DIPLOMARBEIT

Objektorientierte Entwicklung eines GUI-basierten Tools für die Simulation ereignisbasierter verteilter Systeme

Durchgeführt an der

Fachhochschule Aachen
Fachbereich Elektrotechnik und Informationstechnik

Eupener Str. 70
D-52066 Aachen

mit Erstprüfer und Betreuer Prof. Dr.-Ing. Martin Oßmann

und Zweitprüfer Prof. Dr. rer. nat. Heinrich Fassbender

durch

Paul C. Bütow

Matr.Nr.: 266617

Matthiashofstr. 15

D-52064 Aachen

Aachen, den 6. August 2008

Danksagungen

Ohne die Hilfe folgender Personen wäre die Anfertigung dieser Diplomarbeit in diesem Maße nicht möglich gewesen. Daher möchte ich mich bedanken bei:

- Prof. Oßmann als 1. Prüfer sowie Prof. Fassbender als 2. Prüfer
- Andre Herbst
- Carrie Callahan
- Florian Bütow
- Jörn Bütow
- Leslie Bütow

Auch vielen Dank an die Open Source Gemeinde, denn diese Diplomarbeit wurde ausschließlich mit Hilfe von Open Source Software angefertigt

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1.

Einleitung

1.1. Motivation

In der Literatur findet man viele verschiedene Definitionen eines verteilten Systems. Vieler dieser Definitionen unterschieden sich untereinander, so dass es schwer fällt eine Definition zu finden, die als Alleinige als die Richtige gilt. Andrew Tanenbaum und Marten van Steen haben für die Beschreibung eins verteilten Systems die Folgende lockere Charakterisierung formuliert:

[Tan03] *"Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Anwender wie ein einzelnes, kohärentes System erscheinen"*

Der Anwender muss sich nur mit dem lokalen vor ihm befindenden Computer auseinandersetzen (Abbildung 1.1) während die Software des lokalen Computers die reibungslose Kommunikation mit den anderen beteiligten Computern des verteilten Systems sicherstellt.

Diese Diplomarbeit soll den Gebrauchern die Betrachtung von verteilten Systemen aus einer anderen Perspektive erleichtern. Es soll nicht die Sichtweise eines Endbenutzers eingenommen werden, sondern es sollen die Funktionsweisen von Protokollen und deren Prozesse in verteilten Systemen begreifbar gemacht werden. Es sollen alle relevanten Ereignisse eines verteilten Systems transparent dargestellt werden können.

Um dieses Ziel zu erreichen soll ein Simulator entwickelt werden, der dies ermöglicht. Der Simulator soll insbesondere für Lehr- und Lernzwecke an der Fachhochschule Aachen entwickelt werden. Beispielsweise sollen Protokolle aus den verteilten Systemen mit ihren wichtigsten Einflussfaktoren simuliert werden können. Der Simulator soll zu verstehen helfen wie die gegebenen Protokolle funktionieren und es soll viel Spielraum für eigene Experimente zur Verfügung stehen. Der Simulator soll nicht auf eine feste Anzahl von Protokollen beschränkt sein. Es muss daher dem Gebraucher ermöglicht werden, eigene Protokolle zu entwerfen.

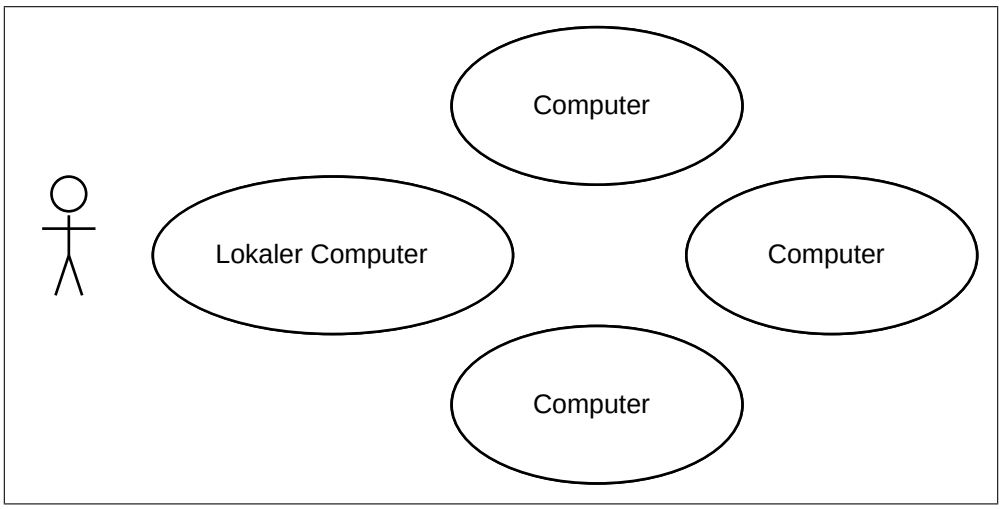

Abbildung 1.1.: Ein verteiltes System bestehend aus 4 Computern

1.2. Grundlagen

Für das Grundverständnis werden im Folgenden einige Grundlagen erläutert. Eine Vertiefung findet erst in den späteren Kapiteln statt.

Client/Server Modell

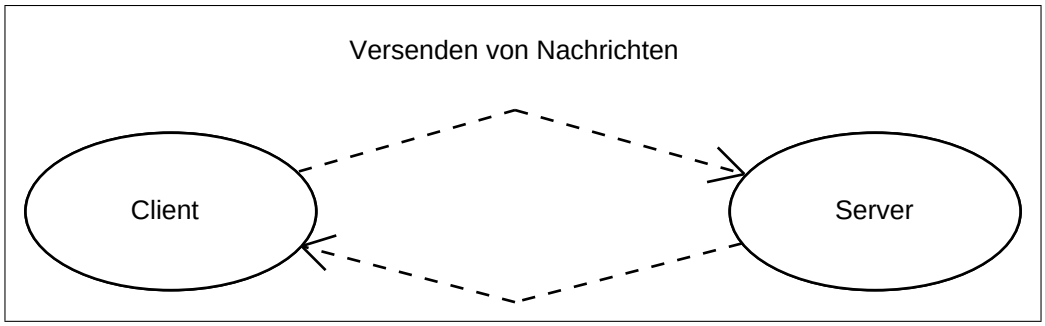

Abbildung 1.2.: Client/Server Modell

Der Simulator basiert auf dem Client/Server Prinzip. Jeder Simulation besteht in der Regel aus einen teilnehmenden Client und einen Server, die miteinander über Nachrichten kommunizieren (Abbildung 1.2). Bei komplexen Simulationen können auch mehrere Clients und/oder Server mitwirken. In der Regel empfangen Server nur Nachrichten, die von Clients verschickt wurden und virce versa.

Prozesse und deren Rollen

Ein verteiltes System wird anhand von Prozessen simuliert. Jeder Prozess nimmt hierbei eine oder mehrere Rollen ein. Beispielsweise kann ein Prozess die Rolle eines Clients einnehmen und ein weiterer Prozess die Rolle eines Servers. Ein Prozess kann auch Client und Server gleichzeitig sein. Es besteht auch die Möglichkeit, dass ein Prozess die Rollen mehrerer Server und Clients gleichzeitig einnimmt. Ob das sinnvoll ist hängt vom simulierten Szenario ab. Um einen Prozess zu kennzeichnen besitzt jeder Prozess eine **eindeutige** Prozess-Identifikationsnummer (PID).

Nachrichten

In einem verteiltem System müssen Nachrichten verschickt werden können. Eine Nachricht kann von einem Client- oder Serverprozess verschickt werden und kann beliebig viele Empfänger haben. Der Inhalt einer Nachricht hängt vom verwendeten Protokoll ab. Was unter einem Protokoll zu verstehen ist, wird später behandelt. Um eine Nachricht zu kennzeichnen besitzt jede Nachricht eine **eindeutige** Nachrichten-Identifikationsnummer (NID).

Lokale und globale Uhren

In einer Simulation gibt es **genau eine** globale Uhr. Sie stellt die aktuelle und **immer korrekte** Zeit dar. Eine globale Uhr geht nie falsch.

Zudem besitzt jeder beteiligter Prozess eine eigene lokale Uhr. Sie stellt die aktuelle Zeit des jeweiligen Prozesses dar. Im Gegensatz zu der globalen Uhr können lokale Uhren eine falsche Zeit anzeigen. Wenn die Prozesszeit nicht global-korrekt ist (nicht der globalen Zeit gleicht beziehungsweise eine falsche Zeit anzeigt), dann wurde sie entweder im Laufe einer Simulation neu gestellt, oder sie geht wegen einer Uhrabweichung falsch. Die Uhrabweichung gibt an, um welchen Faktor die Uhr falsch geht. Hierauf wird später genauer eingegangen.

Neben den normalen Uhren sind auch die **Vektor-Zeitstempel** sowie die **logischen Uhren von Lamport** von Interesse. Jeder Prozess besitzt zusätzlich einen Vektor-Zeitstempel für seine Vektorzeit, sowie einen Lamport-zeitstempel für seine Lamportzeit. Für die Vektor- und Lamportzeiten gibt es hier, im Gegensatz zu der normalen Zeit, keine globalen Äquivalente.

Konkrete Beispiele zu den Lamport- und Vektorzeiten werden später anhand einer Simulation behandelt.

Ereignisse

Eine Simulation besteht aus der Hintereinanderausführung von endlich vielen Ereignissen. Beispielsweise kann es ein Ereignis geben, welches einen Prozess eine Nachricht verschicken lässt. Denkbar wäre auch ein Prozessabsturzereignis. Jedes Ereignis tritt zu einem bestimmten Zeitpunkt ein. Wenn es zeitgleiche Ereignisse gibt, so werden sie in Wirklichkeit ebenso hintereinander ausgeführt, erscheinen aber in der Simulation als ob sie parallel ausgeführt würden. Dieser Umstand ist auf die Implementierung des Simulators zurückzuführen, worauf später noch genauer eingegangen wird. Den Anwender des Simulators hindert dies jedoch nicht, da Ereignisse aus seiner Sicht parallel ausgeführt werden.

Protokolle

Eine Simulation besteht auch aus der Anwendung von Protokollen. Es wurde bereits erwähnt, dass ein Prozess die Rollen von Servern und/oder Clients annehmen kann. Bei jeder Server- und Clientrolle muss zusätzlich das dazugehörige Protokoll spezifiziert werden. Ein Protokoll definiert, wie ein Client und ein Server Nachrichten verschickt und wie bei Ankunft einer Nachricht reagiert wird. Ein Protokoll legt auch fest, welche Daten in einer Nachricht enthalten sind. Ein Prozess verarbeitet eine empfangene Nachricht nur, wenn er das jeweilige Protokoll versteht.

In Abbildung 1.3 sind 3 Prozesse dargestellt. Prozess 1 unterstützt serverseitig das Protokoll “A” und clientseitig das Protokoll “B”. Prozess 2 unterstützt clientseitig das Protokoll “A” und Prozess 3 serverseitig das Protokoll “B”. Das heißt, dass Prozess 1 mit Prozess 2 via Protokoll “A” und mit Prozess 3 via Protokoll “B” kommunizieren kann. Die Prozesse 2 und 3 sind zueinander inkompatibel und können voneinander erhaltene Nachrichten nicht verarbeiten.

Clients können nicht mit Clients, und Server nicht mit Server kommunizieren. Für eine Kommunikation wird stets mindestens ein Client und ein Server benötigt. Diese Einschränkung kann aber umgangen werden, indem Prozesse ein gegebenes Protokoll sowohl server- als auch clientseitig unterstützen (siehe Broadcast-Sturm Protokoll später). Alle vom Simulator verfügbaren Protokolle werden später genauer behandelt.

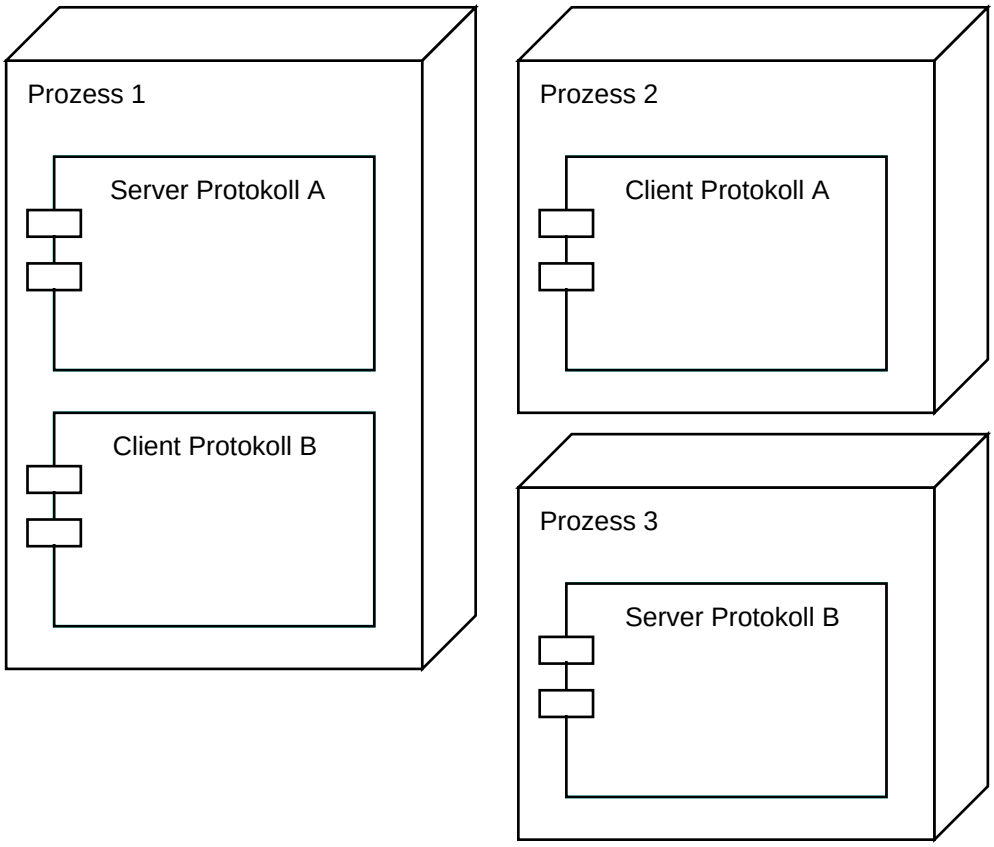

Abbildung 1.3.: Client/Server Protokolle

Kapitel 2.

Der Simulator

2.1. Die grafische Benutzeroberfläche (GUI)

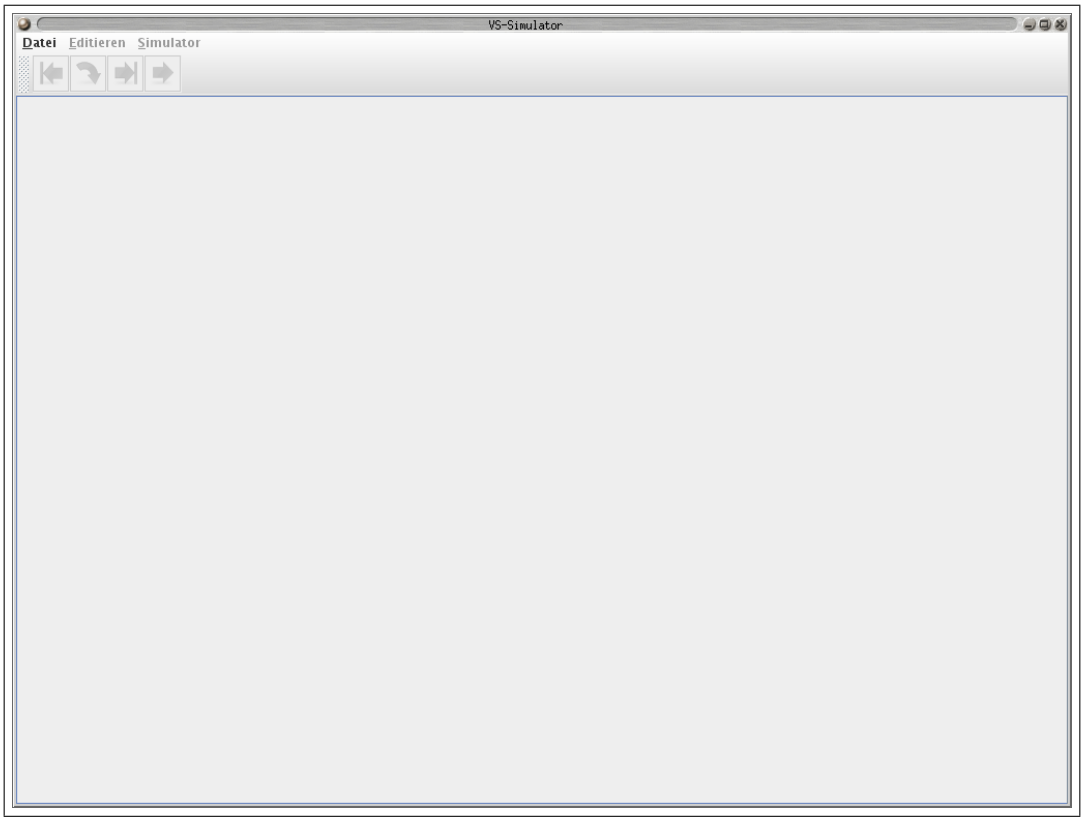

Abbildung 2.1.: Der Simulator nach dem ersten Starten

Der Simulator läßt sich mit dem Befehl *java -jar VS-Sim.jar* starten und präsentiert sich nach dem ersten Starten wie auf Abbildung 2.1. Für die Erstellung einer neuen Simulation wird im Menü “Datei” (Abbildung 2.2) der Punkt

“Neue Simulation” ausgewählt, wo anschließend das Einstellungsfenster für die neue Simulation erscheint. Auf die einzelnen Optionen wird später genauer eingegangen und es werden nun nur die Standardeinstellungen übernommen. Die GUI mit einer frischen Simulation sieht aus wie auf Abbildung 2.3.

Abbildung 2.2.: Datei-Menü

Die Menüzeile

Im Datei-Menü (Abbildung 2.2) lassen sich neue Simulationen erstellen oder die aktuell geöffnete Simulation schließen. Neue Simulationen öffnen sich standardmäßig in einem neuen Tab. Es können allerdings auch neue Simulationsfenster, die wiederum eigene Tabs besitzen, geöffnet oder geschlossen werden. In jedem Tab befindet sich eine von den Anderen vollständig unabhängige Simulation. Es können somit beliebig viele Simulationen parallel ausgeführt werden. Die Menüeinträge “Öffnen”, “Speichern” und “Speichern unter” dienen für das Laden und Speichern von Simulationen.

Über das Editieren-Menü gelangt der Anwender zu den Simulationseinstellungen, worauf später genauer eingegangen wird. In diesem Menü werden auch alle beteiligten Prozesse zum Editieren aufgelistet. Wählt der Anwender dort einen Prozess aus, dann öffnet sich der dazugehörige Prozesseditor. Auf diesen wird ebenso später genauer eingegangen. Das Simulator-Menü bietet die selben Optionen wie die Toolbar, welche im nächsten Teilkapitel beschrieben wird, an.

Einige Menüunterpunkte sind erst erreichbar, wenn im aktuellen Fenster bereits eine Simulation erstellt oder geladen wurde.

