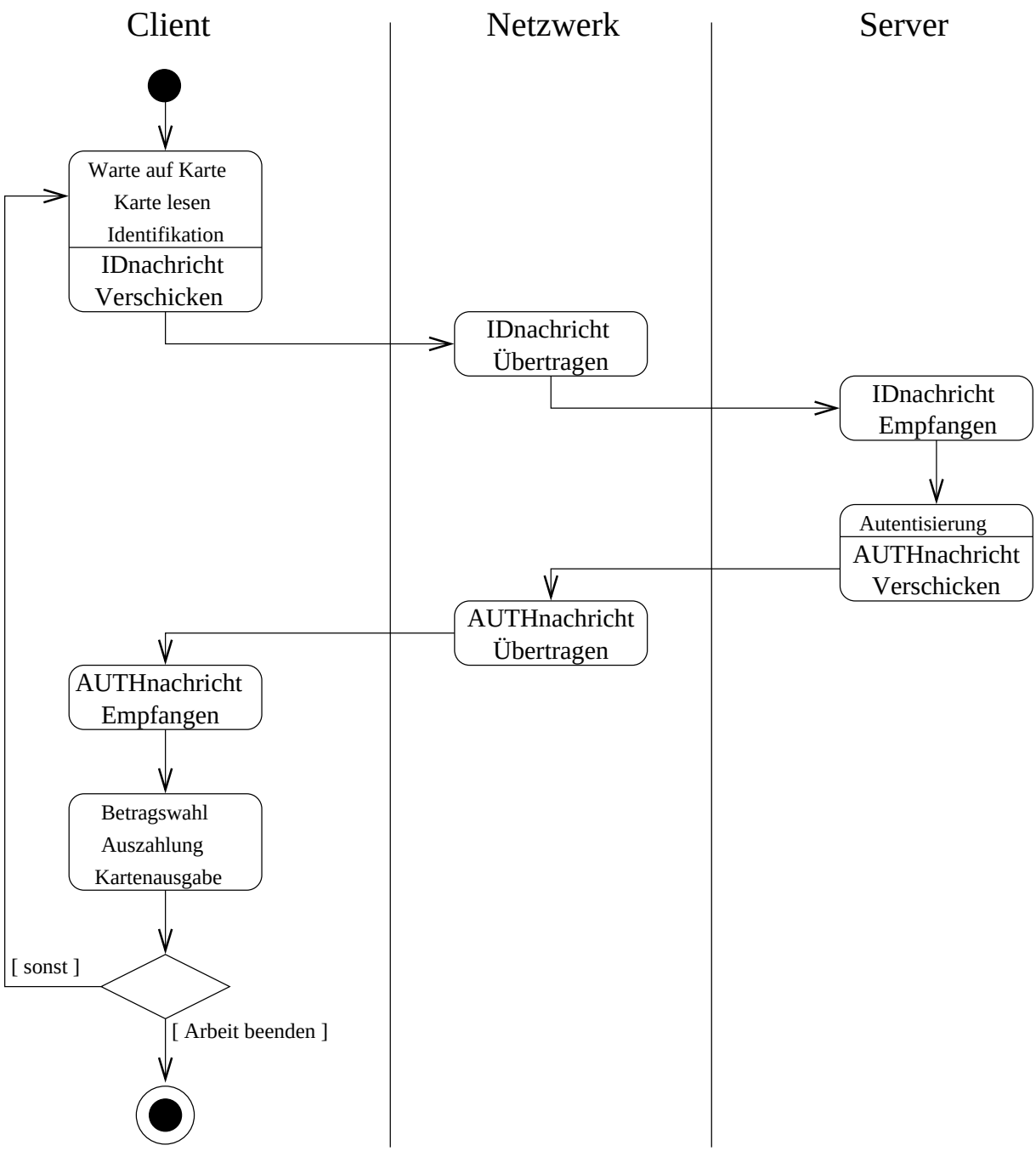

Abbildung 4.6.: Bindung der Aktivitäten an Nachrichten

Um Aktivitäten zu vermeiden, die von einer Nachricht losgelöst sind, kann man einen besonderen Typ von Nachrichten einführen: *NULL-Nachrichten*. Solche Nachrichten werden nicht tatsächlich verschickt. Sie dienen nur als Behälter für Aktivitäten. Mit dieser Vereinbarung ist eine weitere Reduktion der Anzahl von verschiedenen Elementen im Verhaltensmodell möglich. Das Activity Diagramm eines solchen Modells ist 4.7.

Nachdem das typische Verhalten eines Fat-Client Cashautomaten beschrieben wurde, kann man ein Simulationsmodell entwickeln, welches fähig ist solches Verhalten nachzuahmen.

4.4.1. Beschreibung des Modells

Die vollständigen Beschreibung eines Diskrete-Event Simulationsmodells enthält neben der Beschreibung einzelner Modellkomponenten auch ihr Verhalten und eine Umsetzung des Modells in ein lauffähige Software. Im weiteren Verlauf dieses Kapitels ist die Beschreibung aller notwendigen Bestandteile des Modells und ihr Verhalten angegeben. Die Beschreibung der Beispielimplementierung finden sie im Kapitel 5.

Beschreibung der Modelllogik

Aus den Abschnitten 4.2 und 4.3 kann man einige Anforderungen an das Simulationsmodell ableiten:

- Das Simulationsmodell muss auf mehrere Rechner verteilt werden.
- Jede Maschine muss mehrere Clients oder Server imitieren können.
- Clients müssen Antwortzeiten des Systems Protokollieren können.
- Verhaltensmodelle sollen als Parameter des Simulationsmodells dienen.

Aus diesen Anforderungen lassen sich notwendige Entitäten bestimmen:

- Durch die Verteilung des Modells auf mehrere Maschinen muss eine Entität definiert werden, die einen Teil des Simulationsmodells auf einer Maschine repräsentiert. Diese Entität bezeichne ich *Simulationsplattform*.
  - Datenpackete, die über das Netz verschickt werden sollen, müssen in ein zulässiges Format übertragen werden, und an das ComDevice übergeben werden. Diese Aufgabe kann die Entität *SendAgent* übernehmen.
  - Datenpackete, die Empfangen wurden, müssen von dem Netzformat in das von der Simulationssoftware verwendete Format, übertragen werden und an die Stelle, an die die Pakete adressiert sind, übergeben werden. Diese Aufgabe kann die Entität *ReceiveAgent* übernehmen.

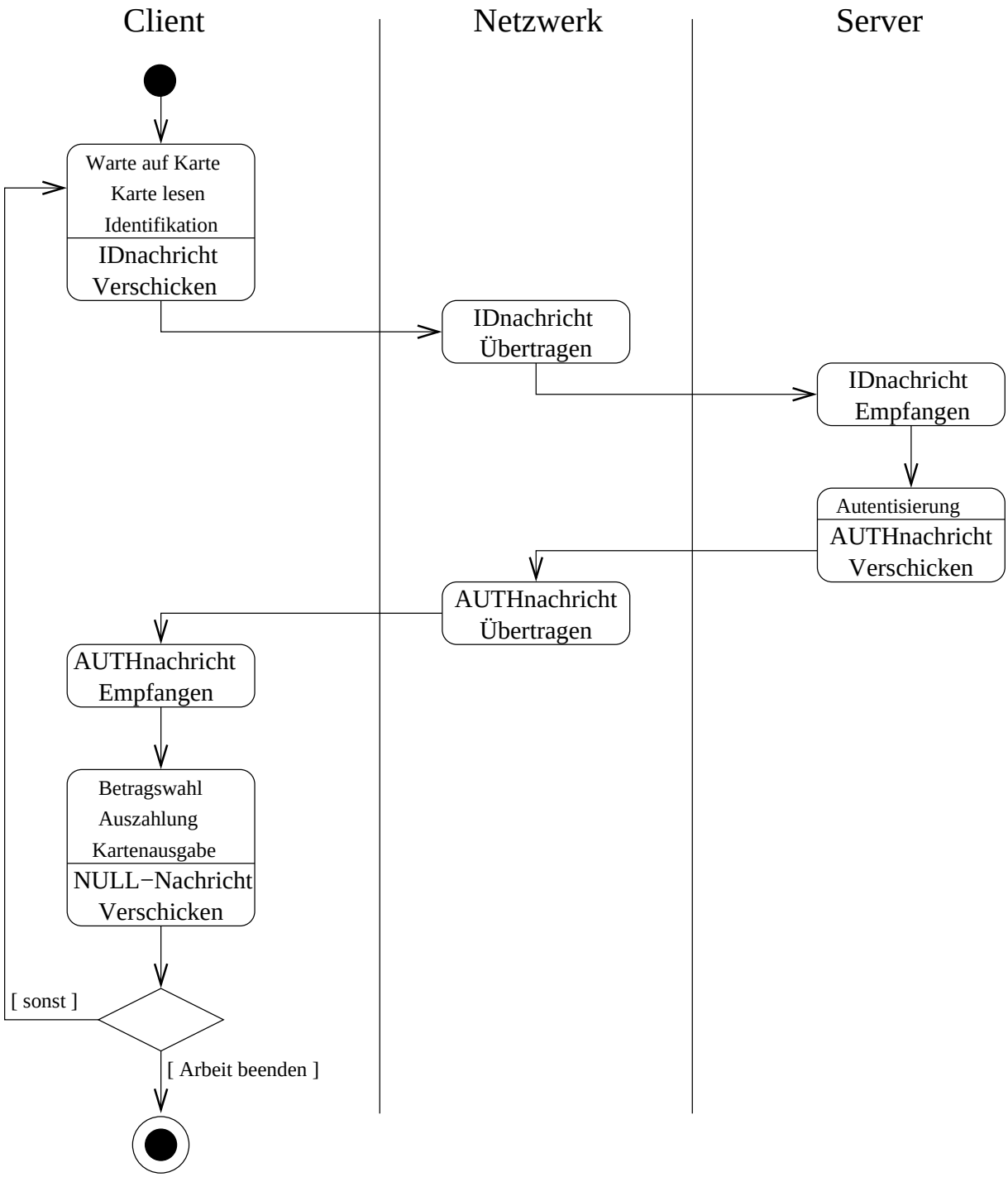

Abbildung 4.7.: Einführung der NULL-Nachricht.

- Weiterhin müssen einzelne Simulationsplattformen unter einander kommunizieren können, um Datenpackete über das Netzwerk verschicken und empfangen zu können. Diese Forderung macht eine weitere Entität notwendig, die als Schnittstelle zum Netzwerk dient. Diese Entität wird im weiteren Verlauf der Arbeit *ComDevice* genannt.
  - Innerhalb dieser Entität soll die Übertragung von Befehlen an einzelne Simulationsplattformen ermöglicht werden. Diese Aufgabe kann die Entität *DatagramAgent* übernehmen.
  - Die Übertragung komplexer Daten (z.B. Konfiguration der Simulationsplattform) kann hier ebenfalls angeboten werden. Diese Aufgabe kann die Entität *DataAgent* übernehmen.
  - Datenpackete, die von den Stellen des Typs Client oder Server verschickt und empfangen werden, können von der Entität *SimAgent* an das Netz übergeben und vom Netz empfangen werden.
- Eine Simulationsplattform muss über ein *User Interface* verfügen um von dem Durchführendem eines Simulationsexperimentes steuerbar und konfigurierbar zu sein.
- Eine Simulationsplattform muss über eine oder mehrere *Stellen* verfügen. Diese Stellen müssen das Verhalten von *Clients* oder *Servern* imitieren können.
- Clients und Stellen kommunizieren miteinander durch verschicken von *Nachrichten*. Wobei diese erst in *Datenpackete* zerlegt und dann erst verschickt werden.
- Clients müssen Antwortzeiten auf einzelne Nachrichten protokollieren können. Diese Einträge können in der Entität *Protokoll* gespeichert werden.
- Die Konfiguration des Simulationsmodells mittels eines Verhaltensmodells wird durch die Entität *SimConfig* vorgenommen.

Im Strukturdiagramm 4.8 wird die Struktur einer Simulationsplattform gezeigt. In den folgenden Abschnitten werden einzelne Komponenten des Simulationsmodells und ihr Verhalten genau beschrieben.

Simulationsplattform

Die *Simulationsplattform* ist die zentrale Komponente des Simulationsmodells. Diese Station dient als Verbindung für alle anderen Teile des Modells. Die einzelnen Komponenten und Routinen einer Simulationsplattform werden in der Tabelle 4.1 angegeben.

Alle genannten Entitäten werden im weiteren Verlauf beschrieben. Die beiden Buffer p_receiv und p_tosend sind durch die Klasse *SynObjCont* implementiert. Die komplette Beschreibung der Implementierung der übrigen Methoden befindet sich im Kapitel 5.

Komponenten der Simulationsplattform

com : ComDevice	Eine Referenz auf die Entität ComDevice.
config : SimConfig	Eine Referenz auf die Entität SimConfig.
stelle : Stelle[]	Ein Array mit Referenzen auf Stellen-Entitäten des Typs Client und Server.
recvAgent : Agent	Eine Referenz auf die Entität ReceiveAgent.
sendAgent : Agent	Eine Referenz auf die Entität SendAgent.
p_receiv : Buffer	Ein Zwischenspeicher für UDP-Pakete die vom Netzwerk empfangen wurden.
p_tosend : Buffer	Ein Zwischenspeicher für Datenpackete die an das Netzwerk übergeben werden sollen.

Routinen der Simulationsplattform

configure(String)	Methode zur Konfiguration einer Simulationsplattform. Die Zeichenkette, die als Parameter übergeben wird, soll ein Verhaltensmodell des Simulationsmodells beinhalten.
getProtokoll() : String	Methode zum Sammeln von Protokollen der Lokalen Stellen. Das gewonnene Protokoll enthält alle Protokolle der lokalen Stellen vom Typ Client.
getProtokollFrom(String) : String	Methode zum Sammeln von Protokollen von entfernten Stellen.
receiveFromNet (DatagramPacket)	Methode zum Empfangen von UDP-Paketen vom Netzwerk.
sendToNet (DatagramPacket)	Methode zum Senden von UDP-Paketen an das Netzwerk.
sendToStelle(int, Datenpacket)	Methode zum Übermitteln von Datenpacketen an eine lokale Stelle, wobei der erste Parameter die Adresse der Stelle in dem Array der Stellen anzeigt an die das Datenpacket übergeben werden soll, und der zweite Parameter das zu übermittelnde Datenpacket.
startSimulationPlatform()	Methode zum Starten der lokalen Simulationsplattform.
stopSimulationPlatform()	Methode zum Stoppen der lokalen Simulationsplattform.
runSlaveSim()	Methode zum Betreiben der lokalen Simulationsplattform im Slave-Modus. In diesem Modus wartet die Simulationsplattform auf Anweisungen die das ComDevice vom Netzwerk empfängt, und führt diese aus.
runTestSim()	Methode zum Betreiben der lokalen Simulationsplattform im Test-Modus. In diesem Modus läuft die Simulationsplattform eine Kette von Anweisungen ab und führt diese aus. Die Anweisungskette ist im Activity Diagramm 4.9 dargestellt.
runMasterSim()	Methode zum Betreiben der lokalen Simulationsplattform im Master-Modus. In diesem Modus soll eine Simulationsplattform interaktiv vom Durchführendem des Simulationsexperimentes gesteuert werden.

Tabelle 4.1.: Komponenten und Routinen der Simulationsplattform

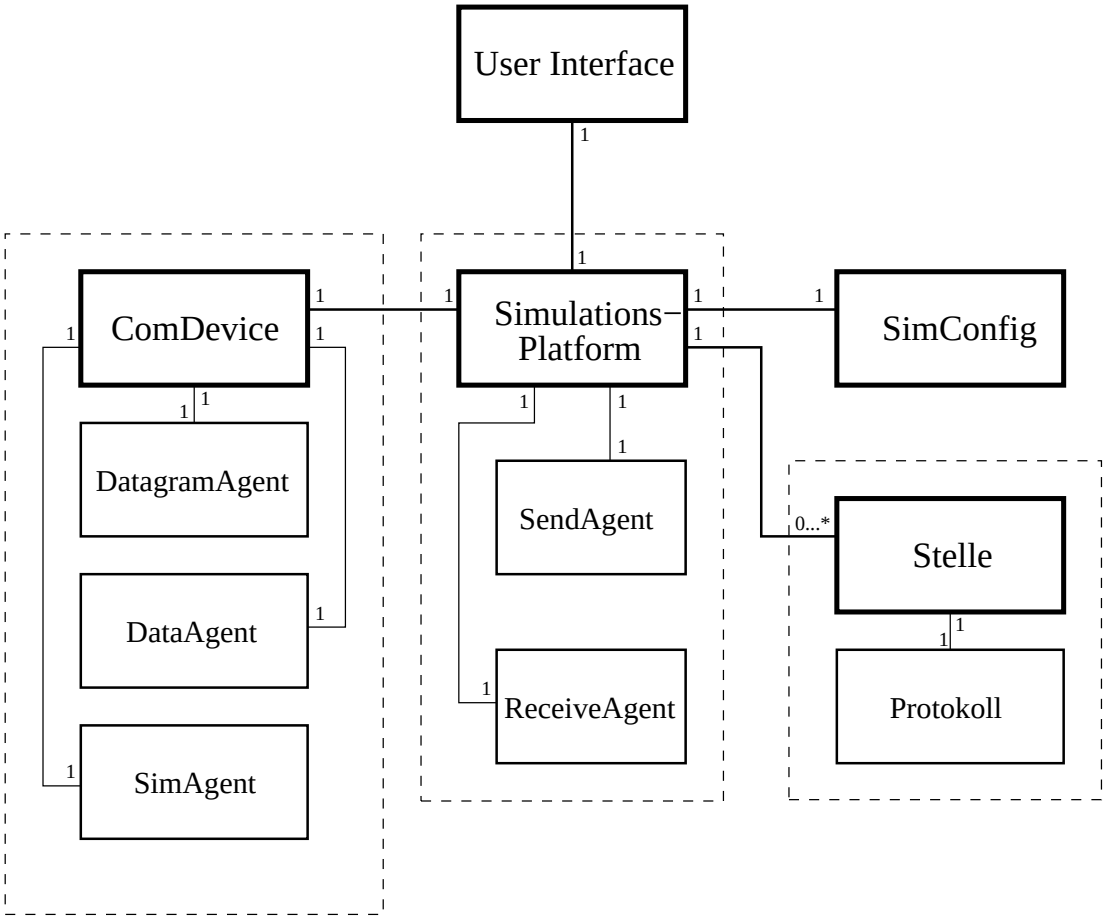

Abbildung 4.8.: Strukturdiagramm einer Simulationsplattform

SendAgent

Diese Entität implementiert einen Berechnungsfaden, der dem p_tosend-Buffer der Simulationsplattform an die er gebunden ist ein *Datenpacket* entnimmt, dieses in ein Datagram umrechnet, und dieses Datagram über die Simulationsplattform zum Versenden an die Kommunikationseinheit ComDevice übergibt. Die einzelnen Schritte, die dabei gemacht werden, werden im Activity Diagramm 4.10 beschrieben. Die Beschreibung der Implementierung befindet sich im Kapitel 5.

ReceiveAgent

Diese Entität implementiert einen Berechnungsfaden, der dem p_received-Buffer der Simulationsplattform an die er gebunden ist ein Datagram entnimmt, dieses in ein Datenpacket umrechnet, und in die Liste p_received der Stelle einträgt, für die das Datenpacket bestimmt ist. Die Übergabe des Datenpacketes an die entsprechende Stelle wird mittels der Methode sendToStelle() der Simulationsplattform bewerkstelligt. Die einzelnen Schritte, die dabei gemacht werden, werden im Activity Diagramm

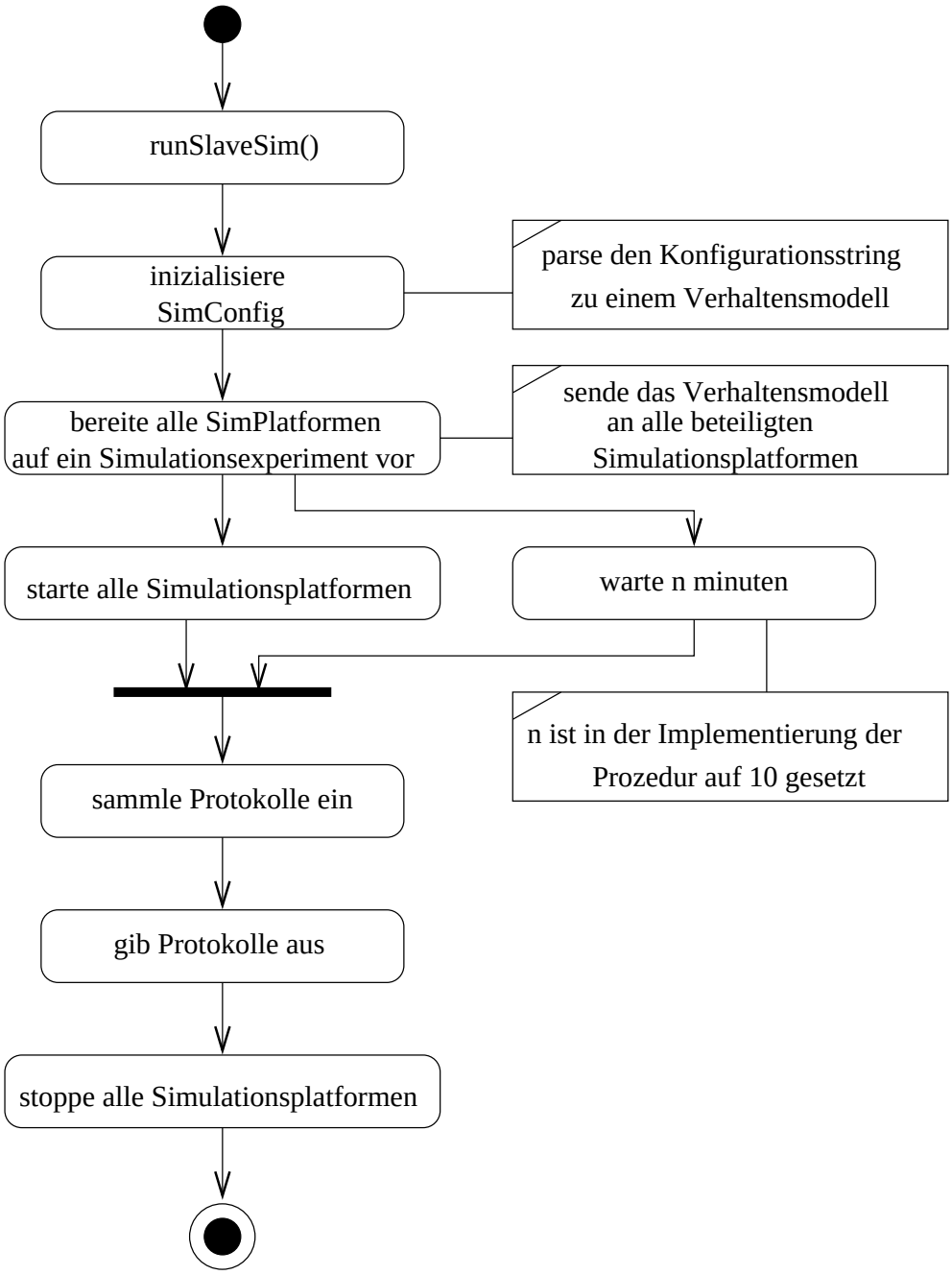

Abbildung 4.9.: Activity Diagramm der Methode runTestSim().

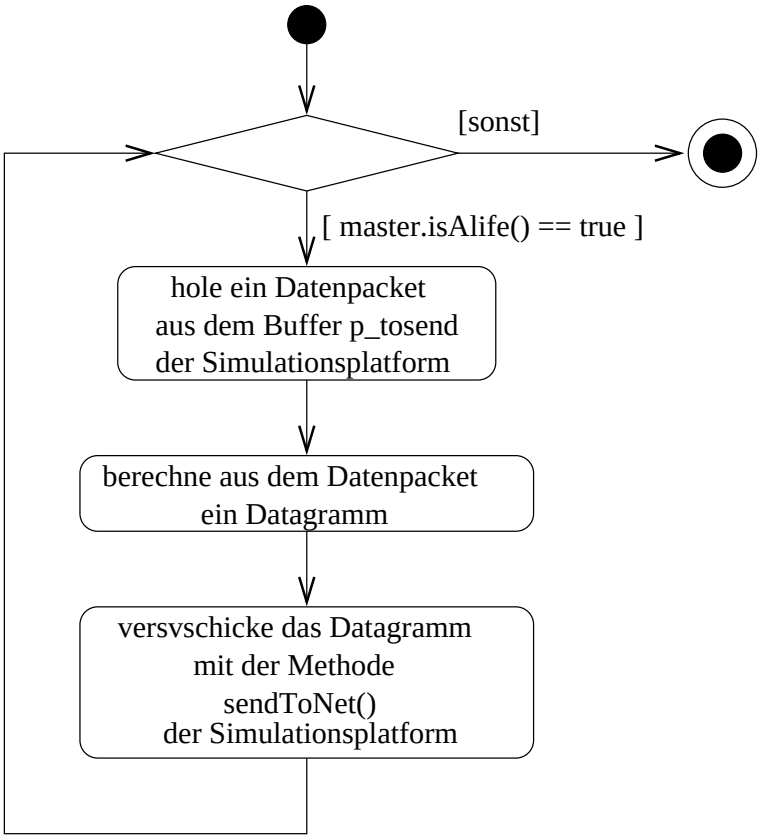

Abbildung 4.10.: Activity Diagramm der Hauptmethode der Entität SendAgent.

4.11 beschrieben. Die Beschreibung der Implementierung befindet sich im Kapitel 5.

ComDevice

Diese Entität stellt die Schnittstelle der Simulationsplattform zum Netzwerk dar. Die notwendigen Komponenten und Methoden dieser Entität werden in der Tabelle 4.2 beschrieben. Die Dokumentation der Routinen und ihre Implementierung finden sie im Kapitel 5.

DatagramAgent

Diese Entität implementiert einen Agenten zum Senden und Empfangen von Kommando-Datagramen. Empfangene Datagrame werden zwischengespeichert um von der Entität ComDevice abgerufen zu werden. Notwendige Komponenten und Methoden dieser Entität sind in der Tabelle 4.3 aufgeführt. Details der Implementierung können im Kapitel 5 eingesehen werden.

Komponenten des ComDevice	
master : Simulationsplattform	Eine Referenz auf die Entität Simulationsplattform.
cmdMgr : DatagramAgent	Eine Referenz auf die Entität DatagramAgent.
datMgr : DataAgent	Eine Referenz auf die Entität DataAgent.
simMgr : SimulationAgent	Eine Referenz auf die Entität SimulationAgent.

Routinen des ComDevice	
ComDevice (Simulationsplattform)	Erzeugt ein neues ComDevice, wobei die folgende Ports genutzt werden: DatagramAgent(5000,5001), DataAgent(5004), SimulationAgent(5006,5007).
sendProtokoll()	Sendet das Protokoll der lokalen Simulationsplattform an den Simulationsmaster.
sendComand()	Sendet über den DatagramAgenten ein Kommando an einen Empfänger.
configureSlave (String)	Konfiguriert die Simulationsplattform mit dem übergebenen Konfigurationsstring.
startSimulationAgent()	Startet den SimulationsAgenten. (Ports:5006, 5007)
stopSimulationAgent()	Stoppt den SimulationsAgenten.
startSimulationPlatform()	Startet die Simulation auf der Simulationsplattform.
stopSimulationPlatform()	Stoppt die Simulation auf der Simulationsplattform.
sendSimulationDatagram (DatagramPacket)	Sendet das Datagram über den SimulationsAgenten.
receiveSimulationDatagram (DatagramPacket)	Übergibt das Datagram an die Simulationsplattform.
sendString (String)	Sendet über den DataAgenten ein String an einen Empfänger.
receiveString() : String	Empfängt über den DataAgenten einen String.
slaveModeComunication()	Methode zur Kommunikation im Slave-Modus: Empfange einen Befehl vom Netz und führe diesen unverzüglich aus.

Tabelle 4.2.: Komponenten und Routinen der Entität ComDevice.

Komponenten des DatagramAgenten	
master : ComDevice	Eine Referenz auf die Entität ComDevice.
slave : DatagramLurker	Eine Referenz auf die einen Agenten, der Datagrame vom Netz empfängt und diese Zwischenspeichert.
cmdMasterSocket : DatagramSocket	Socket zum Senden von Kommandos.

Routinen des DatagramAgenten	
receiveComand() : DatagramPacket	Methode zum Empfangen eines Kommandos.
sendComand(String, String)	Methode zum Senden eines Kommandos an eine entfernte Simulationsplattform.
executeComand (DatagramPacket)	Methode zum Ausführen eines KommandoDatagrams.

Tabelle 4.3.: Komponenten und Routinen der Entität DatagramAgent.

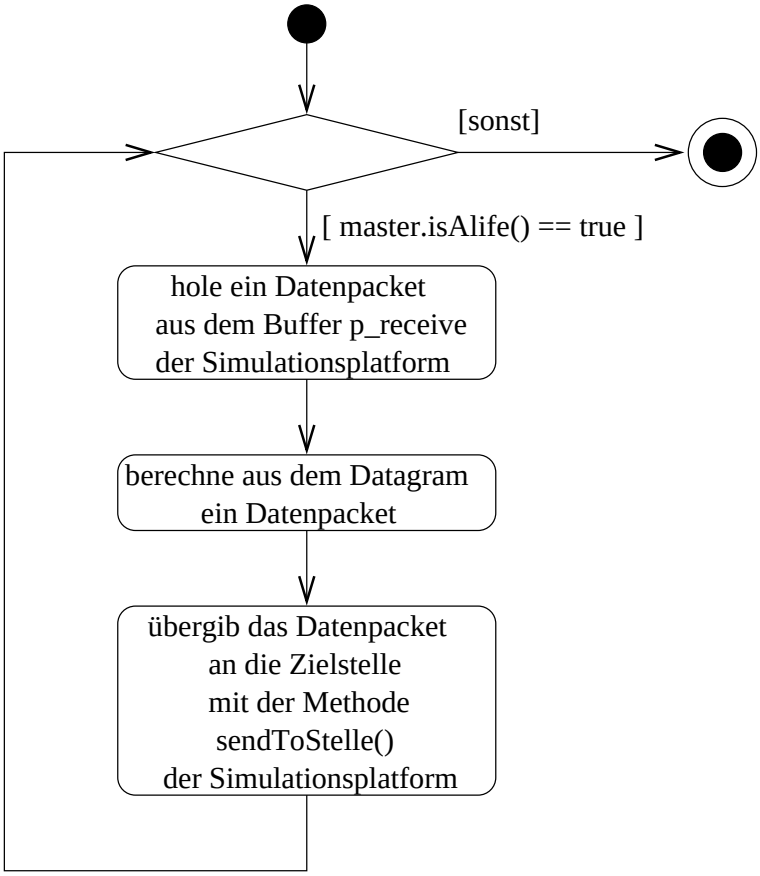

Abbildung 4.11.: Activity Diagramm der Hauptmethode der Entität ReceiveAgent.

DataAgent

Diese Entität implementiert einen Agenten zum Senden und Empfangen von komplexen Daten. Zu solchen Daten gehören z.B.: Strings, Protokolle von Simulationsläufen und Konfigurationen von Simulationsplattformen. Notwendige Komponenten und Methoden dieser Entität sind in der Tabelle 4.4 aufgeführt. Details der Implementierung können im Kapitel 5 eingesehen werden.

SimAgent

Diese Entität implementiert einen Agenten zum Senden und Empfangen von SimulationsDatagramen. Notwendige Komponenten und Methoden dieser Entität sind in der Tabelle 4.5 aufgeführt. Details der Implementierung können im Kapitel 5 eingesehen werden.

Server

Serversationen sind Stationen, die Nachrichten von Clients empfangen, die Bearbeitung dieser Nachrichten durch eine bestimmte Wartezeit simulieren, und wenn

Komponenten des DataAgenten	
master : ComDevice	Eine Referenz auf die Entität ComDevice.
dataReceiveSocket : ServerSocket	Socket zum Empfangen von Daten.
dataSendSocket : Socket	Socket zum Senden von Daten.

Routinen des DataAgenten	
receiveObject() : Object	Methode zum Empfangen eines Objektes.
sendObject(InetAddress, Object)	Methode zum Senden eines Objektes an eine entfernte Simulationsplattform.

Tabelle 4.4.: Komponenten und Routinen der Entität DataAgent.

Komponenten des SimAgenten	
master : ComDevice	Eine Referenz auf die Entität ComDevice.
simulationSocket : DatagramSocket	Socket zum Senden von SimulationsDatagramen.
receivingSocket : DatagramSocket	Socket zum Empfangen von SimulationsDatagramen.
MaxSizeOfDatagram : Integer	Maximale Größe eines zu empfangenen Datagrams (in Bytes).

Routinen des SimAgenten	
receive() : DatagramPacket	Methode zum Empfangen eines DatagramPacket.
send(DatagramPacket)	Methode zum Senden eines DatagramPacket an eine entfernte Simulationsplattform.
run()	Hauptmethode des Simulationsagenten. Das Das Verhalten dieser Methode wird in dem Activity Diagramm 4.12 beschrieben.

Tabelle 4.5.: Komponenten und Routinen der Entität SimAgent.

notwendig dem entsprechenden Client eine Antwort schicken. Bestandteile einer Serverstation sind in der Tabelle 4.6 angegeben. Die Implementierung der Serverstation in der Programmiersprache Java finden sie in dem Kapitel 5.

Die Entität *Server* ist implementiert durch die Klasse *Stelle*. Das Attribut S_{id} ist in dieser Implementierung zusammengesetzt aus zwei Komponenten:

- Der Id der Stelle vom Typ N_id,
- und der Ortsangabe der Stelle vom Typ *Ort*.

Das Verhalten einer Serverstation ist sehr einfach und ist im Activity Diagramm 4.13 beschrieben.