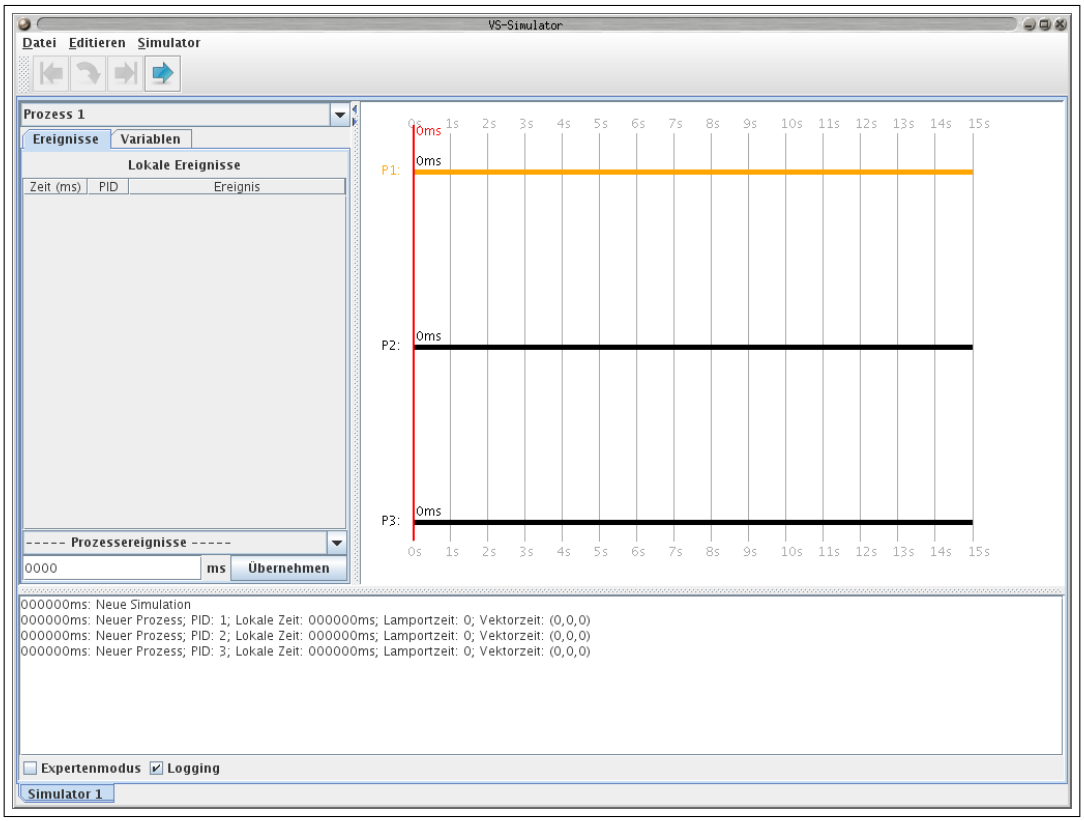

Abbildung 2.3.: Eine neue Simulation

Die Toolbar

Oben links im Simulator befindet sich die Toolbar (Abbildung 2.4). Die Toolbar enthält die Funktionen die vom Anwender am häufigsten benötigt werden.

Abbildung 2.4.: Die Menüzeile inklusive Toolbar

Die Toolbar bietet vier verschiedene Funktionen an:

- Zurücksetzen der Simulation; kann nur betätigt werden, wenn die Simulation pausiert wurde oder wenn die Simulation abgelaufen ist.
- Wiederholen der Simulation; kann nicht betätigt werden, wenn die Simulation noch nicht gestartet wurde.
- Pausieren der Simulation; kann nur betätigt werden, wenn die Simulation derzeit läuft.
- Starten der Simulation; kann nur betätigt werden, wenn die Simulation derzeit nicht läuft.

Die Toolbar lässt sich nach Wunsch repositionieren (z.B. links, rechts oder unten des Simulatorfensters). Hierfür muss sie per “Drag-n-Drop” zur Zielposition gezogen werden.

Die Visualisierung

Mittig rechts befindet sich die grafische Simulationsvisualisierung. Die X-Achse gibt die Zeit in Millisekunden an und auf der Y-Achse sind alle beteiligten Prozesse aufgeführt. Unsere Demo-Simulation endet nach genau 15 Sekunden. Auf Abbildung 2.5 sind 3 Prozesse (mit den PIDs 1, 2 und 3) dargestellt, die jeweils einen eigenen horizontalen schwarzen Balken besitzen. Auf diesen Prozessbalken kann der Anwender die jeweilige lokale Prozesszeit ablesen. Die vertikale rote Linie stellt die globale Simulationszeit dar.

Die Prozessbalken dienen auch für Start- und Zielpunkte von Nachrichten. Wenn beispielsweise Prozess 1 eine Nachricht an Prozess 2 verschickt, so wird eine Linie vom einen Prozessbalken zum Anderen gezeichnet. Nachrichten, die ein Prozess an sich selbst schickt, werden nicht visualisiert. Sie werden aber im Loggfenster (mehr dazu später) protokolliert.

Eine andere Möglichkeit einen Prozesseditor aufzurufen ist ein Linksklick auf den zum Prozess gehörigen Prozessbalken. Dies muss also nicht immer über das Simulator-Menü geschehen. Ein Rechtsklick hingegen öffnet

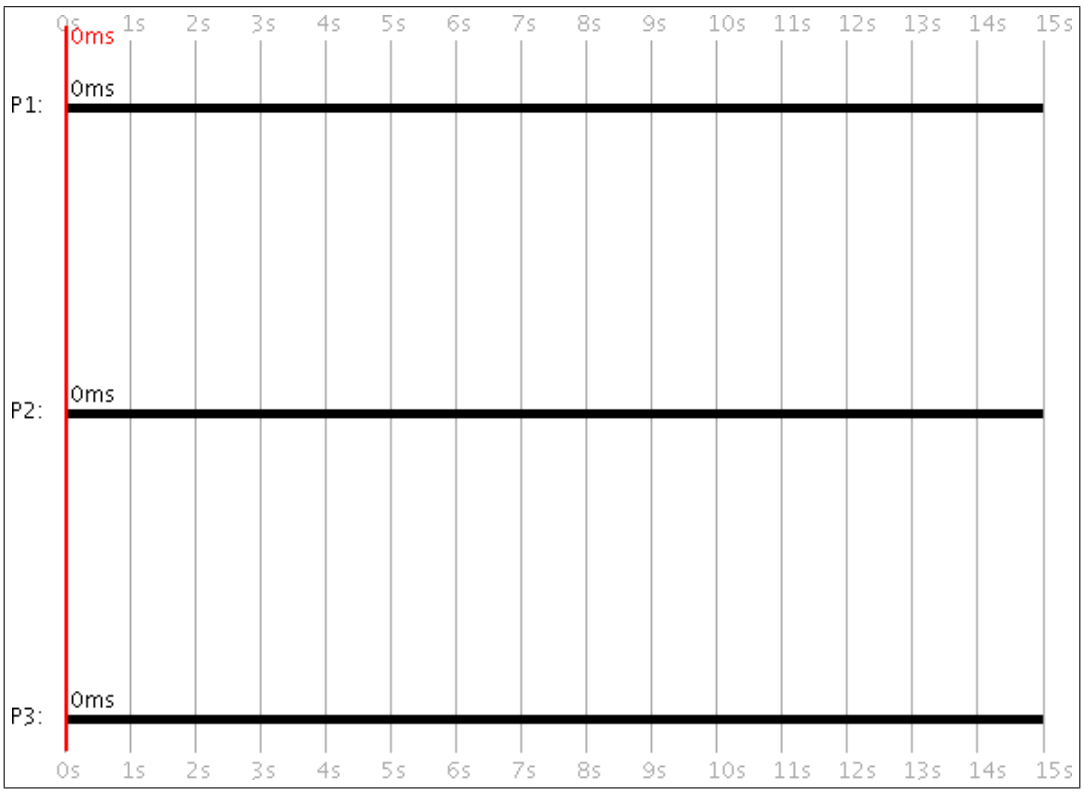

Abbildung 2.5.: Visualisierung einer noch nicht gestarteten Simulation

ein Popup-Fenster mit weiteren Auswahlmöglichkeiten (Abbildung 2.6). Ein Prozess kann über das Popup-Menü nur während einer laufenden Simulation zu einem Absturz oder einer Wiederbelebung bewegt werden.

Generell kann die Anzahl der Prozesse nach belieben variieren. Die Dauer der Simulation beträgt mindestens *5* und höchstens *120* Sekunden. Die Simulation endet erst, wenn sie die globale Zeit die angegebene Simulationsendzeit (hier *15* Sekunden) erreicht hat, und nicht, wenn eine lokale Prozesszeit diese Endzeit erreicht.

Farbliche Differenzierung

Farben helfen dabei die Vorgänge einer Simulation besser zu deuten. Standardmäßig werden die Prozesse (Prozessbalken) und Nachrichten mit den Farben wie in Tabelle 2.1 aufgelistet dargestellt. Dies sind lediglich die Standardfarben, welche über die Einstellungen geändert werden können.

Die Sidebar

Mithilfe der Sidebar lassen sich Prozessereignisse programmieren. Oben auf Abbildung 2.7 ist der zu verwaltende Prozess selektiert (hier mit der PID 1). In dieser Prozessauswahl gibt es auch die Möglichkeit “Alle Prozesse” aus-

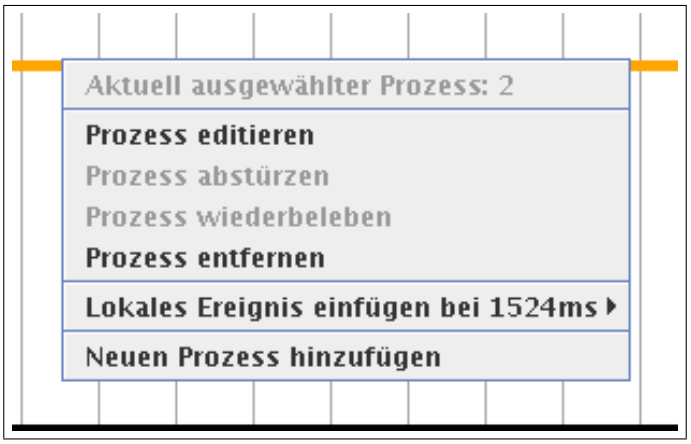

Abbildung 2.6.: Rechtsklick auf einen Prozessbalken

Prozessfarbe	**Bedeutung**
Schwarz	Die Simulation läuft derzeit nicht
Orange	Die Maus befindet sich über den Prozessbalken
Rot	Der Prozess ist abgestürzt
Nachrichtenfarbe	**Bedeutung**
Grün	Die Nachricht ist noch unterwegs und hat das Ziel noch nicht erreicht
Blau	Die Nachricht hat das Ziel erfolgreich erreicht
Rot	Die Nachricht ging verloren

Tabelle 2.1.: Farbliche Differenzierung von Prozessen und Nachrichten

zuwählen, womit die Ereignisse aller Prozesse gleichzeitig verwaltet werden können. Unter “Lokale Ereignisse” versteht man diejenigen Ereignisse, die auftreten, wenn eine bestimmte lokale Zeit des dazugehörigen Prozesses eingetreten ist. Die darunterliegende Ereignistabelle listet alle programmierten Ereignisse (hier noch keine vorhanden) mitsamt Eintrittszeiten sowie den PIDs auf.

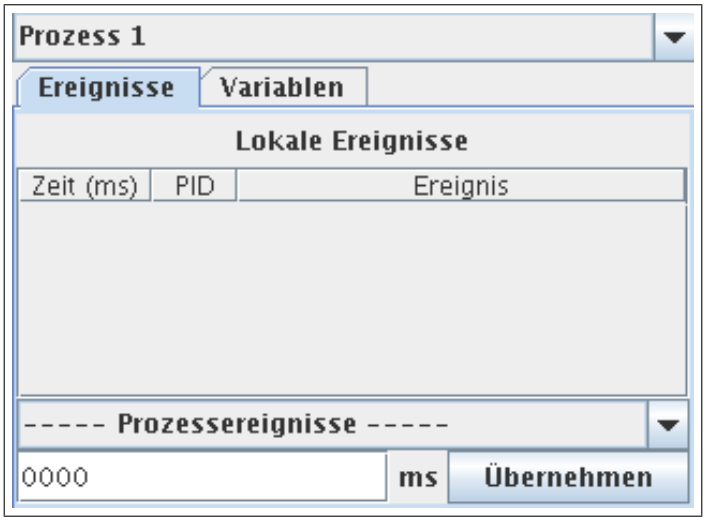

Abbildung 2.7.: Die Sidebar mit leerem Ereigniseditor

Für die Erstellung eines neuen Ereignisses kann der Anwender entweder mit einem Rechtsklick auf einen Prozessbalken (Abbildung 2.6) klicken und dort “Lokales Ereignis einfügen” wählen, oder unterhalb der Ereignistabelle ein Ereignis auswählen (Abbildung 2.8), im darunter liegendem Textfeld die Ereigniseintrittszeit eintragen und auf “Übernehmen” gehen. Beispielsweise wurden auf Abbildung 2.9 drei Ereignisse hinzugefügt: Absturz nach *123ms*, Wiederbelebung nach *321ms* und erneuter Absturz nach *3000ms* des Prozesses mit der ID 1.

Mit einem Rechtsklick auf den Ereigniseditor lassen sich alle selektierten Ereignisse entweder kopieren oder löschen. Mithilfe der Strg-Taste können auch mehrere Ereignisse gleichzeitig markiert werden. Die Einträge der Spalten für die Zeit und der PID lassen sich nachträglich editieren. Somit besteht eine komfortable Möglichkeit bereits programmierte Ereignisse auf eine andere Zeit zu verschieben oder einen anderen Prozess zuzuweisen. Allerdings sollte der Anwender darauf achten, dass er nach dem Ändern der Ereigniseintrittszeit die Enter-Taste betätigt, da sonst die Änderung unwirksam ist.

In der Sidebar gibt es neben dem Ereignis-Tab einen weiteren Tab “Variablen”. Hinter diesem Tab verbirgt sich der Prozesseditor des aktuell ausgewählten Prozesses. Dort können alle Variablen des Prozesses editiert werden und ist somit eine weitere Möglichkeit einen Prozesseditor aufzurufen. Der Prozesseditor wird später genauer behandelt.

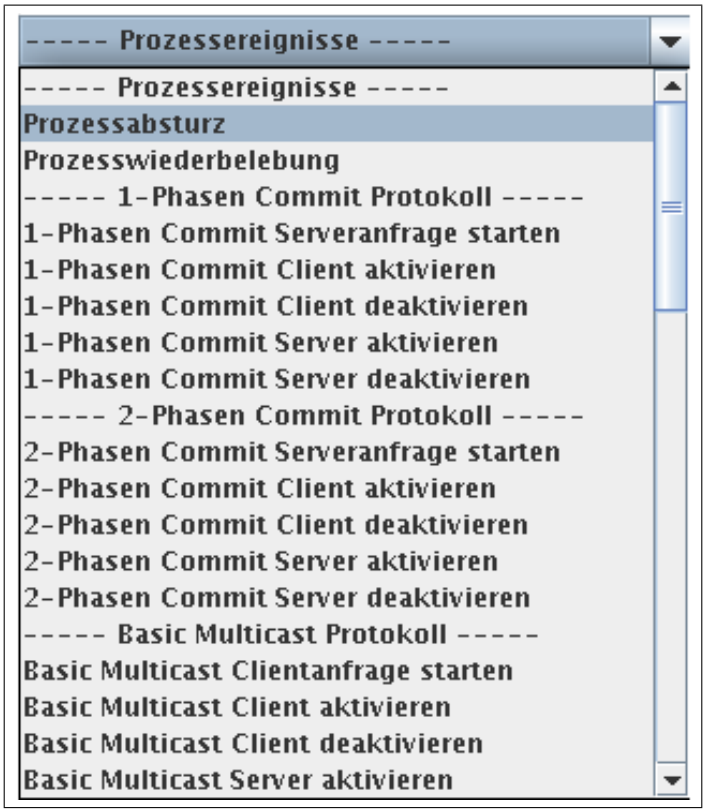

Abbildung 2.8.: Die Ereignisauswahl via Sidebar

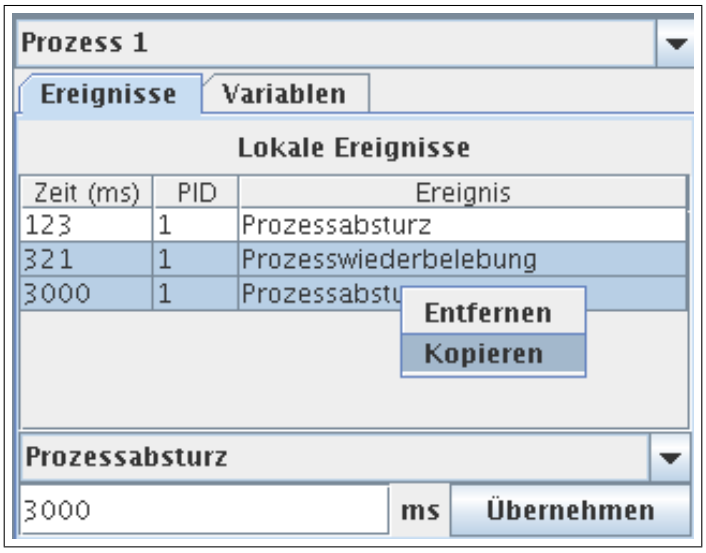

Abbildung 2.9.: Der Ereigniseditor mit 3 programmierten Ereignissen

Das Loggfenster

Das Loggfenster (Abbildung 2.3, unten) protokolliert in chronologischer Reihenfolge alle eingetroffenen Ereignisse. Auf Abbildung 2.10 ist das Loggfenster nach Erstellung der Demo-Simulation zu sehen, an welcher 3 Prozesse beteiligt sind. Am Anfang eines Loggeintrages wird stets die globale Zeit in Millisekunden protokolliert. Bei jedem Prozess werden ebenso seine lokale Zeiten sowie die Lamport- und die Vektor-Zeitstempel aufgeführt. Letztere werden später genauer behandelt. Hinter den Zeitangaben werden weitere Angaben, wie beispielsweise welche Nachricht mit welchem Inhalt verschickt wurde und welchem Protokoll sie angehört, gemacht. Dies wird später noch anhand von Beispielen demonstriert.

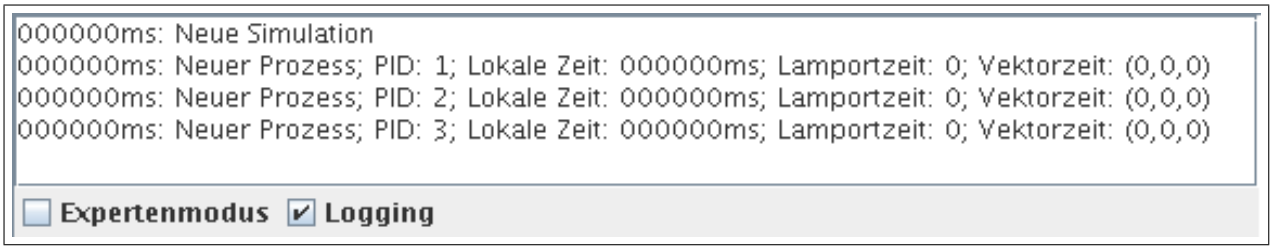

Abbildung 2.10.: Das Loggfenster

Mit dem Deaktivieren der Checkbox “Logging” läßt sich das Loggen von Nachrichten temporär einstellen. Mit deaktiviertem Loggen werden keine neuen Nachrichten mehr ins Loggfenster geschrieben. Nach Reaktivieren der Checkbox werden alle ausgelassenen Nachrichten nachträglich in das Fenster geschrieben. Ein deaktiviertes Loggen kann zu verbessertem Leistungsverhalten des Simulators führen (z.B. kein Rucklen; ist vom verwendeten Computer, auf dem der Simulator läuft, abhängig). Dieser Umstand ist der sehr langsamen Java-Implementierung der JTextArea-Klasse zu verdanken, die schnelle Updates nur sehr träge durchführt.

Über die Checkbox “Expertenmodus” wird der Expertenmodus aktiviert beziehungsweise deaktiviert.

2.2. Der Expertenmodus

Der Simulator kann in zwei verschiedenen Modi betrieben werden. Es gibt einen einfachen- und einen Expertenmodus. Der Simulator startet standardmäßig im einfachen Modus, sodass sich der Anwender nicht mit der vollen Funktionalität des Simulators auf einmal auseinandersetzen muß. Der einfache Modus ist übersichtlicher, bietet jedoch weniger Funktionen an. Der Expertenmodus eignet sich mehr für erfahrene Anwender und bietet dementsprechend auch mehr Flexibilität. Der Expertenmodus kann über die gleichnamige Checkbox unterhalb des Loggfensters oder über die Simulationseinstellungen aktiviert oder deaktiviert werden. Auf Abbildung 2.11 ist der Simulator im Expertenmodus zu sehen. Wenn der Expertenmodus mit dem normalen Modus verglichen wird, dann fallen einige Unterschiede auf:

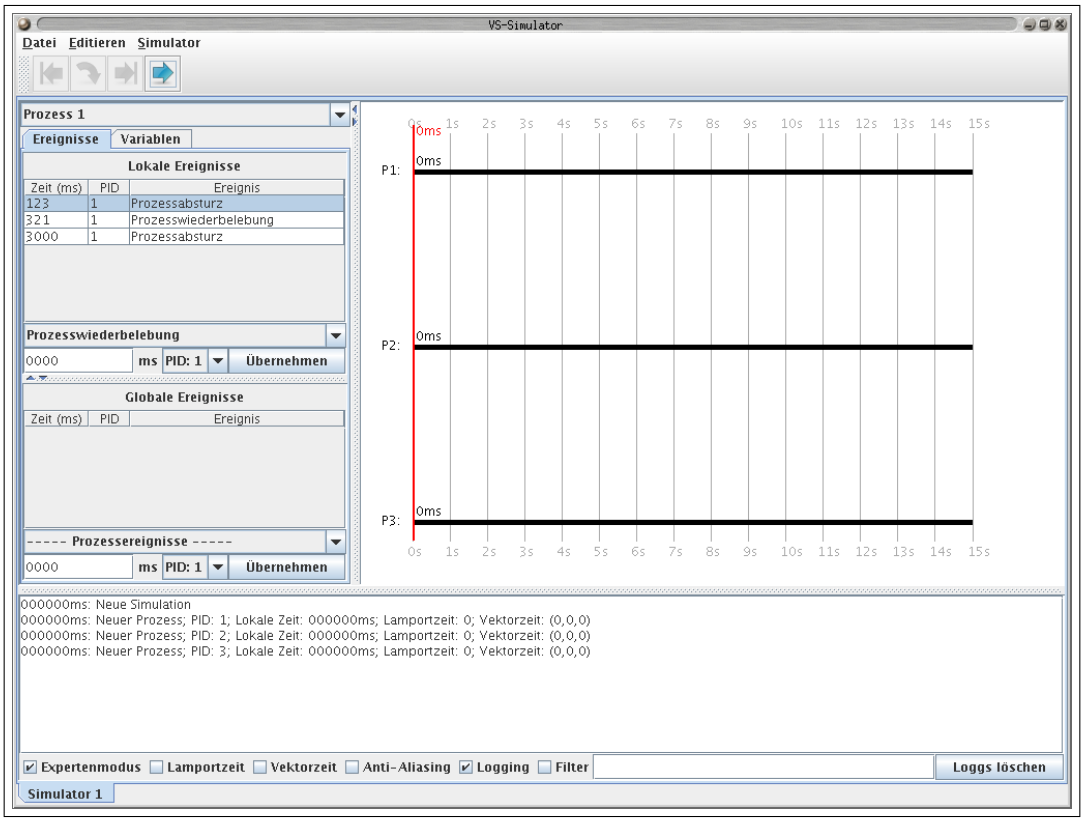

Abbildung 2.11.: Der Simulator im Expertenmodus

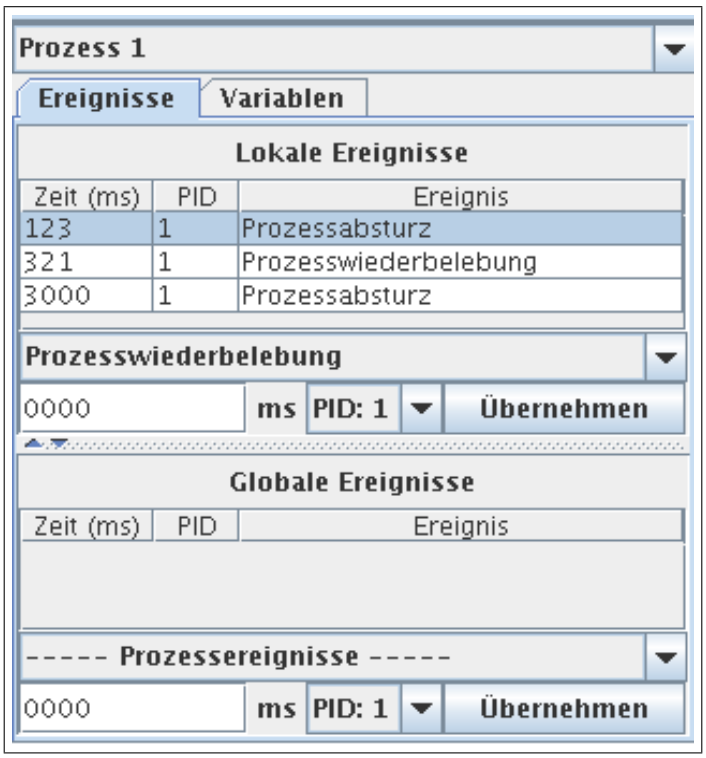

Abbildung 2.12.: Die Sidebar im Expertenmodus

Neue Funktionen in der Sidebar

Der erste Unterschied ist in der Sidebar erkennbar (Abbildung 2.12). Dort sind nun, zusätzlich den lokalen Ereignissen, auch globale Ereignisse editierbar. Wie bereits erwähnt sind unter lokale Ereignisse diejenigen Ereignisse zu verstehen, die auftreten, wenn eine bestimmte lokale Zeit des dazugehörigen Prozesses eingetreten ist. Globale Ereignisse hingegen sind diejenigen Ereignisse, die auftreten, wenn eine bestimmte globale Zeit eingetreten ist. Ein globales Ereignis nimmt die globale Zeit- und ein lokales Ereignis die lokale Prozesszeit als Eintrittskriterium. Globale Ereignisse machen somit nur einen Unterschied, wenn sich die lokalen Prozesszeiten von der globalen Zeit unterscheiden.

Des Weiteren kann der Anwender bei der Programmierung eines neuen Ereignisses direkt die dazugehörige PID selektieren. Im einfachen Modus wurde hier immer standardmäßig die PID des aktuell (in der obersten Combo-Box) ausgewählten Prozesses verwendet (hier mit PID 1). In dieser Combo-Box sollte der Anwender gegebenenfalls “Alle Prozesse” selektieren, damit im Ereigniseditor stets die Ereignisse aller Prozesse aufgelistet werden.

Lamportzeit, Vektorzeit und Anti-Aliasing Schalter

Weitere Unterschiede machen sich unterhalb des Loggfensters bemerkbar. Dort gibt es unter Anderem zwei neue Checkboxen “Lamportzeit” und “Vektorzeit”. Aktiviert der Anwender eine dieser beiden Checkboxen, so wird die Lamport- beziehungsweise Vektorzeit in der Visualisierung dargestellt. Damit die Übersichtlichkeit nicht leidet, kann der Anwender nur jeweils eine dieser beiden Checkboxen zur gleichen Zeit aktiviert haben.

Die Anti-Aliasing-Checkbox ermöglicht dem Anwender Anti-Aliasing zu aktivieren beziehungsweise zu deaktivieren. Mit aktiviertem Anti-Aliasing werden alle Grafiken der Visualisierung gerundet dargestellt. Aus Performancegründen ist Anti-Aliasing standardmäßig nicht aktiv.

Der Loggfilter

Je komplexer eine Simulation wird, desto unübersichtlicher werden die Einträge im Loggfenster. Hier fällt es zunehmend schwerer die Übersicht aller Ereignisse zu behalten. Um dem entgegenzuwirken gibt es im Expertenmodus einen Loggfilter, welcher es ermöglicht nur die wesentlichen Daten aus den Loggs zu filtern.

Der Loggfilter wird anhand der dazugehörigen Checkbox “Filter” aktiviert beziehungsweise deaktiviert. In der dahinterliegenden Eingabezeile kann ein regulärer Ausdruck in Java-Syntax angegeben werden. Beispielsweise werden mit “*PID: (1|2)*” nur Loggzeilen angezeigt, die entweder “*PID: 1*” oder “*PID: 2*” beinhalten. Alle anderen Zeilen, die zum Beispiel nur “*PID: 3*” beinhalten, werden dabei nicht angezeigt. Mit aktivem Loggfilter werden nur die Loggzeilen angezeigt, auf die der reguläre Ausdruck passt. Der Loggfilter kann auch nachträglich aktiviert werden, da bereits protokollierte Ereignisse nach jeder Filteränderung erneut gefiltert werden.

Der Loggfilter kann auch während einer laufenden Simulation verwendet werden. Bei Loggfilterdeaktivierung werden alle Nachrichten wieder angezeigt. Loggnachrichten, die aufgrund des Filters noch nie angezeigt wurden, werden dann nachträglich angezeigt.

2.3. Ereignisse

Es wird zwischen zwei Haupttypen von Ereignissen unterschieden: Programmierbare Ereignisse und nicht-programmierbare Ereignisse. Programmierbare Ereignisse lassen sich im Ereigniseditor programmieren und editieren und deren Eintrittszeiten hängen von den lokalen Prozessuhren oder der globalen Uhr ab. Nicht-programmierbare Ereignisse lassen sich hingegen nicht im Ereigniseditor programmieren und treten nicht wegen einer bestimmten Uhrzeit ein, sondern aufgrund anderer Gegebenheiten wie zum Beispiel das Eintreffen einer Nachricht oder das Ausführen einer Aktion aufgrund eines Weckers, worauf weiter unten nochmal genauer eingegangen wird.

Prozessabsturz- und Wiederbelebung (programmierbar)

Die beiden einfachsten Ereignisse sind “Prozessabsturz” sowie “Prozesswiederbelebung”. Wenn ein Prozess abgestürzt ist, so wird sein Prozessbalken in rot dargestellt. Ein abgestürzter Prozess kann keine weiteren Ereignisse mehr verarbeiten und wenn bei ihm eine Nachricht eintrifft, dann kann sie nicht verarbeitet werden und geht deshalb verloren. Die einzige Ausnahme bildet ein Wiederbelebungsereignis. Ein abgestürzter Prozess kann nichts, außer wiederbelebt werden. Während eines Prozessabsturzes läuft die lokale Prozessuhr, abgesehen der Lamport- und Vektor-Uhren, normal weiter. Das heißt es besteht die Möglichkeit, dass ein Prozess einige seiner Ereignisse gar nicht ausführt, da er zu den Ereigniseintrittszeiten abgestürzt ist. Wenn im echten Leben ein Computer abstürzt oder abgeschaltet wird, dann läuft seine Hardware-Uhr unabhängig vom Betriebssystem auch weiter.

Aktivierung und Deaktivierung von Protokollen sowie Starten von Anfragen (programmierbar)

Es ist bereits bekannt, dass ein Prozess mehrere Protokolle client- und auch serverseitig unterstützen kann. Welches Protokoll von einem Prozess unterstützt wird, kann der Anwender anhand von Protokollaktivierungs- und Protokolldeaktivierungsereignissen konfigurieren. Somit besteht die Möglichkeit, dass ein gegebener Prozess ein bestimmtes Protokoll erst zu einem bestimmten Zeitpunkt unterstützt und gegebenenfalls ein anderes Protokoll ablöst. Jedes Protokoll kann entweder server- oder clientseitig aktiviert beziehungsweise deaktiviert werden. Welche Protokolle es gibt wird später behandelt. Der Anwender hat die Auswahl zwischen fünf verschiedenen Protokollereignistypen:

- Aktivierung des Clients eines gegebenen Protokolls
- Aktivierung des Servers eines gegebenen Protokolls
- Deaktivierung des Clients eines gegebenen Protokolls
- Deaktivierung des Servers eines gegebenen Protokolls
- Starten einer Client/Server-Anfrage eines gegebenen Protokolls

Ob sich das Ereignis für das Starten einer Anfrage auf einen Client oder einen Server bezieht hängt vom verwendeten Protokoll ab. Es gibt Protokolle, wo der Client die Anfragen starten muss, und es gibt Protokolle, wo der Server diese Aufgabe übernimmt. Beispielsweise startet bei dem “Ping-Pong Protokoll” der Client- und bei dem “Commit-Protokollen” der Server immer die erste Anfrage. Es gibt kein Protokoll, wo der Client und der Server jeweils Anfragen starten können.

Bei allen dieser fünf Ereignissen kann der betroffene Prozess noch beliebig andere Dinge, abhängig vom Protokoll, tun. Beispielsweise kann er den Inhalt der Nachricht generieren oder lokale Variablen initialisieren oder eine der lokalen Uhzeiten ändern oder einen Wecker für “Callback Ereignisse” setzen (mehr dazu später) und vieles mehr.

Nachrichtenempfang sowie Antwortnachrichten (nicht-programmierbar)

Nachdem ein Prozess eine Nachricht empfängt wird zuerst überprüft ob er das dazugehörige Protokoll unterstützt. Wenn der Prozess das Protokoll unterstützt, wird geschaut ob es sich um eine Client- oder eine Servernachricht handelt. Wenn es sich um eine Clientnachricht handelt, so muß der Empfängerprozess das Protokoll serverseitig unterstützen und virce versa. Wenn alles passt, dann führt der Empfängerprozess die vom Protokoll definierten Aktionen aus. In der Regel berechnet der Prozess einen bestimmten Wert und schickt ihn über eine Antwortnachricht zurück. Es können aber auch beliebig andere Aktionen ausgeführt werden. Welche dies sind hängt vom Protokoll ab.

Callback-Ereignisse (nicht-programmierbar)

Ein Callback-Ereignis kann von einem Protokoll ausgelöst werden. Das Protokoll setzt einen Wecker, der angibt zur welcher lokalen Uhrzeit eine weitere Aktion ausgeführt werden soll. Zum Beispiel lassen sich hiermit Timeouts realisieren: Wenn ein Protokoll eine Antwort erwartet, diese aber nicht eintrifft, dann kann nach einer bestimmten Zeit eine Anfrage erneut verschickt werden! Es können beliebig viele Callback-Ereignisse definiert werden. Wenn sie noch nicht ausgeführt wurden und aufgrund eines anderen Ereignisses nicht mehr benötigt werden, dann können sie vom Protokoll wieder nachträglich entfernt werden. Wenn ein Callback-Ereignis ausgeführt wird, dann kann es sich selbst wieder für eine weitere Ausführung erneut planen. So lassen sich periodisch wieder-eintreffende Ereignisse realisieren. Beispielsweise verwenden die “Commit-Protokolle” (mehr dazu später) Callback-Ereignisse, indem solange Anfragen verschickt werden, bis alle benötigten Antworten vorliegen.

Zufallsereignisse (nicht-programmierbar)

Die Eintrittszeit eines Zufallsereignisses wird vom Simulator zufällig gewählt. Es besteht lediglich die Möglichkeit die Wahrscheinlichkeit, dass das Ereignis überhaupt eintritt, einzustellen. Ein Beispiel ist ein zufälliger Prozessabsturz, dessen Wahrscheinlichkeit unter den Prozessvariablen konfiguriert werden kann. Diese Variable wird im Abschnitt über den Prozesseditor noch ausführlicher beschrieben.

Prefix	Beschreibung
Boolean	boolschen Wert, z.B. *true* oder *false*
Color	Java-Farbobjekt
Float	Fließkommazahl einfacher genauigkeit
Integer[]	Integervektor
Integer	Einfache Integerzahl
Long	Einfache Long-Zahl
String	Java-Stringobjekt

Tabelle 2.2.: Verfügbare Datentypen für editierbare Variablen

2.4. Einstellungen

In diesem Abschnitt wird genauer auf die möglichen Konfigurationsmöglichkeiten eingegangen. Zunächst gibt es globale Simulationseinstellungen. Diese beinhalten Variablen die die gesamte Simulation betreffen. Zudem hat jeder Prozess seine eigenen lokale Einstellungen. Darüber hinaus kann jedes Protokoll für jeden Prozess separat eingestellt werden.

2.4.1. Variablendatentypen

Der Simulator unterscheidet zwischen mehreren Datentypen, in denen die einstellbaren Variablen vorliegen können (Tabelle 2.2). Im folgenden bedeutet *Prefix*: *wert*, dass die Variable vom Typ *Prefix* ist, und standardmässig den Wert *wert* zugewiesen hat. Vom Anwender lassen sich lediglich die Variablenwerte, jedoch nicht die Variablentypen sowie Variablennamen, ändern.

2.4.2. Simulationseinstellungen

Beim Erstellen einer neuen Simulation erscheint zunächst das dazugehörige Einstellungsfenster (Abbildung 2.13). In der Regel reicht es, wenn der Anwender hier, bis auf die Anzahl beteiligter Prozesse, die Standardwerte übernimmt. Es besteht auch die Möglichkeit die Einstellungen nachträglich zu editieren, indem das Einstellungsfenster via “Editieren ⟶ Einstellungen” erneut aufgerufen wird.

Im Folgenden werden alle in den Simulationseinstellungen verfügbaren Variablen beschrieben. Die Klammern geben die Typen und die Standardwerte an, in denen die Variablen vorliegen.