Die Verhaltensmuster der Hilfsmethoden respond(), sendNachricht(), checkNachricht(), receivePacket() und sendPacket() sind in den Activity Diagrammen 4.14, 4.15, 4.16, 4.17 und 4.18 angegeben.

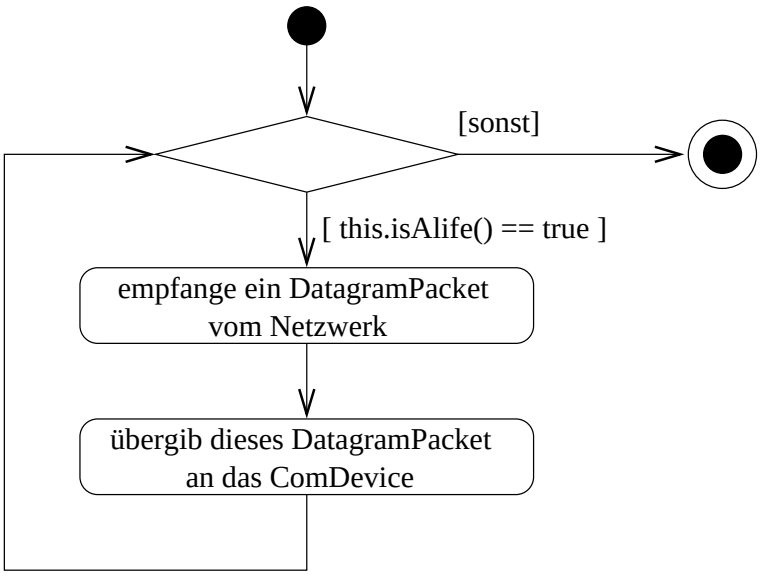

Abbildung 4.12.: Activity Diagramm der Hauptmethode der Entität SimAgent.

Komponenten der Entität Server

S_{id}	Der Name der Serverstation.
p_received : SynObjCont	Eine Liste der empfangenen Datenpackete.
p_tosend : SynObjCont	Eine Liste der Datenpackete, die verschickt werden sollen.
n_incomplete : java.util.Vector	Eine Liste der empfangenen Nachrichten, die noch nicht vollständig sind (einige Datenpackete fehlen).
n_received : java.util.Vector	Eine Liste der empfangenen Nachrichten.
n_tosend : java.util.Vector	Eine Liste der Nachrichten, die verschickt werden sollen.

Routinen der Entität Server

$f_{serv}(N)$	Eine Funktion, die die Bearbeitungszeit der Nachricht durch den Server berechnet.
$f_{reply}(N)$	Eine Funktion, welche die Antwortnachricht auf die aktuell bearbeitete Nachricht berechnet.
run()	Hauptroutine der Entität Server.
respond()	Methode zur Bearbeitung empfangener Nachrichten.
sendNachricht()	Methode zum Senden von Nachrichten.
checkNachricht()	Methode zur Überprüfung der Liste n_incomplete auf enthalten vollständiger Nachrichten.
receivePacket()	Methode zum Empfangen von Datenpacketen.
sendPacket()	Methode zum Senden von Datenpacketen.
sendToSim (Object)	Methode zum Senden von Objekten an die Simulationsplattform.

Tabelle 4.6.: Komponenten und Routinen der Entität Server.

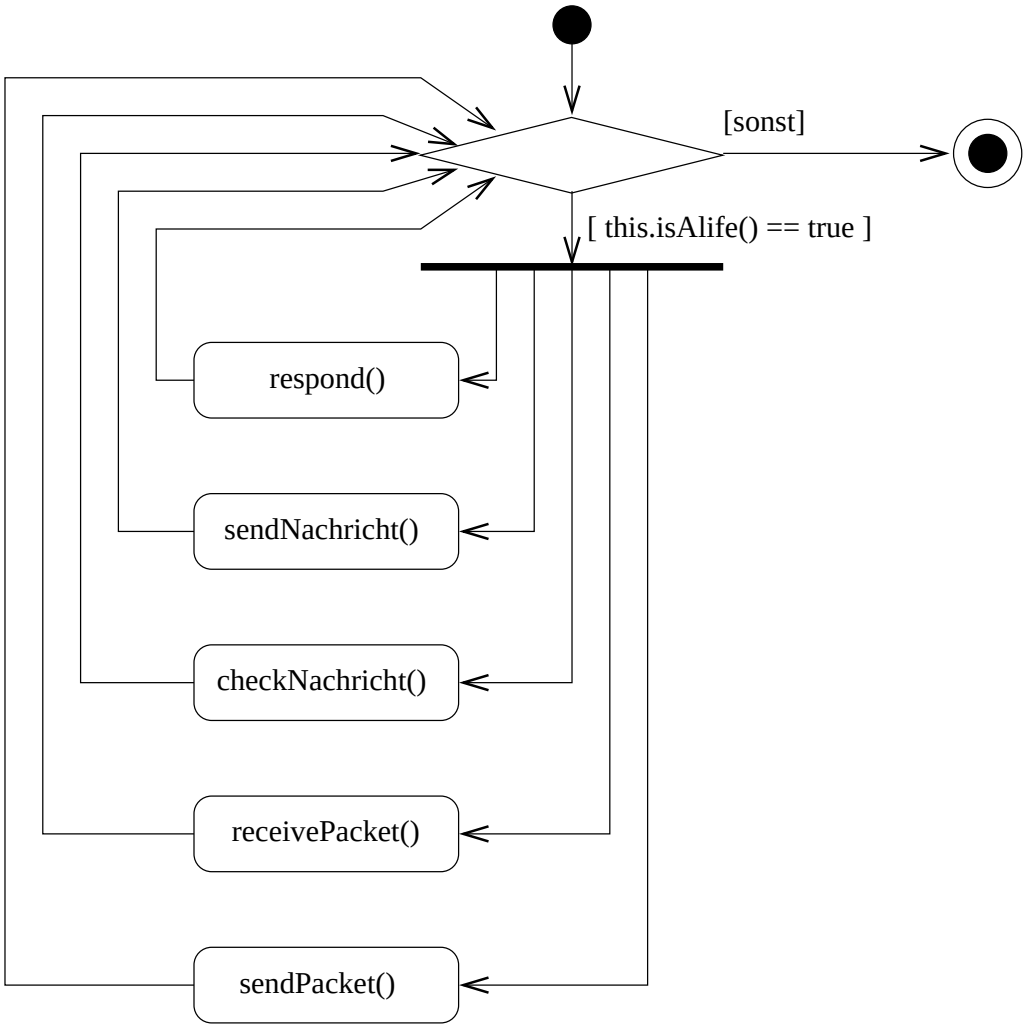

Abbildung 4.13.: Activity Diagramm der Hauptmethode der Entität Server.

Das Verhaltensmodell, das zur Konfiguration der Stelle Server verwendet wird und in dem die Funktionen $f_{serv}(N)$ und $f_{reply}(N)$ beschrieben werden finden sie im Abschnitt 4.4.2.

Client

Clientstationen sind Stationen, die in unserem Modell das Verhalten von Clients modellieren. Das Modell dieser Station ist der Station Server sehr ähnlich. Die Bestandteile von Clientstationen unterscheiden sich kaum von denen einer Serverstation. Den Unterschied zwischen den beiden Stationsarten macht ihr Verhalten aus. Die Funktionen $f_{wait}(N)$, $f_{start}(C)$ und die Markierung C_{inuse} sind neu dazugekommen. Die Auflistung aller notwendigen Komponenten dieser Entität ist in der Tabelle 4.7 angegeben.

Das Attribut C_{id} hat dieselbe Beschaffenheit wie das Attribut S_{id} der Entität

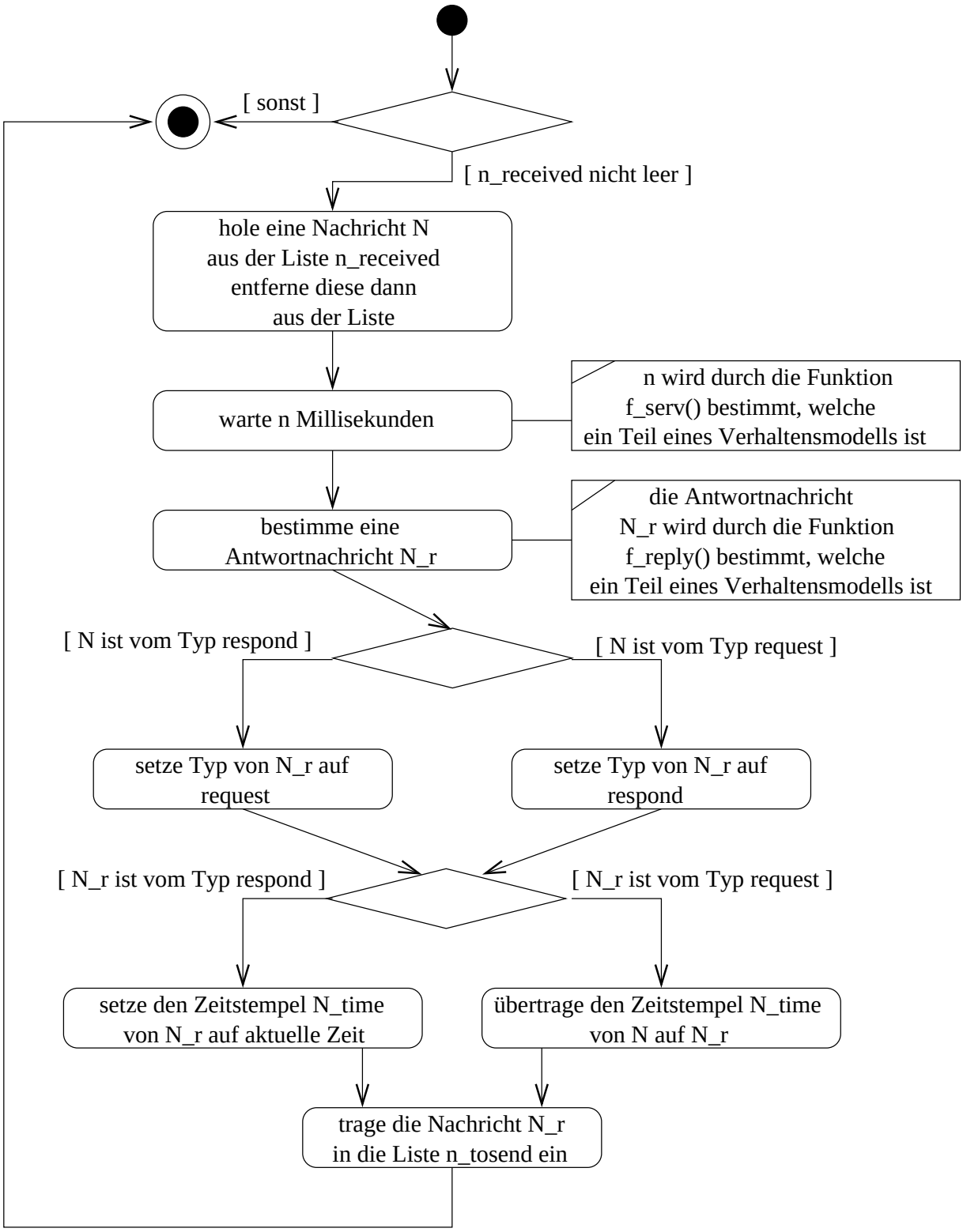

Abbildung 4.14.: Activity Diagramm der Methode der respond() der Entität Server.

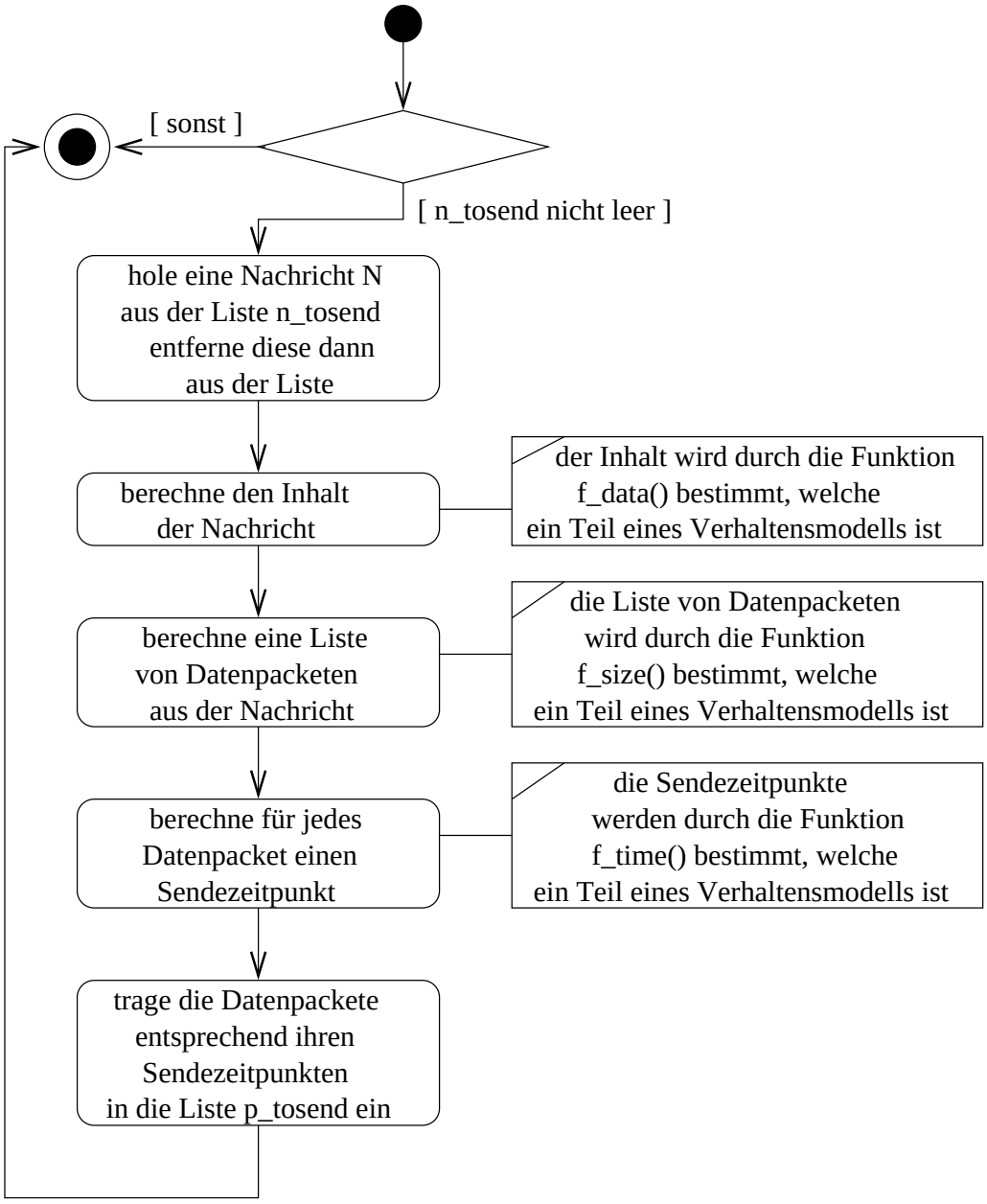

Abbildung 4.15.: Activity Diagramm der Methode sendNachricht().

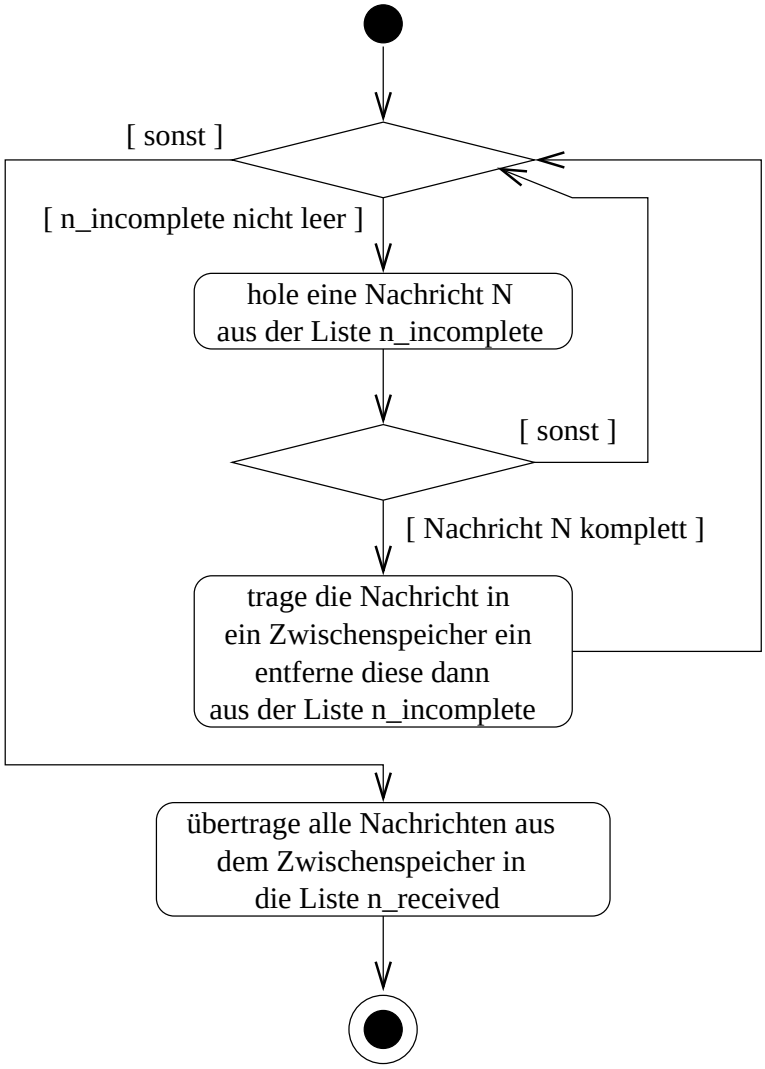

Abbildung 4.16.: Activity Diagramm der Methode checkNachricht().

Server.

Die grundlegenden Verarbeitungsmuster für Datenpackete und Nachrichten stimmen mit denen des Servers in den meisten Punkten überein. Die Methode mit dem abweichendem Verhalten ist Methode respond(). Die Protokollierung der Systemantwortzeiten wird nur auf dem Client durchgeführt, da der Grundsatz der Erwartungskonformität nur auf dem Client einen Sinn ergibt. Das Activity Diagramm 4.19 dokumentiert die Veränderung im Verhalten.

Die Methode start() wird nur auf dem Client ausgeführt. Diese Methode simuliert die Ankunft eines Kunden an einem freien Terminal. Die Simulation dieses Vorgangs wird im Activity Diagramm 4.20 beschrieben.

Weiterhin muss eine Veränderung in der Hauptmethode der Entität Server vorgenommen werden um diese an die Entität Client anzupassen. Diese Veränderung ist im Activity Diagramm 4.21

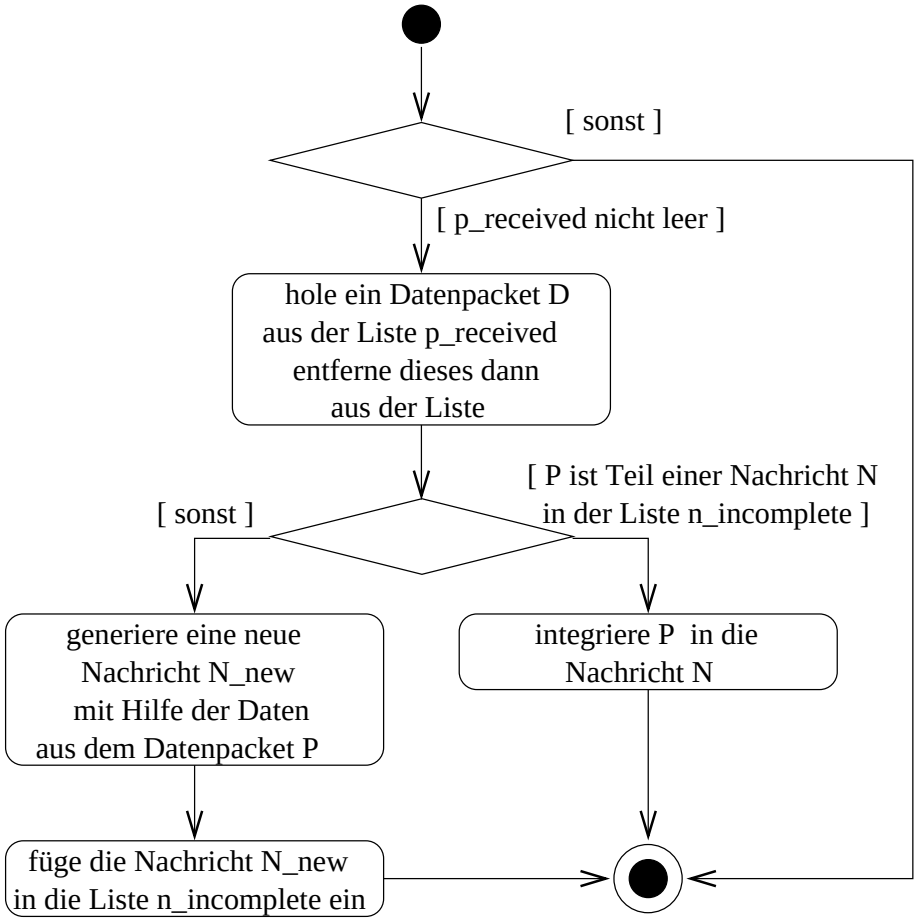

Abbildung 4.17.: Activity Diagramm der Methode receivePacket().

Die Entität *Client* ist implementiert durch die Klasse *ClientStelle*. Die Details der Implementierung finden sie im Kapitel 5.

Die Möglichkeiten der Modellierung des Verhaltens eines Clients müssen sehr flexibel sein, da die Studie Zusammenhänge zwischen Clientverhalten und Netzbelastung untersucht. Deshalb sollte wie auch bei der Serverstation eine Möglichkeit bestehen das Verhalten der Station durch ein Verhaltensmodell zu verändern. Das Verhalten dieser Station kann durch die Anpassung der Funktionen $f_{serv}(N)$, $f_{reply}(N)$, $f_{wait}(N)$ und $f_{start}(C)$ genügend variiert werden. Eine Beschreibung des Verhaltensmodells finden sie in dem Abschnitt 4.4.2.

Protokoll

Diese Entität soll, gebunden an eine Station des Typs Client Daten über die Antwortzeiten des Systems sammeln. Eine mögliche Implementierung dieser Entität könne sie im Kapitel 5 finden.

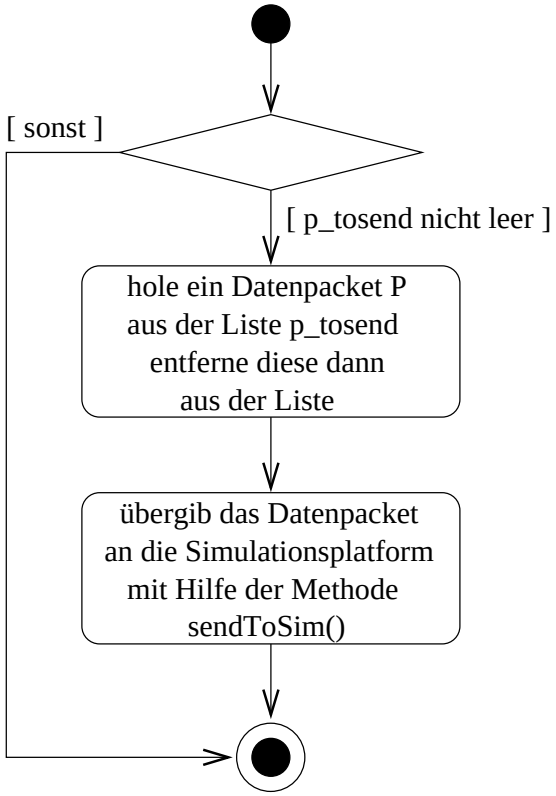

Abbildung 4.18.: Activity Diagramm der Methode sendPacket().

Nachricht

Nachricht ist eine dynamische Entität innerhalb des Simulationsmodells. Diese Entität wird zur Kommunikation zwischen Server und Client verwendet. Die Komponenten einer Nachricht sind in der Tabelle 4.8 zu finden.

Die Implementierungsbeschreibung dieser Entität befindet sich in Kapitel 5. Informationen zum Verhaltensmodell befinden sich in Abschnitt 4.4.2.

Datenpacket

Datenpacket ist eine dynamische Entität innerhalb des Simulationsmodells. Diese Entität enthält einen Teil einer Nachricht. Datenpackete werden gebraucht, um eine Nachricht in Einheiten bestimmter Größe über das Netzwerk zu verschicken. Die Komponenten eines Datenpacketes sind in der Tabelle 4.9 aufgelistet.

Die Implementierungsbeschreibung dieser Entität befindet sich in Kapitel 5.

User Interface

Aus der Sicht dieser Arbeit reicht ein rudimentäres User Interface aus um Simulationsexperimente durchzuführen. Die Implementierung eines einfachen Interfaces finden Sie in Kapitel 5.

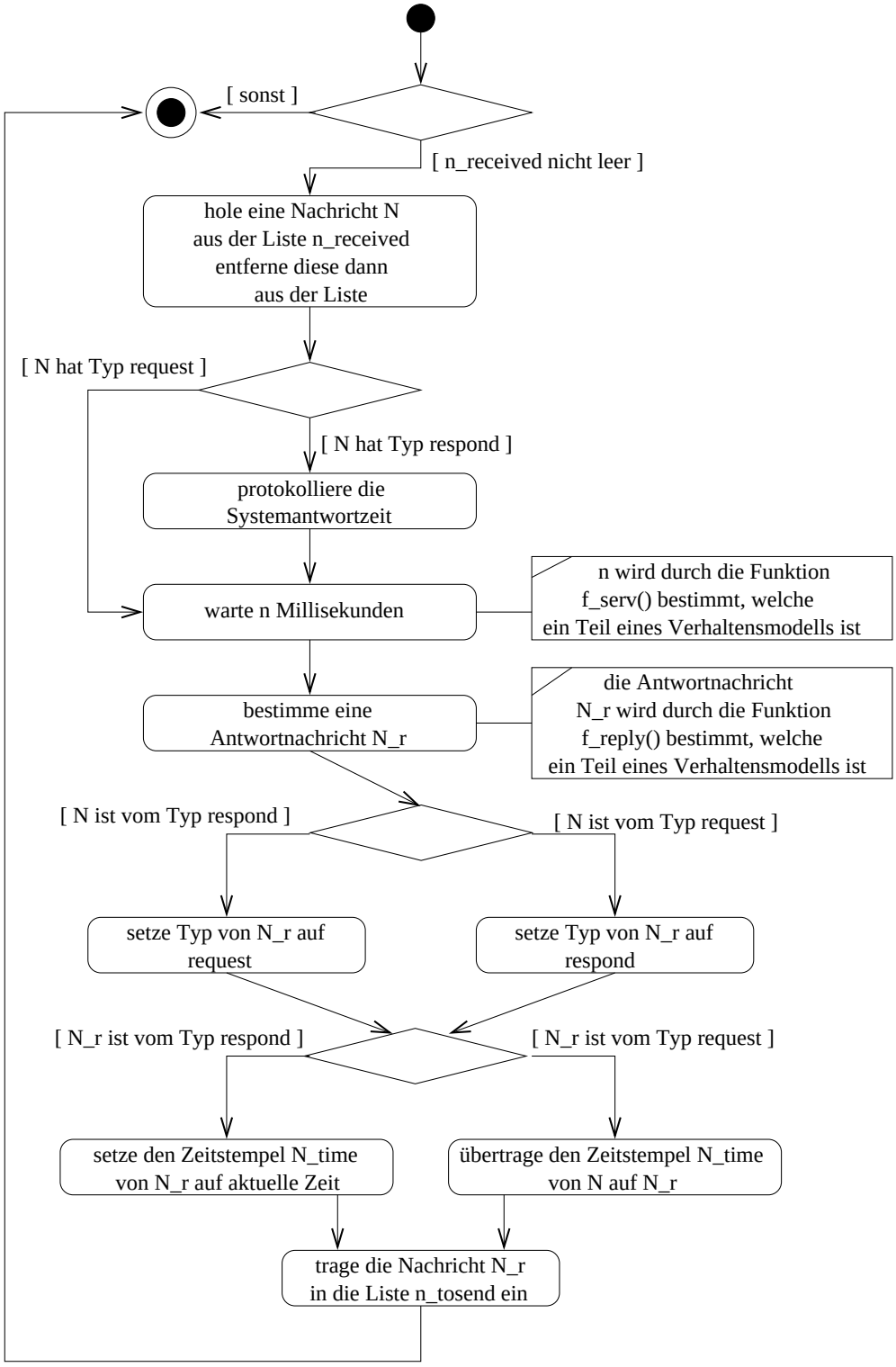

Abbildung 4.19.: Activity Diagramm der Methode respond() der Entität Client.

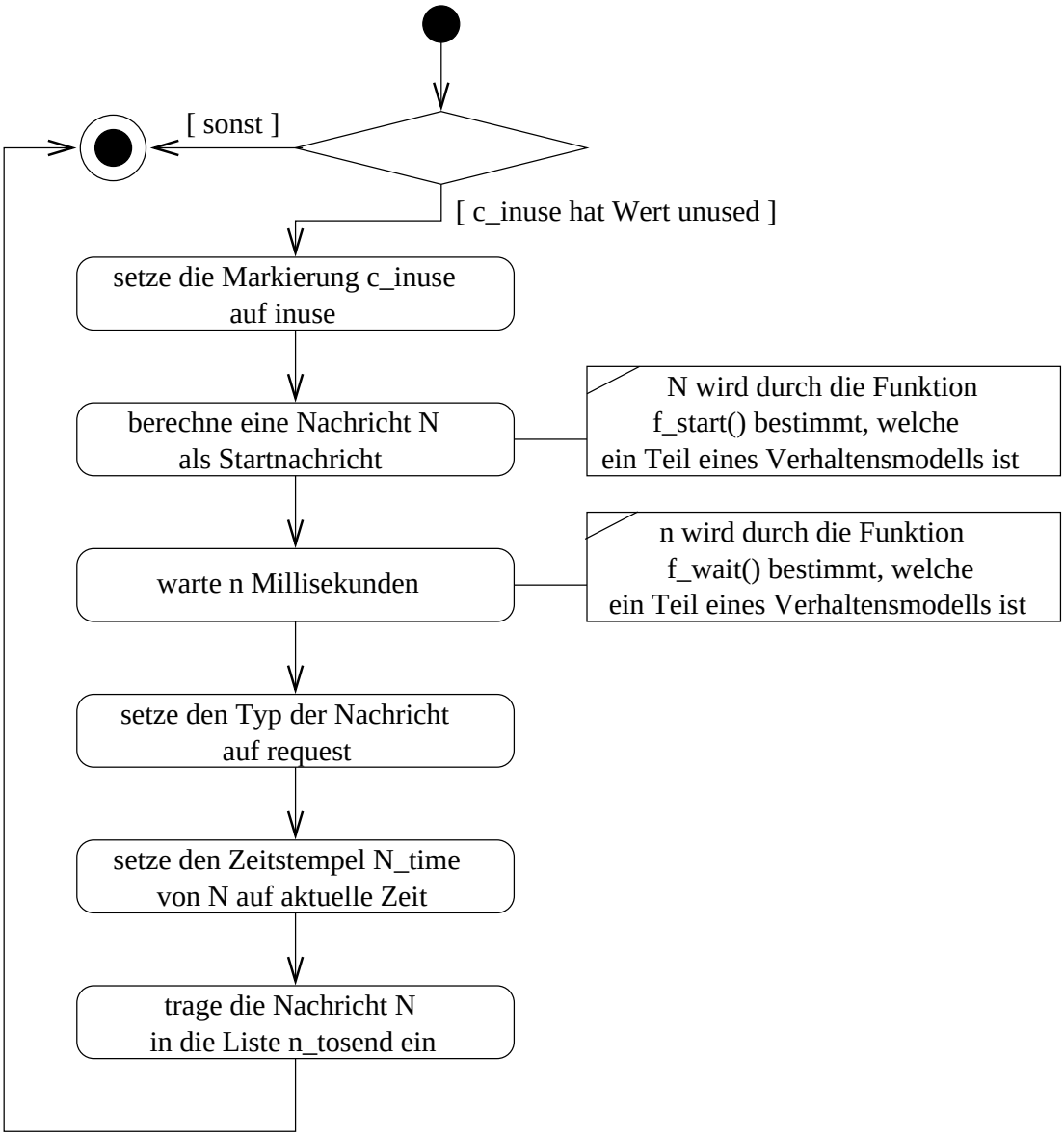

Abbildung 4.20.: Activity Diagramm der Methode start() der Entität Client.