- **Prozesse empfangen eigene Nachrichten** *(Boolean: false)*: Standardmäßig können Prozesse keine Nachrichten empfangen, die sie selbst verschickt haben. Dies trägt zur Übersichtlichkeit der Simulation bei. Wenn diese Variable jedoch auf *true* gesetzt wird, dann kann ein Prozess auch selbst verschickte

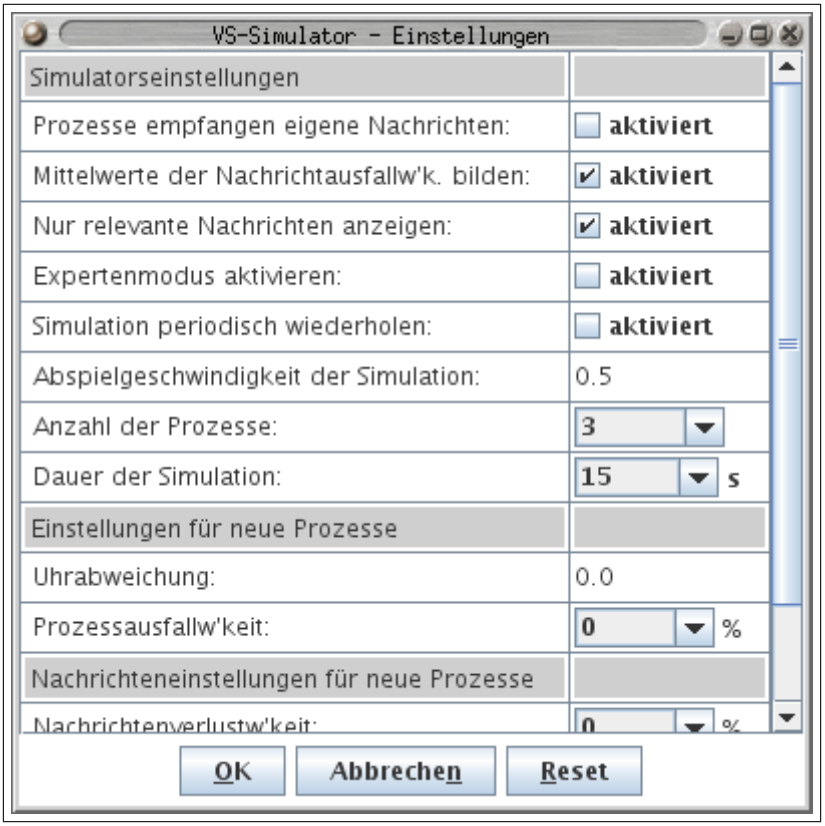

Abbildung 2.13.: Das Fenster zu den Simulationseinstellungen

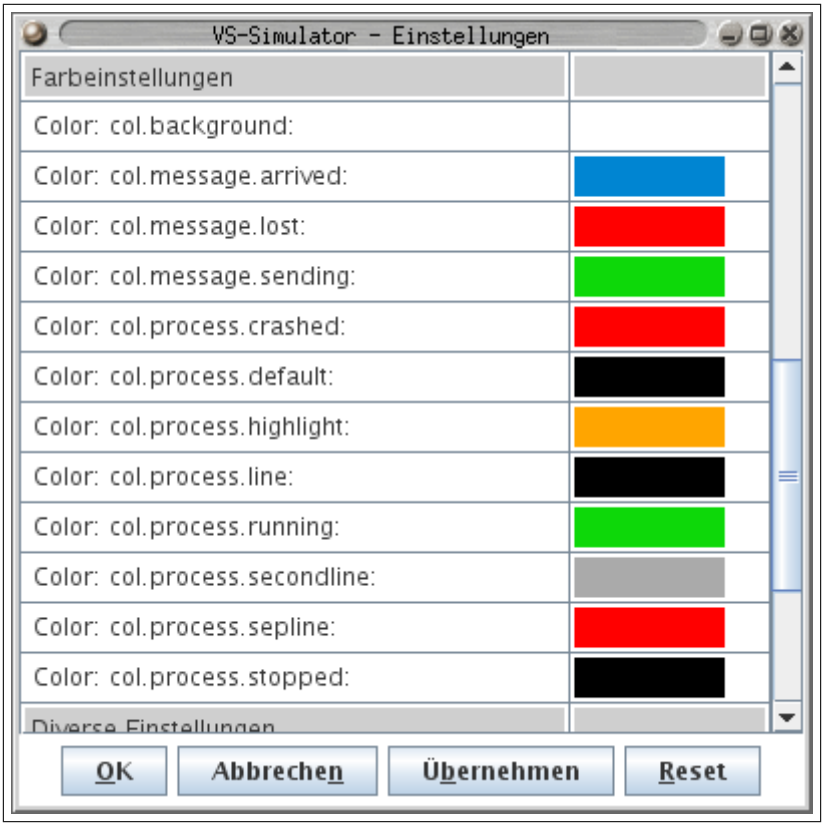

Abbildung 2.14.: Weitere Simulationseinstellungen im Expertenmodus

Nachrichten emfpangen und auf diese ebenso antworten. Die Zeit für das Versenden und Empfangen einer Nachricht an sich selbst beträgt jedoch stets *0ms*. Diese Variable sollte mit Vorsicht verwendet werden, da hierdurch, bedingt aus den *0ms*, Endlosschleifen entstehen können.

- **Mittelwerte der Nachrichtenverlustwahrscheinlichkeiten bilden** *(Boolean, true)*: Jede Nachricht die verschickt wird hat, je nach Einstellungen, eine vom verschickenden Prozess abhängige zufällige Übertragungszeit bis sie ihr Ziel erreicht. Wenn diese Option aktiviert ist, so wird der Mittelwert vom Sende- und Empfangsprozess gebildet. Ansonsten wird stets die Übertragungszeit, die beim Senderprozesses angegeben wurde, verwendet.

- **Nur relevante Nachrichten anzeigen** *(Boolean: true)*: Wenn nur alle relevanten Nachrichten angezeigt werden, so werden Nachrichten an einen Prozess die er selbst nicht verarbeiten kann, weil er das dazugehörige Protokoll nicht unterstützt, nicht angezeigt. Hierdurch wird eine Simulation viel übersichtlicher dargestellt.

- **Expertenmodus aktivieren** *(Boolean, false)*: Hier lässt sich der Expertenmodus aktivieren beziehungsweise deaktivieren. Alternativ kann dies über die gleichnamige Checkbox unterhalb des Loggfensters geschehen.

- **Simulation periodisch wiederholen** *(Boolean: false)*: Wenn diese Variable auf *true* gesetzt ist, so wird die Simulation jedes Mal nach Ablauf automatisch erneut gestartet.

- **Lamportzeiten betreffen alle Ereignisse** *(Boolean: false)*: Wenn diese Variable auf *true* gesetzt ist, so werden bei jedem Ereignis alle Lamportzeitstempel aller Prozesse jeweils inkrementiert. Bei einem Wert *false* inkrementieren sich die Lamportzeitstempel jeweils nur, wenn eine Nachricht empfangen oder verschickt wurde.

- **Vektorzeiten betreffen alle Ereignisse** *(Boolean: false)*: Wenn diese Variable auf *true* gesetzt ist, so werden bei jedem Ereignis alle Vektor-Zeitstempel aller Prozesse jeweils inkrementiert. Bei einem Wert *false* inkrementieren sich die Vektor-Zeitstempel jeweils nur, wenn eine Nachricht empfangen oder verschickt wurde.

- **Abspielgeschwindigkeit der Simulation** *(Float: 0.5)*: Gibt den Faktor der Simulationsabspielgeschindigkeit an. Wenn als Faktor *1* gewählt wird, dann dauert eine simulierte Sekunde so lange wie eine echte Sekunde. Der Faktor *0.5* gibt somit an, dass die Simulation mit halber Echtzeitgschwindigkeit abgespielt wird.

- **Anzahl der Prozesse** *(Integer: 3)*: Gibt an wieviele Prozesse an der Simulation teilnehmen sollen. Der Anwender kann auch nachträglich via Rechtsklick auf den Prozessbalken den jeweiligen Prozess aus der Simulation entfernen oder weitere Prozesse hinzufügen.

- **Dauer der Simulation** *(Integer: 15)*: Gibt die Dauer der Simulation in Sekunden an.

Die weiteren Simulationseinstellungen unter “Einstellungen für neue Prozesse” sowie “Nachrichteneinstellungen für neue Prozesse” geben lediglich Standardwerte an, die für neu zu erstellende Prozesse verwendet werden. Die dort verfügbaren Variablen werden im folgenden Teilkapitel genauer beschrieben.

2.4.3. Prozess- und Protokolleinstellungen

Jeder Prozess besitzt folgende Variablen, die entweder via dem Variablen-Tab in der Sidebar oder “Editieren → Prozess *PID*” oder Linksklick auf den Prozessbalken editiert werden können. Auf allen drei Wegen kommt jeweils der selbe Prozesseditor zum Vorschein.

- **Uhrabweichung** *(Float: 0.0)*: Gibt den Faktor an, um den die lokale Prozessuhr abweicht. Der Faktor *0.0* besagt beispielsweise, dass die Uhr keine Abweichung hat und somit global-korrekt läuft. Ein Faktor von *1.0* würde hingegen bedeuten, dass die Uhr mit doppelter Geschwindigkeit- und ein Faktor von *-0.5*, dass die lokale Prozessuhr mit halber Geschwindigkeit der globalen Uhr fortschreitet. Es sind nur Werte > *-1.0* erlaubt, da sonst die Prozessuhr rückwärts laufen könnte. Bei allen anderen Werten wird der Faktor wieder automatisch auf *0.0* gesetzt. Da der Simulator intern mit Fließkommazahlen doppelter Genauigkeit arbeitet, kann es zu kleinen, jedoch vernachlässigbaren, Rundungsfehlern kommen.

- **Prozessausfallwahrscheinlichkeit** *(Integer: 0)*: Gibt eine Wahrscheinlichkeit in Prozent an, ob der gegebene Prozess während der Simulation zufällig abstürzt. Die Wahrscheinlichkeit bezieht sich auf die komplette Simulationsdauer. Bei einer Einstellung von *100* Prozent und der Simulationsdauer von *15* Sekunden stürzt der Prozess auf jeden Fall zwischen *0ms* und *15000ms* ab. An welcher Stelle dies geschieht wird zufällig bestimmt. Wenn der Prozess nach seinem Absturz wiederbelebt wird, stürzt er nicht noch einmal zufällig ab. Dies gilt allerdings nicht, wenn die Prozesseinstellungen nach dem Zufallsabsturz erneut geändert und übernommen werden, da dann das Zufallsabstürzereignis erneut erstellt wird.

- **Lokale Zeit** *(Long: 0)*: Gibt die lokale Prozesszeit in Millisekunden an.

- **Nachrichtenverlustwahrscheinlichkeit** *(Integer: 0)*: Gibt eine Wahrscheinlichkeit in Prozent an, ob eine vom aktuell ausgewählten Prozess verschickte Nachricht unterwegs verloren geht. An welcher Stelle die Nachricht zwischen dem Sende- und Empfangsprozess verloren geht wird vom Simulator zufällig gewählt.

- **Maximale Übertragungszeit** *(Long: 2000)*: Gibt die Dauer in Millisekunden an, die eine vom Prozess verschickte Nachricht maximal benötigt, bis sie einen Empfängerprozess erreicht. Im weiteren Verlauf wird dieser Wert mit t_{max} bezeichnet. Der tatsächlich verwendete Wert wird zufällig zwischen der minimalen- und der maximalen Zeit (jeweils inklusive) gewählt.

Schlüssel	Beschreibung
col.background	Die Hintergrundfarbe der Simulation
col.message.arrived	Nachrichtenfarbe wenn sie ihr Ziel erreicht hat
col.message.lost	Nachrichtenfarbe wenn sie verloren ging
col.message.sending	Nachrichtenfarbe wenn sie noch unterwegs ist
col.process.crashed	Prozessfarbe wenn er abgestürzt ist
col.process.default	Prozessfarbe wenn die Simulation aktuell nicht läuft und der Prozess aktuell nicht abgestürzt ist
col.process.highlight	Prozessfarbe wenn die Maus über seinem Balken liegt
col.process.line	Farbe, in der die kleine "Prozessfane" an der auch die lokale Prozesszeit angegeben wird, dargestellt wird
col.process.running	Prozessfarbe wenn er nicht abgestürzt ist und die Simulation aktuell läuft
col.process.secondline	Farbe in der die Sekunden-Zeitgitter dargestellt werden
col.process.sepline	Farbe der globalen Zeitachse
col.process.stopped	Prozessfarbe wenn die Simulation pausiert wurde

Tabelle 2.3.: Farbeinstellungen

- **Minimale Übertragungszeit** *(Long: 500)*: Gibt die Dauer in Millisekunden an, die eine vom Prozess verschickte Nachricht minimal benötigt, bis sie einen Empfängerprozess erreicht. Im weiteren Verlauf wird dieser Wert mit t_{min} bezeichnet. Der tatsächlich verwendete Wert wird zufällig zwischen der minimalen- und der maximalen Zeit (jeweils inklusive) gewählt.

  Wenn die Übertragungszeit einer Nachricht immer exakt die selbe Zeit in Anspruch nehmen soll, dann müssen die Prozesseinstellungen mit $t_{min} = t_{max}$ konfiguriert werden.

Im selben Fenster lassen sich auch die Protokollvariablen editieren. Die Protokollvariablen werden jedoch später bei den Protokollen beschrieben.

2.4.4. Einstellungen im Expertenmodus

Im Expertenmodus lassen sich zusätzliche Variablen, wie beispielsweise diverse Farbwerte und Anzahl oder Pixel verschiedener der GUI-Elemente, editieren. Auf Abbildung 2.14 sieht der Anwender alle einstellbaren Farben. Die fett-gedruckten Schlüssel in Tabelle 2.3 dienen nur als Standardwerte für die neu zu erstellenen Prozesse und sind auch jeweils in den Prozesseinstellungen für jeden Prozess separat editierbar.

2.5. Protokolle

Im Folgenden werden alle verfügbaren Protokolle behandelt. Wie bereits beschrieben wird bei Protokollen zwischen Server- und Clientseite unterschieden. Server können auf Clientnachrichten, und Client auf Servernachrichten antworten. Jeder Prozess kann beliebig viele Protokolle sowohl clientseitig als auch serverseitig unterstützen. Theoretisch ist es auch möglich, dass ein Prozess für ein bestimmtes Protokoll gleichzeitig der Server und der Client ist. Der Anwender kann auch weitere eigene Protokolle in der Programmiersprache Java mittels einer speziellen API (Application Programming Interface) erstellen. Wie eigene Protokolle erstellt werden können wird später behandelt.

2.5.1. Beispiel (Dummy) Protokoll

Das Dummy-Protokoll dient lediglich als leeres Template für die Erstellung eigener Protokolle. Bei der Verwendung des Dummy-Protokolls werden bei Ereignissen lediglich Loggnachrichten ausgegeben. Es werden aber keine weiteren Aktionen ausgeführt.

2.5.2. Das Ping-Pong Protokoll

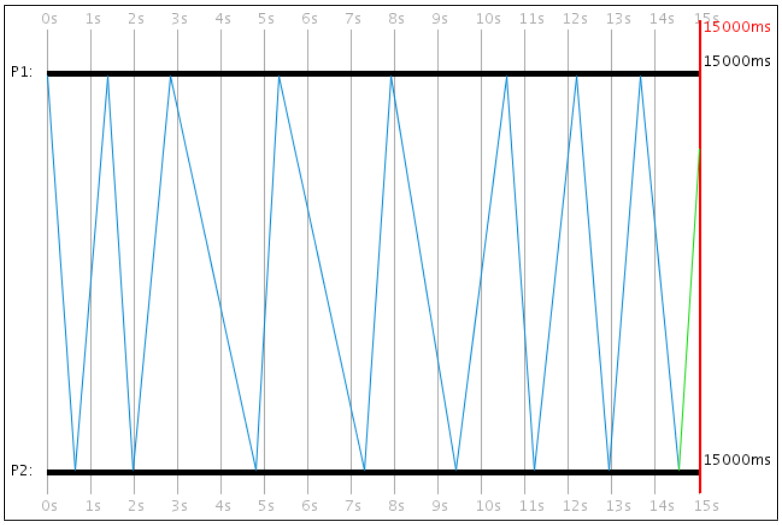

Abbildung 2.15.: Das Ping-Pong Protokoll

Bei dem Ping-Pong Protokoll (Abbildung 2.15) werden zwischen zwei Prozessen, Client P1 und Server P2, ständig Nachrichten hin- und hergeschickt. Der Ping-Pong Client startet die erste Anfrage, worauf der Server dem Client antwortet. Auf diese Antwort wird vom Client ebenfalls geantwortet und so weiter. Jeder Nachricht wird ein Zähler mitgeschickt, der bei jeder Station um eins inkrementiert- und jeweils im Loggfenster protokolliert wird. In Tabelle

Zeit (ms)	PID	Ereignis
0	1	Ping Pong Client aktivieren
0	2	Ping Pong Server aktivieren
0	1	Ping Pong Clientanfrage starten

Tabelle 2.4.: Programmierte Ping-Pong Ereignisse

2.4 sind alle für dieses Beispiel programmierten Ereignisse aufgeführt. Wichtig ist, dass Prozess 1 seinen Ping-Pong Client aktiviert, bevor er eine Ping-Pong Clientanfrage startet! Wenn die Eintrittszeiten für die Aktivierung des Protokolls und das Starten der Anfrage identisch sind, so ordnet der Task-Manager (mehr dazu später) diese Ereignisse automatisch in der richtigen Reihenfolge an. Wenn der Ping-Pong Client nicht aktiviert werden würde, dann könnte P1 auch keine Ping-Pong Anfrage starten. Bevor ein Prozess eine Anfrage starten kann, muss er das dazugehörige Protokoll unterstützen beziehungsweise aktiviert haben. Dies gilt natürlich für alle anderen Protokolle analog. Anhand diesem Beispiel ist erkennbar, dass die noch nicht ausgelieferte Nachrichten grün eingefärbt ist. Alle ausgelieferten Nachrichten tragen bereits die Farbe Blau.

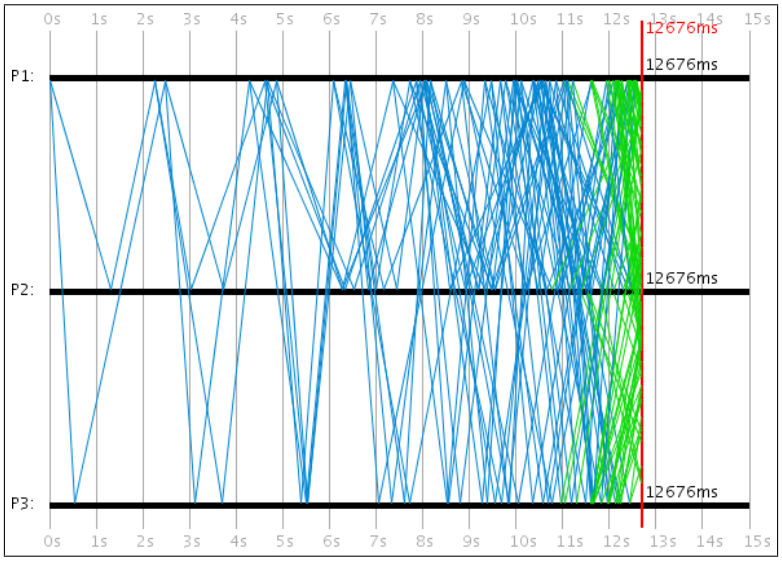

Abbildung 2.16.: Das Ping-Pong Protokoll (Sturm)

Werden die Ereignisse wie in Tabelle 2.5 abgeändert, so lässt sich ein Ping-Pong Sturm realisieren. Dort wurde ein neuer Prozess 3 eingeführt, der als zusätzlicher Ping-Pong Server agiert. Da auf jede Clientnachricht stets zwei Serverantworten folgen, verdoppelt sich bei jedem Ping-Pong Durchgang die Anzahl der kursierenden Nachrichten. Auf Abbildung 2.16 ist der dazugehörige Simulationsverlauf bis zum Zeitpunkt *12676ms* dargestellt.