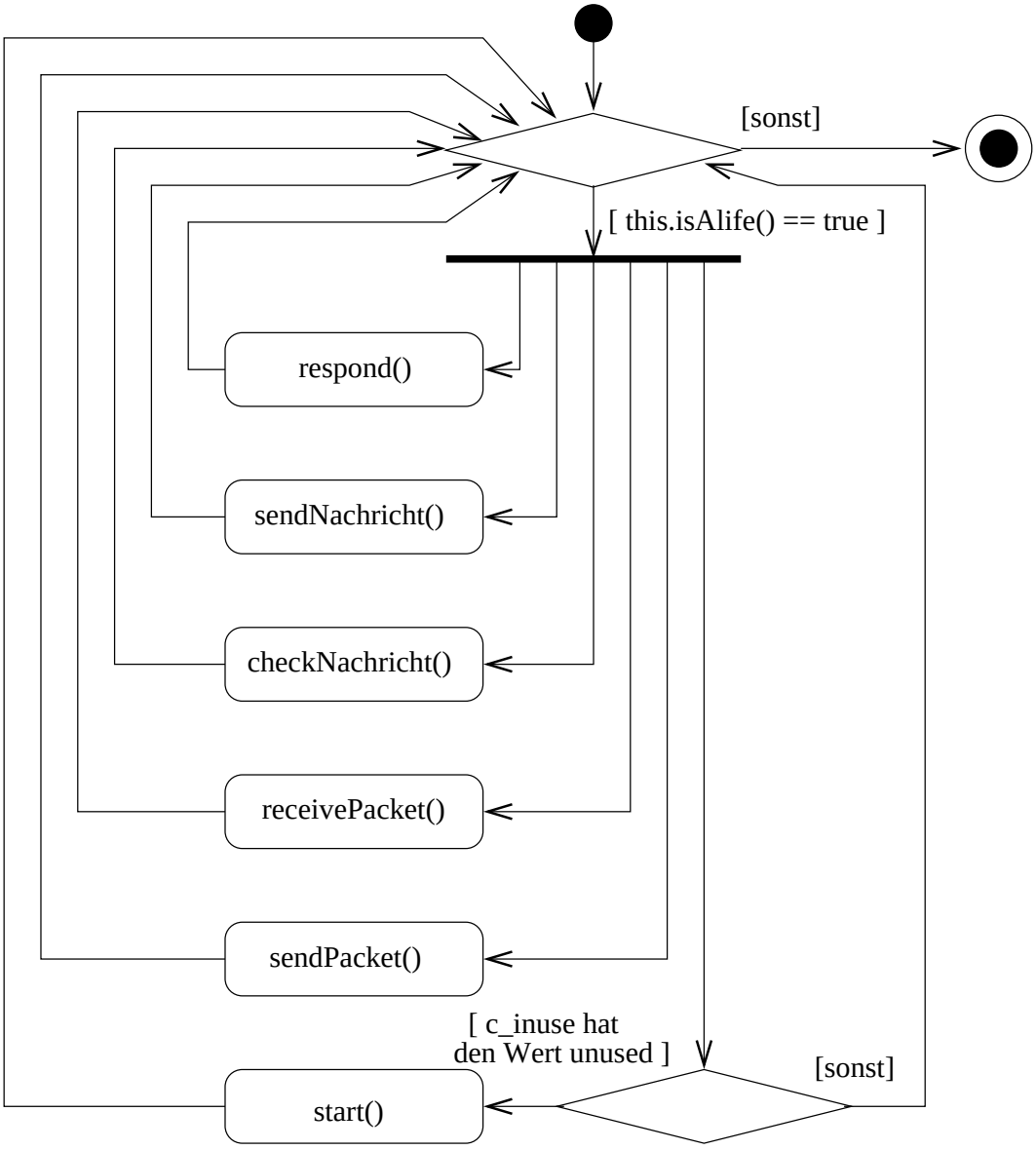

Abbildung 4.21.: Activity Diagramm der Hauptmethode der Entität Client.

Komponenten der Entität Client

C_{id}	Der Name der Serverstation.
C_{inuse}	Eine Markierung der Clientstation, die anzeigt ob ein Client momentan in Benutzung ist. Mögliche Werte der Markierung sind *inuse* und *unused*.
p_received : SynObjCont	Eine Liste der empfangenen Datenpackete.
p_tosend : SynObjCont	Eine Liste der Datenpackete, die verschickt werden sollen.
n_incomplete : java.util.Vector	Eine Liste der empfangenen Nachrichten, die noch nicht vollständig sind (einige Datenpackete fehlen).
n_received : java.util.Vector	Eine Liste der empfangenen Nachrichten.
n_tosend : java.util.Vector	Eine Liste der Nachrichten, die verschickt werden sollen.

Routinen der Entität Client

$f_{serv}(N)$	Eine Funktion, die die Bearbeitungszeit der Nachricht durch den Client berechnet.
$f_{reply}(N)$	Eine Funktion, welche die Antwortnachricht auf die aktuell bearbeitete Nachricht berechnet.
$f_{wait}(N)$	Eine Funktion, welche die Wartezeit zwischen den Aktivierungen des Clients berechnet.
$f_{start}(C)$	Eine Funktion, welche die Startnachricht bei der Aktivierung des Clients berechnet.
run()	Hauptroutine der Entität Server.
start()	Methode zum Starten einer Kommunikationssequenz.
respond()	Methode zur Bearbeitung empfangener Nachrichten.
sendNachricht()	Methode zum Senden von Nachrichten.
checkNachricht()	Methode zur Überprüfung der Liste n_incomplete auf enthalten vollständiger Nachrichten.
receivePacket()	Methode zum Empfangen von Datenpacketen.
sendPacket()	Methode zum Senden von Datenpacketen.
sendToSim (Object)	Methode zum Senden von Objekten an die Simulationsplattform.

Tabelle 4.7.: Komponenten und Routinen der Entität Client.

Der Entwurf und die Implementierung eines komfortablen User Interfaces geht weit über den Rahmen dieser Arbeit hinaus. Eine solche Aufgabe kann aber durchaus im Rahmen einer anderen Diplomarbeit erfolgreich erledigt werden.

SimConfig

Dieser Teil des Modells eines Simulationsexperimentes. In der Entität *SimConfig* enthält ein Verhaltensmodell aller am Experiment beteiligten Simulationsplattformen. Die Komponenten eines Simulationsmodells sind in der Tabelle 4.10 aufgelistet. Details zum Verhaltensmodell finden sie im Abschnitt 4.4.2.

Komponenten der Entität Nachricht	
N_{id}	Der Name der Nachricht.
N_{type}	Das Typatribut der Nachricht, mit möglichen Werten *request* und *respond*.
N_{time}	Der Zeitstempel der Nachricht.
A	Der Name des Absenders der Nachricht.
Z	Der Name des Zielortes der Nachricht.
$N_{packets}$	Eine Liste mit Datenpacketen der Nachricht.
Routinen der Entität Nachricht	
$f_{data}(N)$	Eine Funktion, die die Menge der Daten in einer Nachricht berechnet. Diese Funktion ist Bestandteil des Verhaltensmodells.
$f_{size}(N)$	Eine Funktion, die die Größe einzelner Datenpackete einer Nachricht berechnet. Diese Funktion ist Bestandteil des Verhaltensmodells.
$f_{time}(N)$	Eine Funktion, die die Sendezeitpunkte einzelner Datenpackete berechnet.. Diese Funktion ist Bestandteil des Verhaltensmodells.

Tabelle 4.8.: Komponenten und Routinen der Entität Nachricht.

Komponenten der Entität Datenpacket	
D_{id}	Der Name der Nachricht. Anhand der Daten dieses Attributes soll es möglich sein, das Datenpacket einer Nachricht zusammen mit anderen Datenpacketen derselben Nachricht, zu der ursprünglichen Nachricht zusammenzusetzen.
N_{data} : Byte[]	Dieses Attribut repräsentiert einen Teil der Daten der Nachricht zu der das Datenpacket gehört. Innerhalb des Simulationsplattform wird dieses Attribut verwendet, um die Bytecodierung des Datenpacketes zu speichern. Wenn die Bytecodierung nicht ausreicht um die geforderte Größe des Datenpacketes zu erreichen, wird ein Array passender Größe mit Zufallszahlen gefüllt und an die Datenkodierung angehängt.
A	Der Name des Absenders des Datenpacketes.
Z	Der Name des Zielortes des Datenpacketes.

Tabelle 4.9.: Komponenten und Routinen der Entität Datenpacket.

Komponenten der Entität SimConfig

configString : String	Die Stringkodierung eines Verhaltensmodells.
platformen : PlatformConfig	Die Konfiguration einzelner am Experiment beteiligter Simulationsplattformen, mit Angabe der Namen der Maschinen, auf denen die Simulationsplattformen laufen sollen.
stellen : StellenConfig	Die Konfiguration einzelner am Experiment beteiligter Stellen.
nachrichten : NachrichtenConfig	Die Konfiguration einzelner am Experiment beteiligter Nachrichten. In diesem Attribut sind auch die Funktionen kodiert, die von den Stellen Server und Client geraucht werden, um jede am Experiment beteiligte Nachricht zu verarbeiten.

Routinen der Entität SimConfig

parseConfig (String)	Eine Funktion zum Decodieren des Verhaltensmodells.

Tabelle 4.10.: Komponenten und Routinen der Entität SimConfig.

4.4.2. Verhaltensmodell

Um die Implementierung des Simulationsmodells nicht bei jedem einzelnen Experiment ändern zu müssen, habe ich ein *Verhaltensmodell* für das Simulationsmodell entworfen, welches als Konfiguration des Simulationsmodells verwendet werden kann.

Das Verhaltensmodell besteht aus drei Teilen:

- *PlatformConfig* gibt an:
  - alle am Experiment beteiligten Rechner.
  - Anzahl der Stellen, die auf den entsprechenden Rechnern laufen sollen.
  - Art der Stellen, die auf den entsprechenden Rechnern laufen sollen.

  Die Art der Stellen muss in der StellenConfig angegeben sein.

- *StellenConfig* Definiert Stellenarten der an dem Simulationsexperiment beteiligten Stellen. Bestandteile der Definition sind:
  - Bezeichnung der Stelle.
  - Typ der Stelle (Client oder Server).
  - Die Startnachricht der Stelle.

  Die Startnachricht ist für Stellen des Typs Client wichtig und wird in der Methode start() der Entität Client verwendet.

- *NachrichtenConfig* Dieser Teil des Verhaltensmodells bestimmt das das Verhalten einzelner Stellen. Hier werden Nachrichten definiert, die zur Kommunikation zwischen Client und Server verwendet werden. Innerhalb der Nachrichtendefinitionen werden auch Regeln zur Bearbeitung der einzelnen Nachrichten festgelegt. Eine Definition einer Nachricht besteht demnach aus folgenden Komponenten:

- Bezeichnung der Nachricht.
- Typ des Absenders.
- Typ des Empfängers.
- Funktionsdefinition der Funktion f_{data}.
- Funktionsdefinition der Funktion f_{size}.
- Funktionsdefinition der Funktion f_{time}.
- Funktionsdefinition der Funktion f_{serv}.
- Funktionsdefinition der Funktion f_{wait}.
- Funktionsdefinition der Funktion f_{reply}.

In dem Activity Diagramm 4.7 ist gezeigt, dass das Verhalten von Stellen durch Definition von geeigneten Nachrichten beeinflusst werden kann. Wenn innerhalb einer Nachricht die Funktionen f_{reply}, f_{wait}, f_{serv}, f_{time}, f_{size} und f_{data} definiert werden, kann das Verhalten des Simulationsmodells über diesen Weg gesteuert werden. Die genannten Funktionen werden von den Entitäten Client und Server verwendet, um Nachrichten zu verarbeiten.

Als Funktionen habe ich im Verhaltensmodell stochastische Funktionen gewählt, die entsprechend einer Wahrscheinlichkeitsverteilung einen Wert zurückgeben. Bei allen Funktionen genügt es als Definition der Funktion die Angabe einer diskreten Wahrscheinlichkeitsverteilung mit möglichen Rückgabewerten. Eine diskrete Wahrscheinlichkeit genügt hier, weil die Anzahl möglicher Rückgabewerte bei allen Funktionen sehr gering ist.

Eine Notation für das Verhaltensmodell, die von der Implementierung des Simulationsmodells verwendet wird, wird im Abschnitt 5.3.1 beschrieben.

4.4.3. Notwendige Daten

Um Simulationsexperimente durchführen zu können, müssen die Parameter der Implementierung des bereits beschriebenen Modells festgelegt werden. Für jedes Experiment ist eine eigene Konfiguration notwendig. Im Abschnitt 5.3.1 ist eine Beispielkonfiguration mit Erklärungen zu den einzelnen Parametern angegeben.

Das Verhalten der Simulationsautomaten wird zum größten Teil durch die Konfiguration der Nachrichten bestimmt, welche zwischen einzelnen Stellen ausgetauscht werden. Die in diesem Kapitel angegebene Beispielkonfiguration bezieht sich auf den im Kapitel 4 vorgestellten Cashautomaten. Es sind auch wesentlich komplexere Verhaltensmuster modellierbar.

Hier stellt sich die Frage:
Wie komme ich zu den Parametern eines Simulationsexperimentes?

Es gibt zwei unterschiedliche Situationen in denen die Festlegung von Parametern des Modells vorgenommen werden kann:

1. Es existieren bereits Automaten deren Verhalten simuliert werden soll.

2. Es soll das Verhalten von Automaten Simuliert werden, die noch nicht gebaut worden sind.

In beiden Fällen ist es sehr wichtig, dass bei der Festlegung der Parameter Personal aus dem Entwicklerteam einbezogen wird, welches die entsprechenden Automaten entwickelt bzw. entwickelt hat. Durch die Kenntnis der Funktionsweise der zu imitierenden Automaten wird die Validität des Simulationsexperimentes erhöht.

Messen und Auswerten

Im Fall der existierenden Automaten hat der Durchführende des Simulationsexperimentes, neben dem Einblick in die Funktionsweise des Automaten, die Möglichkeit dessen Verhalten zu beobachten. Der Firma WINCOR NIXDORF GmbH & Co. KG stehen einige Werkzeuge zur Verfügung mit deren Hilfe Netzwerkverkehr aufgezeichnet und Analysiert werden kann.

Schätzen

Im Fall noch nicht existierender Automaten bleibt dem Verantwortlichen für die Durchführung des Experimentes nur die Schätzung der Parameter. Diese Schätzungen müssen sich auf die Aussagen des Entwicklerteams über das zukünftige Verhalten des Automaten stützen. Außer der Schätzung der Funktionsweise des Automaten müssen Schätzungen über das Benutzungsverhalten der Nutzer solcher Automaten gemacht werden. Hier können Methoden aus dem Bereich *Human Computer Interface* und der *Usibility Engeneering* hilfreich sein.

4.5. Migration des Modells zum Thin-Client

In diesem Abschnitt werden Änderungen am Simulationsmodell besprochen, die bei der Simulation von Thin-Clients notwendig werden. Dazu ist es notwendig zu wissen, inwiefern sich das Verhalten eines Thin-Clients von dem eines Fat-Clients unterscheidet.

4.5.1. Verhalten eines Thin-Clients

In dem Activity Diagramm 4.22 ist das Verhalten eines Thin-Client Geldautomaten modelliert. An diesem Modell ist zu sehen, dass sich das Verhalten eines Thin-Clients sich nicht sehr von dem im Activity Diagramm 4.2 modellierten Verhalten eines Fat-Client unterscheidet.

4.5.2. Modellbeschreibung

Die neuen Komponenten des Modells sind ein zusätzlicher Server (UI-Server) und die Kommunikation zwischen diesem Server und dem Client. Da das in dem Ab-

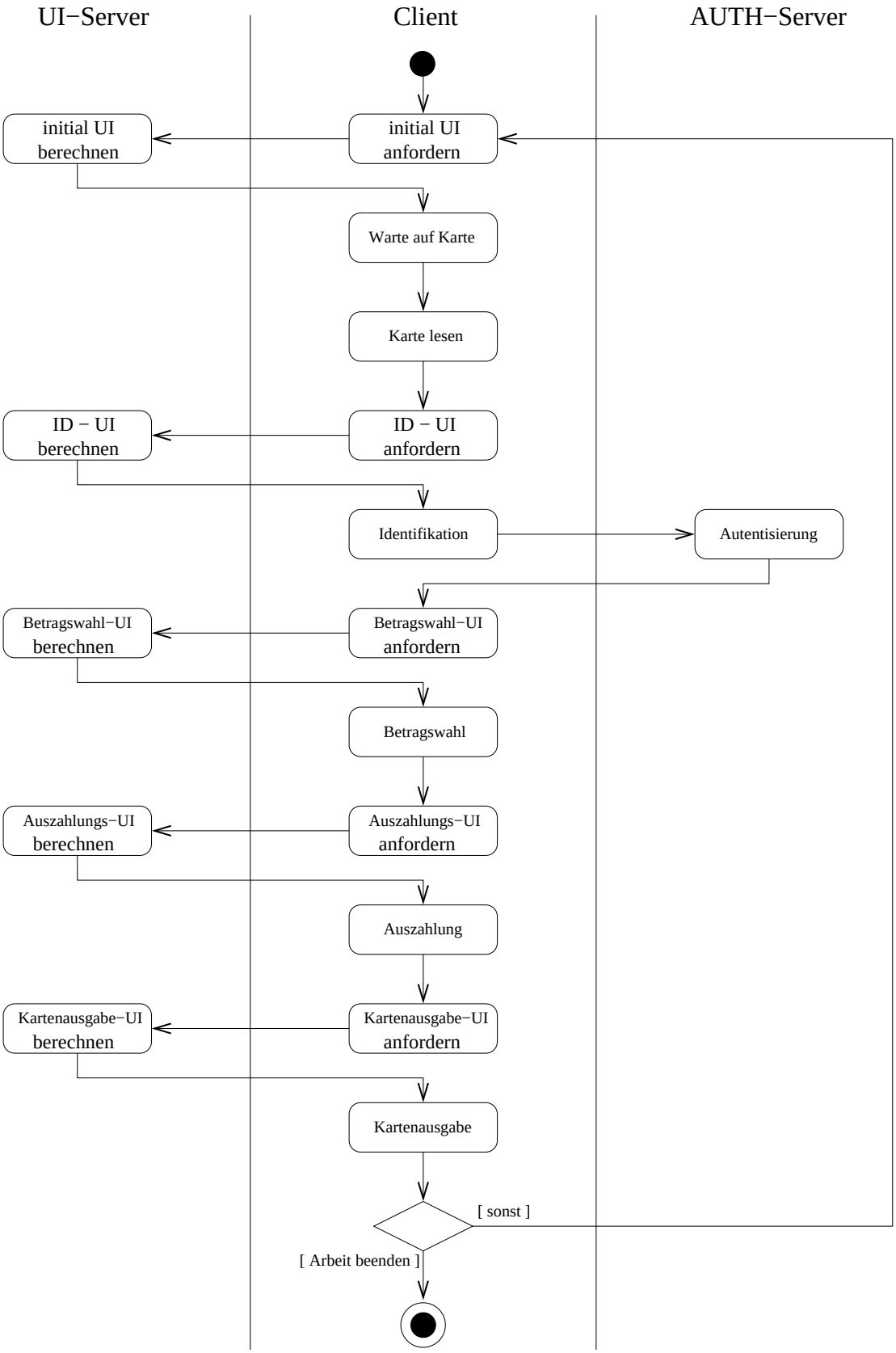

Abbildung 4.22.: Activity Diagramm eines Thin-Client Cashautomaten.

schnitt 4.4.1 beschriebene Modell in der Lage ist mehrere Serverentitäten zu simulieren ist keine Veränderung des Simulationsmodells notwendig um Thin-Clients zu simulieren. Hier ist lediglich eine Anpassung des Verhaltensmodells notwendig. Eine Beispielkonfiguration eines Verhaltensmodells für einen Thin-Client befindet sich im Kapitel 5.

4.6. Mögliche Erweiterungen des Modells

Dieser Abschnitt beschäftigt sich mit folgenden Fragen:

- Was sind die Anforderungen an das Modell?
- Wie können diese Anforderungen im Modell realisiert werden?

4.6.1. Anforderungen an das Modell

Die Anforderungen an das Modell können aus den möglichen Einsatzszenarien dessen Implementierungen entnommen werden. Die im Kapitel 2 genannten typische Szenarien für die Fragestellungen sind:

1. **Szenario:** Ein Kunde möchte Clients älterer Architekturen durch neue ersetzen.
2. **Szenario:** Ein Kunde hat Interesse an einer Anzahl von Clients einer bestimmten Architektur. Diese sollen an das bestehende Netz angeschlossen werden.
3. **Szenario:** Ein Kunde hat den Wunsch Clients verschiedener Architekturen anzuschaffen und sie an verschiedenen Punkten in gegebener Anzahl ans Netz anzuschließen.

Die daraus abgeleiteten Anforderungen sind:

1. Einem Client muss es möglich sein das Verhalten mehrerer Automaten zu simulieren.
2. Es muss möglich sein auch komplexe Verhaltensmuster einzelner Automaten zu simulieren.
3. Da es denkbar ist Clients so zu konstruieren, dass diese mit mehreren Servern kommunizieren (Web-Server, Autorisierungsserver, Datenbankserver, ..), muss das Modell diese Möglichkeiten berücksichtigen.
4. Die Implementierung des Modells muss ein verteiltes System sein, da die Netzwerkkomponente nicht mit modelliert wird. D.h. werden Clients an verschiedenen Punkten an das Netz Angeschlossen, so muss an den entsprechenden Punkten auch eine Simulationsplattform angeschlossen werden. Diese Simulationsplattform muss die für sie notwendigen Konfigurationsanweisungen über das Netzwerk beziehen.

4.6.2. Realisierung der Anforderungen

Anforderung 1

Um diese Forderung zu erfüllen gibt es mehrere Möglichkeiten. Man kann auf derselben Simulationsplattform mehrere Client-Prozesse starten oder einen Client durch eine Modifikation dazu bringen sich wie mehrere Clients zu verhalten.

Um es einem Client zu ermöglichen das Verhalten mehrerer (n) Clients zu simulieren sind einige wenige Änderungen am Modell des Clients notwendig:

- Die Markierung C_{inuse} muss durch einen Zähler C_{usenum} ersetzt werden.
- An jeder Stelle an der die Markierung C_{inuse} auf *unused* gesetzt wird, muss diese Anweisung durch $C_{usenum} = C_{usenum} - 1$ ersetzt werden.
- An jeder Stelle an der die Markierung C_{inuse} auf *inuse* gesetzt wird muss diese Anweisung durch $C_{usenum} = C_{usenum} + 1$ ersetzt werden.
- An jeder Stelle an der gefragt wird ob C_{inuse} den Wert *unused* hat muss diese Anfrage durch die Anfrage ersetzt werden, ob der Wert des Zählers C_{usenum} kleiner ist als n.

Nach diesen Änderungen ist es möglich mittels eines Clients das Verhalten von n Clients zu Simulieren.

Anforderung 2

Das vorgestellte Modell erlaubt bereits das Modellieren komplexer Verhaltensmuster. Dies geschieht durch die geeignete Wahl der Funktionen $f_{reply}(N)$ und $f_{start}(C)$. Die dabei zur Verfügung stehenden Möglichkeiten werden in den folgenden Kapiteln anhand von Beispielen demonstriert.

Anforderung 3

Auch diese Forderung ist bereits im Modell implementiert. Durch eine geeignete Wahl der Funktionen $f_{reply}(N)$ und $f_{start}(C)$ ist es Möglich auch diese Anforderung zu erfüllen. Auch die Erfüllung dieser Anforderung wird in den nächsten Kapiteln anhand von Beispielmodellen demonstriert.

Anforderung 4

Auch diese Forderung ist bereits im Modell implementiert. Durch Angabe der an dem Simulationsexperiment beteiligten Maschinen in dem Teil PlatformConfig des Verhaltensmodells kann eine verteilte Simulation durchgeführt werden. Im Kapitel 5 wird Anhand einer konkreten Beispielimplementierung eine Möglichkeit angegeben diese Anforderung zu erfüllen.

5. Implementierung

In diesem Kapitel wird die Kodierung des im Kapitel 4 entwickelten Simulationsmodells in eine auf Rechnern lauffähige Software. Außerdem wird eine Notation für das Verhaltensmodell angegeben, welches zur Konfiguration des Simulationsmodells notwendig ist. Das Softwaremodell des Simulationsmodells ist im Kapitel A zu finden.

5.1. Implementierung des Modells

Ich habe das Modell in der Programmiersprache Java umgesetzt. Dafür gibt es mehrere gute Gründe:

- Zum einen wird die neue Generation der Automaten bei WINCOR NIXDORF GmbH & Co. KG in Java implementiert. Die Tatsache, dass sowohl die Automaten als auch die Simulationssoftware auf derselben Plattform basiert erhöht die Validität des Simulationsmodells, da nur die Kontrollstruktur beider Softwarepakete sich unterscheidet. Die Basis (Virtuelle Maschine, NetzwekBiblioteken) ist bei beiden Softwarepaketen die gleiche.

- Als Zweites spricht für Java die Plattformunabhängigkeit des Modells. Eine Virtuelle Maschine die Java-Bytecode interpretieren kann, ist mittlerweile für fast alle Betriebssysteme erhältlich. In der Implementierung wurden keine Plattformspezifischen Bibliotheken verwendet.

Die Simulationsplattform besteht aus mehreren Entitäten, die miteinander verbunden sind. Die zentrale Rolle hat die Entität Simulation inne. Diese Entität verbindet folgende Bestandteile des Systems miteinander:

- **User Interface**: Dieser Teil der Simulationsplattform ist durch zwei Bestandteile repräsentiert. Der erste Teil wird durch die Klasse *LastGenerator* dargestellt. Diese Klasse implementiert ein Kommandozeileninterface zur Steuerung der Simulationsplattform. Der zweite Bestandteil besteht aus der Klasse *SimConfig*. Diese Klasse stellt eine Konfigurationsmöglichkeit für ein Simulationsexperiment zur Verfügung.

- **Schnittstelle zum Netzwerk**: Diese Schnittstelle bietet die Klasse *ComDevice*. Hier sind auf der Basis der Klassenbibliotek *java.net* Möglichkeiten

implementiert entfernte Simulationsplattformen zu konfigurieren und einzelne Datenpackete zu senden und zu empfangen.

- **Kern des Simulationsmodells**: Dieser wird durch die Klassen *Stelle* und *ClientStelle* dargestellt, welche die Stellen *Server* und *Client* aus dem Kapitel 4 implementieren.

Die Dokumentation der Klassen und den Quellcode der Beispielimplementierung finden Sie im KapitelA. Notwendige Angaben zur Parametrisierung des Simulationsmodells können Sie im Abschnitt 5.3.1 einsehen.

5.2. Softwaremodell des Simulationsmodells

Die Struktur des Softwaremodells ist fast Identisch mit der Struktur des Simulationsmodells. Im Klassendiagramm 5.1 ist die Struktur des Softwarepaketes dargestellt. Die Implementierung des Softwaremodells und seine Dokumentation befindet sich in Kapitel A.

5.3. Parametrisierung des Modells

Zur Parametrisierung des Simulationsmodells muss ein Verhaltensmodell angegeben werden, dabei wird die in diesem Abschnitt beschriebene Notation benutzt.

5.3.1. Beschreibung der Konfiguration

Die Konfiguration eines Simulationsexperimentes mit Hilfe der Beispielimplementierung des Simulationsmodells wird mittels einer Konfigurationsdatei vorgenommen. Diese Datei beinhaltet Konfigurationsanweisungen im ASCII-Code. Hier ein Beispiel:

```
##### PlatformConfig #####
{
[nexus : (Autorisierungsserver 1)]
[naxos : (Cashautomat 20)]
}
##### StellenConfig #####
{
[Cashautomat : client : Identifikation]
[Autorisierungsserver : server : Autorisierung]
}
##### NachrichtenConfig #####
{
Identifikation : Cashautomat -> Authentisierungsserver
f_data: [ (50.0 2048) (50.0 4096) ]
f_size: [ (12.3 1024) (67.7 2048) (20 4096) ]
```

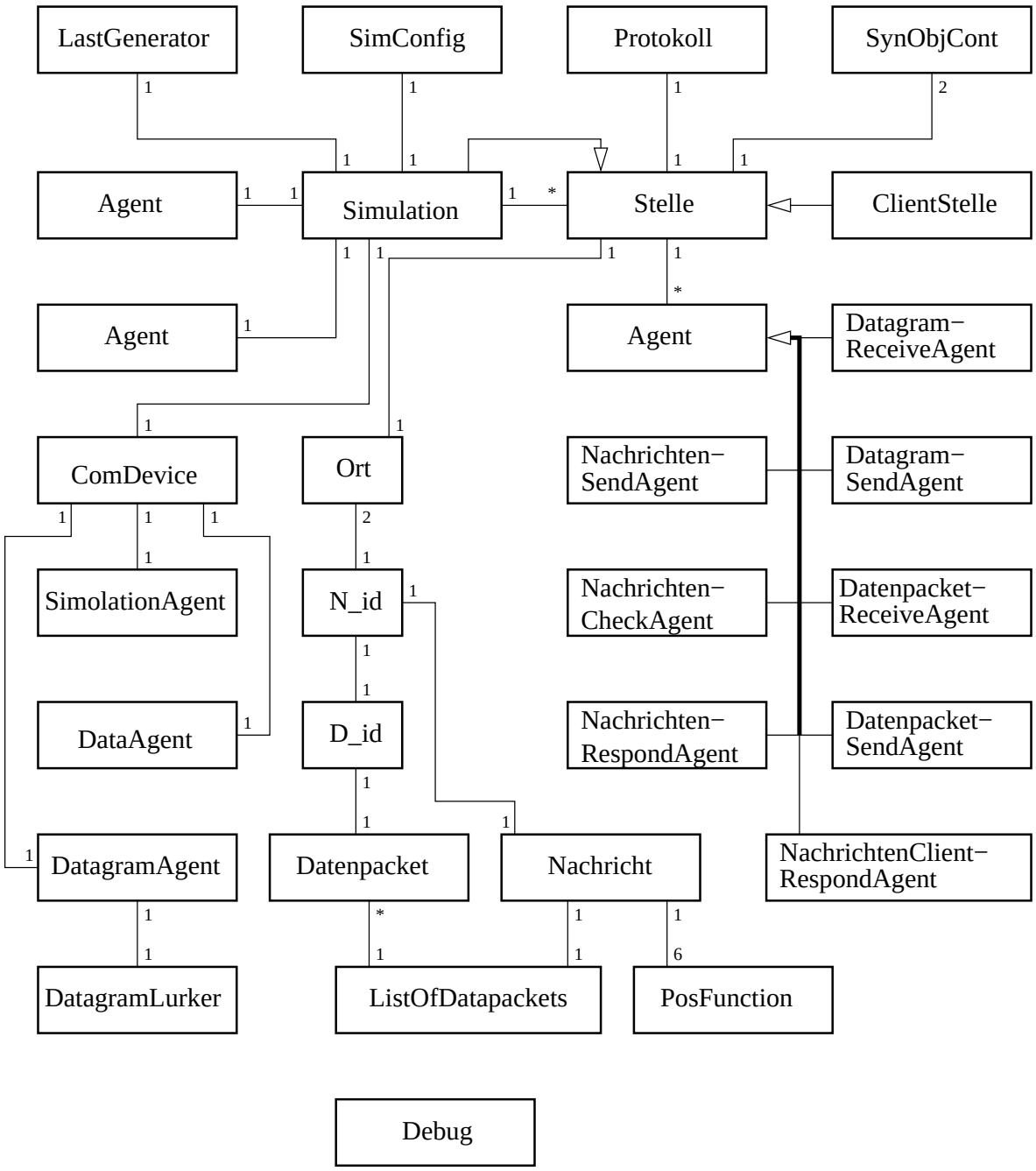

Abbildung 5.1.: Klassendiagramm des Softwarepaketes.

```
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000)]
f_reply: [ (1.0 Autorisierung) ]
}
{
Autorisierung : Authentisierungsserver -> Cashautomat
f_data: [ (50.0 2048) (50.0 4096) ]
f_size: [ (12.3 1024) (67.7 2048) (20 4096) ]
f_time: [ (50.0 0) (50.0 100) ]
f_serv: [ (33.3 1000) (33.3 2000) (33.3 3000) ]
f_wait: [ (100.0 0) ]
f_reply: [ (0.9 ---null---) (0.1 Identifikation) ]
}
{
---null--- : null -> null
f_data: [ (0 0) ]
f_size: [ (0 0) ]
f_time: [ (0 0) ]
f_serv: [ (0 0) ]
f_wait: [ (0 0) ]
f_reply: [ (0 ---null---) ]
}
##### EndOfConfig #####
```

Die Konfigurationsdatei ist in drei Teile unterteilt. Als Konfigurationsanweisungen werden alle Daten angesehen, die sich zwischen den Schlüsselworten **PlatformConfig**, **StellenConfig**, **NachrichtenConfig** und **EndOfConfig** befinden.

Vereinbarungen für die Konfigurationsdatei:

- Alles vor **PlatformConfig** und nach **EndOfConfig** wird als Kommentar angesehen und ignoriert.
- Kommentare werden mit dem Zeichen **;** angefangen und gelten bis zum Ende der Zeile.
- Alle Zeitangaben sind in Millisekunden angegeben.
- Alle Lastangaben sind in Bytes angegeben.

5.3.2. Konfiguration von Plattformen

Der erste Teil der Konfiguration beginnt mit dem Schlüsselwort **PlatformConfig** und endet mit **StellenConfig**. Dieser Teil der Konfiguration enthält die Konfiguration einzelner Simulationsplattformen.

```
##### PlatformKonfig #####
--- Start Fileformat ----
{
[<rechnername> : (<Bezeichnung_der_Stelle> <#Stellen>) (...) ... ]
[nexus : (Authentisierungsserver 1) (Webserver 1)]
[naxos : (Cashautomat 20) (Infoterminal 5)]
[pissarro : (Cashautomat 50)]
}
--- End of Fileformat ---
##### StellenConfig #####
```

Dabei sind folgende Punkte zu beachten:

- **{ }** Klammerung um die Gesamte Konfiguration.
- **[ ]** Klammerung um die Konfiguration einer Simulationsplattform.
- ( ) Klammerung um eine Gruppe gleichartiger Stellen.
- **rechnername**: Bezeichnung einer Maschine, wird von der Plattform in eine Internet Adresse (IP) umgewandelt.
- **Bezeichnung_der_Stelle**: Typ des zu simulierenden Automaten. Der festgelegte Typ muss in der Konfiguration der Stellen definiert werden.
- #**Stellen**: Anzahl der zu simulierenden Automaten desselben Typs. Bei Stellen die den Typ **server** haben, ist die Anzahl = 1

5.3.3. Konfiguration von Stellen

Der zweite Teil der Konfiguration beginnt mit dem Schlüsselwort **StellenConfig** und endet mit **NachrichtenConfig**. Hier werden einzelne Stellen definiert.

```
##### StellenConfig #####
--- Start Fileformat ----
{
[<Bezeichnung_der_Stelle> : <Typ_der_Stelle> : <StartNachricht> ]
[Authentisierungsserver : server : null ]
[Webserver : server : null ]
[Cashautomat : client : CashautomatStart]
[Infoterminal : client : InfoterminalStart]
}
--- End of Fileformat ---
##### NachrichtenConfig #####
```

Dabei sind folgende Punkte zu beachten:

- **{ }** Klammerung um die Gesamte Konfiguration.
- **[ ]** Klammerung um die Konfiguration einer Stelle
- **Bezeichnung_der_Stelle**: z.B.: Authentisierungsserver, Webserver, Cashautomat, Infoterminal, ...
- **Typ_der_Stelle**: **client** oder **server**
- **StartNachricht**: Nachricht, die als erstes auf den Clients gestartet wird. Die hier stehende Nachricht muss in der Konfiguration der Nachrichten vorkommen.

5.3.4. Konfiguration von Nachrichten

Der dritte Teil der Konfiguration beginnt mit dem Schlüsselwort **Nachrichten-Config** und endet mit **EndOfConfig**. Hier wird das Kommunikationsverhalten der Stellen definiert.

```
##### NachrichtenConfig #####
--- Start Fileformat ----
{
<BezeichnungDerNachricht> : <SenderStelle> -> <EmpfängerStelle>
f_size: [ (<Möglichkeit> <Wert>) (12.3 1024) (87.7 2048) (...) ... ]
f_time: [ ... ]
f_serv: [ ... ]
f_wait: [ ... ]
f_reply: [ (1.0 AntwortNachricht) ]
}
{
...
}
...
##### EndOfConfig #####
--- End of Fileformat ---
```

Dabei sind folgende Punkte zu beachten:

- **{ }** Klammerung um die Konfiguration einer Nachricht.
- **[ ]** Klammerung um die Konfiguration einer Funktion
- ( ) Klammerung um eine Möglichkeit einer Funktion
- **BezeichnungDerNachricht** z.B.: AuthentisierungsAnfrage, Authentisierung, StartseiteAnfrage, StartseiteAntwort, ...
- **SenderStelle** eine der Stellen in der Stellen Konfiguration

- **EmpfängerStelle** eine der Stellen in der Stellen Konfiguration
- **f_*** Funktionen des Verhaltensmodells (siehe 4.4.2).
- **Möglichkeit** Der Anteil des Wertes an der Wahrscheinlichkeit ausgewählt zu werden. Gemessen an der Summe der Möglichkeiten.
- **Wert** Der Rückgabewert der Funktion für die entsprechende Möglichkeit.
- **f_data** beschreibt die Verteilung der Möglichkeiten von Größen einer Nachricht.
- **f_size** beschreibt die Verteilung der Möglichkeiten von Größen der Datenpackete einer Nachricht.
- **f_reply** beschreibt die Verteilung der möglichen Antworten auf diese Nachricht.
- **f_wait** beschreibt die Verteilung der Wartezeiten zwischen den Benutzungen eines Automaten.
- **f_serv** beschreibt die Verteilung der Bearbeitungszeiträume der Nachricht durch eine Stelle.
- **f_time** beschreibt die Möglichkeitsverteilung für die Zeiträume zur Versendung von Paketen.

5.4. Beispiele von Verhaltensmodellen

Im folgenden sind zwei Verhaltensmodelle angegeben, die Abschätzungen des Verhaltens eines Fat-Clients und eines Thin-Clients. Aufgrund fehlender statistischer Daten über das Verhalten echter Maschinen, sind diese Verhaltensmodelle nur Schätzungen. Sie sollten daher nicht als Grundlage eines Simulationsexperimentes dienen, da die eine Abschätzung der Qualität der Ergebnisse des Experimentes nicht möglich ist.

5.4.1. Beispiel eines Fat-Clients

Das hier angegebene Modell ist abgeleitet vom Activity Diagramm 4.2. Die statistischen Werte, die in diesem Verhaltensmodell verwendet wurden beruhen zum einen Teil auf Schätzungen der Mitarbeiter der Firma WINCOR NIXDORF GmbH & Co. KG, zum anderen Teil auf eigenen Erfahrungen mit einem Geldautomaten. Die Validität des Modells kann unter diesen Umständen nicht garantiert werden.

```
##### PlatformConfig #####
{
[standby : (Authentisierungsserver 1)]
[knecht  : (Cashautomat 1)]
}
##### StellenConfig #####
{
[Cashautomat : client : Identifikation ]
[Authentisierungsserver : server : ---END--- ]
}
##### NachrichtenConfig #####
{
Identifikation : Cashautomat -> Authentisierungsserver
f_data: [ (50.0 4096) (50.0 8192) ]
f_size: [ (12.3 1024) (67.7 2048) (20 4096) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (1.0 Autorisierung) ]
}
{
Autorisierung : Authentisierungsserver -> Cashautomat
f_data: [ (50.0 2048) (50.0 4096) ]
f_size: [ (12.3 1024) (67.7 2048) (20 4096) ]
f_time: [ (50.0 0) (50.0 100) ]
f_serv: [ (33.3 1000) (33.3 2000) (33.3 3000) ]
f_wait: [ (100.0 0) ]
f_reply: [ (0.9 ---END---) (0.1 Identifikation) ]
}
{
---END--- : null -> null
f_data: [ (0 0) ]
f_size: [ (0 0) ]
f_time: [ (0 0) ]
f_serv: [ (0 0) ]
f_wait: [ (0 0) ]
f_reply: [ (0 ---END---) ]
}
##### EndOfConfig #####
```

5.4.2. Beispiel eines Thin-Clients

Auch dieses Verhaltensmodell basiert nur auf Schätzungen, weshalb seine Validität nicht gegeben ist.

```
##### PlatformConfig #####
{
[standby : (Authentisierungsserver 1)]
[standby : (UserInterfaceServer 1)]
[knecht  : (CashautomatThin 1)]
}
##### StellenConfig #####
{
[CashautomatThin : client : Identifikation ]
[Authentisierungsserver : server : ---END--- ]
[UserInterfaceServer : server : ---END--- ]
}
##### NachrichtenConfig #####
{
InitUIReq : CashautomatThin -> UserInterfaceServer
f_data: [ (1.0 100000) ]
f_size: [ (1.0 1024) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (1.0 InitUIRes) ]
}
{
InitUIRes : UserInterfaceServer -> CashautomatThin
f_data: [ (1.0 100000) ]
f_size: [ (1.0 1024) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (1.0 IDUIReq) ]
}
{
IDUIReq : CashautomatThin -> UserInterfaceServer
f_data: [ (1.0 100000) ]
f_size: [ (1.0 1024) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (1.0 IDUIRes) ]
}
{
IDUIRes : UserInterfaceServer -> CashautomatThin
f_data: [ (1.0 100000) ]
f_size: [ (1.0 1024) ]
```

```
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (1.0 Identifikation) ]
}
{
Identifikation : CashautomatThin -> Authentisierungsserver
f_data: [ (50.0 4096) (50.0 8192) ]
f_size: [ (12.3 1024) (67.7 2048) (20 4096) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (1.0 Autorisierung) ]
}
{
Autorisierung : Authentisierungsserver -> CashautomatThin
f_data: [ (50.0 2048) (50.0 4096) ]
f_size: [ (12.3 1024) (67.7 2048) (20 4096) ]
f_time: [ (50.0 0) (50.0 100) ]
f_serv: [ (33.3 1000) (33.3 2000) (33.3 3000) ]
f_wait: [ (100.0 0) ]
f_reply: [ (1.0 BetrUIReq) ]
}
{
BetrUIReq : CashautomatThin -> UserInterfaceServer
f_data: [ (1.0 100000) ]
f_size: [ (1.0 1024) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (1.0 BetrUIRes) ]
}
{
BetrUIRes : UserInterfaceServer -> CashautomatThin
f_data: [ (1.0 100000) ]
f_size: [ (1.0 1024) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (1.0 AuszUIReq) ]
}
{
AuszUIReq : CashautomatThin -> UserInterfaceServer
f_data: [ (1.0 100000) ]
```

```
f_size: [ (1.0 1024) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (1.0 AuszUIRes) ]
}
{
AuszUIRes : UserInterfaceServer -> CashautomatThin
f_data: [ (1.0 100000) ]
f_size: [ (1.0 1024) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (1.0 KartReq) ]
}
{
KartReq : CashautomatThin -> UserInterfaceServer
f_data: [ (1.0 100000) ]
f_size: [ (1.0 1024) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (1.0 KartRes) ]
}
{
KartRes : UserInterfaceServer -> CashautomatThin
f_data: [ (1.0 100000) ]
f_size: [ (1.0 1024) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000) ]
f_reply: [ (0.1 InitUIReq) (0.9 ---END---) ]
}
{
---END--- : null -> null
f_data: [ (0 0) ]
f_size: [ (0 0) ]
f_time: [ (0 0) ]
f_serv: [ (0 0) ]
f_wait: [ (0 0) ]
f_reply: [ (0 ---END---) ]
}
##### EndOfConfig #####
```

5.5. Anleitung zur Software

An dieser Stelle wird eine kurze Beschreibung der Benutzungsschnittstelle der Simulationssoftware gegeben. Da bisher kein Interaktives Interface implementiert wurde ist die Steuerung der Simulationsexperimente ausschließlich durch das Verhaltensmodell möglich.

Die Dauer des Simulationsexperimentes ist in der Methode *runTestSim()* der Klasse Simulation festgelegt und kann an dieser Stelle, durch eine Änderung der Implementierung, geändert werden.

5.5.1. Kurzanleitung

Mit Hilfe des Implementierten Simulationsmodells können Simulationsexperimente durchgeführt werden. Dies geschieht in folgenden Schritten:

- Erstellen eines Verhaltensmodells für das Simulationsmodell.
- Starten der Simulationsplattformen in dem Slave-Modus auf allen am Simulationsexperiment beteiligten Maschinen.
- Starten einer Simulationsplattform im Test-Modus auf einer nicht am Experiment beteiligten Maschine mit dem Verhaltensmodell als Parameter.

Nach Ablauf der Simulationszeit sammelt die im Test-Modus gestartete Simulationsplattform Protokolle aller am Experiment beteiligter Maschinen ein und gibt diese aus.

Der Start einer Simulationsplattform erfolgt aus der Kommandozeile. Das Kommando zum Start einer Simulationsplattform im Slave-Modus hat folgende Form:

```
> java LastGenerator -s
```

Das Kommando zum Start einer Simulationsplattform im Test-Modus hat folgende Form:

```
> java LastGenerator -t <Verhaltensmodell>
```

Dabei ist <Verhaltensmodell> die Bezeichnung einer ASCII-Datei, in der das Verhaltensmodell Abgelegt ist. Das Verhaltensmodell muss in der im Abschnitt 5.3.1 angegebenen Notation notiert sein.

5.5.2. Voraussetzungen der Beispielimplementierung

Folgende Bedingungen müssen auf jeder am Experiment beteiligter Maschinen erfüllt werden um ein Simulationsexperiment mit Hilfe der vorliegender Beispielimplementierung durchzuführen:

- Installierte JavaVM ab Version 1.2.
- Maschine muss über das TCP/IP Protokoll über das Netzwerk mit allen am Experiment beteiligten Maschinen kommunizieren können.
- Die Namen aller am Experiment beteiligter Maschinen müssen zu IP's aufgelöst werden können.

6. Zusammenfassung und Ausblick

Der Grund für die Entstehung dieser Diplomarbeit war die Suche nach Möglichkeiten festzustellen ob Bankautomaten der Firma WINCOR NIXDORF GmbH & Co. KG in bestimmten Situationen erwartungskonform funktionieren würden. Als wichtiger Aspekt hat sich dabei die *Antwortzeit* des Client-Server Systems erwiesen.

Ich habe festgestellt, dass die Antwortzeit maßgeblich von der Auslastung des Netzwerkes beeinflusst wird, welches Client und Server verbindet. Durch eine Abwägung der zur Verfügung stehenden Methoden bin ich zur Ansicht gelangt, dass Simulation eine geeignete Methode darstellt die Antwortzeiten eines geplanten Systems abzuschätzen.

Um den Mitarbeitern der Firma WINCOR NIXDORF GmbH & Co. KG zu ermöglichen Simulationsexperimente zu diesem Zweck durchzuführen, habe ich eine Einführung in das Themengebiet der Simulation gegeben, ein geeignetes Simulationsmodell entworfen und eine Beispielimplementierung einer Simulationsplattform geschrieben.

Diese Diplomarbeit stellt somit keine vollständige Simulationsstudie dar, sondern eine Anleitung zu einer Simulationsstudie. Mit Hilfe der Beispielimplementierung ist es nun möglich, durch eine geeignete Konfiguration der Simulationsplattform, mittels eins geeigneten Verhaltensmodells, Simulationsexperimente durchzuführen.

Vor der Planung der Experimente ist jedoch zu beachten, dass ich aus Mangel an Daten keine Validierung des Modells durchführen konnte. Sollten dem Verantwortlichen einer Simulationsstudie mehr Daten zur Verfügung stehen, ist es notwendig eine Validierung des Simulationsmodells durchzuführen.

Im Verlauf geplanter Simulationsexperimente werden eventuell auch Änderungen am Simulationsmodell und dem von mir entwickelten Softwaremodell notwendig. Beide Modelle sind modular entworfen, um Änderungen einfacher vornehmen zu können.

Die Planung solcher Experimente, ihre Durchführung und die Auswertung der Ergebnisse muss von den Mitarbeitern der Firma WINCOR NIXDORF GmbH & Co. KG umgesetzt werden.

A. Sourcecode

A.1. Simulationsplatform

A.1.1. Simulation

```
import java.net.InetAddress;
import java.net.DatagramPacket;
/**
 * Diese Klasse impementiert eine Simulationsplatform.
 *
 * @author xelahr
 * @version 0.3
 */
class Simulation extends Stelle
{
/**
 * Die Konfiguration des Simulationsexperiments.
 */
private SimConfig config = null;
/**
 * Die lokalen Stellen der Simulation.
 */
private Stelle[] stelle = null;
/**
 * Die Schnittstelle der Simulationsplatform zum Netzwerk.
 */
private ComDevice com = null;
//private Agent simMaster = null;
/**
 * Agent zum Versenden von Datagramen.
 */
private Agent sendAgent = null;
/**
 * Agent zum Empfangen von Datagramen.
 */
private Agent recvAgent = null;

/**
 * Kreiert eine neue Simulationsplatform.
 */
Simulation()
{
```

```
this.config = new SimConfig();
this.com = new ComDevice(this);
this.p_receiv = new SynObjCont();
this.p_tosend = new SynObjCont();
this.sendAgent = new DatagramSendAgent(this);
this.recvAgent = new DatagramReceiveAgent(this);
this.alife = true;
}

/**
 * Methode zur Übermittlung eines Datenpaketes an eine lokale Stelle.
 * @param addr Nummer der entsprechenden Stelle.
 * @param pack das zu übermittelnde Datenpacket.
 */
void sendToStelle(int addr, Datenpacket pack)
{
try
{
stelle[addr].addToP_receiv(pack);
}
catch (Exception e)
{
Debug.print(10,"Simulation","sendToStelle","stelle nut founnd: "+e);
}
}

/**
 * Methode zur Übermittlung eines Datagrams an das Netzwerk.
 * @param dat das zu übermittelnde Datagram.
 */
void sendToNet(DatagramPacket dat)
{
this.com.sendSimulationDatagram(dat);
}

/**
 * Methode zum Empfangen eines Datagrams vom Netzwerk.
 * @param dat das zu übermittelnde Datagram.
 */
void receiveFromNet(DatagramPacket dat)
{
/*
byte[] inhalt = dat.getData();
String buff = new String();
buff += "[ ";
for ( int i = 0; i < inhalt.length ; i++ )
buff += inhalt[i] + " ";
buff += "]";
*/
this.addToP_receiv(dat);
}
```

```
/**
 * Methode zum Starten der lokalen Simulationsplatform.
 */
void startSimulationPlatform()
{
this.alife = true;
this.p_receiv = new SynObjCont();
this.p_tosend = new SynObjCont();
this.com.startSimulationAgent();
this.sendAgent.start();
this.recvAgent.start();
for ( int i = 0 ; i < stelle.length ; i++ )
{
stelle[i].setAlife(true);
stelle[i].start();
}
}

/**
 * Methode zum Stoppen der lokalen Simulationsplatform.
 */
void stopSimulationPlatform()
{
if (this.stelle != null)
for ( int i = 0 ; i < stelle.length ; i++ )
if (stelle[i] != null ) stelle[i].stopStelle();
this.alife = false;
this.com.stopSimulationAgent();
}

/**
 * Methode zur Konfiguration der lokalen Simulationsplatform.
 * @param conf Ein Konfigurationsstring.
 */
void configure(String conf)
{
this.config = new SimConfig(conf);
this.configureSimulationPlatform();
}

/**
 * Methode zum Suchen einer Adresse.
 * @param server Bezeichnung einer Stelle.
 * @return Der Ort der Stelle.
 */
Ort getServerAddresseFromConfig(String server)
{
String[] platforms = this.config.getPlatformenFor(server);
if ( platforms == null )
{
return null;
}
```

```
int i = (int)Math.round((Math.random() * platforms.length) - 0.5 );
Ort platform = new Ort();
platform.name = platforms[i];
platform.addr = com.getByName(platform.name);
platform.port = 5007;
return platform;
}

/**
 * Methode zum Suchen einer Nachricht.
 * @param name Bezeichnung einer Nachricht.
 * @return die Entsprechende Nachricht.
 */
Nachricht getNachrichtFromConfig(String name)
{
return this.config.getNachricht(name);
}

/**
 * Methode zur Konfiguration der lokalen Simulationsplatform.
 * ( Nach dem parsen eines Konfigstrings. )
 */
void configureSimulationPlatform()
{
if ( this.config == null )
{
Debug.print(10,"Simulation","configureSimulationPlatform","no config available");
return;
}

// eigenen namen feststellen.
InetAddress addr = null;
String name = null;
addr = com.getLocalHost();
name = addr.getHostName();

this.setAddresse(name,addr,5007);
this.setId("simulation",name,0);

// hole platform config für sich
// stellenbezeichnungen
String[] stellenBezeichnungen = this.config.getStellenFor(name);

// anzahl der jeweiligen stellen
int[]  stellenAnzahl = new int[stellenBezeichnungen.length];
for ( int i = 0 ; i < stellenAnzahl.length ; i++ )
{
stellenAnzahl[i] = this.config.getNumOfStellenFor(name,stellenBezeichnungen[i]);
}
// typen der stellen
String[] stellenTypen = new String[stellenBezeichnungen.length];
for ( int i = 0 ; i < stellenTypen.length ; i++ )
```

```
{
stellenTypen[i] = this.config.getTypeOfStelle(stellenBezeichnungen[i]);
}

// namen der startnachrichten
String[] startNachrichten = new String[stellenBezeichnungen.length];
for ( int i = 0 ; i < startNachrichten.length ; i++ )
{
startNachrichten[i] = this.config.getStartNachricht(stellenBezeichnungen[i]);
}

// konfiguriere stellen
this.stelle = new Stelle[stellenBezeichnungen.length];
for ( int i = 0 ; i < this.stelle.length ; i++ )
{
if ( stellenTypen[i].compareTo("server") == 0 )
this.stelle[i] = new Stelle(this);
else
this.stelle[i] = new ClientStelle(this);
// config id's
this.stelle[i].setAddresse(this.getAddresse());
this.stelle[i].setId(stellenBezeichnungen[i],(new Integer(i)).toString(),0);
this.stelle[i].initProtokoll();
// config startnachricht
if ( stellenTypen[i].compareTo("client") == 0 )
{
Nachricht tmp = this.getNachrichtFromConfig(startNachrichten[i]);
tmp.setId(tmp.getIdName(),this.stelle[i].getIdOrt(),this.stelle[i].getHash());
tmp.setRequest();
tmp.setAbsender(this.stelle[i].getAddresse());
tmp.setZielort(this.getServerAddresseFromConfig(tmp.getTypeOfZ()));
((ClientStelle)this.stelle[i]).setStartNachricht(tmp);
}
// anzahl der clients bestimmen
if ( stellenTypen[i].compareTo("client") == 0 )
((ClientStelle)this.stelle[i]).setMax(stellenAnzahl[i]);
}
}

/**
 * Methode zur Durchführung einer Simulation im Master-Modus.
 * @param conf Ein Konfigurationsstring.
 */
public void runMasterSim(String conf)
{
Debug.print(10,"Simulation","runMasterSim","not implemented");
}

/**
 * Methode zur Durchführung einer Simulation im Test-Modus.
 * @param conf Ein Konfigurationsstring.
 */
```

```
public void runTestSim(String conf)
{
this.start();
// setup config
this.config = new SimConfig(conf);
// make slavelist
String[] slaves = this.config.getSimSlaveList();
// prepare slaves for Simulation
for ( int i = 0 ; i < slaves.length ; i++ )
{
com.sendComand(slaves[i],"resetMaster");
com.sendInetAddress(slaves[i],com.getOwnAddress());
com.sendComand(slaves[i],"receiveConfig");
com.sendString(slaves[i],conf);
}
// start SimAgents on Slaves
// start test
for ( int i = 0 ; i < slaves.length ; i++ )
{
com.sendComand(slaves[i],"startSimulationPlatform");
}

//run test
try
{
//this.sleep( 60000 ); // 1 min
this.sleep( 300000 ); // 5 min
//this.sleep( 600000 ); // 10 min
//this.sleep( 3600000 ); // 60 min
//this.sleep( 36000000 ); // 10 std
} catch (Exception e) { Debug.print(5,"Simulation","runTestSim","wait failed " + e);
}

// get protokolls
String protokoll = new String();
for ( int i = 0 ; i < slaves.length ; i++ )
{
protokoll += this.getProtokollFrom(slaves[i]);
}

// print protokolls
System.out.println("TEST Protokoll (begin)");
System.out.println(protokoll);
System.out.println("TEST Protokoll (end)");

// stop test
for ( int i = 0 ; i < slaves.length ; i++ )
{
com.sendComand(slaves[i],"stopSimulationPlatform");
}

// terminate slaves
```

```
try
{
 this.sleep( 5000 );
} catch (Exception e) { Debug.print(5,"Simulation","runTestSim","wait failed " + e);
}
for ( int i = 0 ; i < slaves.length ; i++ )
{
com.sendComand(slaves[i],"endOfComunication");
}
// terminate self
this.stopSimulationPlatform();
this.com.suicide();
}

/**
 * Methode zur Sammeln von Protokollen der entfernten Simulationsplatform.
 * @param slave Bezeichnung der entfernten Simulationsplatform.
 * @return Das Protokoll der entfernten Simulationsplatform.
 */
String getProtokollFrom(String slave)
{
com.sendComand(slave,"sendProtokoll");
return com.receiveString();
}

/**
 * Methode zur Sammeln von Protokollen der lokalen Simulationsplatform.
 * @return Das Protokoll der lokalen Stellen.
 */
String getProtokoll()
{
String protokoll = new String();
for ( int i = 0 ; i < this.stelle.length ; i++ )
if ( this.stelle[i] != null ) protokoll += this.stelle[i].getProtokoll();
if ( protokoll.compareTo("") == 0 ) protokoll += "\n no protokoll available \n";
return protokoll;
}

/**
 * Methode zur Durchführung einer Simulation im Slave-Modus.
 */
public void runSlaveSim()
{
com.slaveModeComunication();
}

public void run()
{
this.runSlaveSim();
}
}
```

A.1.2. DatagramReceiveAgent

```
import java.net.DatagramPacket;
/**
 * Diese Klasse impementiert einen Berechnungsfaden, der der Simulationsplatform
 * an die er
 * gebunden ist ein Datagram entnimmt, dieses in ein Datenpacket umrechnet,
 * und in die Liste P_received der Stelle einträgt, für die das Datenpacket
 * bestimmt ist.
 * @author xelahr
 * @version 0.3
 */
class DatagramReceiveAgent extends Agent
{
/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle,
 * und integriert ihn in eine ThreadGroup, aus welcher auf den Agenten
 * zugegriffen werden kann.
 * @param master Stelle an die der Agent gebunden wird.
 * @param agents ThreadGroup in die der Agent integriert wird.
 */
DatagramReceiveAgent(Stelle master,ThreadGroup agents)
{
this.super(master,agents);
}

/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle.
 * @param master Stelle an die der Agent gebunden wird.
 */
DatagramReceiveAgent(Stelle master)
{
this.super(master);
}

/**
 * Diese Methode ist Bestandteil des Runnable Interfaces.
 * @see java.lang.Runnable
 */
public void run()
{
while(master.isAlife())
{
try
{
// hole ein datagram
DatagramPacket dat = (DatagramPacket)master.getFromP_receiv();
if (dat != null )
{
// baue ein datenpacket daraus
Datenpacket pack = new Datenpacket(dat);
// liefere das datenpacket an die zugehörige stelle
int addr = Integer.parseInt(pack.getNId().ort);
```

```
((Simulation)master).sendToStelle(addr, pack);
}
}
catch (Exception e)
{
Debug.print(120,"DatagramReceiveAgent","run","no packet received");
}
try
{
 this.sleep( 1 );
} catch (Exception e)
{
Debug.print(25,"DatagramReceiveAgent","run","wait failed " + e);
}
}
}
}
```

A.1.3. DatagramSendAgent

```
import java.net.DatagramPacket;
/**
 * Diese Klasse impementiert einen Berechnungsfaden, der der Simulationsplatform
 * an die er
 * gebunden ist ein Datenpacket entnimmt, dieses in ein Datagram umrechnet,
 * und dieses Datagram uber die Simulationsplatform zum Versenden an die
 * Kommunikationseinheit ComDevice übergibt.
 * @author xelahr
 * @version 0.3
 */
class DatagramSendAgent extends Agent
{
/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle,
 * und integriert ihn in eine ThreadGroup, aus welcher auf den Agenten
 * zugegriffen werden kann.
 * @param master Stelle an die der Agent gebunden wird.
 * @param agents ThreadGroup in die der Agent integriert wird.
 */
DatagramSendAgent(Stelle master,ThreadGroup agents)
{
this.super(master,agents);
}

/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle.
 * @param master Stelle an die der Agent gebunden wird.
 */
DatagramSendAgent(Stelle master)
{
this.super(master);
}
```

```
/**
 * Diese Methode ist Bestandteil des Runnable Interfaces.
 * @see java.lang.Runnable
 */
public void run()
{
while(master.isAlife())
{
try
{
// hole ein datenpacket
Datenpacket pack = (Datenpacket)master.getFromP_tosend();
// zerlege es in ein datagram
DatagramPacket dat = pack.toDatagramPacket();
// verschicke es ueber die com einheit
((Simulation)master).sendToNet(dat);
}
catch (Exception e)
{
Debug.print(110,"DatagramSendAgent","run","no packet send");
}
try
{
 this.sleep( 1 );
} catch (Exception e)
{ Debug.print(25,"DatagramSendAgent","run","wait failed " + e); }
}
}
}
```

A.2. ComDevice

```
import java.net.InetAddress;
import java.net.DatagramPacket;
/**
 * Diese Klasse impementiert die Schnittstelle der Simulationsplatform zum Netzwerk.
 *
 * @author xelahr
 * @version 0.3
 */

class ComDevice extends Thread
{
/**
 * Die zugehörige Simulationsplatform
 */
private Simulation master = null;
private boolean alife;

/**
 * Die Internet Adresse der Maschine, die Configurationen verschickt, und
```

```
 * Protokolle der Simulationen einsammelt.
 */
private InetAddress masterOfSimulation;
/**
 * Die Internet Adresse der Maschine, die aktuell Befehle oder Daten empfängt.
 */
private InetAddress actualSlave;
/**
 * Die Internet Adresse eigenen Maschine.
 */
private InetAddress ownAddress;

/**
 * Ein Buffer.
 */
private String stringBuffer = null;

/**
 * Agent zum verschicken und empfangen von Kommando-Datagrammen
 */
private DatagramAgent cmdMgr;

/**
 * Agent zum verschicken und empfangen von Daten (Objecten)
 */
private DataAgent datMgr;

// Agent zum verschicken und empfangen von Nachrichten
//private MessageAgent msgMgr;

/**
 *  Agent zum durchfuehren einer Simulation
 */
private SimulationAgent simMgr;

/**
 * Erzeugt ein neues ComDevice, wobei die folgende Ports genutzt werden:
 * DatagramAgent(5000,5001), DataAgent(5004), SimulationAgent(5006,5007).
 * @param master Die zugehörige Simulationsplatform.
 */
ComDevice(Simulation master)
{
this.master = master;
this.alife = false;

// cmdSlaveSocket: 5000 ; cmdMasterSocket: 5001
// zum empfangen       und senden von Kommandos
cmdMgr = new DatagramAgent(this, 5000, 5001);

//    SlaveSocket: 5002 ;    MasterSocket: 5003
// zum empfangen       und senden von Nachrichten
//msgMgr = new MessageAgent(this, 5002, 5003);
```

```
// dataToSend Port : 5004
datMgr = new DataAgent(this, 5004);

// simulationSocket Port : 5006
// simulationReceivingPort : 5007
simMgr = null; //new SimulationAgent(this, 5006, 5007);

masterOfSimulation = this.getByName("nexus.upb.de");
ownAddress = this.getLocalHost();
actualSlave = this.getLocalHost();

if (this.masterOfSimulation != null) this.alife = true;

}

/**
 * Gibt die Internet Adresse des Kommunikationspartners zurück.
 * @return Internet Adresse des Kommunikationspartners.
 */
InetAddress getActualSlave() { return this.actualSlave; }
/**
 * Setzt Adresse des aktuellen Kommunikationspartners..
 * @param addr Adresse des Kommunikationspartners.
 */
void setActualSlave(String name) { this.actualSlave = this.getByName(name); }
/**
 * Gibt die eigene Internet Adresse zurück.
 * @return eigene Internet Adresse
 */
InetAddress getOwnAddress() { return this.ownAddress; }
/**
 * Gibt Internet Adresse des Simulationsmasters zurück.
 * @return Internet Adresse des Simulationsmasters
 */
InetAddress getMasterOfSimulation() { return this.masterOfSimulation; }
/**
 * Setzt den Simulationsmaster.
 * @param addr Adresse des neuen SimulationMasters.
 */
void setMasterOfSimulation(String name)
{ this.masterOfSimulation = this.getByName(name); }
/**
 * Setzt den Simulationsmaster.
 * @param addr neuer SimulationMaster.
 */
void setMasterOfSimulation(InetAddress addr) { this.masterOfSimulation = addr; }
/**
 * Gibt den Buffer zurück.
 * @return Buffer String.
 */
String getStringBuffer() { return this.stringBuffer; }
```

```
/**
 * Setzt den Buffer.
 * @param str neuer Buffer String.
 * @return neuer Buffer String.
 */
String setStringBuffer(String str) { this.stringBuffer = str; return str; }
/**
 * Empfängt über den DataAgenten einen String.
 * @return Empfangener String.
 */
String receiveString() { return this.datMgr.receiveString(); }
/**
 * Empfängt über den DataAgenten eine Internet Adresse.
 * @return Empfangene Adresse.
 */
InetAddress receiveInetAddress() { return this.datMgr.receiveInetAddress(); }
/**
 * Sendet über den DataAgenten eine Internet Adresse an den Empfänger.
 * @param rcp Empfänger
 * @param addr String
 */
void sendInetAddress(String rcp, InetAddress addr)
{ this.datMgr.sendInetAddress(rcp,addr); }
/**
 * Sendet über den DataAgenten ein String an aktuellen Kommunikationspartner.
 * @param str String
 */
void sendString(String str) { this.datMgr.sendString(str); }
/**
 * Sendet über den DataAgenten ein String an den Empfänger.
 * @param rcp Empfänger
 * @param str String
 */
void sendString(String rcp, String str) { this.datMgr.sendString(rcp,str); }
/**
 * Sendet über den DataAgenten ein String an den Empfänger.
 * @param rcp Empfänger
 * @param str String
 */
void sendString(InetAddress rcp, String str) { this.datMgr.sendString(rcp,str); }
/**
 * Sendet das Protokoll an den Simulationsmaster.
 */
void sendProtokoll()
{ this.sendString(this.masterOfSimulation,master.getProtokoll()); }
/**
 * Sendet über den DatagramAgenten ein Kommando an den Empfänger.
 * @param slave Empfänger
 * @param comand Kommando
 */
void sendComand(String slave, String comand)
{ this.cmdMgr.sendComand(slave,comand); }
```

```
/**
 * Konfiguriert die Simulationsplatform mit dem übergebenen Konfigurationsstring.
 * @param conf Konfigurationsstring
 */
void configureSlave(String conf) { master.configure(conf); }
/**
 * Startet den SimulationsAgenten. (Ports:5006, 5007)
 */
void startSimulationAgent()
{ this.simMgr=new SimulationAgent(this, 5006, 5007);this.simMgr.start(); }
/**
 * Stoppt den SimulationsAgenten.
 */
void stopSimulationAgent() { if (this.simMgr!=null) this.simMgr.suicide(); }
/**
 * Startet die Simulation auf der Simulationsplatform.
 */
void startSimulationPlatform() { master.startSimulationPlatform(); }
/**
 * Stoppt die Simulation auf der Simulationsplatform.
 */
void stopSimulationPlatform() { master.stopSimulationPlatform(); }
/**
 * Sendet das Datagram über den SimulationsAgenten.
 * @param dat Das zu sendende Datagram
 */
void sendSimulationDatagram(DatagramPacket dat) { this.simMgr.send(dat); }
/**
 * Übergibt das Datagram an die Simulationsplatform.
 * @param dat Das zu übergebende Datagram
 */
void receiveSimulationDatagram(DatagramPacket dat)
{ this.master.receiveFromNet(dat); }
/**
 * Ermittelt die Internet Adresse der Maschiene deren Name übergeben wurde,
 * und gibt diese zurück.
 * @return  falls möglich Internet Adresse der eigenen Maschiene, null sonst.
 */
static InetAddress getByName(String name)
{
InetAddress addr = null;
try
{
addr = InetAddress.getByName(name);
}
catch (Exception e)
{
Debug.print(10,"ComDevice","getByName","no name service: "+e);
}
return addr;
}
/**
```

```
 * Ermittelt die Internet Adresse der eigenen Maschiene, und gibt diese zurück.
 * @return  falls möglich Internet Adresse der eigenen Maschiene, null sonst.
 */
static InetAddress getLocalHost()
{
InetAddress addr = null;
try
{
addr = InetAddress.getLocalHost();
}
catch (Exception e)
{
Debug.print(10,"ComDevice","getLocalHost","no name service: "+e);
}
return addr;
}
/**
 * Gibt den Namen des Rechners zurück, falls möglich.
 * @return Name des Rechners falls vorhanden, sonst "unknown".
 */
String getLocalName()
{
String name = "unknown";
if (this.ownAddress!=null) name = this.ownAddress.getHostName();
return name;
}
/**
 * Gibt den Simulationsport zurück.
 * @return Simulationsport.
 */
static int getSimulationPort()
{
return 5007;
}
/**
 * Methode zur Kommunikation im Slave-Modus: Empfange einen Befehl vom Netz
 * und führe diesen unverzüglich aus.
 */
void slaveModeComunication()
{
while(this.alife) cmdMgr.executeComand(cmdMgr.receiveComand());
}
/**
 * Methode zur Beendihung jeglicher Aktivität und der Freigabe der besetzten
 * Ports.
 */
void suicide()
{
this.alife = false;
this.cmdMgr.suicide();
this.datMgr.suicide();
}
```

```
void runSimulation(){}
}
```