Zeit (ms)	PID	Ereignis
0	1	Ping Pong Client aktivieren
0	2	Ping Pong Server aktivieren
0	3	Ping Pong Server aktivieren
0	1	Ping Pong Clientanfrage starten

Tabelle 2.5.: Programmierte Ping-Pong Ereignisse (Sturm)

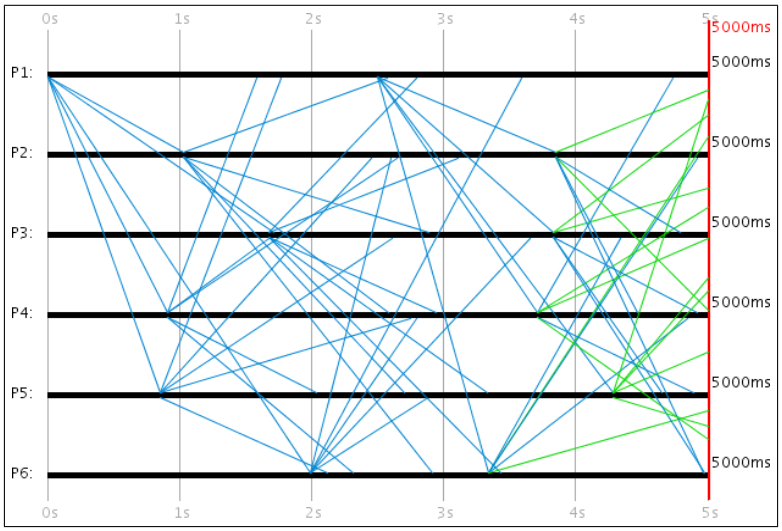

Abbildung 2.17.: Das Broadcast Protokoll

Zeit (ms)	PID	Ereignis
0000	1	Broadcast Client aktivieren
0000	2	Broadcast Client aktivieren
0000	3	Broadcast Client aktivieren
0000	4	Broadcast Client aktivieren
0000	5	Broadcast Client aktivieren
0000	6	Broadcast Client aktivieren
0000	1	Broadcast Server aktivieren
0000	2	Broadcast Server aktivieren
0000	3	Broadcast Server aktivieren
0000	4	Broadcast Server aktivieren
0000	5	Broadcast Server aktivieren
0000	6	Broadcast Server aktivieren
0000	1	Broadcast Clientanfrage starten
2500	1	Broadcast Clientanfrage starten

Tabelle 2.6.: Programmierte Broadcast Ereignisse

2.5.3. Das Broadcast Protokoll

Das Broadcast Protokoll verhält sich ähnlich wie das Ping-Pong Protokoll. Der Unterschied besteht darin, dass sich das Protokoll anhand einer eindeutigen Broadcast-ID merkt, welche Nachrichten bereits verschickt wurden. Das Broadcast Protokoll (server- und clientseitig) verschickt alle erhaltenen Nachrichten, sofern sie vom jeweiligen Prozess noch nicht schon einmal verschickt wurden, erneut. Der Server und der Client unterscheiden sich in diesem Fall nicht und führen bei Ankunft einer Nachricht jeweis die selben Aktionen durch. Somit lässt sich, unter Verwendung mehrerer Prozesse (hier 6), wie auf Abbildung 2.17, ein Broadcast erzeugen. P1 ist der Client und startet je eine Anfrage nach *0ms* und *2500ms*. Die Simulationsdauer beträgt hier genau *5000ms*. Da ein Client nur Servernachrichten und ein Server nur Clientnachrichten empfangen kann, ist in dieser Simulation jeder Prozess, wie in Tabelle 2.6 angegeben, gleichzeitig Server und Client.

2.5.4. Das Protokoll zur internen Synchronisierung in einem synchronen System

Bisher wurden nur Protokolle vorgeführt, in denen die beteiligten Prozesse keine Uhrabweichung eingestellt hatten. Das Protokoll zur internen Synchronisierung ist ein Protokoll zur Synchronisierung der lokalen Prozesszeit, welches beispielsweise angewendet werden kann, wenn eine Prozesszeit aufgrund einer Uhrabweichung falsch geht. Wenn der Client seine falsche lokale Zeit t_c mit einem Server synchronisieren möchte, so schickt er ihm eine Clientanfrage. Der Server schickt als Antwort seine eigene lokale Prozesszeit t_s zurück, womit der Client seine neue und genauere Prozesszeit berechnen kann. Wie genau die neue Prozesszeit berechnet wird, wird im

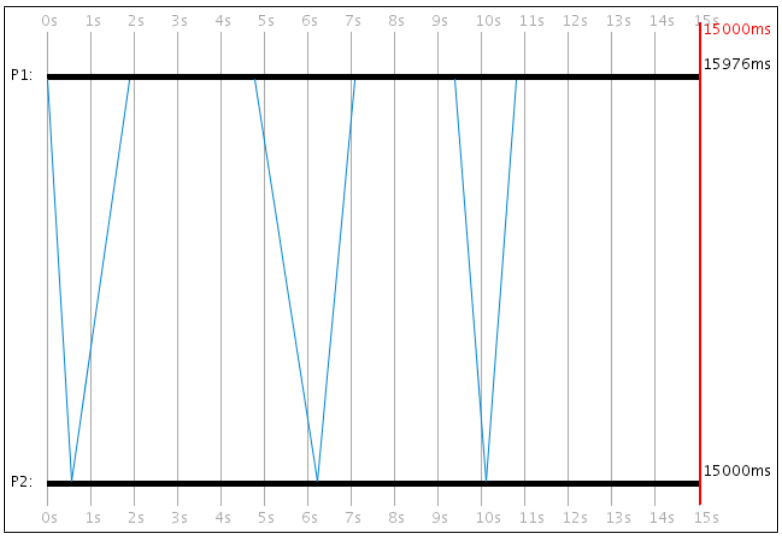

Abbildung 2.18.: Das Protokoll zur internen Synchronisierung

Zeit (ms)	PID	Ereignis
00000	1	Interne Sync. Client aktivieren
00000	2	Interne Sync. Server aktivieren
00000	1	Interne Sync. Clientanfrage starten
05000	1	Interne Sync. Clientanfrage starten
10000	1	Interne Sync. Clientanfrage starten

Tabelle 2.7.: Programmierte Ereignisse zur internen Synchronisierung

Folgenden beschrieben.

Hier (Abbildung 2.18) stellt P1 den Client und P2 den Server dar. Da die Übertragungszeit t_u einer Nachricht angenommen zwischen t'_{min} und t'_{max} liegt, setzt der Client P1 nach Empfang der Serverantwort seine lokale Prozesszeit auf

$$t_c := t_s + \frac{1}{2}(t'_{min} + t'_{max})$$

Somit wurde die lokale Zeit von P1, bis auf einen Fehler von $< \frac{1}{2}(t'_{max} - t'_{min})$, synchronisiert.

Der Clientprozess hat in der Abbildung 2.18 als Uhrabweichung den Wert *0.1* und der Server hat als Uhrabweichung den Wert *0.0* konfiguriert. Der Client startet, wie in Tabelle 2.7 angegeben, nach *0ms*, *5000ms* und *10000ms* seiner lokalen Prozesszeit jeweils eine Clientanfrage. In der Abbildung lässt sich erkennen, dass die 2. und die 3. Anfrage nicht synchron zu der globalen Zeit (siehe Sekunden-Gatter) gestartet werden, was auf die Uhrabweichung von P1 zurückzuführen ist. Nach Simulationsende ist die Zeit von P1 bis auf *15000ms - 15976ms = -976ms* synchronisiert.

Protokollvariablen

Dieses Protokoll verwendet folgende zwei clientseitige Variablen, die in den Prozesseinstellungen unter dem Punkt “Interne Sync. Client” konfiguriert werden können. Serverseitig gibt es hier keine Variablen.

- **Min. Übertragungszeit** *(Long: 500)*: Gibt den Wert t'_{min} in Millisekunden an
- **Max. Übertragungszeit** *(Long: 2000)*: Gibt den Wert t'_{max} in Millisekunden an

t'_{min} und t'_{max} sind die bei den Protokollberechnungen verwendeten Werte. Sie können sich allerdings von den tatsächlichen Nachrichtenübertragungszeiten t_{min} und t_{max} (siehe Sektion über Prozesseinstellungen) unterscheiden. Somit lassen sich auch Szenarien simulieren, in denen das Protokoll falsch eingestellt wurde und wo in der Zeitsynchronisierung größere Fehler auftreten können.

2.5.5. Christians Methode zur externen Synchronisierung

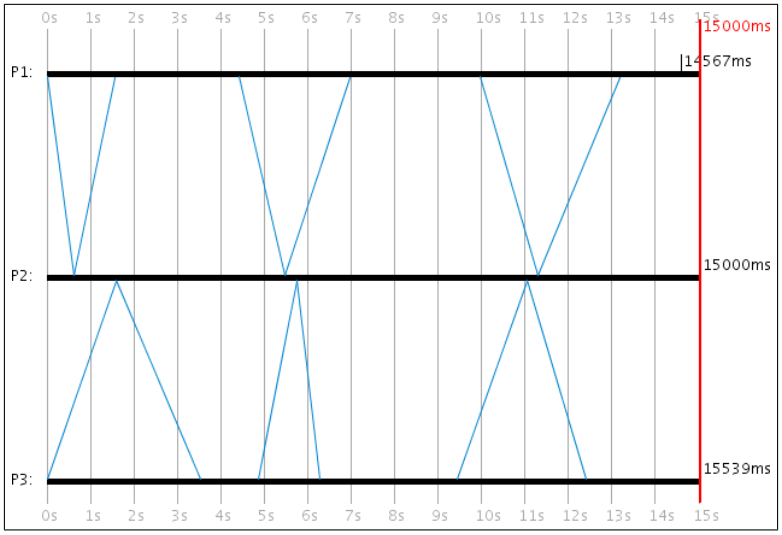

Abbildung 2.19.: Interne Synchronisierung und Christians Methode im Vergleich

Ein weiteres Protokoll für die Synchronisierung von Uhrzeiten funktioniert nach der Christians Methode zur externen Synchronisierung. Die Christians Methode benutzt die RTT (Round Trip Time) t_{rtt}, um die Übertragungszeiten von einzelnen Nachrichten zu approximieren.

Wenn der Client seine lokale Zeit t_c bei einem Server synchronisieren möchte, so verschickt er eine Anfrage, und misst dabei bis zur Ankunft der Serverantwort die dazugehörige RTT t_{rtt}. Die Serverantwort beinhaltet die lokale Prozesszeit t_s vom Server von dem Zeitpunkt, als der Server die Antwort verschickte. Der Client setzt dann seine lokale Zeit neu auf:

Zeit (ms)	PID	Ereignis
00000	1	Interne Sync. Client aktivieren
00000	1	Interne Sync. Clientanfrage starten
00000	2	Christians Server aktivieren
00000	2	Interne Sync. Server aktivieren
00000	3	Christians Client aktivieren
00000	3	Christians Clientanfrage starten
05000	1	Interne Sync. Clientanfrage starten
05000	3	Christians Clientanfrage starten
10000	1	Interne Sync. Clientanfrage starten
10000	3	Christians Clientanfrage starten

Tabelle 2.8.: Programmierte Ereignisse, Vergleich interne und externe Synchronisierung

$$t_c := t_s + \frac{1}{2} t_{rtt}$$

und zwar mit einer Genauigkeit von $\pm(\frac{1}{2}t_{rtt} - u_{min})$ wenn u_{min} eine Schranke für eine Nachrichtenübertragung mit $t_{rtt} < u_{min}$ ist (siehe [Oßm07]).

Im Prinzip sieht eine Christians-Simulation so aus wie auf Abbildung 2.18, daher wird hier auf eine einfache Abbildung vom Christians-Protokoll verzichtet. Viel Interessanter ist der direkte Vergleich zwischen dem Protokoll zur internen Synchronisierung und der Christians Methode der externen Synchronisierung (Abbildung 2.19). Hier stellt P1 den Client zur internen Synchronisierung und P3 den Client zur externen Synchronisierung dar. P2 fungiert für beide Protokolle gleichzeitig als Server. P1 und P3 starten jeweils zu den lokalen Prozesszeiten *0ms*, *5000ms* und *10000ms* eine Clientanfrage (Tabelle 2.8). P1 und P3 haben als Uhrabweichung *0.1* eingestellt und die Simulationsdauer beträgt insgesamt *15000ms*.

Auf der Abbildung 2.19 ist ablesbar, dass nach Ablauf der Simulation P1 seine Zeit bis auf *15000ms - 14567ms = 433ms* und P3 seine Zeit bis auf *15000ms - 15539ms = -539ms* synchronisiert hat. In diesem Beispiel hat also das Protokoll zur internen Synchronisierung ein besseres Ergebnis geliefert. Dies ist allerdings nicht zwingend immer der Fall, da nach einer erneuten Ausführung alle Nachrichten jeweils eine neue zufällige Übertragungszeit zwischen t_{min} und t_{max} haben werden, die auf das eine oder andere Protokoll wieder andere Auswirkungen haben können.

2.5.6. Der Berkeley Algorithmus zur internen Synchronisierung

Der Berkeley Algorithmus zur internen Synchronisierung ist eine weitere Möglichkeit lokale Uhrzeiten abzugleichen. Dies ist das erste Protokoll, wo der Server die Anfragen startet. Der Server stellt den Koordinator des

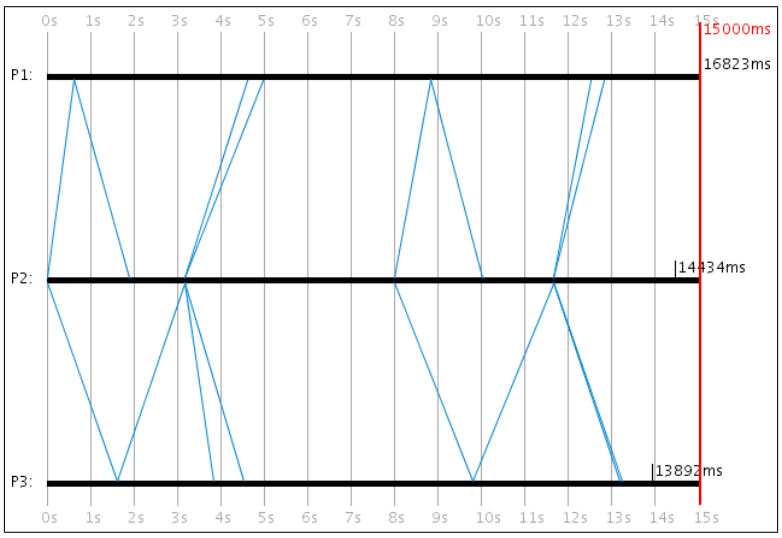

Abbildung 2.20.: Der Berkeley Algorithmus zur internen Synchronisierung

Zeit (ms)	PID	Ereignis
0000	1	Berkeley Client aktivieren
0000	2	Berkeley Server aktivieren
0000	3	Berkeley Client aktivieren
0000	2	Berkeley Serveranfrage starten
7500	2	Berkeley Serveranfrage starten

Tabelle 2.9.: Programmierte Ereignisse zum Berkeley Algorithmus

Protokolls dar. Die Clients sind somit passiv und müssen warten, bis eine Serveranfrage eintrifft. Hierbei muss der Server wissen, welche Clientprozesse an dem Protokoll teilnehmen, was sich in den Protokolleinstellungen des Servers einstellen lässt.

Wenn der Server seine eigene lokale Zeit t_s und auch die lokalen Prozesszeiten t_i der Clients ($i = 1, \ldots, n$) synchronisieren möchte, so verschickt er eine Serveranfrage. n sei hierbei die Anzahl beteiligter Clients. Die Clients senden dann ihre lokalen Prozesszeiten in einer Nachricht zurück zum Server. Der Server hat dabei die RTTs r_i bis zur Ankunft aller Clientantworten gemessen.

Nachdem alle Antworten vorliegen, setzt er zunächst seine eigene Zeit t_s auf den Mittelwert t_{avg} aller bekannten Prozesszeiten (seiner eigenen Prozesszeit eingeschlossen). Die Übertragungszeit einer Clientantwort wird auf die hälfte der RTT geschätzt und wird in der Berechnung berücksichtigt:

$$t_{avg} := \frac{1}{n+1}(t_s + \sum_{i=1}^{n} \frac{r_i}{2} + t_i)$$

$$t_s := t_{avg}$$

Anschließend berechnet der Server für jeden Client einen Korrekturwert $k_i := t_{avg} - t_i$, den er jeweils in einer separaten Nachricht zurückschickt. Die Clients setzten dann jeweils die lokale Prozesszeit auf $t'_i := t'_i + k_i$. Hierbei stellt t'_i die derzeit aktuelle Prozesszeit des jeweiligen Clients dar. Denn bis zum Eintreffen des Korrekturwertes ist inzwischen wieder neue Zeit verstrichen.

Im Beispiel auf Abbildung 2.20 gibt es die 2 Clientprozesse P1 und P3 sowie den Serverprozess P2. Der Server startet nach jeweils *0ms* und *7500ms* eine Synchronisierungsanfrage (Tabelle 2.9). Hier fällt auf, dass der Server stets 2 Korrekturwerte verschickt, die jeweils P1 und P3 erreichen. Es werden hier also pro Synchronisierungsvorgang insgesamt 4 Korrekturwerte ausgeliefert. Eine Korrekturnachricht enthält neben dem Korrekturwert k_i auch die PID des Prozesses, für den die Nachricht bestimmt ist. Indem das Protokoll die PID überprfüft verarbeitet ein Client so nur die für ihn bestimmten Korrekturwerte.

Protokollvariablen

Dieses Protokoll verwendet folgende serverseitige Variable, die in den Prozesseinstellungen unter dem Punkt “Berkeley Server” konfiguriert werden kann. Clientseitig gibt es hier keine Variablen.

- **PIDs beteiliger Prozesse** *(Integer[]: [1,3])*: Dieser Vektor aus Integerwerten beinhaltet alle PIDs der Berkeley Clientprozesse, mit denen der Berkeley Server die Zeit synchronisieren soll. Das Protokoll funktioniert nicht wenn hier eine PID angegeben wird die gar nicht existiert oder nicht das Berkeley Protokoll clientseitig unterstütz. In diesem Fall würde ewig auf eine fehlende Clientantwort gewartet werden.

2.5.7. Das Ein-Phasen Commit Protokoll

Das Ein-Phasen Commit Protokoll ist dafür gedacht beliebig vielen Clients zu einer Festschreibung zu bewegen. Im realen Leben könnte dies beispielsweise das Erstellen oder Löschen einer Datei sein, von der auf jedem Client eine lokale Kopie existiert. Der Server ist der Koordinator und auch derjenige, der einen Festschreibewunsch initiiert. Hierbei verschickt der Server periodisch so oft den Festschreibewunsch, bis er von jedem Client eine Bestätigung erhalten hat. Der Server muss dabei die PIDs aller beteiligten Clientprozesse sowie einen Wecker für erneutes Versenden des Festschreibewunsches eingestellt bekommen.

Die programmierten Ereignisse des Beispiels auf Abbildung 2.21 sind in Tabelle 2.10 aufgelistet. P1 und P3 simulieren jeweils einen Client und P2 den Server. Damit die Simulation mehrere Festschreibewünsche verschickt, stürzt in der Simulation P1 nach *1000ms* ab und nach *5000ms* steht er wieder zur Verfügung. Die ersten beide Festschreibewünsche erreichen dadurch P1 nicht und erst der dritte Versuch verläuft erfolgreich. Bevor die Bestätigung von P1 bei P2 eintrifft, läuft jedoch der Wecker erneut ab, so dass ein weiterer Festschreibewunsch

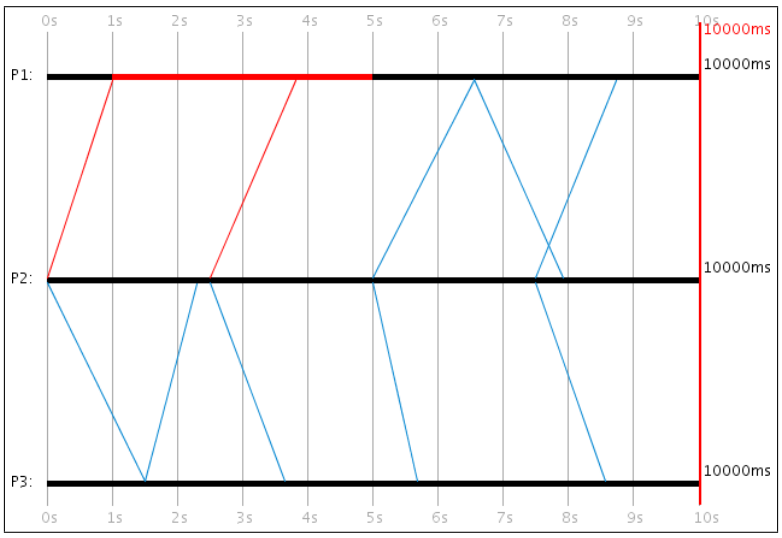

Abbildung 2.21.: Das Ein-Phasen Commit Protokoll

Zeit (ms)	PID	Ereignis
0000	1	1-Phasen Commit Client aktivieren
0000	2	1-Phasen Commit Server aktivieren
0000	3	1-Phasen Commit Client aktivieren
0000	2	1-Phasen Commit Serveranfrage starten
1000	1	Prozessabsturz
5000	1	Prozesswiederbelebung

Tabelle 2.10.: Programmierte Ein-Phasen Commit Ereignisse

versendet wird. Da P1 und P3 jeweils schon eine Bestätigung verschickt haben, wird diese Festschreibewunschnachricht ignoriert. Jeder Client bestätigt auf einen Festschreibewunsch nur ein einziges Mal.

Protokollvariablen

Dieses Protokoll verwendet folgende serverseitige Variablen, die in den Prozesseinstellungen unter dem Punkt “1-Phasen Commit Server” konfiguriert werden können. Clientseitig gibt es hier keine Variablen.

- **Zeit bis erneute Anfrage** *(Long: 2500)*: Gibt die Anzahl von Millisekunden an, die gewartet werden sollen, bis der Festschreibewunsch erneut verschickt wird.

- **PIDs beteiliger Prozesse** *(Integer[]: [1,3])*: Dieser Vektor aus Integerwerten beinhaltet alle PIDs der Clientprozesse, die festschreiben sollen.

Zeit (ms)	PID	Ereignis
0000	1	2-Phasen Commit Client aktivieren
0000	2	2-Phasen Commit Server aktivieren
0000	3	2-Phasen Commit Client aktivieren
0000	2	2-Phasen Commit Serveranfrage starten

Tabelle 2.11.: Programmierte Zwei-Phasen Commit Ereignisse

2.5.8. Das Zwei-Phasen Commit Protokoll

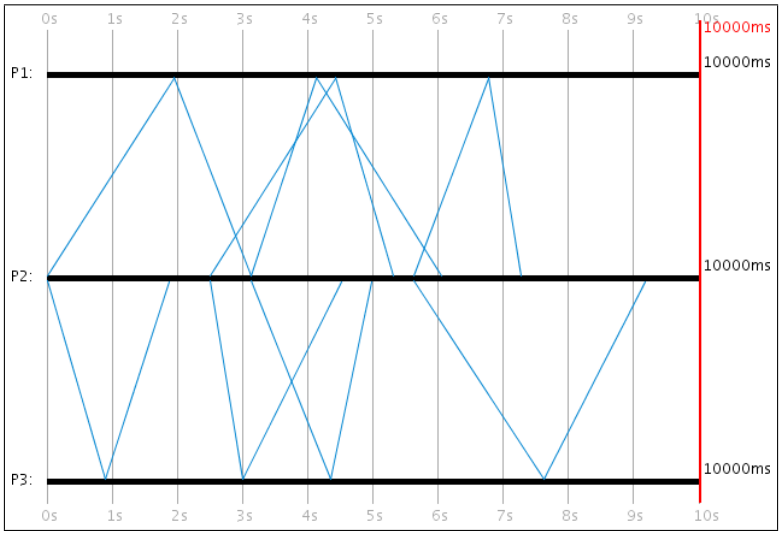

Abbildung 2.22.: Das Zwei-Phasen Commit Protokoll

Das Zwei-Phasen Commit Protokoll ist eine Erweiterung des Ein-Phasen Commit Protokolls. Der Server startet zunächst eine Anfrage an alle beteiligten Clients, ob festgeschrieben werden soll. Jeder Client antwortet dann mit *true* oder *false*. Der Server fragt so oft periodisch nach, bis alle Ergebnisse aller Clients vorliegen. Nach Erhalt aller Abstimmungen überprüft der Server, ob alle mit *true* abgestimmt haben. Für den Fall dass mindestens ein Client mit *false* abgestimmt hat, wird der Festschreibevorgang abgebrochen und als globales Abstimmungsergebnis *false* verschickt. Wenn jedoch alle mit *true* abstimmten, soll festgeschrieben werden. Dabei wird das globale Abstimmungsergebnis *true* verschickt. Das globale Abstimmungsergebnis wird periodisch so oft erneut verschickt, bis von jedem Client eine Bestätigung des Erhalts vorliegt.

In dem Beispiel (Abbildung 2.22) sind P1 und P3 Clients und P2 der Server. Der Server verschickt nach *0ms* seine erste Anfrage (Tabelle 2.11). Da diese Simulation recht unübersichtlich ist, liegen in den Tabellen 2.12 und 2.13 Auszüge aus dem Loggfenster vor. Auf die Lamport- und Vektorzeitstempel sowie die lokalen Prozesszeiten wurde hier wegen Irrelevanz verzichtet. Da keine Uhrabweichungen konfiguriert wurden, sind die lokalen Prozesszeiten stets gleich der globalen Zeit und deswegen wird hier pro Loggeintrag jeweils nur eine Zeit angegeben. Anhand der

Nachrichten IDs lassen sich dort die einzelnen Sendungen zuordnen. In den Loggs wird auch ständig der Inhalt der verschickten Nachricht sowie die dazugehörigen Datentypen aufgeführt. Hier stimmen P1 und P3 jeweils mit *true*, d.h. es soll festgeschrieben werden, ab.

Protokollvariablen

Dieses Protokoll verwendet folgende serverseitige Variablen, die in den Prozesseinstellungen unter dem Punkt "2-Phasen Commit Server" konfiguriert werden können:

- **Zeit bis erneute Anfrage** *(Long: 2500)*: Gibt die Anzahl von Millisekunden an, die gewartet werden sollen, bis der Festschreibewunsch erneut verschickt wird.
- **PIDs beteiliger Prozesse** *(Integer[]: [1,3])*: Dieser Vektor aus Integerwerten beinhaltet alle PIDs der Clientprozesse, die über eine Festschreibung abstimmen, und anschließend gegebenenfalls festschreiben sollen.

Und folgende Clientvariable kann unter den Prozesseinstellungen unter dem Punkt "2-Phasen Commit Client" konfiguriert werden:

- **Festschreibwahrscheinlichkeit** *(Integer: 50)*: Gibt die Wahrscheinlichkeit in Prozent an, die der Client mit *true*, also für das Festschreiben, abstimmt.

2.5.9. Der ungenügende (Basic) Multicast

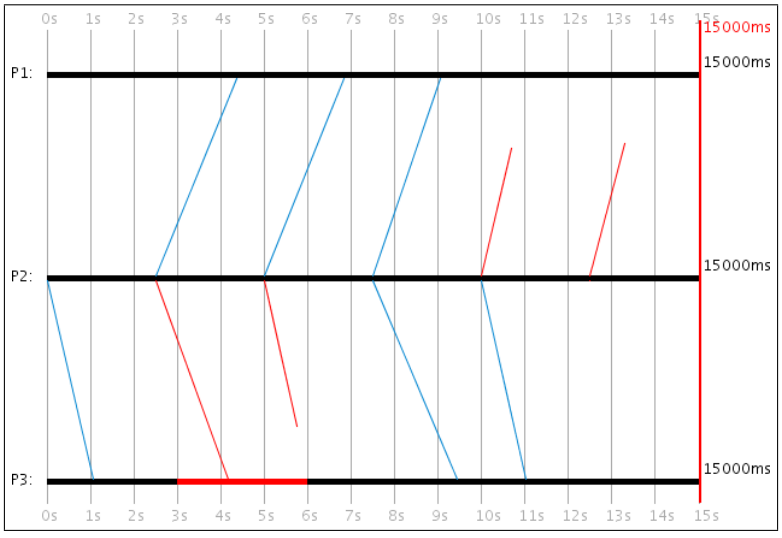

Abbildung 2.23.: Das Basic-Multicast Protokoll

Zeit (ms)	PID	Loggnachricht
000000		Simulation gestartet
000000	1	2-Phasen Commit Client aktiviert
000000	2	2-Phasen Commit Server aktiviert
000000	2	Nachricht versendet; ID: 94; Protokoll: 2-Phasen Commit Boolean: wantVote=true
000000	3	2-Phasen Commit Client aktiviert
000905	3	Nachricht erhalten; ID: 94; Protokoll: 2-Phasen Commit
000905	3	Nachricht versendet; ID: 95; Protokoll: 2-Phasen Commit Integer: pid=3; Boolean: isVote=true; vote=true
000905	3	Abstimmung true versendet
001880	2	Nachricht erhalten; ID: 95; Protokoll: 2-Phasen Commit
001880	2	Abstimmung von Prozess 3 erhalten! Ergebnis: true
001947	1	Nachricht erhalten; ID: 94; Protokoll: 2-Phasen Commit
001947	1	Nachricht versendet; ID: 96; Protokoll: 2-Phasen Commit Integer: pid=1; Boolean: isVote=true; vote=true
001947	1	Abstimmung true versendet
002500	2	Nachricht versendet; ID: 97; Protokoll: 2-Phasen Commit Boolean: wantVote=true
003006	3	Nachricht erhalten; ID: 97; Protokoll: 2-Phasen Commit
003006	3	Nachricht versendet; ID: 98; Protokoll: 2-Phasen Commit Integer: pid=3; Boolean: isVote=true; vote=true
003006	3	Abstimmung true versendet
003137	2	Nachricht erhalten; ID: 96; Protokoll: 2-Phasen Commit
003137	2	Abstimmung von Prozess 1 erhalten! Ergebnis: true
003137	2	Abstimmungen von allen beteiligten Prozessen erhalten! Globales Ergebnis: true
003137	2	Nachricht versendet; ID: 99; Protokoll: 2-Phasen Commit Boolean: isVoteResult=true; voteResult=true
004124	1	Nachricht erhalten; ID: 99; Protokoll: 2-Phasen Commit
004124	1	Globales Abstimmungsergebnis erhalten. Ergebnis: true
004124	1	Nachricht versendet; ID: 100; Protokoll: 2-Phasen Commit Integer: pid=1; Boolean: isAck=true
004354	3	Nachricht erhalten; ID: 99; Protokoll: 2-Phasen Commit
004354	3	Globales Abstimmungsergebnis erhalten. Ergebnis: true
004354	3	Nachricht versendet; ID: 101; Protokoll: 2-Phasen Commit Integer: pid=3; Boolean: isAck=true
004434	1	Nachricht erhalten; ID: 97; Protokoll: 2-Phasen Commit
004434	1	Nachricht versendet; ID: 102; Protokoll: 2-Phasen Commit Integer: pid=1; Boolean: isVote=true; vote=true

Tabelle 2.12.: Auszug aus der Loggausgabe des Zwei-Phasen Commit Beispiels

Zeit (ms)	PID	Loggnachricht
004434	1	Abstimmung true versendet
004527	2	Nachricht erhalten; ID: 98; Protokoll: 2-Phasen Commit
004975	2	Nachricht erhalten; ID: 101; Protokoll: 2-Phasen Commit
005311	2	Nachricht erhalten; ID: 102; Protokoll: 2-Phasen Commit
005637	2	Nachricht versendet; ID: 103; Protokoll: 2-Phasen Commit Boolean: isVoteResult=true; voteResult=true
006051	2	Nachricht erhalten; ID: 100; Protokoll: 2-Phasen Commit
006051	2	Alle Teilnehmer haben die Abstimmung erhalten
006766	1	Nachricht erhalten; ID: 103; Protokoll: 2-Phasen Commit
006766	1	Globales Abstimmungsergebnis erhalten. Ergebnis: true
006766	1	Nachricht versendet; ID: 104; Protokoll: 2-Phasen Commit Integer: pid=1; Boolean: isAck=true
007279	2	Nachricht erhalten; ID: 104; Protokoll: 2-Phasen Commit
007618	3	Nachricht erhalten; ID: 103; Protokoll: 2-Phasen Commit
007618	3	Globales Abstimmungsergebnis erhalten. Ergebnis: true
007618	3	Nachricht versendet; ID: 105; Protokoll: 2-Phasen Commit Integer: pid=3; Boolean: isAck=true
009170	2	Nachricht erhalten; ID: 105; Protokoll: 2-Phasen Commit
010000		Simulation beendet

Tabelle 2.13.: Auszug aus der Loggausgabe des Zwei-Phasen Commit Beispiels (2)

Zeit (ms)	PID	Ereignis
00000	2	Basic Multicast Client aktivieren
00000	3	Basic Multicast Server aktivieren
00000	2	Basic Multicast Clientanfrage starten
02500	1	Basic Multicast Server aktivieren
02500	2	Basic Multicast Clientanfrage starten
03000	3	Prozessabsturz
05000	2	Basic Multicast Clientanfrage starten
06000	3	Prozesswiederbelebung
07500	2	Basic Multicast Clientanfrage starten
10000	2	Basic Multicast Clientanfrage starten
12500	2	Basic Multicast Clientanfrage starten

Tabelle 2.14.: Programmierte Basic-Multicast Ereignisse

Das Basic-Multicast Protokoll ist sehr einfach aufgebaut. Im Beispiel auf Abbildung 2.23 sind P1 und P3 Server und P2 der Client. Bei diesem Protokoll startet der Client immer die Anfrage, welche bei diesem Protokoll eine einfache Multicast-Nachricht darstellen soll. Die Basic-Multicast Server dienen lediglich für den Empfang einer Nachricht. Es werden keine Bestätigungen verschickt. Wie in Tabelle 2.14 aufgeführt verschickt P2 alle *2500ms* jeweils eine Multicast-Nachricht, die alle voneinander völlig unabhängig sind.

P1 kann jedoch erst nach *2500ms* Multicast-Nachrichten empfangen, da er vorher das Protokoll nicht unterstützt während P3 von *3000ms* bis *6000ms* abgestürzt ist und in dieser Zeit auch keine Nachrichten empfangen kann. Je nach Interpretation könnte P1 einen Server simulieren, der erst später ans Netz angeschlossen wird. Da die Einstellung “Nur relevante Nachrichten anzeigen” aktiviert ist, wird die erste Multicast-Nachricht von P2 an P1 nicht dargestellt. Bei jedem Prozess wurde die Nachrichtenverlustwahrscheinlichkeit auf *30* Prozent gestellt, weswegen alle in dieser Simulation verschickten Nachrichten mit einer Wahrscheinlichkeit von *30* Prozent ausfallen.

In diesem Beispiel ging die 3. Multicast-Nachricht auf den Weg zu P3- und die 5. sowie 6. Nachricht auf den Weg zu P1 verloren. Lediglich die 4. Multicast-Nachricht hat alle beiden Ziele aufeinmal erreicht.

2.5.10. Der zuverlässige (Reliable) Multicast

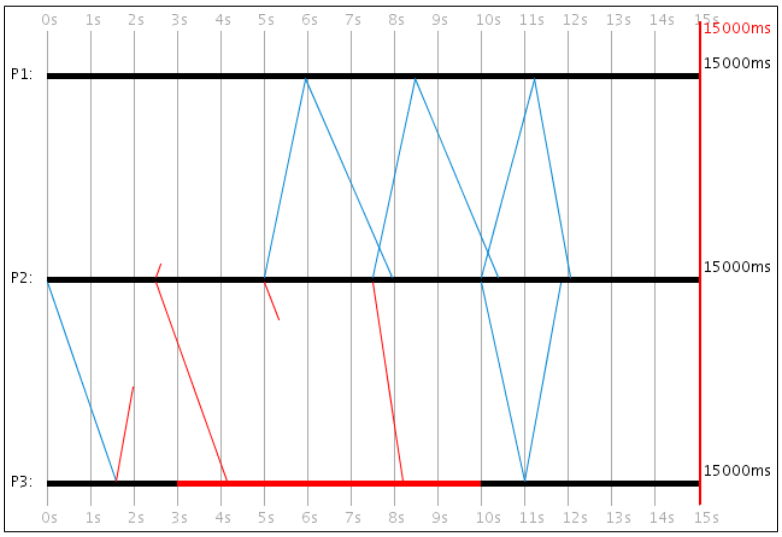

Abbildung 2.24.: Das Reliable-Multicast Protokoll

Bei dem zuverlässigen (Reliable) Multicast verschickt der Client so oft periodisch seine Multicast-Nachricht erneut, bis er von allen beteiligten Servern eine Bestätigung erhalten hat. Nach jedem erneuten Versuch vergisst der Client, von welchen Servern er bereits eine Bestätigung erhalten hat, womit jeder erneuter Versuch von allen Teilnehmern aufs Neue bestätigt werden muss. In dem Beispiel (Abbildung 2.24, Tabelle 2.15, sowie den Loggs in den Tabellen 2.16 und 2.17) ist P2 der Multicast-verschickende Client, während P1 und P3 die Server darstellen.

Zeit (ms)	PID	Ereignis
00000	3	Reliable Multicast Server aktivieren
00000	2	Reliable Multicast Client aktivieren
00000	2	Reliable Multicast Clientanfrage starten
02500	1	Reliable Multicast Server aktivieren
03000	3	Prozessabsturz
10000	3	Prozesswiederbelebung

Tabelle 2.15.: Programmierte Reliable-Multicast Ereignisse

Bei *0ms* initiiert der Client seine Multicast-Nachricht. Die Nachrichtenverlustwahrscheinlichkeiten sind bei allen Prozessen auf *30* Prozent eingestellt.