A.2.1. DatagramAgent

```
import java.net.DatagramSocket;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.util.Vector;
/**
 * Diese Klasse impementiert einen Agenten zum Senden und Empfangen von
 * Kommando Datagramen.
 *
 * @author xelahr
 * @version 0.3
 */
class DatagramAgent extends Thread
{
/**
 * Das zugehörige ComDevice.
 */
private ComDevice master = null;

private boolean alife = true;

/**
 * Portlauscher zum Empfangen von Kommandos
 */
private DatagramLurker slave = null;

/**
 * Socket zum Senden von Kommandos
 */
private DatagramSocket cmdMasterSocket = null;

//private Vector incomming = new Vector();

/**
 * Kreiert einen DatagramAgenten.
 * @param newMaster Das ComDevice
 * @param cmdSlavePort Port für das Empfangen von Komandos
 * @param cmdMasterPort Port für das Senden von Komandos
 */
DatagramAgent(ComDevice newMaster, int cmdSlavePort, int cmdMasterPort)
{
this.master = newMaster;
try
{
this.cmdMasterSocket   = new DatagramSocket(cmdMasterPort);
}
catch (Exception e)
{ Debug.print(10,"DatagramAgent","DatagramAgent","socket not open: "+e); }
```

```
this.slave = new DatagramLurker(this, cmdSlavePort);
this.slave.start();
}

public void run()
{
Debug.print(10,"DatagramAgent","run","not implemented");
}

/**
 * Methode zur Beendihung jeglicher Aktivität und der Freigabe der besetzten
 * Ports.
 */
void suicide()
{
this.cmdMasterSocket.close();
this.slave.suicide();
this.alife = false;
this.master = null;
}

/**
 * Methode zum Empfangen eines Kommandos.
 * @return Ein kommando Datagram.
 */
synchronized DatagramPacket receiveComand()
{
DatagramPacket comandPacket = null;
int time = 0;
boolean received = false;
while(!received)
{
time++;
try
{
this.wait(1000);
} catch (Exception e)
{ Debug.print(25,"DatagramAgent","receiveComand: ","wait failed: "+e); }
if (slave.hasNextPacket())
{
comandPacket = slave.getNextPacket();
received = true;
time = 0;
}
}
return comandPacket;
}

/**
 * Methode zum Senden eines Kommandos.
 * @param toSend Ein kommando Datagram zum Senden.
 */
```

```
void send(DatagramPacket toSend)
{
try
{
this.cmdMasterSocket.send(toSend);
}
catch (Exception e)
{
Debug.print(25,"DatagramAgent","send","send failed: "+e);
}
}

/**
 * Methode zum Senden eines Kommandos. Kommando wird an den aktuellen
 * Kommunikationspartner gesendet.
 * @param komando Ein kommando String zum Senden.
 */
void sendComand(String komando)
{
this.sendComand(master.getActualSlave(),komando);
}

/**
 * Methode zum Senden eines Kommandos.
 * @param slave Der Empfänger.
 * @param komando Ein kommando String zum Senden.
 */
void sendComand(String slave, String komando)
{
this.sendComand(master.getByName(slave),komando);
}

/**
 * Methode zum Senden eines Kommandos.
 * @param slave Der Empfänger.
 * @param komando Ein kommando String zum Senden.
 */
void sendComand(InetAddress slave,String komando)
{
DatagramPacket comandPacket = this.makeComDatagram(slave,komando);
this.send(comandPacket);
}

/**
 * Methode zum Generieren eines KommandoDatagrams.
 * Kommando wird an den aktuellen
 * Kommunikationspartner gesendet.
 * @param komando Ein kommando String.
 * @return Das entsprechende KommandoDatagram.
 */
DatagramPacket makeComDatagram(String komando)
{
```

```
return this.makeComDatagram(master.getActualSlave(), komando);
}

/**
 * Methode zum Generieren eines KommandoDatagrams.
 * @param slave Ein Empfänger.
 * @param komando Ein kommando String.
 * @return Das entsprechende KommandoDatagram.
 */
DatagramPacket makeComDatagram(InetAddress slave, String komando)
{
byte[] hashCode = new byte[9];
for (int i = 0; i < hashCode.length; i++)
   hashCode[i] = (byte)(Math.random() * 255 + 0.5);
return this.makeComDatagram(slave, komando, hashCode);
}

/**
 * Methode zum Generieren eines KommandoDatagrams.
 * @param slave Ein Empfänger.
 * @param komando Ein kommando String.
 * @param message Zusatznachricht (z.B. ein HashCode).
 * @return Das entsprechende KommandoDatagram.
 */
DatagramPacket makeComDatagram(InetAddress slave, String komando, byte[] message)
{
byte[] cmdBuf = new byte[message.length+1];
cmdBuf[0] = 42;
for (int i=1 ; i < cmdBuf.length; i++) cmdBuf[i] = message[i-1];
DatagramPacket tmpDatagramPacket = new DatagramPacket(cmdBuf, cmdBuf.length);

tmpDatagramPacket.setAddress(slave);
tmpDatagramPacket.setPort(5000);

if      (komando.compareTo("runSimulation")==0) cmdBuf[0] =  0;
else if (komando.compareTo("generateDataToSend")==0) cmdBuf[0] =  1;
else if (komando.compareTo("resetMaster")==0) cmdBuf[0] =  2;
else if (komando.compareTo("startSimAgents")==0) cmdBuf[0] =  3;
else if (komando.compareTo("stopSimAgents")==0) cmdBuf[0] =  4;
else if (komando.compareTo("startSimulationPlatform")==0) cmdBuf[0] =  5;
else if (komando.compareTo("stopSimulationPlatform")==0) cmdBuf[0] =  6;
else if (komando.compareTo("endOfComunication")==0) cmdBuf[0] =  9;
else if (komando.compareTo("receivePacket")==0) cmdBuf[0] = 10;
else if (komando.compareTo("receivePacketList")==0) cmdBuf[0] = 11;
else if (komando.compareTo("receiveConfig")==0) cmdBuf[0] = 17;
else if (komando.compareTo("receiveString")==0) cmdBuf[0] = 18;
else if (komando.compareTo("receiveData")==0) cmdBuf[0] = 19;
else if (komando.compareTo("sendPacket")==0) cmdBuf[0] = 20;
else if (komando.compareTo("sendPacketList")==0) cmdBuf[0] = 21;
else if (komando.compareTo("sendProtokoll")==0) cmdBuf[0] = 27;
else if (komando.compareTo("sendString")==0) cmdBuf[0] = 28;
else if (komando.compareTo("sendData")==0) cmdBuf[0] = 29;
```

```
else if (komando.compareTo("acknowledgePacket")==0) cmdBuf[0] = 30;
else if (komando.compareTo("acknowledgePacketList")==0) cmdBuf[0] = 31;
else if (komando.compareTo("acknowledgeString")==0) cmdBuf[0] = 38;
else if (komando.compareTo("acknowledgeData")==0) cmdBuf[0] = 39;
else cmdBuf[0] = 42;

tmpDatagramPacket.setData(cmdBuf, 0, cmdBuf.length);

return tmpDatagramPacket;
}

/**
 * Methode zum Ausführen eines KommandoDatagrams.
 * @param komandoDatagram Ein Kommando.
 */
void executeComand(DatagramPacket komandoDatagram)
{
String komando = this.interpretComDatagram(komandoDatagram);
if      (komando.compareTo("runSimulation")==0) master.runSimulation();
//else if (komando.compareTo("generateDataToSend")==0)
//   master.generateDataToSend();
//else if (komando.compareTo("startSimAgents")==0) master.startSimAgents();
//else if (komando.compareTo("stopSimAgents")==0) master.stopSimAgents();
else if (komando.compareTo("startSimulationPlatform")==0)
   master.startSimulationPlatform();
else if (komando.compareTo("stopSimulationPlatform")==0)
   master.stopSimulationPlatform();
else if (komando.compareTo("endOfComunication")==0) master.suicide();
else if (komando.compareTo("receiveConfig")==0)
   master.configureSlave(master.receiveString());
else if (komando.compareTo("sendProtokoll")==0) master.sendProtokoll();
else if (komando.compareTo("receiveString")==0)
   master.setStringBuffer(master.receiveString());
else if (komando.compareTo("sendString")==0)
   master.sendString(master.getStringBuffer());
else if (komando.compareTo("resetMaster")==0)
   master.setMasterOfSimulation(master.receiveInetAddress());
//else if (komando.compareTo("receivePacket")==0)
//   master.packetBuffer = master.datMgr.receivePacket();
//else if (komando.compareTo("receivePacketList")==0)
//   master.receivePacketList();
else
   Debug.print(15,"DatagramAgent","executeComand","unknown comand: "+komando);
}

/**
 * Methode zum Interpretieren eines KommandoDatagrams.
 * @param tmpDatagramPacket Ein KommandoDatagram.
 * @return Die Interpretation des KommandoDatagrams.
 */
String interpretComDatagram(DatagramPacket tmpDatagramPacket)
{
```

```
String buffer = new String();
byte[] cmdBuf = tmpDatagramPacket.getData();
if      (cmdBuf[0]== 0) buffer+="runSimulation";
else if (cmdBuf[0]== 1) buffer+="generateDataToSend";
else if (cmdBuf[0]== 2) buffer+="resetMaster";
else if (cmdBuf[0]== 3) buffer+="startSimAgents";
else if (cmdBuf[0]== 4) buffer+="stopSimAgents";
else if (cmdBuf[0]== 5) buffer+="startSimulationPlatform";
else if (cmdBuf[0]== 6) buffer+="stopSimulationPlatform";
else if (cmdBuf[0]== 9) buffer+="endOfComunication";
else if (cmdBuf[0]==10) buffer+="receivePacket";
else if (cmdBuf[0]==11) buffer+="receivePacketList";
else if (cmdBuf[0]==17) buffer+="receiveConfig";
else if (cmdBuf[0]==18) buffer+="receiveString";
else if (cmdBuf[0]==19) buffer+="receiveData";
else if (cmdBuf[0]==20) buffer+="sendPacket";
else if (cmdBuf[0]==21) buffer+="sendPacketList";
else if (cmdBuf[0]==27) buffer+="sendProtokoll";
else if (cmdBuf[0]==28) buffer+="sendString";
else if (cmdBuf[0]==29) buffer+="sendData";
else if (cmdBuf[0]==30) buffer+="acknowledgePacket";
else if (cmdBuf[0]==31) buffer+="acknowledgePacketList";
else if (cmdBuf[0]==38) buffer+="acknowledgeString";
else if (cmdBuf[0]==39) buffer+="acknowledgeData";
else if (cmdBuf[0]==42) Debug.print(15,"DatagramAgent","interpretDatagram","blank comand");
else Debug.print(15,"DatagramAgent","interpretDatagram","unknown comand");
return buffer;
}

}
```

A.2.2. DatagramLurker

```
import java.net.DatagramSocket;
import java.net.DatagramPacket;
import java.util.Vector;
/**
 * Diese Klasse impementiert einen Agenten zum Empfangen und zwischenspeichern von
 * Datagramen.
 *
 * @author xelahr
 * @version 0.3
 */
class DatagramLurker extends Thread
{
/**
 * Der zugehörige DatagramAgent.
 */
private DatagramAgent master = null;

/**
 * Socket zum Empfangen von Kommandos
```

```
 */
private DatagramSocket cmdSlaveSocket = null;

private boolean alife = true;

/**
 * Buffer für Empfangene Datagrame
 */
private Vector received = new Vector();

/**
 * Kreiert einen DatagramLurker.
 * @param newMaster Der DatagramAgent
 * @param DataPort Port zum Empfangen von Datagramen.
 */
DatagramLurker(DatagramAgent newMaster, int cmdSlavePort)
{
this.master = newMaster;
try
{
this.cmdSlaveSocket = new DatagramSocket(cmdSlavePort);
}
catch ( java.io.IOException e )
{
Debug.print(10,"DatagramLurker","DatagramLurker","port not open: "+e);
}
}

/**
 * Solange der Agent läuft werden Datagramme empfangen und zwischengespeichert.
 */
public void run()
{
while(this.alife)
{
this.addToReceived(this.receiveDatagram());
}
}

/**
 * Abfrage ob der Buffer leer ist.
 * @return "true" falls Buffer leer, "false" sonst.
 */
boolean hasNextPacket()
{
return !this.received.isEmpty();
}

/**
 * Gibt das älteste Datagram zurück, und löscht es aus dem Buffer.
 * @return Das älteste Datagram.
 */
```

```
DatagramPacket getNextPacket()
{
DatagramPacket tmpPacket = (DatagramPacket)(this.received.firstElement());
this.received.remove(0);
return tmpPacket;
}

/**
 * Fügt ein Datagram in den Buffer ein.
 * @param packet Ein Datagram.
 */
private void addToReceived(DatagramPacket packet)
{
this.received.addElement(packet);
}

/**
 * Gibt ein empfangenes Datagram zurück.
 * @return Das älteste Datagram.
 */
private DatagramPacket receiveDatagram()
{
DatagramPacket datagramPacket = new DatagramPacket( new byte[10], 10 );
try
{
this.cmdSlaveSocket.receive(datagramPacket);
} catch (java.io.IOException e)
{ Debug.print(10,"DatagramLurker","receiveDatagram","receive comand failed: "+e); }
return datagramPacket;
}

/**
 * Methode zur Beendihung jeglicher Aktivität und der Freigabe der besetzten
 * Ports.
 */
void suicide()
{
this.alife = false;
this.cmdSlaveSocket.close();
this.master = null;
}
}
```

A.2.3. DataAgent

```
import java.net.ServerSocket;
import java.net.Socket;
import java.net.InetAddress;
import java.io.OutputStream;
import java.io.ObjectOutputStream;
import java.io.InputStream;
import java.io.ObjectInputStream;
```

```
/**
 * Diese Klasse impementiert einen Agenten zum Senden und Empfangen von
 * Komplexen Daten.
 *
 * @author xelahr
 * @version 0.3
 */
class DataAgent extends Thread
{
/**
 * Das zugehörige ComDevice.
 */
private ComDevice master = null;

private boolean alife = false;

/**
 * Socket fuer den Datentransfer
 */
private ServerSocket dataReceiveSocket;
/**
 * Socket fuer den Datentransfer
 */
private Socket dataSendSocket;

/**
 * Internet Adresse des letzten Kommunikationspartners.
 */
private InetAddress last;

/**
 * Kreiert einen DataAgent.
 * @param newMaster Das ComDevice
 * @param DataPort Port für das DatenSocket
 */
DataAgent(ComDevice newMaster, int DataPort)
{
this.master = newMaster;
try
{
this.dataReceiveSocket = new ServerSocket(DataPort);
this.dataSendSocket    = new Socket(InetAddress.getLocalHost(),DataPort);
this.dataReceiveSocket.accept();
this.last = master.getOwnAddress();
this.alife = true;
}
catch (Exception e)
{
Debug.print(10,"DataAgent","DataAgent","no network: "+e);
}
}
```

```
/**
 * Methode zur Beendihung jeglicher Aktivität und der Freigabe der besetzten
 * Ports.
 */
void suicide()
{
try
{
this.dataSendSocket.close();
this.dataReceiveSocket.close();
this.alife = false;
this.master = null;
} catch (Exception e) { Debug.print(10,"DataAgent","suicide","failed"); }
}

/**
 * Methode zum Senden von Objekten.
 * @param recipient Empfänger
 * @param toSend das zu sendende Object.
 */
void sendObject(InetAddress recipient, Object toSend)
{
try
{
dataSendSocket = new Socket(recipient, 5004);
OutputStream o = dataSendSocket.getOutputStream();
ObjectOutputStream s = new ObjectOutputStream(o);

s.writeObject(toSend);
//s.writeChars(toSend.toString());
s.flush();
s.close();
}
catch (Exception e)
{
Debug.print(10,"DataAgent","sendObject","send object failed: "+e);
}
}

/**
 * Methode zum Empfangen von Objekten.
 * @return das empfangene Object.
 */
synchronized Object receiveObject()
{
Object receivedObject = null;
try
{
Socket soc = dataReceiveSocket.accept();
this.last = soc.getInetAddress();
InputStream o = soc.getInputStream();
try
```

```
{
 this.wait( 1000 );
} catch (Exception e)
{ Debug.print(5,"DataAgent","receiveObject","wait failed " + e); }
if(o.available()>0)
{
ObjectInputStream s = new ObjectInputStream(o);
receivedObject = s.readObject();
s.close();
}
//else
// receivedObject = this.receiveObject();
}
catch ( Exception e )
{
Debug.print(10,"DataAgent","receiveObject","receive object failed: "+e);
}
return receivedObject;
}

/**
 * Methode zum Senden von Objekten. Das Objekt wird an den aktuellen
 * Kommunikationspartner geschickt.
 * @param toSend das zu sendende Object.
 */
void sendObject(Object toSend)
{
this.sendObject(master.getActualSlave(), toSend);
}

/**
 * Methode zum Senden von Objekten.
 * @param recipient Empfänger
 * @param toSend das zu sendende Object.
 */
void sendObject(String recipient, Object toSend)
{
this.sendObject(master.getByName(recipient), toSend);
}

/**
 * Methode zum Senden von Strings. Das String wird an den aktuellen
 * Kommunikationspartner geschickt.
 * @param toSend das zu sendende String.
 */
void sendString(String toSend)
{
this.sendObject(toSend);
}

/**
 * Methode zum Senden von Strings.
```

```
 * @param recipient Empfänger
 * @param toSend das zu sendende String.
 */
void sendString(String recipient, String toSend)
{
this.sendObject(recipient,toSend);
}

/**
 * Methode zum Senden von Strings.
 * @param recipient Empfänger
 * @param toSend das zu sendende String.
 */
void sendString(InetAddress recipient, String toSend)
{
this.sendObject(recipient,toSend);
}

/**
 * Methode zum Senden von Internet Adressen.
 * @param recipient Empfänger
 * @param toSend die zu sendende Adresse.
 */
void sendInetAddress(String recipient, InetAddress toSend)
{
this.sendObject(recipient,toSend);
}

/**
 * Methode zum Empfangen von Strings.
 * @return das empfangene String.
 */
String receiveString()
{
String buf = (String)this.receiveObject();
return buf;
}

/**
 * Methode zum Empfangen von Internet Adressen.
 * @return die empfangene Adresse.
 */
InetAddress receiveInetAddress()
{
InetAddress addr = (InetAddress)this.receiveObject();
return this.last;
}

}
```

A.2.4. SimulationAgent

```
import java.net.DatagramSocket;
import java.net.DatagramPacket;
/**
 * Diese Klasse impementiert einen Agenten zum Senden und Empfangen von
 * SimulationsDatagramen.
 *
 * @author xelahr
 * @version 0.3
 */
class SimulationAgent extends Thread
{
boolean alife = true;

/**
 * Maximale Grösse eines zu empfangenen Datagramms (in bytes).
 */
public static final int MAX_SIZE_OF_DATAGRAM = 32768;

/**
 * Das zugehörige ComDevice.
 */
private ComDevice master = null;

//private SynObjCont received = null;

/**
 * Socket fuer den Simulationsdatenverkehr.
 */
private DatagramSocket simulationSocket;

/**
 * Socket fuer den Simulationsdatenverkehr.
 */
private DatagramSocket receivingSocket;

/**
 * Kreiert einen neuen SimulationAgent.
 * @param newMaster zugehöriges ComDevice
 * @param simulationPort port zum senden von Datagramen
 * @param receivingPort port zum empfangen von Datagramen
 */
SimulationAgent(ComDevice newMaster, int simulationPort, int receivingPort)
{
this.master = newMaster;

try
{
this.simulationSocket = new DatagramSocket(simulationPort);
this.receivingSocket = new DatagramSocket(receivingPort);
this.alife = true;
}
```

```
catch (Exception e)
{
Debug.print(10,"SimulationAgent","creation","sockets not free"+e);
this.alife = false;
}
}

/**
 * Solange der Agent läuft werden Datagramme empfangen und durch das
 * ComDevice an die Simulationsplatform weitergegeben.
 */
public void run()
{
while(this.alife)
{
master.receiveSimulationDatagram(this.receive());
try
{
 this.sleep( 1 );
} catch (Exception e)
{ Debug.print(25,"SimulationAgent","run","wait failed " + e); }
}
}

/**
 * Methode zum Senden von Datagramen.
 * @param dat das zu sendende Datagram.
 */
void send(DatagramPacket dat)
{
try
{
this.simulationSocket.send(dat);
}
catch ( Exception e )
{
Debug.print(10,"SimulationAgent","send",
             "datagram not sent: "+ dat+" -> "+e);
}
}

/**
 * Methode zum Empfangen von Datagramen.
 * @return Das empfangene Datagram.
 */
DatagramPacket receive()
{
DatagramPacket tmp = new DatagramPacket(new byte[this.MAX_SIZE_OF_DATAGRAM],
                                        this.MAX_SIZE_OF_DATAGRAM);
try
{
this.receivingSocket.receive(tmp);
```

```
} catch (Exception e)
{
Debug.print(10,"DatagramLurker","receive","receive comand failed: "+e);
}
return tmp;
}

/**
 * Methode zum Darstellen von Datagramen.
 * @param tmp Das darzustellende Datagram.
 * @return Die Stringrepräsentation des Datagrams.
 */
static String datagramToString(DatagramPacket tmp)
{
String buffer = new String();
buffer += "ADDR: " + tmp.getAddress();
buffer += " PORT: " + tmp.getPort();
buffer += " SIZE: " + tmp.getLength();
return buffer;
}

/**
 * Methode zur Beendihung jeglicher Aktivität und der Freigabe der besetzten
 * Ports.
 */
void suicide()
{
this.alife = false;
this.simulationSocket.close();
this.receivingSocket.close();
}
}
```

A.3. Kern des Simulationsmodells

A.3.1. Server (Stelle)

```
import java.util.Vector;
import java.util.Iterator;
import java.util.Date;
import java.net.InetAddress;
/**
 * Diese Klasse implementiert die Entität Server.
 *
 * @author xelahr
 * @version 0.3
 */
class Stelle extends Thread
{
protected boolean alife = false;
/**
 * Die zugehörige Simulationsplatform.
```

```
 */
protected Simulation master = null;
/**
 * Das Id-Attribut der Stelle.
 */
protected N_id id = null;
/**
 * Die Ortsangabe der Stelle.
 */
protected Ort addresse = null;
/**
 * Die Liste P_received der Stelle.
 */
protected SynObjCont p_receiv = null;
/**
 * Die Liste P_tosend der Stelle.
 */
protected SynObjCont p_tosend = null;
/**
 * Die Liste N_incomplete der Stelle.
 */
protected Vector n_incomplete = null;
/**
 * Die Liste N_tosend der Stelle.
 */
protected Vector n_tosend = null;
/**
 * Die Liste N_received der Stelle.
 */
protected Vector n_received = null;
/**
 * Die Protokollmitschrift der Stelle.
 */
protected Protokoll protokoll = null;
protected static int hash = 0;
/**
 * Eine Liste aller gestarteter Agenten der Stelle.
 */
protected static ThreadGroup agents = null;

/**
 * Generiert eine neue Stelle des Typs "server".
 */
Stelle()
{
this.p_receiv = new SynObjCont();
this.p_tosend = new SynObjCont();
this.n_incomplete = new Vector();
this.n_tosend = new Vector();
this.n_received = new Vector();
this.protokoll = new Protokoll();
this.agents = new ThreadGroup("simAgents");
```

```
this.alife = true;
}

/**
 * Generiert eine neue Stelle des Typs "server",
 * und bindet diese an die angegebene Simulationsplatform.
 * @param master Eine Simulationsplatform
 */
Stelle(Simulation master)
{
this.master = master;
this.p_receiv = new SynObjCont();
this.p_tosend = new SynObjCont();
this.n_incomplete = new Vector();
this.n_tosend = new Vector();
this.n_received = new Vector();
this.protokoll = new Protokoll();
this.agents = new ThreadGroup("simAgents");
this.alife = true;
}

/**
 * Methode zum Stoppen der Aktivitäten der Stelle.
 */
void stopStelle()
{
this.alife = false;
this.p_receiv = new SynObjCont();
this.p_tosend = new SynObjCont();
this.n_incomplete = new Vector();
this.n_tosend = new Vector();
this.n_received = new Vector();
this.agents = new ThreadGroup("simAgents");
}

/**
 * Initialisiert ein Protokoll.
 */
void initProtokoll()
{
this.protokoll = new Protokoll(this.getIdName()+" "+this.getIdOrt(),
                               this.getAddresse().addr);
}

/**
 * Generiert eine HashZahl und gibt diese zurück.
 * @return Eine HashZahl.
 */
synchronized int getHash()
{
this.hash++;
return this.hash;
```

```
}
/**
 * Setzt die Adresse dieser Stelle.
 * @param name Der Stellenbezeichnung.
 * @param adr Die Internet Addresse der Stelle.
 * @param port Der Port der Internet Addresse der Stelle.
 */
void setAddresse(String name, InetAddress addr, int port)
{
this.addresse = new Ort();
this.addresse.name = name;
this.addresse.addr = addr;
this.addresse.port = port;
}
/**
 * Setzt die Adresse dieser Stelle.
 * @param ort Die Adresse der Stelle.
 */
void setAddresse(Ort addr) { this.addresse = addr; }
/**
 * Gibt die Adresse dieser Stelle zurück.
 * @return Die Adresse der Stelle.
 */
Ort getAddresse() { return this.addresse; }
/**
 * Gibt die Simulationsplatform dieser Stelle zurück.
 * @return Die Simulationsplatform der Stelle.
 */
Simulation getMaster() { return this.master; }
/**
 * Abfrage ob diese Stelle aktiv ist.
 * @return "true" wenn die Stelle aktiv ist, "false" sonst.
 */
boolean isAlife() {return this.alife; }
/**
 * Setzt das Attribut alife.
 * @param life Das Attribut alife.
 */
void setAlife(boolean life) { this.alife=life; }
/**
 * Gibt die Bezeichnung dieser Stelle zurück.
 * @return  Die Bezeichnung dieser Stelle.
 */
String getIdName() { return this.id.name; }
/**
 * Gibt die Ortsangabe dieser Stelle zurück.
 * @return  Die Ortsangabe dieser Stelle.
 */
String getIdOrt() { return this.id.ort; }
/**
 * Gibt den HashCode dieser Stelle zurück.
 * @return  Der HashCode dieser Stelle.
```

```
 */
int getIdHash() { return this.id.hash; }
/**
 * Gibt die Id dieser Stelle zurück.
 * @return  Die Id dieser Stelle.
 */
N_id getId() { return this.id; }
/**
 * Setzt die Id dieser Stelle.
 * @param id  Die Id dieser Stelle.
 */
void setId(N_id id) { this.id = id; }
/**
 * Setzt die Id dieser Stelle.
 * @param name Die Bezeichnung dieser Stelle.
 * @param ort  Die Ortsangabe dieser Stelle.
 * @param hash Der HashCode dieser Stelle.
 */
void setId(String name, String ort, int hash)
{
this.id = new N_id();
this.id.name = name;
this.id.ort = ort;
this.id.hash = hash;
}

/**
 * Gibt die Protokollmitschrift dieser Stelle zurück.
 * @return  Die Protokollmitschrift dieser Stelle.
 */
String getProtokoll()
{
String buff = new String();
if (this.protokoll != null) buff += this.protokoll.toString();
return buff;
}

/**
 * Fügt einen Eintrag zur Protokollmitschrift dieser Stelle hinzu.
 * @param art Bezeichnung der Anfrageart.
 * @param abs Absendezeitpunkt der Anfrage.
 * @param ank Ankunftszeitpunkt der Antwort.
 */
void addToProtokoll(String art, Date abs, Date ank)
{
if (this.protokoll == null)
{
try
{
this.protokoll = new Protokoll();
this.protokoll.setClientName(this.getIdName());
this.protokoll.setAddr(ComDevice.getLocalHost());
```

```
}
catch(Exception e)
{
Debug.print(10,"Stelle","addToProtokoll",e.toString());
}
}
this.protokoll.addDate( art,  abs,  ank);
}

// interface Nachrichten Agents ###################################
/**
 * Fügt eine Nachricht in die Liste N_incomplete ein.
 * @param msg Eine Nachricht.
 */
void addIncomplete(Nachricht msg)
{
synchronized(this.n_incomplete)
{
this.n_incomplete.add(msg);
}
}
/**
 * Fügt ein Datenpacket in die Liste N_incomplete ein.
 * @param pkg Ein Datenpacket.
 */
void addIncomplete(Datenpacket pkg)
{
N_id id = pkg.getNId();
synchronized(this.n_incomplete)
{
Nachricht inc = null;
// Durchsuche die Liste nach der richtigen Nachricht
Iterator iter = this.n_incomplete.iterator();
while(iter.hasNext())
{
Nachricht tmp = (Nachricht)iter.next();
if ( tmp.getIdName().compareTo(id.name) == 0 && tmp.getIdHash() == id.hash )
{
inc = tmp;
break;
}
}
// falls nicht in der liste
if ( inc == null )
{
inc = master.getNachrichtFromConfig(id.name);
inc.setId(id);
inc.setAbsender(pkg.getAbsender());
inc.setZielort(pkg.getZielort());
this.n_incomplete.add(inc);
}
// integriere datenpacket
```

```
inc.receiveDatenpacket(pkg);
}
}
/**
 * Gibt ein Array mit allen Nachrichten der Liste N_incomplete zurück, die vollständig sind.
 * @return Array mit Nachrichten wenn welche komplett sind, "null" sonst.
 */
Nachricht[] getComplete()
{
Vector buff = new Vector();
synchronized(this.n_incomplete)
{
Iterator iter = this.n_incomplete.iterator();
while (iter.hasNext())
{
Nachricht tmp = (Nachricht)iter.next();
if (tmp.isComplete()) buff.add(tmp);
}
this.n_incomplete.removeAll(buff);
}
if ( buff.size() < 1 )
return null;
Nachricht[] retarr = new Nachricht[buff.size()];
for ( int i = 0 ; i < retarr.length ; i++ )
retarr[i] = (Nachricht)buff.elementAt(i);
return retarr;
}

/**
 * Fügt eine Nachricht in die Liste N_tosend ein.
 * @param msg Eine Nachricht.
 */
void addTosend(Nachricht msg)
{
synchronized(this.n_tosend)
{
this.n_tosend.add(msg);
}
}
/**
 * Fügt eine Nachricht in die Liste N_received ein.
 * @param msg Eine Nachricht.
 */
void addReceived(Nachricht msg)
{
synchronized(this.n_received)
{
this.n_received.add(msg);
}
}
/**
 * Gibt eine Nachricht in die Liste N_tosend zurück, und löscht diese aus der Liste.
```

```
 * @return Die älteste Nachricht in der Liste.
 */
Nachricht getTosend()
{
Nachricht tmp = null;
synchronized(this.n_tosend)
{
try{
tmp = (Nachricht)this.n_tosend.firstElement();
this.n_tosend.removeElementAt(0);
} catch(Exception e){}
}
return tmp;
}
/**
 * Gibt eine Nachricht in die Liste N_received zurück, und löscht diese aus der Liste.
 * @return Die älteste Nachricht in der Liste.
 */
Nachricht getReceived()
{
Nachricht tmp = null;
synchronized(this.n_received)
{
try{
tmp = (Nachricht)this.n_received.firstElement();
this.n_received.removeElementAt(0);
} catch(Exception e){}
}
return tmp;
}

/**
 * Gibt eine Nachricht mit der angegebenen Bezeichnung zurück.
 * @param name Eine Bezeichnung einer Nachricht.
 * @return Eine Nachricht mit der entsprechenden Bezeichnung.
 */
Nachricht getNachrichtByName(String name)
{
return this.master.getNachrichtFromConfig(name);
}

// interface to Simulation
/**
 * Sendet ein Objekt an die Simulationsplatform.
 * @param obj Ein Objekt.
 */
void sendToSim(Object obj)
{
this.master.addToP_tosend(obj);
}

/**
```

```
 * Empfängt ein Objekt von der Simulationsplatform.
 * @param obj Ein Objekt.
 */
void receiveFromSim(Object obj)
{
this.p_receiv.addObject(obj);
}

// packet functions
/**
 * Fügt ein Objekt in die Liste P_tosend ein.
 * @param obj Ein Objekt.
 */
void addToP_tosend(Object obj)
{
this.p_tosend.addObject(obj);
}

/**
 * Fügt ein Objekt in die Liste P_received ein.
 * @param obj Ein Objekt.
 */
void addToP_receiv(Object obj)
{
this.p_receiv.addObject(obj);
}

/**
 * Gibt ein Objekt aus der Liste P_tosend zurück.
 * @return obj Ein Objekt.
 */
Object getFromP_tosend()
{
return this.p_tosend.getNextObject();
}

/**
 * Gibt ein Objekt aus der Liste P_received zurück.
 * @return obj Ein Objekt.
 */
Object getFromP_receiv()
{
return this.p_receiv.getNextObject();
}

// run
/**
 * Hauptmethode der Stelle. Hier werden entsprechend dem Modell
 * Berechnungsfäden (Agenten) gestartet.
 */
public void run()
{
```

```
while(this.alife)
{
// N_received
if(!this.n_received.isEmpty())
   this.startAgent(new NachrichtenRespondAgent(this,this.agents));
// N_tosend
if(!this.n_tosend.isEmpty())
   this.startAgent(new NachrichtenSendAgent(this,this.agents));
// N_incomplete
if(!this.n_incomplete.isEmpty())
   this.startAgent(new NachrichtenCheckAgent(this,this.agents));
// P_received
if(!this.p_receiv.isEmpty())
   this.startAgent(new DatenpacketReceiveAgent(this,this.agents));
// P_tosend
if(!this.p_tosend.isEmpty())
   this.startAgent(new DatenpacketSendAgent(this,this.agents));
try
{
 this.sleep( 1 );
}
catch (Exception e) { Debug.print(25,"Stelle","run","wait failed " + e); }

}
}
/**
 * Methode zum Starten von Berechnungsfäden (Agenten).
 * @param agt Ein Agent.
 */
protected static void startAgent(Agent agt)
{
if (!agt.isAlive()) agt.start();
}
}
```