In diesem Beispiel benötigt der Client bis zur erfolgreichen Auslieferung des zuverlässigen Multicasts genau 5 Versuche:

1. Versuch:
   - P1 unterstützt das Reliable-Multicast Protokoll noch nicht, und kann somit weder Multicast-Nachricht erhalten noch eine Bestätigung verschicken.
   - P3 empfängt die Multicast-Nachricht, jedoch geht seine Bestätigungsnachricht verloren.
2. Versuch:
   - P1: Die Multicast-Nachricht geht unterwegs zu P1 verloren.
   - P3: Die Multicast-Nachricht erreicht P3, aber P3 ist abgestürzt und kann somit keine Nachricht verarbeiten.
3. Versuch:
   - P1 empfängt die Multicast-Nachricht und seine Bestätigung kommt wie geplant bei P2 an.
   - P3: Die Multicast-Nachricht geht unterwegs zu P3 verloren.
4. Versuch:
   - P1 empfängt die Multicast-Nachricht und seine Bestätigung kommt wie geplant bei P2 an.
   - P3: Die Multicast-Nachricht erreicht P3, aber P3 ist abgestürzt und kann somit keine Nachricht verarbeiten.
5. Versuch:
   - P1 empfängt die Multicast-Nachricht und seine Bestätigung kommt wie geplant bei P2 an.
   - P3 empfängt die Multicast-Nachricht und seine Bestätigung kommt wie geplant bei P2 an.

Zeit (ms)	PID	Loggnachricht
000000		Simulation gestartet
000000	2	Reliable Multicast Client aktiviert
000000	2	Nachricht versendet; ID: 280; Protokoll: Reliable Multicast; Boolean: isMulticast=true
000000	3	Reliable Multicast Server aktiviert
001590	3	Nachricht erhalten; ID: 280; Protokoll: Reliable Multicast
001590	3	Nachricht versendet; ID: 281; Protokoll: Reliable Multicast Integer: pid=3; Boolean: isAck=true
001590	3	ACK versendet
002500	1	Reliable Multicast Server aktiviert
002500	2	Nachricht versendet; ID: 282; Protokoll: Reliable Multicast Boolean: isMulticast=true
003000	3	Abgestürzt
005000	2	Nachricht versendet; ID: 283; Protokoll: Reliable Multicast Boolean: isMulticast=true
005952	1	Nachricht erhalten; ID: 283; Protokoll: Reliable Multicast
005952	1	Nachricht versendet; ID: 284; Protokoll: Reliable Multicast Integer: pid=1; Boolean: isAck=true
005952	1	ACK versendet
007500	2	Nachricht versendet; ID: 285; Protokoll: Reliable Multicast Boolean: isMulticast=true
007937	2	Nachricht erhalten; ID: 284; Protokoll: Reliable Multicast
007937	2	ACK von Prozess 1 erhalten!
008469	1	Nachricht erhalten; ID: 285; Protokoll: Reliable Multicast
008469	1	Nachricht versendet; ID: 286; Protokoll: Reliable Multicast Integer: pid=1; Boolean: isAck=true
008469	1	ACK erneut versendet
010000	2	Nachricht versendet; ID: 287; Protokoll: Reliable Multicast Boolean: isMulticast=true
010000	3	Wiederbelebt
010395	2	Nachricht erhalten; ID: 286; Protokoll: Reliable Multicast
010995	3	Nachricht erhalten; ID: 287; Protokoll: Reliable Multicast
010995	3	Nachricht versendet; ID: 288; Protokoll: Reliable Multicast Integer: pid=3; Boolean: isAck=true
010995	3	ACK erneut versendet
011213	1	Nachricht erhalten; ID: 287; Protokoll: Reliable Multicast
011213	1	Nachricht versendet; ID: 289; Protokoll: Reliable Multicast Integer: pid=1; Boolean: isAck=true

Tabelle 2.16.: Auszug aus der Loggausgabe des Reliable-Multicast Beispiels

Zeit (ms)	PID	Loggnachricht
011213	1	ACK erneut versendet
011813	2	Nachricht erhalten; ID: 288; Protokoll: Reliable Multicast
011813	2	ACK von Prozess 3 erhalten!
011813	2	ACKs von allen beteiligten Prozessen erhalten!
012047	2	Nachricht erhalten; ID: 289; Protokoll: Reliable Multicast
015000		Simulation beendet

Tabelle 2.17.: Auszug aus der Loggausgabe des Reliable-Multicast Beispiels (2)

Protokollvariablen

Dieses Protokoll verwendet folgende serverseitige Variablen, die in den Prozesseinstellungen unter dem Punkt "Reliable Multicast Server" konfiguriert werden können:

- **Zeit bis erneute Anfrage** *(Long: 2500)*: Gibt die Anzahl von Millisekunden an, die gewartet werden sollen, bis der Muticast erneut verschickt wird.
- **PIDs beteiliger Prozesse** *(Integer[]: [1,3])*: Dieser Vektor aus Integerwerten beinhaltet alle PIDs der Serverprozesse, die die Multicast-Nachricht erhalten sollen.

2.6. Weitere Beispiele

Bisher wurden alle verfügbaren Protokolle anhand von Beispielen aufgeführt. Mit dem Simulator lassen sich allerdings noch viel mehr Szenarien simulieren. Daher soll hier auf weitere Anwendungsbeispiele eingegangen werden.

2.6.1. Vektor- und Lamportzeitstempel

Die Vektor- und Lamportzeitstempel lassen sich sehr gut am bereits behandeltem Beispiel des Berkeley-Protokoll's demonstrieren. Nach Aktivierung der Lamportzeit-Checkbox erscheinen bei jedem Ereignis die zum jeweiligen Prozess gehörigen Lamportzeitstempel (Abbildung 2.25). Jeder Prozess besitzt einen eigenen Lamportzeitstempel, der bei jedem Versenden oder Erhalten einer Nachricht inkrementiert wird. Jeder Nachricht wird die aktuelle Lamportzeit $t_l(i)$ des sendenden Prozesses i beigefügt. Wenn ein weiterer Prozess j diese Nachricht erhält, so wird der aktuelle Lamportzeitstempel $t_l(j)$ von Prozess j wie folgt neu berechnet:

$$t_l(j) := 1 + max(t_l(j), t_l(i))$$

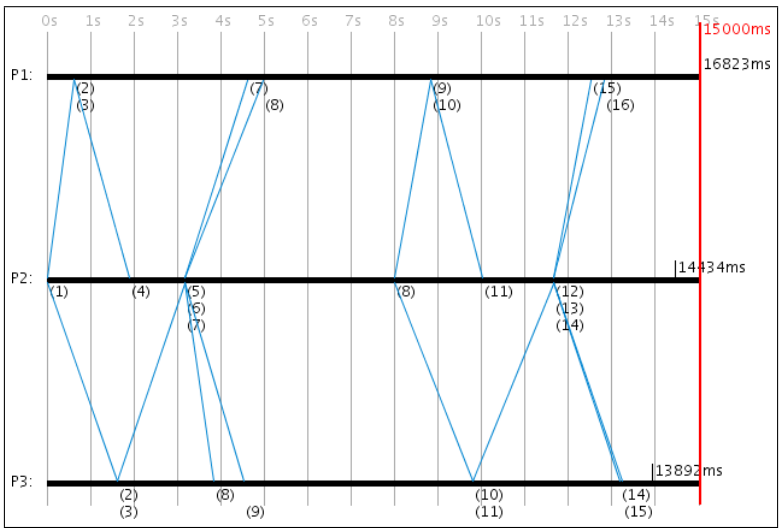

Abbildung 2.25.: Lamportzeitstempel

Es wird also stets die grössere Lamportzeit vom Sender- und Empfängerprozess verwendet und anschließend wird diese um *1* inkrementiert. Nach Ablauf der Berkeley-Simulation hat P1 *(16)*, P2 (*14*) und P3 (*15*) als Lamportzeitstempel abgespeichert.

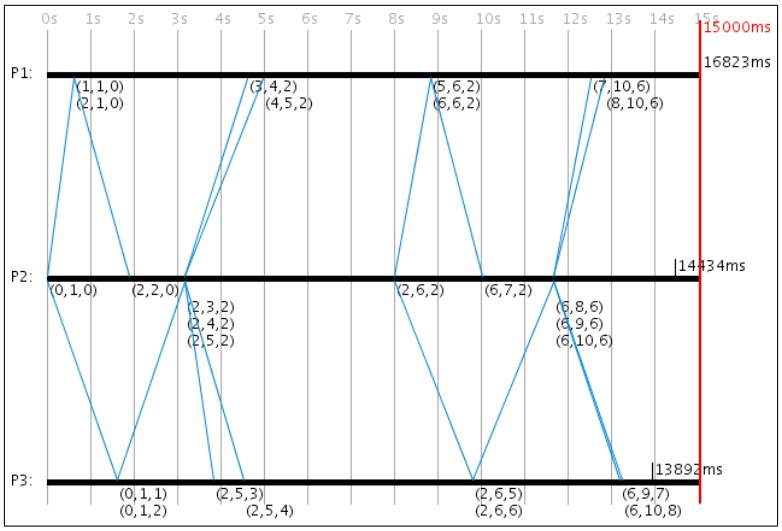

Abbildung 2.26.: Vektorzeitstempel

Mit aktivierter Vektorzeit-Checkbox werden, wie auf Abbildung 2.26, alle Vektor-Zeitstempel angezeigt. Wie bei den Lamportzeitstempeln wird auch hier jeder Nachricht der aktuelle Vektor-Zeitstempel des Senderprozesses beigegefügt. Bei n beteiligten Prozessen hat der Vektor-Zeitstempel v die größe n. Somit gibt es für jeden beteiligten Prozess i einen eigenen Index i. Über $v(i)$ kann jeder Prozess auf seine lokale logische Zeit zugreifen.

Standardmäßig wird der Vektor-Zeitstempel nur inkrementiert, wenn eine Nachricht verschickt- oder erhalten wird.

Bei beiden Fällen inkrementiert der Sender- beziehungsweise Senderprozess seinen eigenen Index im Vektor-Zeitstempel mit $v(i) = v(i) + 1$. Beim Empfang einer Nachricht wird anschließend der lokale Vektor-Zeitstempel mit dem des Senderprozesses verglichen und für alle Indizies stets der größere Wert in den lokalen Vektor-Zeitstempel übernommen.

Im Beispiel auf Abbildung 2.26 hat P1 *(8,10,6)*, P2 *(6,10,6)* und P3 *(6,10,8)* als Vektor-Zeitstempel abgespeichert.

Wenn im Laufe einer Simulation Prozesse entfernt- oder neue Prozesse hinzugefügt werden, so passt sich die Größe der Vektor-Zeitstempel aller anderen Prozesse automatisch der Anzahl der Prozesse an.

Wie bereits beschrieben gibt es in den Simulationseinstellungen die boolschen Variablen “Lamportzeiten betreffen alle Ereignisse” und “Vektorzeiten betreffen alle Ereignisse”, die standardmäßig auf *false* gesetzt sind. Mit *true* werden alle Ereignisse, und nicht nur der Empfang oder das Versenden einer Nachricht, berücksichtigt. Für eine weitere Betrachtung der Lamport- sowie Vektor-Zeitstempel siehe [Oßm07] oder [Tan03].

Kapitel 3.

Die Implementierung

In diesem Kapitel wird auf die Implementierung des Simulators eingegangen. Der Simulator wurde in der Programmiersprache Java entwickelt. Bei der Betrachtung der Zielgruppe wird klar, dass Java für die gestellte Aufgabe die geigneste Programmiersprache ist. Der Simulator ist somit auf jeder Platform verfügbar, für die es die JRE (Java Runtime Environment) gibt und erstreckt sich somit über alle gängigen Betriebssysteme. Da an der Fachhochschule Aachen auch Java gelehrt wird, sollten hier die meisten Studenten auch eigene Erweiterungen, wie eigene Protokolle, entwerfen können. Der Simulator wurde mit dem derzeit aktuellsten Java SDK (Software Development Kit) in der Version 6 (1.6) entwickelt.

Da es sonst den Rahmen sprengen würde, soll im Folgenden der komplette Quelltext nicht bis in das letzte Detail behandelt werden. Der Quelltext erstreckt sich nämlich, einschließlich Kommentare, auf über 15.000 Zeilen und über 60 Dateien. Zudem ist die generierte Quelltext-Dokumentation (Javadoc) über 2MB groß. Alle folgenden UML-Diagramme stellen aufgrund der Übersichtlichkeit lediglich die wesentlichen Dinge dar. Alle Details lassen sich im Quelltext und der dazugehörigen Dokumentation einsehen. Die Paketstruktur des Quelltextes ist in Tabelle 3.1 in alphanumerischer Reihenfolge aufgeführt.

3.1. Programmierrichtlinien

Die Programmierrichtlinien entsprechen in den meisten Fällen denen aus der Vorlesung [Fas06]. Die Main-Methode befindet sich in der Klasse *simulator.VSMain*.

- Alle Klassen- und Interfacenamen beginnen mit großen Buchstaben, während alle Variablen-, Methoden- und Attributnamen mit kleinen Buchstaben beginnen. Namen statischer Variablen und Attribute sind komplett in Großbuchstaben gehalten.
- Alle Quelltext-Dateien besitzen einen Header, der Informationen der verwendeten Lizenz angibt.

Paketname	Beschreibung
core	Klassen für Prozesse und Nachrichten
core.time	Klassen für Zeitformate
events	Basisklassen für Ereignisse
events.implementations	Implementierungen von Ereignissen
events.internal	Implementierungen von internen Ereignissen
exceptions	Klassen für Fehlerbehandlungen
prefs	Klassen für die Einstellungen
prefs.editors	Klassen für die Editoren
protocols	Basisklassen für Protokolle
protocols.implementations	Implementierungen von Protokollen
serialize	Helferklassen für die Serialisierung von Simulationen
simulator	Klassen für die GUI und die Visualisierung
utils	Diverse Helferklassen

Tabelle 3.1.: Die Paketstruktur

- Alle Quelltext-Dateien sind vollständig mit Javadoc dokumentiert worden.
- Der komplette Quelltext inklusive Dokumentation wurde in englischer Sprache verfasst.
- Eine Quelltext-Datei hat eine maximale Zeilenlänge von 80 Zeichen. Eine Ausnahme stellt die Klasse *prefs.VSDefaultPrefs* dar, denn hier befinden sich auch längere Texte die in Strings abgespeichert werden, wo manuelle Zeilenumbrüche wenig Sinn ergeben.
- Es werden zuerst Klassen aus der Java-Standardbibliothek importiert, bevor Klassen aus dem VS-Simulator selbst importiert werden.
- Für die Einrückung des Quelltextes wird das Tool *astyle* mit den Aufrufparametern *–style=java –mode=java* verwendet. Hierbei wird eine Einrückungslänge von 4 Zeichen verwendet.
- Namen abstrakter Klassen tragen stets das Prefix *VSAbstract*.
- Namen aller Klassen und Interfaces tragen als Prefix stets *VS*, was für Verteilte Systeme steht.

3.2. Einstellungen und Editoren

Eine Simulation ist von einer Vielzahl von Einstellungen abhängig. Da auf diese Einstellungen in den weiteren Teilkapitel sets zurückgegriffen wird, macht es Sinn die dazugehörigen Klassen zuerst zu betrachten.

Das Paket *prefs*

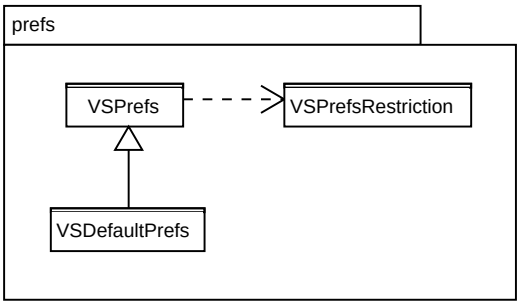

Abbildung 3.1.: Das Paket *prefs*

Auf Abbilung 3.1 ist der Aufbau des Pakets *prefs* zu sehen. In einer Instanz der Klasse *VSPrefs* lassen sich viele verschiedene Daten als Variablen für eine spätere Verwendung dynamisch ablegen und stellt somit einen Container für diese Daten dar. In einem *VSPrefs*-Objekt speichert der Simulator alle seine Einstellungen ab. Zudem besitzt jedes Prozessobjekt und jedes Protokollobjekt für lokale Einstellungen seine eigene Instanz von *VSPrefs*. Selbst Nachrichtenobjekte besitzt hiervon eine eigene Instanz, wobei hier die zu verschickenden Daten abgelegt werden können.

Jede Variable besteht aus einen Datentypen, einen Variablenamen und einer optionalen Beschreibung. Einige Datentypen unterstützen auch die Angabe von Minimum- und Maximumwerten (zum Beispiel besteht eine Prozentangabe aus einen Integerwert zwischen *0* und *100*), was mithilfe der *VSPrefsRestriction*-Klasse implementiert wird. Da man beispielsweise bei Prozent ein *%* und bei Millisekunden ein *ms* hinter der Variable sehen möchte, kann für jede Variable auch ein optionaler Einheiten-String abgespeichert werden. Alle verfügbaren Typen wurden bereits in Tabelle 2.2 aufgelistet. *VSPrefs* stellt für alle Variablen entsprechende Zugriffsmethoden zur Verfügung.

Im Folgenden werden nicht alle exisierenden Methoden aufgelistet, da diese auch in der Quelltext-Dokumentation (Javadoc) eingesehen werden können. Die Methoden werden nun nur anhand des Integer-Datentyps verdeutlicht. Für alle anderen Typen gilt fast alles analog. Für Integer stehen in *VSPrefs* folgende Methoden zur Verfügung:

- *void setInteger(String key, Integer val)*
- *void setInteger(String key, Integer val, String descr)*
- *void setInteger(String key, int val)*
- *void setInteger(String key, int val, String descr)*
- *Integer getIntegerObj(String key)*
- *int getInteger(String key)*

- *java.util.Set<String> getIntegerKeySet()*
- *void initInteger(String key, int val)*
- *void initInteger(String key, int val, String descr)*
- *void initInteger(String key, int val, String descr, int minValue, int maxValue)*
- *void initInteger(String key, int val, String descr, int minValue, int maxValue, String unit)*
- *void initInteger(String key, int val, String descr, VSPrefsRestriction.VSIntegerPrefsRestriction r)*
- *void initInteger(String key, int val, String descr, VSPrefsRestriction.VSIntegerPrefsRestriction r, String unit)*

Hierbei stellt *key* den Variablennamen- und *val* den Variablenwert dar. *descr* ist eine optionale Variablenbeschreibung. Wenn für eine Variable keine Beschreibung existiert so wird, wie auf Abbildung 2.14 anhand der Farbvariablen schon gesehen wurde, für die Anzeige für eine Variable der Datentyp und der Variablenname verwendet. Es können sowohl Java's Integer-Objekte, als auch Java's primitiver Integer-Typ *int* verwendet werden. Ein *int*-Wert wird intern allerdings als Integer-Objekt abgespeichert und macht somit keinen großen Unterschied. Die Methode *getIntegerKeySet* gibt alle vorhandenen Integer-Variablennamen (*key*s) als *Set* zurück.

VSPrefs bietet auch eine Reihe von *initInteger*-Methoden an, welche sich von den *setInteger*-Methoden dadurch unterscheiden, dass sie eine Variable nur einen Wert zuweisen, wenn sie vorher noch nicht initialisiert wurde, was durch *setInteger* oder *initInteger* selbst geschehen sein kann. Eine komplette Übersicht aller Methoden (auch für andere Datentypen) gibt es in der Quelltext-Dokumentation.

VSPrefs speichert alle Integervariablen in einem *HashMap<String,Integer>*-Objekt ab, wobei der String-Wert den Variablenamen *key* angibt. Für die Beschreibung *descr*, den Einheiten-String *unit* sowie möglichen Minimum- und Maximumwerte werden separate Instanzen von *HashMap* verwendet. Da alle *HashMap*-Objekte synchronisiert sind, können alle Methoden von verschiednenen Threads gleichzeitig verwendet werden.