A.3.2. Client (ClientStelle)

```
/**
 * Diese Klasse implementiert die Entität Client.
 *
 * @author xelahr
 * @version 0.3
 */
class ClientStelle extends Stelle
{
/**
 * Das Attribut C_inuse der Entität Client.
 */
private int inuse = Integer.MAX_VALUE;
/**
 * Das Attribut C_max der Entität Client.
 */
```

```
private int max  = 0;
/**
 * Die Startnachricht Entität Client.
 */
private Nachricht startNachricht = null;

/**
 * Setzt das Attribut C_max.
 * @param numOfClients Anzahl der Clients die simuliert werden sollen.
 */
void setMax(int numOfClients) { this.max = numOfClients; }
/**
 * Methode zum Inkrementieren des Attributes C_inuse
 */
void incInuse() { this.inuse++; }
/**
 * Methode zum Dekrementieren des Attributes C_inuse
 */
void decInuse() { this.inuse--; }
/**
 * Gibt die Startnachricht zurück.
 * @return Die Startnachricht.
 */
Nachricht getStartNachricht() { return this.startNachricht; }
/**
 * Setzt die Startnachricht.
 * @param start  Die Startnachricht.
 */
void setStartNachricht(Nachricht start) { this.startNachricht = start; }
/**
 * Gibt die Adresse einer Stelle mit der gegebenen Bezeichnung zurück.
 * @param server Die Bezeichnung der Stelle.
 * @return Die Adresse der Stelle.
 */
Ort getServerAddresse(String server)
{ return master.getServerAddresseFromConfig(server); }

/**
 * Generiert eine neue Stelle des Typs "client",
 * und bindet diese an die angegebene Simulationsplatform.
 * @param master Eine Simulationsplatform
 */
ClientStelle(Simulation master)
{
super(master);
}

/**
 * Hauptmethode der Stelle. Hier werden entsprechend dem Modell
 * Berechnungsfäden (Agenten) gestartet.
 */
public void run()
```

```
{
this.inuse = 0;

//starte startAgenten
///for ( int i = 0 ; i < this.max ; i++ )
//{
// this.inuse = i;
// this.startAgent(new NachrichtenStartAgent(this,this.agents));
//}

while(this.alife)
{
// starte tote Agenten
if(this.inuse < this.max)
{
this.incInuse();
this.startAgent(new NachrichtenStartAgent(this,this.agents));
}
// N_received
if(!this.n_received.isEmpty())
   this.startAgent(new NachrichtenClientRespondAgent(this,this.agents));
// N_tosend
if(!this.n_tosend.isEmpty())
   this.startAgent(new NachrichtenSendAgent(this,this.agents));
// N_incomplete
if(!this.n_incomplete.isEmpty())
   this.startAgent(new NachrichtenCheckAgent(this,this.agents));
// P_received
if(!this.p_receiv.isEmpty())
   this.startAgent(new DatenpacketReceiveAgent(this,this.agents));
// P_tosend
if(!this.p_tosend.isEmpty())
   this.startAgent(new DatenpacketSendAgent(this,this.agents));
try
{
 this.sleep( 1 );
}
catch (Exception e)
{ Debug.print(25,"ClientStelle","run","wait failed " + e); }
}
}
}
```

A.3.3. Agent

```
/**
 * Diese Klasse fungiert als Oberklasse aller in der Simulationsplatform
 * vorkommenden "Agenten". ( Eine Ausnahme bilden Agenten die direkten Zugriff auf
 * das Netzwerk haben.)
 * @author xelahr
 * @version 0.3
 */
```

```
class Agent extends Thread
{
/**
 * Stelle an die der Agent gebunden ist.
 */
protected Stelle master = null;

/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle.
 * @param master Stelle an die der Agent gebunden wird.
 */
Agent(Stelle master)
{
this.master = master;
}

/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle,
 * und integriert ihn in eine ThreadGroup, aus welcher auf den Agenten
 * zugegriffen werden kann.
 * @param master Stelle an die der Agent gebunden wird.
 * @param agents ThreadGroup in die der Agent integriert wird.
 */
Agent(Stelle master, ThreadGroup agents)
{
super(agents, "simAgent");
this.master = master;
}

/**
 * Diese Methode ist Bestandteil des Runnable Interfaces.
 * @see java.lang.Runnable
 */
public void run()
{
Debug.print(10,"Agent","run","not implemented");
}
}
```

A.3.4. DatenpacketReceiveAgent

```
/**
 * Diese Klasse impementiert einen Berechnungsfaden der Stellen:
 * Stelle und ClientStelle. Dieser Agent entnimmt der Liste P_received der Stelle
 * an die er gebunden ist ein Datenpacket D und integriert dieses in die
 * zugehörige Nachricht in der Liste N_incomplete.
 * @author xelahr
 * @version 0.3
 */
class DatenpacketReceiveAgent extends Agent
{
/**
```

```
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle,
 * und integriert ihn in eine ThreadGroup, aus welcher auf den Agenten
 * zugegriffen werden kann.
 * @param master Stelle an die der Agent gebunden wird.
 * @param agents ThreadGroup in die der Agent integriert wird.
 */
DatenpacketReceiveAgent(Stelle master,ThreadGroup agents)
{
this.super(master,agents);
}

/**
 * Diese Methode ist Bestandteil des Runnable Interfaces.
 * @see java.lang.Runnable
 */
public void run()
{
try
{
Datenpacket pkg = (Datenpacket)master.getFromP_receiv();
master.addIncomplete(pkg);
}
catch (Exception e)
{
Debug.print(95,"DatenpacketReceiveAgent","run","failed"+e);
}
}
}
```

A.3.5. DatenpacketSendAgent

```
/**
 * Diese Klasse impementiert einen Berechnungsfaden, der ein Datenpacket aus der
 * Liste P_tosend der zugehörigen Stelle entnimmt und dieses zum versenden über
 * die
 * Stelle an die zugehörige Simulationsplatform übergibt.
 * @author xelahr
 * @version 0.3
 */
class DatenpacketSendAgent extends Agent
{
/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle,
 * und integriert ihn in eine ThreadGroup, aus welcher auf den Agenten
 * zugegriffen werden kann.
 * @param master Stelle an die der Agent gebunden wird.
 * @param agents ThreadGroup in die der Agent integriert wird.
 */
DatenpacketSendAgent(Stelle master,ThreadGroup agents)
{
this.super(master,agents);
}
```

```
/**
 * Diese Methode ist Bestandteil des Runnable Interfaces.
 * @see java.lang.Runnable
 */
public void run()
{
Object tmp = master.getFromP_tosend();
master.sendToSim(tmp);
}
}
```

A.3.6. NachrichtenCheckAgent

```
/**
 * Diese Klasse impementiert einen Berechnungsfaden, der in der Liste
 * N_incomplete nach Nachrichten sucht, die bereits vollständig sind,
 * und diese
 * Nachrichten dann in die Liste N_received einträgt.
 * @author xelahr
 * @version 0.3
 */
class NachrichtenCheckAgent extends Agent
{
/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle,
 * und integriert ihn in eine ThreadGroup, aus welcher auf den Agenten
 * zugegriffen werden kann.
 * @param master Stelle an die der Agent gebunden wird.
 * @param agents ThreadGroup in die der Agent integriert wird.
 */
NachrichtenCheckAgent(Stelle master, ThreadGroup agents)
{
this.super(master,agents);
}

/**
 * Diese Methode ist Bestandteil des Runnable Interfaces.
 * @see java.lang.Runnable
 */
public void run()
{
Nachricht[] complete = master.getComplete();
if ( complete != null )
for ( int i = 0 ; i < complete.length ; i++ )
master.addReceived(complete[i]);
}
}
```

A.3.7. NachrichtenClientRespondAgent

```
import java.util.Date;
```

```
/**
 * Diese Klasse impementiert einen Berechnungsfaden, der an eine Stelle der Klasse
 * ClientStelle gebunden wird. Dieser Agent entnimmt der Liste N_received eine
 * Nachricht N_r.
 *
 * Falls N_r den Typ "respond" hat, wird der Zeitpunkt der Entnahme
 * dieser Nachricht aus der Liste zusammen mit der Bezeichnung derselben und dem
 * Absendezeitpunkt der Ursprungsnachricht protokolliert.
 *
 * Mittels der Funkion f_serv der Nachricht N_r wird ein Zeitraum ermittelt, welcher
 * zur Simulation der Bearbeitungszeit dieser Nachricht gebraucht wird. Für die
 * Dauer dieses Zeitraumes wird dieser Berechnungsfaden "schlafen" gelegt.
 *
 * Nach der Simulation der Bearbeitung wird eine Antwortnachricht mittels der
 * Funktion f_reply bestimmt. Diese wird nach dem im Modell angegebenen Muster
 * bearbeitet und schliesslich in die Liste N_tosend eingetragen.
 *
 * @author xelahr
 * @version 0.3
 */
class NachrichtenClientRespondAgent extends Agent
{
/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle,
 * und integriert ihn in eine ThreadGroup, aus welcher auf den Agenten
 * zugegriffen werden kann.
 * @param master Stelle an die der Agent gebunden wird.
 * @param agents ThreadGroup in die der Agent integriert wird.
 */
NachrichtenClientRespondAgent(ClientStelle master, ThreadGroup agents)
{
this.super(master,agents);
}

/**
 * Diese Methode ist Bestandteil des Runnable Interfaces.
 * @see java.lang.Runnable
 */
public void run()
{
// besorge eine nachricht
Nachricht received = master.getReceived();
if ( received == null ) return;
Date recTime = new Date(System.currentTimeMillis());

// protokoll schreiben
if (received.isRespond())
{
master.addToProtokoll(received.getIdName(), received.getTime(), recTime);
}

// simuliere bearbeitungszeit
```

```
int serv = received.getValueF_serv();
try
{
this.sleep(serv);
}
catch(Exception e)
{
Debug.print(10,"NachrichtenClientRespondAgent","run","sleep failed");
}

// bestimme die Antwortnachricht
String nameOfRespond = received.getValueF_reply();
Nachricht respond = master.getNachrichtByName(nameOfRespond);

// id erzeugen
respond.setId(respond.getIdName(),received.getIdOrt(),received.getIdHash());

// absender und empfänger eintragen
respond.setAbsender(master.getAddresse());
if ( respond.getTypeOfZ().compareTo(received.getTypeOfA()) != 0 )
Debug.print(10,"NachrichtenClientRespondAgent","run",
              "type of Ziel is false: "+respond.getTypeOfZ());
respond.setZielort(received.getAbsenderName(),received.getAbsenderAddr());

if (received.isRequest()) respond.setRespond();
else if (received.isRespond()) respond.setRequest();

if (respond.isRequest()) respond.setCurrTime();
else if (respond.isRespond()) respond.setTime(received.getTime());

// prüfung ob leere nachricht.
if ( respond.getIdName().compareTo("---END---") != 0 )
master.addTosend(respond);
else
((ClientStelle)master).decInuse();
}
}
```

A.3.8. NachrichtenRespondAgent

```
import java.util.Date;
/**
 * Diese Klasse impementiert einen Berechnungsfaden, der an eine Stelle der Klasse
 * Stelle gebunden wird. Dieser Agent entnimmt der Liste N_received eine
 * Nachricht N_r.
 *
 * Falls N_r den Typ "respond" hat, wird der Zeitpunkt der Entnahme
 * dieser Nachricht aus der Liste zusammen mit der Bezeichnung derselben und dem
 * Absendezeitpunkt der Ursprungsnachricht protokolliert.
 *
 * Mittels der Funkion f_serv der Nachricht N_r wird ein Zeitraum ermittelt, welcher
 * zur Simulation der Bearbeitungszeit dieser Nachricht gebraucht wird. Für die
```

```
 * Dauer dieses Zeitraumes wird dieser Berechnungsfaden "schlafen" gelegt.
 *
 * Nach der Simulation der Bearbeitung wird eine Antwortnachricht mittels der
 * Funktion f_reply bestimmt. Diese wird nach dem im Modell angegebenen Muster
 * bearbeitet und schliesslich in die Liste N_tosend eingetragen.
 *
 * BEM: Im Originalmodell ist die Protokollierung der Systemantwortzeiten auf einem
 * Server nicht vorgesehen.
 *
 * @author xelahr
 * @version 0.3
 */
class NachrichtenRespondAgent extends Agent
{
/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle,
 * und integriert ihn in eine ThreadGroup, aus welcher auf den Agenten
 * zugegriffen werden kann.
 * @param master Stelle an die der Agent gebunden wird.
 * @param agents ThreadGroup in die der Agent integriert wird.
 */
NachrichtenRespondAgent(Stelle master,ThreadGroup agents)
{
this.super(master,agents);
}

/**
 * Diese Methode ist Bestandteil des Runnable Interfaces.
 * @see java.lang.Runnable
 */
public void run()
{
// besorge eine nachricht
Nachricht received = master.getReceived();
if ( received == null ) return;

Date recTime = new Date(System.currentTimeMillis());

// protokoll schreiben
if (received.isRespond())
{
master.addToProtokoll(received.getIdName(), received.getTime(), recTime);
}

// simuliere bearbeitungszeit
int serv = received.getValueF_serv();
try
{
this.sleep(serv);
}
catch(Exception e)
{
```

```
Debug.print(10,"NachrichtenRespondAgent","run","sleep failed");
}
// bestimme die Antwortnachricht
String nameOfRespond = received.getValueF_reply();
Nachricht respond = master.getNachrichtByName(nameOfRespond);
// id erzeugen
respond.setId(respond.getIdName(),received.getIdOrt(),received.getIdHash());
// absender und empfänger eintragen
respond.setAbsender(master.getIdName());
if ( respond.getTypeOfZ().compareTo(received.getTypeOfA()) != 0 )
Debug.print(10,"NachrichtenRespondAgent","run","type of Ziel is false");
respond.setZielort(received.getAbsenderName(),received.getAbsenderAddr());

if (received.isRequest()) respond.setRespond();
else if (received.isRespond()) respond.setRequest();

if (respond.isRequest()) respond.setCurrTime();
else if (respond.isRespond()) respond.setTime(received.getTime());

master.addTosend(respond);
}
}
```

A.3.9. NachrichtenSendAgent

```
/**
 * Diese Klasse impementiert einen Berechnungsfaden, der eine Nachricht aus der
 * Liste N_tosend entnimmt. Die Datenmenge dieser Nachricht mittels der Funktion
 * f_data ermittelt. Die Grösse der Datenpackete der Nachricht mittels der Funktion
 * f_size bestimmt. Und den Zeitraum über den die Nachricht versendet wird mittels
 * der Funktion f_time feststellt. Aus den so gewonnenen Daten wird eine Liste mit
 * Datenpacketen generiert. Diese werden dann entsprechend im Absendezeitraum in die
 * Liste P_tosend eingetragen.
 *
 * @author xelahr
 * @version 0.3
 */
class NachrichtenSendAgent extends Agent
{
/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle,
 * und integriert ihn in eine ThreadGroup, aus welcher auf den Agenten
 * zugegriffen werden kann.
 * @param master Stelle an die der Agent gebunden wird.
 * @param agents ThreadGroup in die der Agent integriert wird.
 */
NachrichtenSendAgent(Stelle master,ThreadGroup agents)
{
this.super(master,agents);
}

/**
```

```
 * Diese Methode ist Bestandteil des Runnable Interfaces.
 * @see java.lang.Runnable
 */
public void run()
{
Nachricht msg = master.getTosend();
if ( msg == null ) return;

// daten generierung
int data = msg.getValueF_data();
int size = msg.getValueF_size();
int time = msg.getValueF_time();

msg.generatePacketList(data,size);
msg.generateDelays(time);

// versenden einzelner packete
int[] delays = msg.getDelays();
try
{
for ( int i = 0 ; i < delays.length ; i++ )
{
try
{
this.sleep(delays[i]);
}
catch(Exception e)
{
Debug.print(10,"NachrichtenSendAgent","run","sleep failed");
}
master.addToP_tosend(msg.getDatenpacket(i));
}
}
catch (Exception e)
{
Debug.print(10,"NachrichtenSendAgent","run","send packets failed: "+e);
try
{
((ClientStelle)master).decInuse();
}
catch (Exception e1)
{
Debug.print(10,"NachrichtenSendAgent","run","master modification failed: "+e1);
}
}
}
}
```

A.3.10. NachrichtenStartAgent

```
import java.util.Date;
/**
```

```
 * Diese Klasse impementiert einen Berechnungsfaden, der an eine Stelle der Klasse
 * ClientStelle gebunden wird. Dieser Agent besorgt sich eine Startnachricht N_s
 * von der zugehörigen Stelle.
 *
 * Mittels der Funkion f_wait der Nachricht N_s wird ein Zeitraum ermittelt, welcher
 * zur Simulation der Wartezeit des Clients auf einen neuen User gebraucht wird.
 * Für die Dauer dieses Zeitraumes wird dieser Berechnungsfaden "schlafen" gelegt.
 *
 * Nach der Simulation der Wartezeit die Nachricht nach dem im Modell angegebenen
 * Muster bearbeitet und schliesslich in die Liste N_tosend eingetragen.
 *
 * @author xelahr
 * @version 0.3
 */
class NachrichtenStartAgent extends Agent
{
/**
 * Konstruirt einen Agenten und bindet ihn an die übergebene Stelle,
 * und integriert ihn in eine ThreadGroup, aus welcher auf den Agenten
 * zugegriffen werden kann.
 * @param master Stelle an die der Agent gebunden wird.
 * @param agents ThreadGroup in die der Agent integriert wird.
 */
NachrichtenStartAgent(ClientStelle master, ThreadGroup agents)
{
this.super(master,agents);
}

/**
 * Diese Methode ist Bestandteil des Runnable Interfaces.
 * @see java.lang.Runnable
 */
public void run()
{
// besorge eine nachricht
Nachricht start = ((ClientStelle)master).getStartNachricht();
if ( start == null ) return;

// warte die waittime
int wait = start.getValueF_wait();
try
{
this.sleep(wait);
}
catch(Exception e)
{
Debug.print(10,"NachrichtenStartAgent","run","sleep failed");
}
// id erzeugen
start.setId(start.getIdName(),master.getIdOrt(),master.getHash());

// typ setzen
```

```
start.setRequest();

// zeitstempel setzen
start.setTime(new Date(System.currentTimeMillis()));

// absender und empfänger eintragen
start.setAbsender(master.getAddresse());
start.setZielort(((ClientStelle)master).getServerAddresse(start.getTypeOfZ()));

// nachricht abschicken
master.addTosend(start);
}
}
```

A.3.11. Protokoll

```
import java.util.Vector;
import java.util.Iterator;
import java.util.Date;
import java.net.InetAddress;
/**
 * Diese Klasse impementiert eine Protokollmitschrift einer Simulation, wobei die
 * Anfrageart, der Absendezeitpunkt einer Nachricht und die Ankunftzeit der Antwort
 * mitgeschrieben werden.
 *
 * @author xelahr
 * @version 0.3
 */
class Protokoll
{
/**
 * Internet Adresse der Maschine auf der sich die Stelle befindet.
 */
private InetAddress addr = null;
/**
 * Name der Stelle, die das Protokoll schreibt.
 */
private String clientName = null;
/**
 * Liste mit Anfragearten.
 */
private Vector anfrageart = null;
/**
 * Liste mit Absendezeitpunkten
 */
private Vector absendezeit = null;
/**
 *  Liste mit Ankunftzeiten
 */
private Vector ankunftzeit = null;

/**
```

```
 * Erzeugt ein neues Protokoll.
 * @param name Name der Stelle, die das Protokoll schreibt.
 * @param addr Internet Adresse der Maschine auf der sich die Stelle befindet.
 */
Protokoll(String name,InetAddress addr)
{
this.clientName = name;
this.addr = addr;
this.anfrageart = new Vector();
this.absendezeit = new Vector();
this.ankunftzeit = new Vector();
}

/**
 * Erzeugt ein neues Protokoll.
 */
Protokoll()
{
this.anfrageart = new Vector();
this.absendezeit = new Vector();
this.ankunftzeit = new Vector();
}

/**
 * Setzt eine neue Internet Adresse
 * @param addr Internet Adresse einer Maschine
 */
void setAddr(InetAddress addr) { this.addr = addr; }
/**
 * Setzt einen neuen Stellenamen
 * @param name Name einer Stelle.
 */
void setClientName(String name) { this.clientName = name; }

/**
 * Fügt einen neuen Eintrag ins Protokoll ein.
 * @param art Bezeichnung der Anfrageart.
 * @param abs Absendezeitpunkt der Anfrage.
 * @param ank Ankunftzeitpunkt der Antwort.
 */
void addDate(String art, Date abs, Date ank)
{
synchronized(this)
{
this.anfrageart.add(art);
this.absendezeit.add(abs);
this.ankunftzeit.add(ank);
}
}

/**
 * Gibt das Protokoll als String zurück.
```

```
 * @return Stringrepräsentation des Protokolls.
 */
public String toString()
{
String buff = new String();
synchronized(this)
{
try
{
try { buff += "\n   Protokoll for: "+clientName+" at( "+addr.toString()+" )\n"; }
catch (Exception e) { buff += "\n   Protokoll for unknown SimPlatform \n"; }
Iterator iter1 = this.anfrageart.iterator();
Iterator iter2 = this.absendezeit.iterator();
Iterator iter3 = this.ankunftzeit.iterator();
while( iter1.hasNext() && iter2.hasNext() && iter3.hasNext() )
{
String art = (String)iter1.next();
Date abs = (Date)iter2.next();
Date ank = (Date)iter3.next();
buff += abs.toString();
buff += " ( " + ( ank.getTime() - abs.getTime() ) + " )";
buff += " - "+ art +"\n";
}
}
catch ( Exception e )
{
Debug.print(10,"Protokoll","toString","failed " + e);
}
}
return buff;
}

/**
 * Gibt das Protokoll in eine Datei aus.
 * @param Dateiname
 */
public void toFile(String filename)
{
Debug.print(10,"Protokoll","toFile","not implemented");
}
}
```

A.3.12. SynObjCont

```
import java.util.Vector;
/**
 * Diese Klasse impementiert einen Container, der synchronisierten
 * Zugriff auf
 * seinen Inhalt sicherstellt.
 *
 * @author xelahr
 * @version 0.3
```

```
 */
class SynObjCont
{
/**
 * gekapselter Container.
 */
private Vector buffer = null;

/**
 * Erzeugt einen neuen Container.
 */
SynObjCont()
{
this.buffer = new Vector();
}

/**
 * Check ob der Container leer ist.
 * @return "true" wenn der Container leer ist und "false" sonst
 */
boolean isEmpty()
{
boolean tmp = true;
synchronized(this.buffer)
{
if (this.buffer.isEmpty()) tmp = true;
else  tmp = false;
}
return tmp;
}

/**
 * Fügt ein Object in den Container ein.
 * @param obj Object welches eingefügt werden soll.
 */
void addObject(Object obj)
{
synchronized(this.buffer)
{
this.buffer.add(obj);
}
}

/**
 * Gibt das älteste Element aus dem Container zurück, und entfernt es dabei aus
 * dem Container.
 * @return Das älteste Element im Container.
 */
Object getNextObject()
{
Object tmp = null;
synchronized(this.buffer)
```

```
{
if (!this.buffer.isEmpty())
{
tmp = this.buffer.firstElement();
this.buffer.removeElementAt(0);
}
}
return tmp;
}

/**
 * Gibt alle Elemente im Container im Array angeordnet zurück, und löscht sie
 * dabei aus dem Container.
 * @return "true" wenn der Container leer ist und "false" sonst
 */
Object[] getAllObjects()
{
Object[] buff = null;
synchronized(this.buffer)
{
 buff = this.buffer.toArray();
 this.buffer = new Vector();
}
return buff;
}
}
```

A.4. Dynamische Entitäten

A.4.1. Nachricht

```
class Nachricht implements Serializable
{
/**
 * Attribut N_id der Entität Nachricht
 */
private N_id id = null;

/**
 * Attribut N_type der Entität Nachricht:
 * 0 - unknown
 * 1 - request
 * 2 - respond
 */
private byte type = 1;// => N_id

/**
 * Attribut N_time der Entität Nachricht
 */
private Date time = null;

/**
```

```
 * Attribut D der Entität Nachricht
 * ( wird in der Simulation nicht verwendet )
 */
private int dataLength = 0;

/**
 * Teil des Attributs A der Entität Nachricht
 */
private Ort absender = null;
/**
 * Teil des Attributs A der Entität Nachricht
 */
private String typeOfA = null;

/**
 * Teil des Attributs Z der Entität Nachricht
 */
private Ort zielort = null;
/**
 * Teil des Attributs Z der Entität Nachricht
 */
private String typeOfZ = null;

/**
 * Attribut N_packets der Implementierung der Entität Nachricht
 */
private ListOfDatapackets packets = null;

/**
 * Implementierung der Funktion f_data
 */
private PosFunction f_data = null;

/**
 * Implementierung der Funktion f_size
 */
private PosFunction f_size = null;

/**
 * Implementierung der Funktion f_time
 */
private PosFunction f_time = null;

/**
 * Implementierung der Funktion f_serv
 */
private PosFunction f_serv = null;

/**
 * Implementierung der Funktion f_wait
 */
private PosFunction f_wait = null;
```

```
/**
 * Implementierung der Funktion f_reply
 */
private PosFunction f_reply = null;

// Datenpacket Operationen
/**
 * Integriert das empfangene Datenpacket.
 * @param pkg Ein Datenpacket.
 */
void receiveDatenpacket(Datenpacket pkg)
{
if ( this.packets == null )
this.packets = new ListOfDatapackets(this);
this.packets.receiveDatenpacket(pkg);
}

/**
 * Gibt das Datenpacket mit der entsprechenden Nummer zurück.
 * @param nr Nummer des Datenpacketes.
 * @return Ein Datenpacket.
 */
Datenpacket getDatenpacket(int nr)
{
return this.packets.getDatenpacket(nr);
}

/**
 * Check ob die Nachricht Komplett ist.
 * @return "true" falls Nachricht komplett, "false" sonst.
 */
boolean isComplete()
{
if ( this.packets == null )
this.packets = new ListOfDatapackets(this);
return this.packets.isComplete();
}

/**
 * Generiert eine Packetliste entsprechend den Parametern.
 * @param dataSize Gesamtmenge der Daten (in Bytes).
 * @param packetSize Grösse der Datenpackete.
 */
void generatePacketList(int dataSize, int packetSize)
{
this.packets = new ListOfDatapackets(this);
this.packets.generateList(dataSize,packetSize);
}

/**
 * Generiert Absendeverzögeungen der einzelner Datenpackete.
```

```
 * @param time Zeitintrvall in dem die Nachricht gesendet wird.
 */
void generateDelays(int time)
{
this.packets.generateDelays(time);
}

/**
 * Gibt das Array mit den Absendeverzögeungen der Datenpackete zurück.
 * @return Das Array mit den Absendeverzögeungen.
 */
int[] getDelays()
{
return this.packets.getDelays();
}

// clone Grundoperationen
/**
 * Methode zum Kopieren einer Nachricht.
 * @return Die kopierte Nachricht.
 */
protected Object clone()
{
Nachricht tmp = new Nachricht();
// Id
tmp.id = new N_id();
tmp.id.name = this.getIdName();
// Absenderart
tmp.typeOfA = this.getTypeOfA();
// Zielortart
tmp.typeOfZ = this.getTypeOfZ();
// funktionen
tmp.f_data = this.getF_data();
tmp.f_size = this.getF_size();
tmp.f_time = this.getF_time();
tmp.f_serv = this.getF_serv();
tmp.f_wait = this.getF_wait();
tmp.f_reply = this.getF_reply();

return tmp;
}

// function Grundoperationen
/**
 * Gibt die Funktion f_data zurück.
 * @return Die Funktion f_data.
 */
PosFunction getF_data() { return this.f_data; }
/**
 * Gibt die Funktion f_size zurück.
 * @return Die Funktion f_size.
 */
```

```
PosFunction getF_size() { return this.f_size; }
/**
 * Gibt die Funktion f_time zurück.
 * @return Die Funktion f_time.
 */
PosFunction getF_time() { return this.f_time; }
/**
 * Gibt die Funktion f_serv zurück.
 * @return Die Funktion f_serv.
 */
PosFunction getF_serv() { return this.f_serv; }
/**
 * Gibt die Funktion f_wait zurück.
 * @return Die Funktion f_wait.
 */
PosFunction getF_wait() { return this.f_wait; }
/**
 * Gibt die Funktion f_reply zurück.
 * @return Die Funktion f_reply.
 */
PosFunction getF_reply() { return this.f_reply; }
/**
 * Gibt einen Wert entsprechend der Dichteverteilung der
 * Funktion f_data zurück.
 * @return Ein Funktionswert.
 */
int getValueF_data()
{
if (this.f_data == null || this.f_data.getProbValue() == null) return 0;
return Integer.parseInt((String)this.f_data.getProbValue());
}
/**
 * Gibt einen Wert entsprechend der Dichteverteilung der
 * Funktion f_size zurück.
 * @return Ein Funktionswert.
 */
int getValueF_size()
{
if (this.f_size == null || this.f_size.getProbValue() == null) return 0;
return Integer.parseInt((String)this.f_size.getProbValue());
}
/**
 * Gibt einen Wert entsprechend der Dichteverteilung der
 * Funktion f_time zurück.
 * @return Ein Funktionswert.
 */
int getValueF_time()
{
if (this.f_time == null || this.f_time.getProbValue() == null) return 0;
return Integer.parseInt((String)this.f_time.getProbValue());
}
/**
```

```
 * Gibt einen Wert entsprechend der Dichteverteilung der
 * Funktion f_serv zurück.
 * @return Ein Funktionswert.
 */
int getValueF_serv()
{
if (this.f_serv == null || this.f_serv.getProbValue() == null) return 0;
return Integer.parseInt((String)this.f_serv.getProbValue());
}
/**
 * Gibt einen Wert entsprechend der Dichteverteilung der
 * Funktion f_wait zurück.
 * @return Ein Funktionswert.
 */
int getValueF_wait()
{
if (this.f_wait == null || this.f_wait.getProbValue() == null) return 0;
return Integer.parseInt((String)this.f_wait.getProbValue());
}
/**
 * Gibt einen Wert entsprechend der Dichteverteilung der
 * Funktion f_reply zurück.
 * @return Ein Funktionswert.
 */
String getValueF_reply()
{
if (this.f_reply == null || this.f_reply.getProbValue() == null) return "";
return (String)this.f_reply.getProbValue();
}

// parse Grundoperationen
private static String  cutBuffer = new String();
private static String restBuffer = new String();
private static String funcBuffer = new String();
/* Bsp:
Identifikation : Cashautomat -> Authentisierungsserver
f_size: [ (12.3 1024) (67.7 2048) (20 4096) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (25.0 10000) (25.0 300000) (25.0 600000) (25.0 1800000)]
f_reply: [ (1.0 Autorisierung) ]
*/
/**
 * Methode zum Parsen einer Nachricht
 * @param config Ein Konfigurationsstring für eine Nachricht.
 * @return Die geparste Nachricht aus dem Konfigurationsstring.
 */
static Nachricht parseNachricht(String config)
{
Nachricht tmpNachricht = new Nachricht();
StringTokenizer str = new StringTokenizer(config);
String name = str.nextToken();
```

```
if ( (str.nextToken()).compareTo(":") != 0 )
Debug.print(10,"Nachricht","parseNachricht","Syntax error");
String from = str.nextToken();
if ( (str.nextToken()).compareTo("->") != 0 )
Debug.print(10,"Nachricht","parseNachricht","Syntax error");
String to = str.nextToken();
tmpNachricht.id = new N_id();
tmpNachricht.id.name = name;
tmpNachricht.setTypeOfA(from);
tmpNachricht.setTypeOfZ(to);

// parse functions
tmpNachricht.cutLine(config);
config = tmpNachricht.restBuffer.substring(0);
tmpNachricht.cutLine(config);
config = tmpNachricht.restBuffer.substring(0);
String[] line = new String[6];
for ( int i = 0 ; i < line.length ; i++ )
{
if ( Nachricht.cutLine(config) )
{
line[i] = tmpNachricht.cutBuffer.substring(0);
config = tmpNachricht.restBuffer.substring(0);
}
else Debug.print(10,"Nachricht","parseNachricht",
                 "Syntax error line "+i+" -> "+config);
}

// f_data
for ( int i = 0 ; i < line.length ; i++ )
{
str = new StringTokenizer(line[i]);
if ( (str.nextToken()).compareTo("f_data") == 0 )
tmpNachricht.f_data = PosFunction.parsePosFunction(line[i]);
}
// f_size
for ( int i = 0 ; i < line.length ; i++ )
{
str = new StringTokenizer(line[i]);
if ( (str.nextToken()).compareTo("f_size") == 0 )
tmpNachricht.f_size = PosFunction.parsePosFunction(line[i]);
}
// f_time
for ( int i = 0 ; i < line.length ; i++ )
{
str = new StringTokenizer(line[i]);
if ( (str.nextToken()).compareTo("f_time") == 0 )
tmpNachricht.f_time = PosFunction.parsePosFunction(line[i]);
}
// f_serv
for ( int i = 0 ; i < line.length ; i++ )
{
```

```
str = new StringTokenizer(line[i]);
if ( (str.nextToken()).compareTo("f_serv") == 0 )
tmpNachricht.f_serv = PosFunction.parsePosFunction(line[i]);
}
// f_wait
for ( int i = 0 ; i < line.length ; i++ )
{
str = new StringTokenizer(line[i]);
if ( (str.nextToken()).compareTo("f_wait") == 0 )
tmpNachricht.f_wait = PosFunction.parsePosFunction(line[i]);
}
// f_reply
for ( int i = 0 ; i < line.length ; i++ )
{
str = new StringTokenizer(line[i]);
if ( (str.nextToken()).compareTo("f_reply") == 0 )
tmpNachricht.f_reply = PosFunction.parsePosFunction(line[i]);
}

return tmpNachricht;
}

private static boolean cutLine(String original)
{
cutBuffer = new String();
restBuffer = new String();
int startIndex = 0;
int endIndex = 0;
endIndex = original.indexOf("\n");
if ( startIndex < 0 || endIndex < 0 ) return false;
cutBuffer += original.substring(startIndex,endIndex);
restBuffer += original.substring(endIndex+1,original.length());
return true;
}



private static boolean cutNextPosPoint(String original, String ch1, String ch2)
{
cutBuffer = new String();
restBuffer = new String();
int startIndex = 0;
int endIndex = 0;

startIndex = original.indexOf(ch1);
endIndex = original.indexOf(ch2);
if ( startIndex < 0 || endIndex < 0  || startIndex == endIndex) return false;

cutBuffer += original.substring(startIndex+1,endIndex);
restBuffer += original.substring(0,startIndex)
           + original.substring(endIndex+1,original.length());
return true;
```

```
}

// time Grundoperationen
/**
 * Gibt den Zeitstempel zurück..
 * @return Der Zeitstempel der Nachricht.
 */
Date getTime() { return this.id.time; }
/**
 * Setzt den Zeitstempel.
 * @param time Der Zeitstempel der Nachricht.
 */
void setTime(Date time) { this.id.time = time; this.time = time; }
/**
 * Setzt den Zeitstempel auf die Aktuelle Zeit.
 */
void setCurrTime()
{ this.setTime(new Date(System.currentTimeMillis())); }

// type Grundoperationen
/**
 * Gibt den Typ dieser Nachricht zurück.
 * @return Der Typ dieser Nachricht.
 */
byte getType() { return this.id.type; }
/**
 * Setzt den Typ dieser Nachricht.
 * @param type Der Typ dieser Nachricht.
 */
void setType(byte type) { this.id.type = type; this.type = type; }
/**
 * Setzt den Typ dieser Nachricht auf "request".
 */
void setRequest() { this.setType((byte)1); }
/**
 * Setzt den Typ dieser Nachricht auf "respond".
 */
void setRespond() { this.setType((byte)2); }
/**
 * Check ob der Typ dieser Nachricht "request" ist.
 * @return "true" falls der Typ "request", "false" sonst.
 */
boolean isRequest()
{ if (this.getType() == 1) return true; else return false; }
/**
 * Check ob der Typ dieser Nachricht "respond" ist.
 * @return "true" falls der Typ "respond", "false" sonst.
 */
boolean isRespond()
{ if (this.getType() == 2) return true; else return false; }

// Id Grundoperationen
```

```
/**
 * Gibt die Bezeichnung dieser Nachricht zurück.
 * @return Die Bezeichnung dieser Nachricht.
 */
String getIdName() { return this.id.name; }
/**
 * Gibt die Ortsangabe dieser Nachricht zurück.
 * @return  Die Ortsangabe dieser Nachricht.
 */
String getIdOrt() { return this.id.ort; }
/**
 * Gibt den HashCode dieser Nachricht zurück.
 * @return  Der HashCode dieser Nachricht.
 */
int getIdHash() { return this.id.hash; }
/**
 * Gibt die Id dieser Nachricht zurück.
 * @return  Die Id dieser Nachricht.
 */
N_id getId() { return this.id; }
/**
 * Setzt die Id dieser Nachricht.
 * @param id  Die Id dieser Nachricht.
 */
void setId(N_id id) { this.id = id; }
/**
 * Setzt die Id dieser Nachricht.
 * @param name Die Bezeichnung dieser Nachricht.
 * @param ort  Die Ortsangabe dieser Nachricht.
 * @param hash Der HashCode dieser Nachricht.
 */
void setId(String name, String ort, int hash)
{
this.id = new N_id();
this.id.name = name;
this.id.ort = ort;
this.id.hash = hash;
}


// data Grundoperationen
/**
 * Gibt das dataLength Feld des Datenpackets zurück.
 * @return Das dataLength-Feld.
 */
int getDataLength() { return this.dataLength; }
/**
 * Setzt das dataLength Feld des Datenpackets.
 * @param newDataLength Das neue dataLength-Feld.
 */
void setDataLength(int newDataLength) { this.dataLength = newDataLength; }
```

```
// Serializable functions
private void writeObject(java.io.ObjectOutputStream out) throws IOException
{
out.writeInt(this.dataLength);
out.writeObject(this.id);
out.flush();
}
private void readObject(java.io.ObjectInputStream in)
                                  throws IOException, ClassNotFoundException
{
this.dataLength = in.readInt();
this.id = (N_id)in.readObject();
}

// Ort Grundoperationen
/**
 * Gibt den Typ des Absenders der Nachricht zurück.
 * @return Der Typ des Absenders der Nachricht.
 */
String getTypeOfA() { return this.typeOfA; }
/**
 * Gibt den Typ des Zielortes der Nachricht zurück.
 * @return Der Typ des Zielortes der Nachricht.
 */
String getTypeOfZ() { return this.typeOfZ; }
/**
 * Setzt den Typ des Absenders der Nachricht.
 * @param type Der Typ des Absenders der Nachricht.
 */
void setTypeOfA(String type) { this.typeOfA = type; }
/**
 * Setzt den Typ des Zielortes der Nachricht.
 * @param type Der Typ des Zielortes der Nachricht.
 */
void setTypeOfZ(String type) { this.typeOfZ = type; }
/**
 * Gibt den Absender der Nachricht zurück.
 * @return Der Absender der Nachricht.
 */
Ort getAbsender() { return this.absender; }
/**
 * Gibt den Zielort der Nachricht zurück.
 * @return Der Zielort der Nachricht.
 */
Ort getZielort() { return this.zielort; }
/**
 * Gibt den Absendernamen der Nachricht zurück.
 * @return Der Absendername der Nachricht.
 */
String getAbsenderName() { return this.absender.name; }
/**
 * Gibt den Zielortnamen der Nachricht zurück.
```

```
 * @return Der Zielortnamen der Nachricht.
 */
String getZielortName() { return this.zielort.name; }
/**
 * Gibt die Absenderaddresse der Nachricht zurück.
 * @return Die Internet Addresse des Absenders der Nachricht.
 */
InetAddress getAbsenderAddr() { return this.absender.addr; }
/**
 * Gibt die Zielortadresse der Nachricht zurück.
 * @return Die Internet Addresse des Zielortes der Nachricht.
 */
InetAddress getZielortAddr() { return this.zielort.addr; }
/**
 * Setzt den Absender der Nachricht.
 * @param ort Der Absender der Nachricht.
 */
void setAbsender(Ort addr) { this.absender = addr; }
/**
 * Setzt den Zielort der Nachricht.
 * @param ort Der Zielort der Nachricht.
 */
void setZielort(Ort addr) { this.zielort = addr; }
/**
 * Setzt den Absender der Nachricht.
 * @param ort Der Absendername der Nachricht.
 * @param adr Die Internet Addresse des Absenders der Nachricht.
 * @param port Der Port der Internet Addresse des Absenders.
 */
void setAbsender(String ort, InetAddress adr, int port)
{
this.absender = new Ort();
this.absender.name = ort;
this.absender.addr = adr;
this.absender.port = port;
}
/**
 * Setzt den Absender der Nachricht. Adresse und Port werden
 * dem ComDevice entnommen.
 * @param ort Der Absendername der Nachricht.
 */
void setAbsender(String ort)
{
this.setAbsender(ort, ComDevice.getLocalHost(), ComDevice.getSimulationPort());
}

/**
 * Setzt den Absender der Nachricht.
 * @param ort Der Absendername der Nachricht.
 * @param host Der Name der Maschine der Nachricht.
 * @param port Der Port der Internet Addresse des Absenders.
 */
```

```
void setAbsender(String ort, String host, int port)
{
this.setAbsender(ort, ComDevice.getByName(host), port) ;
}

/**
 * Setzt den Zielort der Nachricht.
 * @param ort Der Zielortname der Nachricht.
 * @param adr Die Internet Addresse des Zielortes der Nachricht.
 * @param port Der Port der Internet Addresse des Zielortes.
 */
void setZielort(String ort, InetAddress adr, int port)
{
this.zielort = new Ort();
this.zielort.name = ort;
this.zielort.addr = adr;
this.zielort.port = port;
}
/**
 * Setzt den Zielort der Nachricht. Der Port wird dem ComDevice entnommen.
 * @param ort Der Zielortname der Nachricht.
 * @param adr Die Internet Addresse des Zielortes der Nachricht.
 */
void setZielort(String ort, InetAddress adr)
{ this.setZielort(ort, adr, ComDevice.getSimulationPort()); }
/**
 * Setzt den Zielort der Nachricht. Der Port wird dem ComDevice entnommen.
 * @param ort Der Zielortname der Nachricht.
 * @param host Der Maschinenname des Zielortes der Nachricht.
 */
void setZielort(String ort, String host)
{ this.setZielort(ort, ComDevice.getByName(host)); }
}
```

A.4.2. N_id

```
import java.io.Serializable;
import java.io.IOException;
import java.util.Date;
/**
 * Diese Klasse implementiert die Id einer Nachricht
 *
 * @author xelahr
 * @version 0.3
 */
class N_id implements Serializable
{
/**
 * Bezeichnung der Nachricht.
 */
String name = null;
/**
```

```
 * Entstehungsort der Nachricht.
 */
String ort = null;
/**
 * Zeitstempel der Nachricht.
 */
Date time = null;
/**
 * Typ der Nachricht.
 */
byte type = 1;
/**
 * HashCode der Nachricht.
 */
int hash = 0;

public String toString()
{
return "N_id: "+ this.name +" t: "+ this.type +" o: "
               + this.ort +" h: "+ this.hash;
}

private void writeObject(java.io.ObjectOutputStream out) throws IOException
{
out.writeObject(this.name);
out.writeObject(this.ort);
out.writeObject(this.time);
out.writeByte(this.type);
out.writeInt(this.hash);
out.flush();
}
private void readObject(java.io.ObjectInputStream in)
                                    throws IOException, ClassNotFoundException
{
this.name = (String)in.readObject();
this.ort = (String)in.readObject();
this.time = (Date)in.readObject();
this.type = in.readByte();
this.hash = in.readInt();
}
}
```

A.4.3. PosFunction

```
import java.util.Vector;
import java.util.Iterator;
import java.io.Serializable;
import java.io.IOException;
import java.util.StringTokenizer;

/**
 * Implementierung einer Möglichkeitsfunktion
```

```
 * @author xelahr
 * @version 0.3
 */

class PosFunction implements Serializable
{
/**
 * Möglichkeitsverteilung (Dichteverteilung)
 */
private double[] possibilities = null;
/**
 * Wahrscheinlichkeitsverteilung
 */
private double[] probabilities = null;
/**
 * Vector mit möglichen Rückgabewerten
 */
private Vector returnValues = null;

/**
 * Methode zur Ausgabe der Funktion als String.
 * @return Darstellung der Funktion als String.
 */
public String toString()
{
String buffer = new String();
buffer += "[ ";
try
{
for ( int i = 0 ; i < possibilities.length ; i++ )
buffer += "( "+this.possibilities[i]+" "
       +this.returnValues.elementAt(i).toString()+" ) ";
} catch (Exception e)
{ Debug.print(10,"PosFunction","toString","no function"+e);  }
buffer += "]\n";
return buffer;
}

// Funktionen zur Bearbeitung des Rückgabevectors
/**
 * Setzt die Menge der Rückgabewerte der Funktion.
 * @param values Die Menge der Rückgabewerte der Funktion.
 */
void setReturnValues(Vector values)
{
this.returnValues = values;
}
/**
 * Gibt die Menge der Rückgabewerte der Funktion zurück.
 * @return Die Menge der Rückgabewerte der Funktion
 */
Vector getReturnValues()
```

```
{
return this.returnValues;
}
/**
 * Setzt die Test-Menge der Rückgabewerte der Funktion: {true,false}.
 */
void setTestValues()
{
this.returnValues = new Vector();
this.returnValues.add(new Boolean(true));
this.returnValues.add(new Boolean(false));
}

// Funktionen zur Bearbeitung der Möglichkeitsverteilung
/**
 * Setzt die Dichteverteilung der Funktion.
 * @param poss Die Dichteverteilung der Funktion.
 */
void setPossibilities(double[] poss)
{
this.possibilities = poss;
}
/**
 * Gibt die Dichteverteilung der Funktion zurück.
 * @return Die Dichteverteilung der Funktion.
 */
double[] getPossibilities()
{
return this.possibilities;
}
/**
 * Setzt die Dichteverteilung auf die Gleichverteilung der Möglichkeiten.
 */
void setEqualPoss()
{
if (this.returnValues == null)
{
Debug.print(10,"PosFunction","setEqualPoss","# of returnValues is null");
}
else
{
this.possibilities = new double[this.returnValues.size()];
for( int i = 0 ; i < this.possibilities.length ; i++ )
this.possibilities[i] = 1.0/this.possibilities.length;
}
}

// Funktionen zur Bearbeitung der Wahrscheinlichkeitsverteilung
/**
 * Setzt die Wahrscheinlichkeitsverteilung der Funktion.
 * @param poss Die Wahrscheinlichkeitsverteilung der Funktion.
 */
```

```
private void setProbabilities(double[] prob)
{
this.probabilities = prob;
}
/**
 * Gibt die Wahrscheinlichkeitsverteilung der Funktion zurück.
 * @return Die Wahrscheinlichkeitsverteilung der Funktion.
 */
private double[] getProbabilities()
{
return this.probabilities;
}
/**
 * Rechnet die Dichteverteilung in eine Wahrscheinlichkeitsverteilung um.
 */
private void possToProb()
{
if (this.possibilities==null)
{
Debug.print(10,"PosFunction","setEqualPoss","# of possibilities is null");
}
else
{
this.probabilities = new double[this.possibilities.length];
double sumOfPoss = 0;
//array initialisieren
for ( int i = 0 ; i < this.probabilities.length ; i++ )
{
sumOfPoss = sumOfPoss + this.possibilities[i];
this.probabilities[i] = sumOfPoss;
}
// Wht berechnen
for ( int i = 0 ; i < this.probabilities.length ; i++ )
this.probabilities[i] = this.probabilities[i]/sumOfPoss;
// sicherstellen dass trotz rundungen summe der Wht = 1
this.probabilities[this.probabilities.length - 1] = 1.0;
}
}

// Schnittstellen nach Aussen
/**
 * Diese Funktion liefert ein Object, wobei die Wahrscheinlichkeit
 * der Rückgabe dieses Objectes durch
 * diese Funktion durch die Möglichkeitsverteilung bestimmt wird.
 */
Object getProbValue()
{
Object retValue = null;
if (this.probabilities==null)
{
Debug.print(10,"PosFunction","setEqualPoss","# of probabilities is null");
return retValue;
```

```
}
double randNum = Math.random();
int i = 0;
while( randNum > this.probabilities[i] ) i++;
retValue = this.returnValues.elementAt(i);
return retValue;
}

// Parse functionality
// Bsp: f_size : [ ( 0.0 string  ) ( 0.2 20  ) ( 0.4 140.1  ) (  1.0 300 ) ]
// buffer
private static String  cutBuffer = new String();
private static String restBuffer = new String();
private static String funcBuffer = new String();

static PosFunction parsePosFunction(String dFunction)
{
PosFunction retFunction = new PosFunction();

funcBuffer = dFunction;

Vector elements = new Vector();

// wegschneiden der aeusseren klammern
cutNextPosPoint(dFunction,"[","]");
funcBuffer = new String(cutBuffer);

// speichern einzelner Punkte der Funktion
while(cutNextPosPoint(funcBuffer,"(",")"))
{
PosFunctionElement tmpEl = new PosFunctionElement();
StringTokenizer st = new StringTokenizer(cutBuffer);
tmpEl.possibility = st.nextToken();
tmpEl.valueOfPoss = st.nextToken();
elements.addElement(tmpEl);
funcBuffer = new String(restBuffer);
}

// write possibilities and functionValues
int numOfEl = elements.size();
double[] possArr = new double[numOfEl];
Vector valueVect = new Vector();
for (int i = 0 ; i < numOfEl ; i++)
{
PosFunctionElement tmpEl = (PosFunctionElement)elements.elementAt(i);
possArr[i] = Double.parseDouble(tmpEl.possibility);
valueVect.add(tmpEl.valueOfPoss);
}
retFunction.setPossibilities(possArr);
retFunction.setReturnValues(valueVect);
// compute probabilities
retFunction.possToProb();
```

```
return retFunction;
}

private static boolean cutNextPosPoint(String original, String ch1, String ch2)
{
cutBuffer = new String();
restBuffer = new String();
int startIndex = 0;
int endIndex = 0;

startIndex = original.indexOf(ch1);
endIndex = original.indexOf(ch2);
if ( startIndex < 0 || endIndex < 0 || startIndex == endIndex) return false;

cutBuffer += original.substring(startIndex+1,endIndex);
restBuffer += original.substring(0,startIndex)
           + original.substring(endIndex+1,original.length());
return true;
}

// Serializable functions
private void writeObject(java.io.ObjectOutputStream out) throws IOException
{
out.writeInt(this.possibilities.length);
for (int i=0; i < this.possibilities.length ; i++ )
   out.writeDouble(this.possibilities[i]);
out.writeInt(this.probabilities.length);
for (int i=0; i < this.probabilities.length ; i++ )
   out.writeDouble(this.probabilities[i]);
out.writeInt(this.returnValues.size());
Iterator iter = this.returnValues.iterator();
while(iter.hasNext()) out.writeObject(iter.next());
out.flush();
}
private void readObject(java.io.ObjectInputStream in)
                        throws IOException, ClassNotFoundException
{
int possLength = in.readInt();
this.possibilities = new double[possLength];
for (int i=0; i < this.possibilities.length ; i++ )
   this.possibilities[i] = in.readDouble();
int probLength = in.readInt();
this.probabilities = new double[probLength];
for (int i=0; i < this.probabilities.length ; i++ )
   this.probabilities[i] = in.readDouble();
int valLength = in.readInt();
this.returnValues = new Vector();
for (int i = 0 ; i < valLength ; i++ )
   this.returnValues.add(in.readObject());
}
```

```
// Clone function (incomplete)
protected Object clone()
{
PosFunction clone = new PosFunction();
if (this.possibilities != null)
{
double[] possClone = new double[this.possibilities.length];
for ( int i = 0 ; i < possClone.length ; i++ )
possClone[i] = this.possibilities[i];
}
if (this.probabilities != null)
{
double[] probClone = new double[this.probabilities.length];
for ( int i = 0 ; i < probClone.length ; i++ )
probClone[i] = this.probabilities[i];
}
if (this.returnValues != null)
{
Vector cloneValues = new Vector();
Iterator iter = this.returnValues.iterator();
while(iter.hasNext())
cloneValues.add(iter.next());
}
return clone;
}
}

class PosFunctionElement
{
String possibility = null;
String valueOfPoss = null;
}
```

A.4.4. ListOfDatapackets

```
import java.io.Serializable;
import java.io.IOException;
import java.util.Date;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.PipedOutputStream;
import java.io.PipedInputStream;
/**
 * Diese Klasse implementiert eine Liste von Datenpacketen.
 *
 * @author xelahr
 * @version 0.3
 */
class ListOfDatapackets
{
/**
 * Die Nachricht zu der die Liste gehört.
```

```
 */
private Nachricht master = null;
/**
 * Die Liste der Datenpackete.
 */
private Datenpacket[] packets = null;
/**
 * Die Liste mit Sendeverzögerungen der Datenpackete.
 */
private int[] sendDelays = null;
/**
 * Die Liste mit den Zuständen der Datenpackete (
 * 0 - unknown,
 * 1 - generated,
 * 2 - delay computed,
 * 3 - send,
 * 4 - not received,
 * 5 - received).
 */
private byte[] states = null;

/**
 * Liefert das Datenpacket mit dem angegebenen index.
 * @param nr Ein Index.
 * @return Das entsprechende Datenpacket.
 */
Datenpacket getDatenpacket(int nr)
{
Datenpacket tmp = null;
try
{
tmp = this.packets[nr];
}
catch (Exception e)
{
Debug.print(105,"ListOfDatapackets","getDatenpacket","failed get nr: "+nr);
}
return tmp;
}

/**
 * Methode zur Integration eines Datenpackets in die Packetliste.
 * @param pack Ein Datenpacket.
 */
void receiveDatenpacket(Datenpacket pack)
{
int nr = pack.getPackNr();
int anz = pack.getPackAnz();
if ( this.packets == null )
{
this.packets = new Datenpacket[anz];
this.states = new byte[anz];
```

```
}

this.packets[nr] = pack;
this.states[nr] = 5;
}

/**
 * Check ob die Packetliste Komplett ist.
 * @return "true" falls Komplett, "false" sonst
 */
boolean isComplete()
{
return this.checkStates(5);
}

/**
 * Check ob die ganze Packetliste einen bestimmten Status hat.
 * @param state Ein Status.
 * @return "true" falls Komplett, "false" sonst
 */
boolean checkStates(int state)
{
boolean result = true;
try
{
for (int i=0; i<this.states.length; i++)
if (this.states[i] != state) result = false;
}
catch (Exception e)
{
Debug.print(55,"ListOfDatapackets","checkStates","failed"+e);
result = false;
}
return result;
}

/**
 * Generiert eine neue Packetliste.
 * @param master Die zugehörige Nachricht.
 */
ListOfDatapackets(Nachricht master)
{
this.master = master;
}

/**
 * Liefert das Array mit den Sendeverzögerungen.
 * @return Array mit Sendeverzögerungen.
 */
int[] getDelays()
{
return sendDelays;
```

```
}

/**
 * Generiert ein Array mit Sendeverzögerungen in dem angegebenen Zeitraum.
 * Die Sendeverzögerungen sind in dem gegebenen Zeitraum gleichverteilt.
 * @param time Zeitraum für Sendeverzögerungen.
 */
void generateDelays(int time)
{
if(this.checkStates(1))
{
this.sendDelays = new int[states.length];
for ( int i = 0 ; i < this.sendDelays.length ; i++ )
this.sendDelays[i] = (int)Math.round((float)(time*Math.random()-0.5));
this.sort(this.sendDelays);
for ( int i = 0 ; i < this.sendDelays.length ; i++ )
{
int last0 = 0;
int last1 = 0;
if (this.sendDelays[i] > 0)
{
last0 = this.sendDelays[i];
this.sendDelays[i] -= last1;
last1 = last0;
}
}
}
else { Debug.print(10,"ListOfDatapackets","generateDelays",
                        "wrong state of Packets"); }
}

/**
 * Methode zum sortieren eines kleinen Arrays.
 * @param toSort Das zu sortierende Array
 * @return Das zu sortierte Array
 */
private int[] sort(int[] toSort)
{
for ( int i = 0 ; i < toSort.length ; i++ )
for ( int j = i ; j < toSort.length ; j++ )
if (toSort[i]>toSort[j])
{
int tmp = toSort[i];
toSort[i] = toSort[j];
toSort[j] = tmp;
}
return toSort;
}

/**
 * Methode zum Generierung einer Packetliste.
 * @param bytes Die Datenmenge der Nachricht.
```

```
 * @param size Die Grösse der Datenpackete.
 */
void generateList(int bytes, int size)
{
int numOfPkg = (int)Math.round(((double)bytes/(double)size)+0.4) ;
NachrichtInfos info = new NachrichtInfos();
info.type = master.getType();
info.time = master.getTime();
info.dataLength = master.getDataLength();
N_id masterId = master.getId();
Ort absender = master.getAbsender();
Ort zielort = master.getZielort();
byte[] infobytes = info.toBytes();

packets = new Datenpacket[numOfPkg];
states = new byte[numOfPkg];

for (int i = 0 ; i < packets.length ; i++ )
{
Datenpacket tmpPkg = new Datenpacket(this);
tmpPkg.setDataLength(size);
tmpPkg.setData(infobytes);
tmpPkg.setId(masterId, i , numOfPkg);
tmpPkg.setAbsender(absender);
tmpPkg.setZielort(zielort);

packets[i] = tmpPkg;
states[i] = 1;
}
}

private class NachrichtInfos implements Serializable
{
byte type = 0;
Date time = null;
int dataLength = 0;

byte[] toBytes()
{
byte[] data = null;
try{
PipedInputStream in = new PipedInputStream();
PipedOutputStream out = new PipedOutputStream(in);
ObjectOutputStream outO = new ObjectOutputStream(out);
this.writeObject(outO);
out.flush();
data = new byte[in.available()];
in.read(data, 0, data.length);
outO.close();
out.close();
in.close();
}
```

```
catch(Exception e)
{
Debug.print(10,"NachrichtInfos","toBytes",e.toString());
}
return data;
}

// Serializable functions ##############################
private void writeObject(java.io.ObjectOutputStream out) throws IOException
{
out.writeByte(this.type);
out.writeObject(this.time);
out.writeInt(this.dataLength);
out.flush();
}
private void readObject(java.io.ObjectInputStream in)
                          throws IOException, ClassNotFoundException
{
this.type = in.readByte();
this.time = (Date)in.readObject();
this.dataLength = in.readInt();
}
}

//void computeDelays(int maxDelay){}

//sendList()
//sendNext()
//sendPacket()
//receivePacket()
//isComplete()
}
```

A.4.5. Datenpacket

```
import java.io.Serializable;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.PipedOutputStream;
import java.io.PipedInputStream;
import java.io.BufferedOutputStream;
import java.io.BufferedInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.util.Date;
import java.net.InetAddress;
import java.net.DatagramPacket;
/**
 * Diese Klasse implementiert die Entität Datenpacket.
 *
 * @author xelahr
```

```
 * @version 0.3
 */
class Datenpacket extends Thread implements Serializable
{
/**
 * Die zugehörige Liste von Datenpacketen.
 */
private ListOfDatapackets masterList = null;

/**
 * Attribut D_p der Entität Datenpacket
 */
private int dataLength = 0;
/**
 * Attribut D_p der Entität Datenpacket
 */
private byte[] data = null;

/**
 * Attribut D_id der Entität Datenpacket
 */
private D_id dataPackId = null;

/**
 * Attribut A der Entität Datenpacket
 */
private Ort absender = null;

/**
 * Attribut Z der Entität Datenpacket
 */
private Ort zielort = null;

/**
 * Attribut t_a der Entität Datenpacket
 */
private Date aTime = null;

/**
 * Attribut t_z der Entität Datenpacket
 */
private Date zTime = null;

/**
 * Generiert ein leeres Datenpacket.
 */
Datenpacket(){}

/**
 * Generiert ein neues Datenpacket.
 * @param masterList Die zugehörige Liste von Datenpacketen.
 */
```

```
Datenpacket(ListOfDatapackets masterList)
{
this.setMasterList(masterList);
}
/**
 * Setzt die zugehörige Liste von Datenpacketen.
 * @param masterList Die zugehörige Liste von Datenpacketen.
 */
void setMasterList(ListOfDatapackets masterList)
{
this.masterList = masterList;
}

// DatagramPacket Operationen ##############################
private class DatenpacketStreamWriter extends Thread
{
byte[] data;
PipedOutputStream out;

DatenpacketStreamWriter(byte[] data,PipedOutputStream out)
{
this.data=data;
this.out=out;
}

public void run()
{
for( int i = 0 ; i < data.length ; i++ )
{
try
{
out.write(data[i]);
out.flush();
}
catch (Exception e)
{
i--;
}
}
}
}

private class DatenpacketStreamReader extends Thread
{
Datenpacket pack;
PipedInputStream in;
ObjectInputStream inO;

boolean done = false;

DatenpacketStreamReader(Datenpacket pack,PipedInputStream in)
{
```

```
this.pack = pack;
this.in = in;
try { this.inO = new ObjectInputStream(in); }
catch (Exception e) { Debug.print(90,"Datenpacket","reader",
                                  "inO failed: "+e); }
}

public void run()
{
try { this.pack.readObject(inO); }
catch (Exception e) { Debug.print(10,"Datenpacket","reader",
                                  "read object failed: "+e); }
this.done = true;
}
}

/**
 * Generiert ein neues Datenpacket aus dem gegebenen Datagram.
 * @param datagram Das Quell-Datagram.
 */
Datenpacket(DatagramPacket datagram)
{
if (datagram == null) return;
try{
byte[] data = datagram.getData();

ByteArrayInputStream inB = new ByteArrayInputStream(data);
ObjectInputStream inO = null;

while ( inB.available() <= 0 )
{
try
{
 this.sleep( 1000 );
} catch (Exception e) { Debug.print(5,"Datenpacket","Datenpacket",
                                    "wait failed " + e); }
}
if (inB.available() > 0)
{
inO = new ObjectInputStream(inB);
this.readObject(inO);
}
}catch(Exception e){Debug.print(110,"Datenpacket",
                                "Datenpacket",e.toString()); }
}

/**
 * Gibt das Datenpacket als ein ByteArray zurück.
 * @return Das Datenpacket als ein ByteArray.
 */
byte[] toBytes()
{
```

```
byte[] data = null;
byte[] buff = null;
try{
PipedInputStream in = new PipedInputStream();
PipedOutputStream out = new PipedOutputStream(in);
ObjectOutputStream outO = new ObjectOutputStream(out);
this.writeObject(outO);
out.flush();
buff = new byte[in.available()];
in.read(buff, 0, buff.length);
outO.close();
out.close();
in.close();
}catch(Exception e){Debug.print(10,"Datenpacket","toBytes",e.toString()); }

// + daten über die Nachricht
data = new byte[buff.length+this.data.length];
for ( int i = 0 ; i < buff.length ; i++ ) data[i] = buff[i];
for ( int i = 0 ; i < this.data.length ; i++ )
   data[i+buff.length] = this.data[i];

return data;
}

/* Datenpacket testToBytes()
{
Datenpacket pack = null;
try{
PipedInputStream in = new PipedInputStream();
PipedOutputStream out = new PipedOutputStream(in);
ObjectOutputStream outO = new ObjectOutputStream(out);
this.writeObject(outO);
out.flush();

ByteArrayInputStream inB = new ByteArrayInputStream(this.toBytes());


while ( in.available() <= 0 )
{
try
{
 this.sleep( 1000 );
}
catch (Exception e)
{
Debug.print(5,"Datenpacket","Datenpacket","wait failed " + e);
}
}
ObjectInputStream inO = new ObjectInputStream(inB);
if (in.available() > 0)
{
pack = new Datenpacket();
```

```
pack.readObject(in0);
}

out0.close();
out.close();
in.close();
in0.close();
}
catch(Exception e)
{
Debug.print(10,"Datenpacket","testToBytes",e.toString());
}

return pack;
}*/

/**
 * Gibt das Datenpacket als ein Datagram zurück.
 * @return Das Datenpacket als ein Datagram.
 */
DatagramPacket toDatagramPacket()
{
DatagramPacket tmpPacket = new DatagramPacket(new byte[this.dataLength],
                                                    this.dataLength);
tmpPacket.setAddress(this.getZielortAddr());
tmpPacket.setPort(this.getZielortPort());
byte[] data = this.toBytes();


// Länge der Packete überprüfen.
byte[] buff = null;
if ( data.length > this.dataLength )
{
Debug.print(10,"Datenpacket","toDatagramPacket","packet too long: "
                                        +data.length+" / "+this.dataLength);
tmpPacket.setData(data,0,data.length);
}
else
{
buff = this.createData(this.dataLength - data.length);
byte[] tmp = new byte[this.dataLength];
for ( int i = 0 ; i < data.length ; i++ ) tmp[i] = data[i];
for ( int i = 0 ; i < buff.length ; i++ ) tmp[i+data.length] = buff[i];
tmpPacket.setData(tmp,0,tmp.length);
}
if ( data.length > SimulationAgent.MAX_SIZE_OF_DATAGRAM )
Debug.print(10,"Datenpacket","toDatagramPacket","packet too long: "
+data.length+" max receiving: "+SimulationAgent.MAX_SIZE_OF_DATAGRAM);
return tmpPacket;
}

// data Operationen ##############################
```

```
/**
 * Setzt das data Feld des Datenpackets.
 * @param data Das neue data-Feld.
 */
void setData(byte[] data) { this.data = data; }
/**
 * Setzt das Attribut dataLength des Datenpackets.
 * @param data Das neue dataLength.
 */
void setDataLength(int length) { this.dataLength = length; }
/**
 * Gibt das data-Feld zurück.
 * @return Das data-Feld.
 */
byte[] getData() { return this.data; }
/**
 * Generiert gleichverteilte Zufallsdaten.
 * @param Die Länge des Arrays.
 * @return ByteArray mit Zufallszahlen.
 */
byte[] createData(int dataLength)
{
byte[] data = null;
data = new byte[dataLength];
for ( int i = 0 ; i < data.length ; i++ )
data[i] = (byte)Math.round((float)(257*Math.random()-0.5));
return data;
}

// Id Grundoperationen ##############################
/**
 * Gibt den N_id-Teil der Id zurück.
 * @return Der N_id-Teil der Id.
 */
N_id getNId() { return this.dataPackId.nId; }
/**
 * Gibt die Nummer des Datenpacketes zurück.
 * @return Die Nummer des Datenpacketes.
 */
int getPackNr() { return this.dataPackId.packNr; }
/**
 * Gibt die Anzahl der Datenpackete in der Nachricht zurück.
 * @return Die Anzahl der Datenpackete.
 */
int getPackAnz() { return this.dataPackId.packAnz; }
/**
 * Setzt die Id des Datenpacketes.
 * @param id Der N_id-Teil der Id.
 * @param num Die Nummer des Datenpacketes.
 * @param anz Die Anzahl der Datenpackete.
 */
void setId(N_id id, int num, int anz)
```

```
{
this.dataPackId = new D_id();
this.dataPackId.nId = id;
this.dataPackId.packNr = num;
this.dataPackId.packAnz = anz;
}

// Ort Grundoperationen ##############################
/**
 * Gibt den Absendernamen des Datenpacketes zurück.
 * @return Der Absendername des Datenpacketes.
 */
String getAbsenderName() { return this.absender.name; }
/**
 * Gibt den Zielortnamen des Datenpacketes zurück.
 * @return Der Zielortnamen des Datenpacketes.
 */
String getZielortName() { return this.zielort.name; }
/**
 * Gibt die Absenderaddresse des Datenpacketes zurück.
 * @return Die Internet Addresse des Absenders des Datenpacketes.
 */
InetAddress getAbsenderAddr() { return this.absender.addr; }
/**
 * Gibt die Zielortadresse des Datenpacketes zurück.
 * @return Die Internet Addresse des Zielortes des Datenpacketes.
 */
InetAddress getZielortAddr() { return this.zielort.addr; }
/**
 * Gibt den Port der Zielortadresse des Datenpacketes zurück.
 * @return Der Port der Internet Addresse des Zielortes.
 */
int getZielortPort() { return this.zielort.port; }
/**
 * Setzt den Absender des Datenpacketes.
 * @param ort Der Absendername des Datenpacketes.
 * @param adr Die Internet Addresse des Absenders des Datenpacketes.
 * @param port Der Port der Internet Addresse des Absenders.
 */
void setAbsender(String ort, InetAddress adr, int port)
{
this.absender.name = ort;
this.absender.addr = adr;
this.absender.port = port;
}
/**
 * Setzt den Absender des Datenpacketes.
 * Adresse und Port werden dem ComDevice entnommen.
 * @param ort Der Absendername des Datenpacketes.
 */
void setAbsender(String ort)
{
```

```
this.setAbsender(ort, ComDevice.getLocalHost(), ComDevice.getSimulationPort());
}
/**
 * Setzt den Absender des Datenpacketes.
 * @param ort Der Absendername des Datenpacketes.
 * @param host Der Name der Maschine des Absenders.
 * @param port Der Port der Internet Addresse des Absenders.
 */
void setAbsender(String ort, String host, int port)
{
this.setAbsender(ort, ComDevice.getByName(host), port) ;
}

/**
 * Setzt den Zielort des Datenpacketes.
 * @param ort Der Zielortname des Datenpacketes.
 * @param adr Die Internet Addresse des Zielortes des Datenpacketes.
 * @param port Der Port der Internet Addresse des Zielortes.
 */
void setZielort(String ort, InetAddress adr, int port)
{
this.zielort.name = ort;
this.zielort.addr = adr;
this.zielort.port = port;
}
/**
 * Setzt den Zielort des Datenpacketes. Der Port wird dem ComDevice entnommen.
 * @param ort Der Zielortname des Datenpacketes.
 * @param adr Die Internet Addresse des Zielortes des Datenpacketes.
 */
void setZielort(String ort, InetAddress adr)
{ this.setZielort(ort, adr, ComDevice.getSimulationPort()); }
/**
 * Setzt den Zielort des Datenpacketes. Der Port wird dem ComDevice entnommen.
 * @param ort Der Zielortname des Datenpacketes.
 * @param host Der Maschinenname des Zielortes des Datenpacketes.
 */
void setZielort(String ort, String host)
{ this.setZielort(ort, ComDevice.getByName(host)); }
/**
 * Setzt den Absender des Datenpacketes.
 * @param ort Der Absender des Datenpacketes.
 */
void setAbsender(Ort ort) { this.absender = ort; }
/**
 * Setzt den Zielort des Datenpacketes.
 * @param ort Der Zielort des Datenpacketes.
 */
void setZielort(Ort ort) { this.zielort = ort; }
/**
 * Gibt den Absender des Datenpacketes zurück.
 * @return Der Absender des Datenpacketes.
```

```
 */
Ort getAbsender() { return this.absender; }
/**
 * Gibt den Zielort des Datenpacketes zurück.
 * @return Der Zielort des Datenpacketes.
 */
Ort getZielort() { return this.zielort; }

// Zeit Grundoperationen ##############################
/**
 * Gibt den Zeitstempel A zurück..
 * @return Der Zeitstempel A.
 */
Date getATime() { return this.aTime; }
/**
 * Gibt den Zeitstempel Z zurück..
 * @return Der Zeitstempel Z.
 */
Date getZTime() { return this.zTime; }
/**
 * Setzt den Zeitstempel A.
 * @param time Der Zeitstempel A.
 */
void setATime(Date time) { this.aTime = time; }
/**
 * Setzt den Zeitstempel Z.
 * @param time Der Zeitstempel Z.
 */
void setZTime(Date time) { this.zTime = time; }
/**
 * Setzt den Zeitstempel A auf die Aktuelle Zeit.
 */
void setCurrATime() { this.aTime = new Date(System.currentTimeMillis()); }
/**
 * Setzt den Zeitstempel Z auf die Aktuelle Zeit.
 */
void setCurrZTime() { this.zTime = new Date(System.currentTimeMillis()); }


// Serializable functions ##############################
private void writeObject(java.io.ObjectOutputStream out) throws IOException
{
// data
out.writeInt(this.dataLength);
// Id's
out.writeObject(this.dataPackId.nId);
out.writeInt(this.dataPackId.packNr);
out.writeInt(this.dataPackId.packAnz);
// A und Z
out.writeObject(this.absender);
out.writeObject(this.zielort);
// Zeiten
```

```
out.writeObject(this.aTime);
out.writeObject(this.zTime);
out.flush();
}
private void readObject(java.io.ObjectInputStream in)
                                throws IOException, ClassNotFoundException
{
// data
this.dataLength = in.readInt();
// Id's
this.dataPackId = new D_id();
this.dataPackId.nId = (N_id)in.readObject();
this.dataPackId.packNr = in.readInt();
this.dataPackId.packAnz = in.readInt();
// A und Z
this.absender = (Ort)in.readObject();
this.zielort = (Ort)in.readObject();
// Zeiten
this.aTime = (Date)in.readObject();
this.zTime = (Date)in.readObject();
}
}
```

A.4.6. D_id

```
/**
 * Diese Klasse implementiert die Id eines Datenpacketes
 *
 * @author xelahr
 * @version 0.3
 */
class D_id
{
/**
 * Id der Nachricht aus der das Datenpacket stammt.
 */
N_id nId;
/**
 * Die Nummer des Datenpackets.
 */
int packNr;
/**
 * Die Anzahl der Datenpackete in der Nachricht..
 */
int packAnz;
}
```

A.4.7. Ort

```
import java.io.Serializable;
import java.io.IOException;
```

```
import java.net.InetAddress;
/**
 * Diese Klasse implementiert eine Ortsangabe.
 *
 * @author xelahr
 * @version 0.3
 */
class Ort implements Serializable
{
/**
 * Bezeichnung des Ortes.
 */
String name = null;
/**
 * Internet Adresse der Maschine.
 */
InetAddress addr = null;
/**
 * Simulationsport.
 */
int port = 5007;
private void writeObject(java.io.ObjectOutputStream out) throws IOException
{
out.writeObject(this.name);
out.writeObject(this.addr);
out.writeInt(this.port);
out.flush();
}
private void readObject(java.io.ObjectInputStream in)
                                    throws IOException, ClassNotFoundException
{
this.name = (String)in.readObject();
this.addr = (InetAddress)in.readObject();
this.port = in.readInt();
}
}
```

A.5. SimConfig

```
import java.io.Serializable;
import java.io.IOException;
import java.io.StringReader;
import java.util.StringTokenizer;
import java.util.Vector;
import java.util.Iterator;
/**
 * Diese Klasse impementiert eine Konfiguration eines Simulationsexperimentes.
 *
 * @author xelahr
 * @version 0.3
 */
class SimConfig implements Serializable
```

```
{
/**
 * Der Konfigurationsstring.
 */
private String configString;
/**
 * Die Platformkonfiguration.
 */
private PlatformConfig platformen;
/**
 * Die Stellenkonfiguration.
 */
private StellenConfig stellen;
/**
 * Die Nachrichtenkonfiguration.
 */
private NachrichtenConfig nachrichten;

/**
 * Kreiert eine leere Konfiguration.
 */
SimConfig()
{
this.configString = null;
this.platformen = null;
this.stellen = null;
this.nachrichten = null;
}

/**
 * Kreiert eine neue Konfiguration.
 * @param conf Ein Konfigurationsstring.
 */
SimConfig(String conf)
{
this.parseConfig(conf);
}

// Interface to Simulation ##############################
/**
 * Gibt eine Liste aller an der Simulation beteiligten Maschinen zurück.
 * @return Array mit Maschinenamen.
 */
String[] getSimSlaveList()
{
Vector buff = new Vector();
for ( int i = 0 ; i < this.platformen.rechner.length ; i++ )
{
//wenn nicht im buffer dazupacken
Iterator iter = buff.iterator();
boolean isIn = false;
while(iter.hasNext())
```

```
{
String tmp = (String)iter.next();
if ( tmp.compareTo(this.platformen.rechner[i]) == 0 ) { isIn = true; break; }
}
if(!isIn) buff.add(this.platformen.rechner[i]);
}
String[] slaves = new String[buff.size()];
for ( int i = 0 ; i < slaves.length ; i++ )
slaves[i] = (String)buff.elementAt(i);

return slaves;
}

/**
 * Gibt eine Liste aller Platformen zurück, auf denen der gegebene Stellentyp
 * simuliert werden soll.
 * @param stelle Ein Stellentyp.
 * @return Array mit Maschinenamen.
 */
String[] getPlatformenFor(String stelle)
{
Vector platformV = new Vector();
for ( int i = 0 ; i < this.platformen.stellen.length ; i++ )
if ( this.platformen.stellen[i].compareTo(stelle) == 0 )
platformV.add(this.platformen.rechner[i]);
if (platformV.size() == 0) return null;
String[] platform = new String[platformV.size()];
for ( int i = 0 ; i < platform.length ; i++ )
platform[i] = (String)platformV.elementAt(i);
return platform;
}

/**
 * Gibt eine Liste aller Stellen zurück, die auf der der gegebenen Platform
 * simuliert werden sollen.
 * @param platform Eine Platform.
 * @return Array mit Stellenbezeichnungen.
 */
String[] getStellenFor(String platform)
{
Vector stellenV = new Vector();
for ( int i = 0 ; i < this.platformen.rechner.length ; i++ )
if ( this.platformen.rechner[i].compareTo(platform) == 0 )
stellenV.add(this.platformen.stellen[i]);
if (stellenV.size() == 0) return null;
String[] stellen = new String[stellenV.size()];
for ( int i = 0 ; i < stellen.length ; i++ )
stellen[i] = (String)stellenV.elementAt(i);
return stellen;
}

/**
```

```
 * Gibt den Typ={client,server} der gegebenen Stelle zurück.
 * @param stelle Eine Stelle.
 * @return Der entsprechende Stellentyp.
 */
String getTypeOfStelle(String stelle)
{
String type = null;
for ( int i = 0 ; i < this.stellen.bezeichnung.length ; i++ )
if ( this.stellen.bezeichnung[i].compareTo(stelle) == 0 )
{
type = this.stellen.typ[i];
break;
}
return type;
}

/**
 * Gibt die Startnachricht zu der gegebenen Stelle zurück.
 * @param stellenBezeichnung Eine Stelle.
 * @return Die entsprechende Startnachricht.
 */
String getStartNachricht(String stellenBezeichnung)
{
String start = null;
for ( int i = 0 ; i < this.stellen.bezeichnung.length ; i++ )
{
if ( this.stellen.bezeichnung[i].compareTo(stellenBezeichnung) == 0 )
{
start = this.stellen.startNachricht[i];
break;
}
}
return start;
}

/**
 * Gibt die Anzahl der Stellen zu der gegebenen Stelle und Platform zurück.
 * @param platform Eine Plattform.
 * @param stelle Eine Stelle.
 * @return Die entsprechende Anzahl.
 */
int getNumOfStellenFor(String platform, String stelle)
{
int num = 0;
for ( int i = 0 ; i < this.platformen.rechner.length ; i++ )
if ( this.platformen.rechner[i].compareTo(platform) == 0
&&  this.platformen.stellen[i].compareTo(stelle) == 0 )
{
num = this.platformen.anzahl[i];
break;
}
return num;
```

```
}

// Interface to Stellen ##############################
/**
 * Gibt die der Bezeinung entsprechende Nachricht zurück.
 * @param nachricht Eine Bezeichnung einer Nachricht.
 * @return Die entsprechende Nachricht.
 */
Nachricht getNachricht(String nachricht)
{
if (nachricht == null) return null;
for ( int i = 0 ; i < this.nachrichten.nachrichten.length ; i++ )
{
if ( this.nachrichten.nachrichten[i].getIdName().compareTo(nachricht) == 0 )
{
return (Nachricht)this.nachrichten.nachrichten[i].clone();
}
}
return null;
}

// Serializable functionality ##############################
private void writeObject(java.io.ObjectOutputStream out) throws IOException
{}

private void readObject(java.io.ObjectInputStream in)
                        throws IOException, ClassNotFoundException
{}


// ##############################
// ##############################
// ##############################

private class PlatformConfig
{
SimConfig master;
String[] rechner;
String[] stellen;
int[] anzahl;

PlatformConfig(SimConfig master)
{
this.master = master;
}

// Bsp.:
/*
{
[ nexus : ( Autorisierungsserver 1 ) ]
[ naxos : ( Cashautomat 20 ) ]
}
```

```
*/
void parseConfig(String config)
{
String buff = new String();
Vector rechnerBuff = new Vector();
Vector stellenBuff = new Vector();
Vector anzBuff = new Vector();
// lese einnzelne teile in die buffer
while(master.cutNextPosPoint(config, "[","]"))
{
config = master.restBuffer.substring(0);
buff = master.cutBuffer.substring(0) + " ; ";
StringTokenizer str = new StringTokenizer(buff);
// lese Rechnernamen
String name = str.nextToken();
str = null;
master.cutNextPosPoint(buff, ":",";");
buff = master.cutBuffer.substring(0);
// lese SimPlatformen
while(master.cutNextPosPoint(buff, "(",")"))
{
str = new StringTokenizer(master.cutBuffer.substring(0));
rechnerBuff.add(name);
stellenBuff.add(str.nextToken());
anzBuff.add(str.nextToken());
buff = master.restBuffer.substring(0);
}
// copy in die arrays
int n = rechnerBuff.size();
this.rechner=new String[n];
this.stellen=new String[n];
this.anzahl=new int[n];
for ( int i = 0 ; i < n ; i++ )
{
this.rechner[i] = (String)rechnerBuff.elementAt(i);
this.stellen[i] = (String)stellenBuff.elementAt(i);
this.anzahl[i] = Integer.parseInt((String)anzBuff.elementAt(i));
}
}
// lese aus den buffern in die arrays
}
}

private class StellenConfig
{
SimConfig master;
String[] bezeichnung;
String[] typ;
String[] startNachricht;
StellenConfig(SimConfig master)
{
this.master = master;
```

```
}
// Bsp:
/*
{
[ Cashautomat : client : Identifikation ]
[ Autorisierungsserver : server : Autorisierung ]
}
*/
void parseConfig(String config)
{
String buff = new String();
Vector bezeichnungBuff = new Vector();
Vector typBuff = new Vector();
Vector startBuff = new Vector();
// lese einnzelne teile in die buffer
while(master.cutNextPosPoint(config, "[","]"))
{
config = master.restBuffer.substring(0);
buff = master.cutBuffer.substring(0);
StringTokenizer str = new StringTokenizer(buff);
String name = str.nextToken();
str.nextToken();
String type = str.nextToken();
str.nextToken();
String start = str.nextToken();
bezeichnungBuff.add(name);
typBuff.add(type);
startBuff.add(start);
}
// lese aus den buffern in die arrays
int n = bezeichnungBuff.size();
this.bezeichnung = new String[n];
this.typ = new String[n];
this.startNachricht = new String[n];
for ( int i = 0 ; i < n ; i++ )
{
this.bezeichnung[i] = (String)bezeichnungBuff.elementAt(i);
this.typ[i] = (String)typBuff.elementAt(i);
this.startNachricht[i] = (String)startBuff.elementAt(i);
}
}
}

private class NachrichtenConfig
{
SimConfig master;
Nachricht[] nachrichten;
NachrichtenConfig(SimConfig master)
{
this.master = master;
}
```

```
// Bsp:
/*
{
Identifikation : Cashautomat -> Authentisierungsserver
f_size: [ (12.3 1024) (67.7 2048) (20 4096) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (20.0 10000) (20.0 300000) (20.0 600000) (20.0 1800000) (20.0 3200000) ]
f_reply: [ (1.0 Autorisierung) ]
}
{
Autorisierung : Authentisierungsserver -> Cashautomat
f_size: [ (12.3 1024) (67.7 2048) (20 4096) ]
f_time: [ (50.0 0) (50.0 100) ]
f_serv: [ (33.3 1000) (33.3 2000) (33.3 3000) ]
f_wait: [ (100.0 0) ]
f_reply: [ (1.0 Identifikation) ]
}
*/
void parseConfig(String config)
{
String buff = new String();
Vector nBuff = new Vector();
while(master.cutNextPosPoint(config, "{","}"))
{
buff = master.cutBuffer.substring(0);
config = master.restBuffer.substring(0);
Nachricht tmp = Nachricht.parseNachricht(buff);
nBuff.add(tmp);
}
int n = nBuff.size();
nachrichten = new Nachricht[n];
for ( int i = 0 ; i < n ; i++ )
nachrichten[i] = (Nachricht)nBuff.elementAt(i);
}
}

public String toString()
{
String buffer = new String();
//buffer += "##### Original Configuration String #####\n";
//buffer += this.configString;
//buffer += "\n##### Parsed Configuration String #####\n";
buffer += "##### PlatformConfig #####\n";
buffer += "{\n";
for (int i = 0; i<this.platformen.rechner.length ; i++)
buffer += "[ "+this.platformen.rechner[i]
       +" : ( "+this.platformen.stellen[i]+" "
       +this.platformen.anzahl[i]+" ) ]\n";
buffer += "}\n";
buffer += "##### StellenConfig #####\n";
buffer += "{\n";
```

```
for (int i = 0; i<this.stellen.bezeichnung.length ; i++)
buffer += "[ "+this.stellen.bezeichnung[i]
+" : "+this.stellen.typ[i]
+" : "+this.stellen.startNachricht[i]+" ]\n";
buffer += "}\n";
buffer += "##### NachrichtenConfig #####\n";
for (int i = 0; i<this.nachrichten.nachrichten.length ; i++)
{
Nachricht tmpNachricht = this.nachrichten.nachrichten[i];
buffer += "{\n";
buffer += tmpNachricht.getIdName()+" : "
+tmpNachricht.getTypeOfA()+" -> "
+tmpNachricht.getTypeOfZ()+"\n";
buffer += "f_size : "+tmpNachricht.getF_size();
buffer += "f_time : "+tmpNachricht.getF_time();
buffer += "f_serv : "+tmpNachricht.getF_serv();
buffer += "f_wait : "+tmpNachricht.getF_wait();
buffer += "f_reply : "+tmpNachricht.getF_reply();
buffer += "}\n";
}
return buffer;
}

// Parse functionality .:############################
// Bsp.:
/*
##### PlatformConfig #####
{
[nexus : (Autorisierungsserver 1)]
[naxos : (Cashautomat 20)]
}
##### StellenConfig #####
{
[Cashautomat : client : Identifikation]
[Autorisierungsserver : server : Autorisierung]
}
##### NachrichtenConfig #####
{
Identifikation : Cashautomat -> Authentisierungsserver
f_size: [ (12.3 1024) (67.7 2048) (20 4096) ]
f_time: [ (33.3 10) (33.3 100) (33.3 1000) ]
f_serv: [ (10.0 1000) (45.0 100) (45.0 200) ]
f_wait: [ (20.0 10000) (20.0 300000) (20.0 600000) (20.0 1800000) (20.0 3200000) ]
f_reply: [ (1.0 Autorisierung) ]
}
{
Autorisierung : Authentisierungsserver -> Cashautomat
f_size: [ (12.3 1024) (67.7 2048) (20 4096) ]
f_time: [ (50.0 0) (50.0 100) ]
f_serv: [ (33.3 1000) (33.3 2000) (33.3 3000) ]
f_wait: [ (100.0 0) ]
f_reply: [ (1.0 Identifikation) ]
```

```
}
##### EndOfConfig #####
*/
// buffer
private static String  cutBuffer = new String();
private static String restBuffer = new String();
//private static String funcBuffer = new String();

/**
 * Parsemethode.
 * @param config Konfigstring
 */
void parseConfig(String config)
{
this.configString = config;
config = SimConfig.correctSyntax(config);
/* unterteile den String in 3 Teile mit Grenzen bei
(##### PlatformConfig ##### - ##### StellenConfig #####)
(##### StellenConfig ##### - ##### NachrichtenConfig #####)
(##### NachrichtenConfig ##### - ##### EndOfConfig #####)
*/
String pConf = null;
String sConf = null;
String nConf = null;
if (cutNextPosPoint( config , "PlatformConfig", "StellenConfig"))
pConf = cutBuffer.substring(0);
else
Debug.print(10,"SimConfig","parseConfig",
                     "Syntax error by reading PlatformConfig");
if (cutNextPosPoint( config , "StellenConfig", "NachrichtenConfig"))
sConf = cutBuffer.substring(0);
else
Debug.print(10,"SimConfig","parseConfig",
                     "Syntax error by reading StellenConfig");
if (cutNextPosPoint( config , "NachrichtenConfig", "EndOfConfig"))
nConf = cutBuffer.substring(0);
else
Debug.print(10,"SimConfig","parseConfig",
                     "Syntax error by reading NachrichtenConfig");

// parse die einzelteile.
this.platformen = new PlatformConfig(this);
this.platformen.parseConfig(pConf);
this.stellen = new StellenConfig(this);
this.stellen.parseConfig(sConf);
this.nachrichten = new NachrichtenConfig(this);
this.nachrichten.parseConfig(nConf);
}

private static boolean cutNextPosPoint(String original,
                                       String ch1, String ch2)
{
```

```
cutBuffer = new String();
restBuffer = new String();
int startIndex = 0;
int endIndex = 0;

startIndex = original.indexOf(ch1);
endIndex = original.indexOf(ch2);
if ( startIndex < 0 || endIndex < 0 || startIndex == endIndex) return false;

cutBuffer = original.substring(startIndex+1,endIndex);
restBuffer = original.substring(0,startIndex)
             + original.substring(endIndex+1,original.length());
return true;
}

private static String correctSyntax(String source)
{
String tmp = new String();
try{
StringReader sr = new StringReader(source);
int i;
while ((i = sr.read()) != -1)
{
char buf[] = {(char)i};
String ch = new String(buf);
// fuer das lesen nur einer zeile
//if (ch.equals("\n") || ch.equals("\r"))
//{
// if (tmp.length() != 0)
// {
// return tmp.trim();
// }
//}
if ( ch.equals("(") || ch.equals(")") || ch.equals("[")
                    || ch.equals("]") || ch.equals(":") )
{
tmp += " " + ch + " ";
}
else
{
tmp += ch;
}
}
}
catch (Exception e) { Debug.print(100,"SimConfig","correctSyntax",
                                  "error: "+e.toString()); }
return tmp;
}
}
```

A.6. User Interface (LastGenerator)

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
/**
 * Diese Klasse impementiert eine Kommandozeilenschnittstelle der
 * Simulationsplatform.
 *
 * @author xelahr
 * @version 0.3
 */
public class LastGenerator
{
/**
 * zugehörige Simulationsplatform
 */
private static Simulation simulationSlave;

/**
 * Hauptmethode der Klasse.
 */
public static void main(String[] args) throws java.io.IOException,
                                       java.net.UnknownHostException,
                                       java.lang.ClassNotFoundException
{
int mode = 6;
int priority = 10;
String configFileName = "test000.lgs";
String config = null;

int i=0;
while ( i < args.length)
{
if (args[i].compareTo("-g")==0) // -g Parameter
{
if ((i+1) < args.length)
{
try
{
priority = Integer.decode(args[i+1]).intValue();
i++;
}
catch(Exception e) { priority = Integer.MAX_VALUE; }
}
else
priority = Integer.MAX_VALUE;

i++;
continue;
}
if (args[i].compareTo("-master")==0) // -m Parameter
{
```

```
mode = 0;
i++;
if ( i < args.length )
{
configFileName = args[i];
i++;
}
continue;
}
if (args[i].compareTo("-slave")==0) // -s Parameter
{
mode = 1;
i++;
continue;
}
if (args[i].compareTo("-help")==0) // -h Parameter
{
mode = 6;
i++;
continue;
}
if (args[i].compareTo("-test")==0) // -t Parameter
{
mode = 7;
i++;
if ( i < args.length )
{
configFileName = args[i];
i++;
}
continue;
}
if (args[i].compareTo("-parse")==0) // -parse Parameter
{
mode = 9;
i++;
if ( i < args.length )
{
configFileName = args[i];
i++;
}
continue;
}
else
{
mode = 6;
i++;
continue;
}
}
Debug.setCurrPrio(priority);
```

```
//readConfig
try
{
config = readConfig(configFileName);
} catch (Exception e)
{ Debug.print(5,"LastGenerator","main:","failed to read config:" + e); }

switch(mode)
{
case  0: {
simulationSlave = new Simulation();
simulationSlave.runMasterSim(config);
  break;
}
case  1: {
simulationSlave = new Simulation();
simulationSlave.runSlaveSim();
break;
}
case  6: {
help();
  break;
}
case  7: {
simulationSlave = new Simulation();
simulationSlave.runTestSim(config);
break;
}
case  9: {
SimConfig conf = new SimConfig();
conf.parseConfig(config);
System.out.println(conf.toString());
break;
}
default: {
Debug.print(15,"LastGenerator","main:","unknown simulation mode");
help();
}
}

}

/**
 * Methode zum einlesen einer Konfigurationsdatei.
 * @param filename Konfigurationsdatei
 * @return Konfigurationsstring
 */
static String readConfig(String filename) throws IOException
{
FileReader fr = new FileReader(filename);
String tmp = new String();
int i = 0; // i zaehlt Symbole in der Zeile
```

```
while (fr.ready())
{
char buf[] = {(char)fr.read()};
String ch = new String(buf);
tmp += ch;
}
fr.close();
return tmp;
}

/**
 * Methode zur Ausgabe des help-Textes
 */
static void help()
{
System.out.println("\nUsage: LastGenerator <options>\n\n"+
"where <options> includes:\n"+

   "-g\t\t\tgenerate all debugging info\n"+
   "-g <priority (int)>\tgenerate only some debugging info\n"+

"-slave\t\t\tslave mode\n"+
"-test <config.lgt>\ttest mode\n"+
"-master <config.lgs>\tmaster mode\n"+
"-parse <config.lgs>\tparse config file\n"
);

}
}
```

A.7. Debug

```
import java.util. *;

/**
 * Implementierung eines Debug-Patterns.
 * @author awagner@upb.de,xelahr@upb.de
 * @version 0.9
 */
public class Debug
{
/**
 * Die momentane Prioritaet der auszugebenden Debug-Meldungen. <br>
 * Meldungen mit prio &gt; currPrio werden nicht ausgegeben.
 */
private static int currPrio = 10;

/**
 * setzt den momentanen Debug-Level auf prio
 * @param prio der neue Debug-Level
 */
public static void setCurrPrio(int prio)
```

```
{
currPrio = prio;
}

/**
 * @param priority Die Prioritaet der Debug-Nachricht
 * @param klasse Die Klasse aus der die Debug-Nachricht kommt
 * @param methode Die Methode aus der die Debug-Nachricht kommt
 * @param message Freitext
 */
public static void print (int priority, String klasse,
                          String methode, String message)
{
if (priority <= currPrio)
   System.out.println("prio = " + priority + "\t" + klasse
                    +" : "+methode+" -> "+message );
}  // print

/**
 * @param priority Die Prioritaet der Debug-Nachricht
 * @param message Freitext
 */
public static void print (int priority, String message)
{
if (priority <= currPrio) System.out.print(message);
}  // print

/**
 * @param priority Die Prioritaet der Debug-Nachricht
 * @param klasse Die Klasse aus der die Debug-Nachricht kommt
 * @param methode Die Methode aus der die Debug-Nachricht kommt
 * @param message Integer-Zahl
 */
public static void print (int priority, String klasse,
                           String methode, int message)
{
if (priority <= currPrio)
   System.out.println("prio = " + priority + "\t" + klasse
                    +" : "+methode+" -> "+message );
}  // print

/**
 * @param priority Die Prioritaet der Debug-Nachricht
 * @param klasse Die Klasse aus der die Debug-Nachricht kommt
 * @param methode Die Methode aus der die Debug-Nachricht kommt
 * @param message Double-Zahl
 */
public static void print (int priority, String klasse,
                           String methode, double message)
{
if (priority <= currPrio)
   System.out.println("prio = " + priority + "\t" + klasse
```

```
                    +" : "+methode+" -> "+message );
}  // print

/**
 * @param priority Die Prioritaet der Debug-Nachricht
 * @param klasse Die Klasse aus der die Debug-Nachricht kommt
 * @param methode Die Methode aus der die Debug-Nachricht kommt
 * @param message Objekt
 */
public static void print (int priority, String klasse,
                          String methode, Object message)
{
if (priority <= currPrio)
   System.out.println("prio = " + priority + "\t" + klasse
                    +" : "+methode+" -> "+message );
}  // print

} // Debug
```

Literaturverzeichnis

[A.N88] Sirjaev A.N. *Wahrscheinlichkeit.* VEB Deutscher Verlag der Wissenschaften, 1988.

[AS00] Christos Alexopoulos and Andrew F. Seila. Proceedings of the 2000 winter simulation conference. In *Output analysis for simulations*, pages 101–108, 2000.

[Ban00] Jerry Banks. Proceedings of the 2000 winter simulation conference. In *Introduction to Simulation*, pages 9–16, 2000.

[BCNN00] J. Banks, J.S. Carson, B.L. Nelson, and D.M. Nicol. *Diskrete event system simulation.* Prentice-Hall, 3. edition, 2000.

[BEPW96] Klaus Backhaus, Bernd Erichson, Wulff Plinke, and Rolf Weiber. *Multivariante Analysemethoden.* Springer, 8. edition, 1996.

[Car93] J.S. Carson. Proceedings of the 1993 winter simulation conference. In *Modelling and simulation world views*, pages 18–23, 1993.

[DA93] Alan Dolan and Joan Aldous. *Networks and algorithms.* John Wiley & Sons, INC., 1993.

[GHJV00] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, 2000.

[GT00] David Goldsman and Gamze Tokol. Proceedings of the 2000 winter simulation conference. In *Output analysis procedures for computer simulations*, pages 39–45, 2000.

[JP91] Jean Jacod and Philip Protter. *Probability Essentials.* Springer Verlag, 1991.

[Lau00] Thorsten Laux. Erfassung und analyse von intranet-verkehrsströmen, 9 2000.

[Lee00] Lawrence Leemis. Proceedings of the 2000 winter simulation conference. In *Input Modeling*, pages 17–25, 2000.

[LK00] Averill M. Law and W. Dawid Kelton. *Simulation Modeling and Analysis.* McGraw-Hill series in industrial engeneering and management science, 3. edition, 2000.

[Men82] Günter Menges. *Die Statistik.* Gabler, 1982.

[Nel95] Randolph Nelson. *Probability, stohastic processes, and queueing theory.* Springer Verlag, 1995.

[Pit93] Jim Pitman. *Probability.* Springer Verlag, 1993.

[Por93] Sidney C. Port. *Theoretical Probability for Applications.* John Wiley & Sons, INC., 1993.

[PSS95] C D. Pegen, R.E. Shannon, and R.P. Sadowski. *Introduction to simulation using SIMAN.* McGraw-Hill, 2. edition, 1995.

[Rip87] Brian D. Ripley. *Stohastic Simulation.* John Wiley & Sons, 1987.

[Sar00] Robert G. Sargent. Proceedings of the 2000 winter simulation conference. In *Verification, validation, and accreditation of simulation models*, pages 50–59, 2000.

[Sha98] Robert E. Shannon. Proceedings of the 1998 winter simulation conference. In *Introduction to the Art and Science of Simulation*, pages 7–14, 1998.

[Sta99] William Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2.* Addison Wesley Longman, 1999.

[Tan98] Andrew S. Tanenbaum. *Computernetzwerke.* Prentice Hall, 3. edition, 1998.

[Ter92] Kornel Terplan. *Communication networks management.* Prentice-Hall, 1992.