Die Klasse *VSDefaultPrefs* erweitert *VSPrefs* und initialisiert bei Instanzierung automatisch alle verfügbaren Simulationsvariablen mit ihren Standardwerten. Dort sind auch alle Spracheinstellungen abgelegt. Sollte jemand den Simulator in eine andere Sprache, zum Beispiel ins Englische, übersetzen wollen, so muß er lediglich diese Datei und die Protokoll-Klassen (mehr dazu später) editieren. Die Spracheinstellungen sind nämlich in einem *VSPrefs*–Objekt als versteckte String-Variablen abgespeichert. Spracheinstellungen für Protokolle wurden in den Protokollklassen direkt angegeben, da dies mehr Komfort für den Protokollentwickler bietet und für jede neue Textausgabe nicht ständig *VSDefaultPrefs.java* editiert werden muss.

Alle Variablen die als Prefix *lang*, *keyevent*, *div* oder *col* tragen, sind versteckte Variablen und werden in einem Editor nicht angezeigt. Im Expertenmodus sind hingegen nur Variablen die mit *lang* und *keyevent* beginnen versteckt. Somit lassen sich im Expertenmodus weitere Variablen vom Anwender editieren.

Variablen-Prefix	Beschreibung	Beispiel
col	Farbvariablen	*Color: col.background = Color-Objekt*
div	Diverse versteckte Variablen	*Integer: div.window.loggsize = 300*
keyevent	Variablen, die Tastaturkürzel definieren	*Integer: keyevent.close = KeyEvent.VK_C*
lang	Variablen, die Text beinhalten	*String: lang.activate = aktivieren*
message	Variablen, die Nachrichten betreffen	*Integer: message.prob.outage = 0*
process	Variablen, die Prozesse betreffen	*Integer: process.prob.crash = 0*
sim	Allgemeine Simulationsvariablen	*Integer: sim.process.num = 3*

Tabelle 3.2.: Konventionen für Variablennamen-Prefixe in *VSDefautPrefs*

Das Paket *prefs.editors*

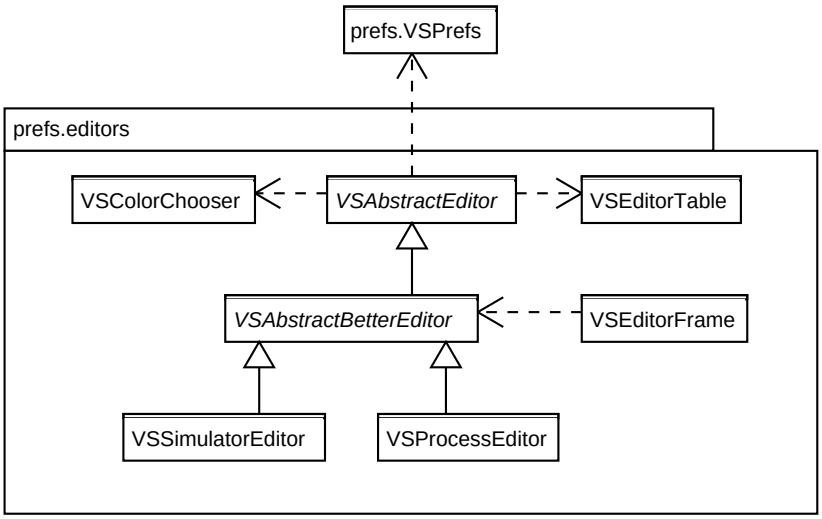

Abbildung 3.2.: Das Paket *prefs.editors*

Wie Variablen intern abgespeichert werden ist bereits bekannt. Für das Editieren der Variablen werden Editor-Objekte verwendet. Auf Abbildung 3.2 ist die Klassenstruktur des dazugehörigen Paketes *prefs.editors* angegeben.

Die Basis eines Editors stellt die abstrakte Klasse *VSAbstractEditor* dar, dem ein *VSPrefs* Objekt zum Editieren übergeben wird. Ein Editor stellt alle verfügbaren und nicht-versteckten Variablen des *VSPrefs*-Objektes im GUI dar und bietet gleichzeitig die Möglichkeit alle Variablen darüber zu editieren an. Für das Editieren von Farbwerten wird auf *VSColorChooser* zurückgegriffen. Die Klasse *VSEditorTable* ist für das *JTable*-Objekt aus Java's Swing-Bibliothek zuständig, welches bei der graphischen Darstellung aller Variablen eingesetzt wird. Die abstrakte Klasse *VSAbstractBetterEditor* wurde, wegen der Übersicht, als Zwischenschritt eingefügt.

Die Klasse *VSSimulatorEditor* dient für das Editieren der globalen Simulationseinstellungen und *VSProcessEditor*

für das Editieren der Prozesseinstellungen sowie der dazugehörigen Protokollvariablen. Da diese beiden Klassen von *VSAbstractBetterEditor* erben, können sie mithilfe von *VSEditorFrame* in einem separaten Fenster angezeigt werden. Alternativ können die Editoren auch in der Sidebar im Tab “Variablen” angezeigt werden. Auf Abbildung 2.13 wurde bereits ein *VSEditorFrame* in Aktion gesehen.

Für Protokolle gibt es keine separate Editor-Klasse, da sie bereits vom Prozesseditor aus editiert werden können. Dabei iteriert der Prozesseditor über alle für den jeweiligen Prozess verfügbaren Protokollobjekte und fügt deren Variablen zusätzlich in den Prozesseditor ein. Somit erscheinen die Prozess- und die dazugehörigen Protokollvariablen im selben Editor, womit dem Benutzer eine bessere Übersicht geboten wird.

3.3. Ereignisse

3.3.1. Die Funktionsweise von Ereignissen

Für jedes Ereignis existiert eine dazugehörige Klasse, welche die auszuführenden Aktionen implementiert. Eine Instanz davon wird, für eine spätere Ausführung, in einem *VSTask*-Objekt verpackt dem Task-Manager übergeben. Auf den Task-Manager wird später noch genauer eingegangen.

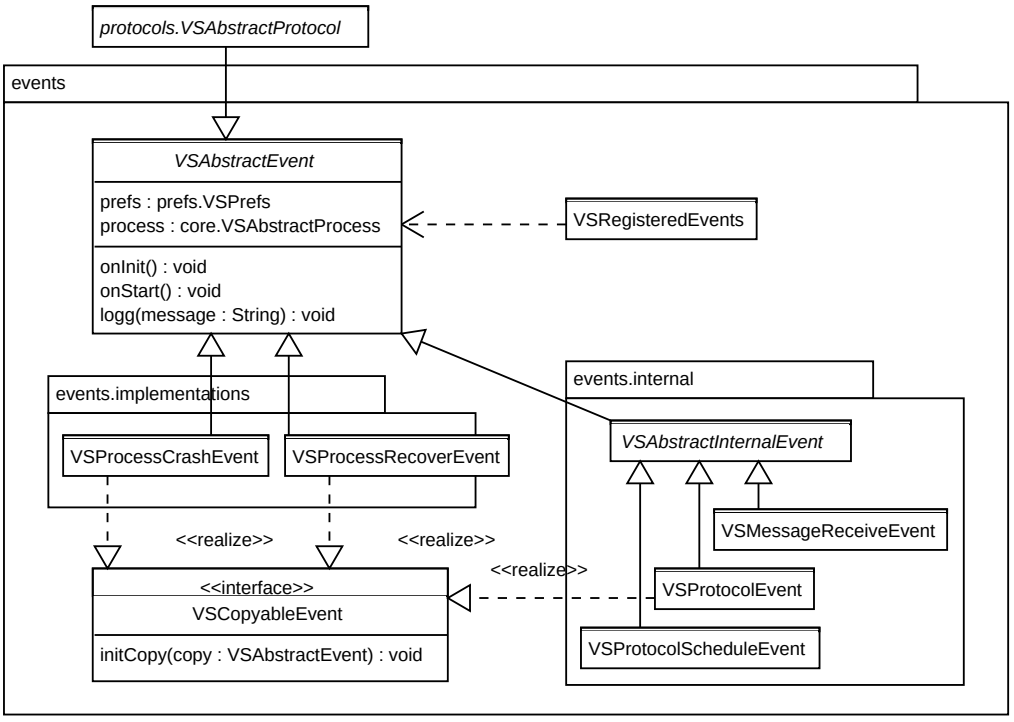

Abbildung 3.3.: Die Pakete *events* und *events.**

Jedes programmierbare Ereignis muß, bevor es vom Simulator verwendet werden kann, in der statischen Klasse *VSRegisteredEvents* registriert werden. Da sich die Anzahl der verfügbaren Ereignisse bei Laufzeit des Simulators nicht ändert, gibt es keine Instanzen von *VSRegisteredEvents*. Alle Methoden und Klassenattribute sind hier statisch. Wenn beispielsweise eigene Ereignisse implementiert werden, dann müssen alle neuen Ereignisse per Hand in die Datei *VSRegisteredEvents.java* übernommen- und der Simulator erneut kompiliert werden.

In der Implementierung wird zwischen drei Haupttypen von Ereignissen unterschieden, die jeweils in verschiedenen Paketen liegen:

1. *events.implementations*: In diesem Paket befinden sich alle Ereignisse, die ohne weitere Spezialbehanldung im Simulator eingesetzt werden können und vom Benutzer direkt im Ereigniseditor programmierbar sind.

    - *VSProcessCrashEvent*: Dieses Ereignis lässt den dazugehörigen Prozess abstürzen.
    - *VSProcessRecoverEvent*: Dieses Ereignis lässt den dazugehörigen Prozess wiederbeleben.

2. *events.internal*: In diesem Paket befinden sich alle Ereignisse, die vom Simulator intern verwendet werden und dadurch eine direkte Programmierung via Ereigniseditor ausschließen.

    - *VSAbstractInternalEvent*: Diese Klasse stellt weitere Methoden zur Verfügung, die von allen internen Ereignissen benötigt werden. Derzeit betrifft dies nur Methoden zur Serialisierung der gegebenen Objekte. Auf die Serialisierung (Abspeichern/Laden) von Simulationen wird später noch genauer eingegangen.
    - *VSMessageReceiveEvent*: Diese Klasse wird für die Ankunft einer Nachricht bei einem Empfängerprozess benötigt. Sie kapselt die eigentliche Nachricht und überprüft, ob der Empfängerprozess das zur Nachricht dazugehörige Protokoll versteht. Diese Klasse überprüft auch die Simulationseinstellung “Nur relevante Nachrichten anzeigen” und entscheidet, ob die Nachricht nach Eintreffen in der Visualisierung und im Loggfenster berücksichtigt werden soll oder nicht.
    - *VSProtocolEvent*: Diese Klasse implementiert gleichzeitig vier verschiedene Ereignisse: Das Aktivieren/Deaktivieren eines Servers/Clients eines gegebenen Protokolls. Der Ereigniseditor berechnet anhand der verfügbaren Protokolle automatisch alle möglichen Kombinationen und bietet sie dem Anwender in seiner Auswahl an. Für alle dieser vier Ereignisse wird jeweils ein Objekt von *VSProtocolEvent* verwendet, jedoch mit jeweils anderen Attributwerten.
    - *VSProtocolScheduleEvent*: Diese Klasse wird für die Wecker-Ereignisse benötigt. Wecker-Ereignisse können nur von Protokollen (mehr dazu später) erstellt werden. *VSProtocolScheduleEvent* besitzt eine Referenz auf das gegebene Protokoll und ruft bei Ereigniseintrittszeit entweder die Methode

onServerScheduleStart bei einem Server- oder *onClientScheduleStart* bei einem Clientprotokoll auf.

3. *protocols.implementations*: In diesem Paket befinden sich alle Protokollimplementierung. Jedes Protokoll besitzt hier seine eigene Klasse. Alle Protokolle erben hierbei von der auf Abbildung 3.3 zu sehenden Klasse *protocols.VSAbstractProtocol*. Da *protocols.VSAbstractProtocol* von *events.VSAbstractEvent* erbt, kann ein Protokollobjekt auch als Ereignis eingesetzt werden. Ein solches Ereignis ruft bei Eintritt entweder die Methode *onServerStart* oder die Methode *onClientStart* des Protokolls auf, was einer Server- beziehungsweise einer Clientanfrage entspricht. Die Implementierung von Protokollen wird später genauer behandelt.

Alle Ereignisse, die das Interface *VSCopyableEvent* implementieren, können vom Anwender im Ereigniseditor mit einem Rechtsklick kopiert werden und müssen die Methode *initCopy(VSAbstractEvent copy)* implementieren. Dort werden alle relevanten Attribute in das neue Ereignis *copy* kopiert.

Alle Ereignisklassen erweitern die abstrakte Klasse *VSAbstractEvent* und müssen folgende abstrakten Methoden implementieren:

- *abstract public void onInit()*: Bevor ein Ereignisobjekt vom Simulator verwendet werden kann, muß es initialisiert werden. Je nach Ereignis können hier verschiedene Werte initialisiert werden. Diese Methode wird pro Ereignisobjekt nach Erstellung nur ein einziges Mal ausgeführt.
- *abstract public void onStart()*: Diese Methode wird jedes Mal ausgeführt, wenn das Ereignis eintritt. Sie stellt somit das Kernstück eines Ereignisses dar.

Des Weiteren werden folgende nicht-abstrakte Methoden von *VSAbstractEvent* vererbt:

- *public void logg(String message)*: Diese Methode schreibt eine Loggnachricht in das Simulationsloggfenster.
- *public VSAbstractEvent getCopy()*: Diese Methode erstellt vom aktuellen Ereignis eine Kopie, wovon eine Referenz zurückgegeben wird. Alle Ereignisse die kopiert werden können müssen ebenso das Interface *VSCopyableEvent* implementieren. Wenn ein Ereignis dies nicht tut und *getCopy()* aufgerufen wird, dann wird von Java die Ausnahme *exceptions.VSEventNotCopyable* geworfen.
- *public VSAbstractEvent getCopy(VSInternalProcess process)*: Diese Methode erstellt vom aktuellen Ereignis ebenfalls eine Kopie, jedoch mit dem Unterschied, dass das Ereignis einem anderen Prozess zugewiesen wird.

Jede Ereiginsklasse hat zudem Zugriff auf folgende Attribute, die von *VSAbstractEvent* vererbt werden:

- *protected VSPrefs prefs*: Eine Referenz auf das Simulationseinstellungsobjekt. Hierüber lassen sich alle Simulationseinstellungen beziehen.
- *protected VSAbstractProcess process*: Eine Referenz auf das Prozessobjekt des jeweiligen Prozesses, auf welches das Ereignis angewendet wird.

3.3.2. Beispielimplementierung eines Ereignisses

Im Folgenden wird als Beispiel die Implementierung des Prozessabsturzereignisses *VSProcessCrashEvent* behandelt. Da die dazugehörige Klasse keine Attribute besitzt, verleibt hier auch die *initCopy*-Methode mit leerem Rumpf. Jede Ereignisklasse muss in *onInit()* mit *setClassname* den eigenen Klassennamen mitteilen. In *onStart()* wird das eigentliche Ereignis ausgeführt. Hier wird obligatorisch überprüft, ob der Prozess bereits abgestürzt (hier eigentlich nicht Notwendig, verbessert aber die Lesbarkeit der Logik) ist und gegebenenfalls wird der Prozess dann zum Absturz bewegt. Der Task-Manager überprüft bereits, ob der Prozess abgestürzt ist oder nicht, d.h. ein Ereignis wird bei einem abgestürztem Prozess gar nicht erst ausgeführt. Die einzige Ausnahme bildet ein Wiederbelebungsereignis (VSProcessRecover), welches vom Task-Manager ausgeführt wird, auch wenn der Prozess abgestürzt ist. Mit *logg* wird eine Nachricht in das Loggfenster geschrieben, welche ueber das *VSPrefs*-Objekt *prefs* bezogen wird:

```
package events.implementations;

import events.*;

public class VSProcessCrashEvent
extends VSAbstractEvent implements VSCopyableEvent {
    private static final long serialVersionUID = 1L;

    public void initCopy(VSAbstractEvent copy) {
    }

    public void onInit() {
        setClassname(getClass().toString());
    }

    public void onStart() {
        if (!process.isCrashed()) {
```

```
                process.isCrashed(true);
                logg(prefs.getString("lang.crashed"));
            }
        }
    }
```

3.4. Zeitformate, Prozesse, Nachrichten sowie Task-Manager

3.4.1. Funktionsweise

Das Paket *core.time* auf Abbildung 3.4 stellt lediglich die Klassen für die Vektor- und Lamportzeitstempel zur Verfügung. Für die normale lokale Prozesszeit wird aus Performancegründen keine eigene Klasse, sondern ein einfaches *long*-Attribut des Prozessobjektes verwendet.

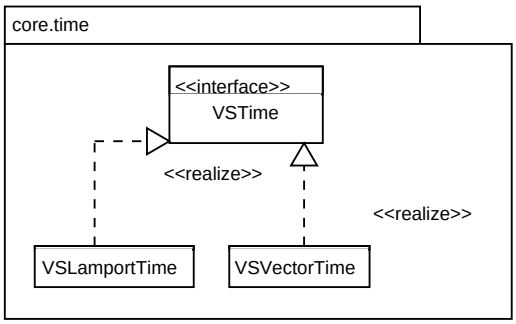

Abbildung 3.4.: Das Paket *core.time*

Auf Abbildung 3.5 ist stark vereinfacht (in Wirklichkeit existieren in den angegebenen Klassen viel mehr Attribute und Methoden) das Paket *core* dargestellt. Für jedes auszuführendes Ereignis wird eine Instanz von *VSTask* benötigt, welche die Ereigniseintrittszeit als Attribut abgespeichert hat sowie eine Referenz auf das Objekt des auszuführenden Ereignisses (*VSAbstractEvent*) und dem Prozessobjekt besitzt. Geplante *VSTask*-Instanzen werden für eine spätere Ausführung dem Task-Manager übergeben.

Die Kapselung eines *VSAbstractEvent*-Objektes in einem *VSTask*-Objekt erlaubt es, dass die selbe *VSAbstractEvent*-Instanz mehrmals auf einmal im Task-Manager geplant werden kann. Ohne dieser Kapselung gäbe es für jedes Ereignis lediglich nur eine einzige mögliche Eintrittszeit. Von dieser Möglichkeit wird zum Beispiel bei den Server- und Clientanfragen eines Protokollobjektes Gebrauch gemacht. Für jedes Protokoll kann der Anwender

in einer Simulation beliebig viele Anfragen programmieren, wobei für jede Anfrage stets das selbe Protokollobjekt als Ereignis verwendet wird.

Jede Simulation besitzt genau eine Instanz von *VSTaskManager*. Eine Instanz dieser Klasse stellt den Task-Manager dar. Er verwaltet alle *VSTask*-Instanzen und überprüft periodisch, ob es auszuführende Ereignisse gibt. Der Task-Manager unterscheidet zwischen globalen und lokalen Ereignissen. Hierbei werden alle globalen Ereignisse (gekapselt in einem *VSTask*-Objekt) in einer Prioritäts-Warteschlange abgelegt. Die Prioritäts-Warteschlange stellt hierbei die korrekte Ereigniseintrittsreihenfolge sicher. Da sich die lokalen Zeiten aller beteiligten Prozesse voneinander unterscheiden können, muss für jeden Prozess eine separate lokale Prioritäts-Warteschlange verwendet werden, auf die jedes Prozessobjekt seine eigene Referenz hat. In den lokalen Warteschlangen sind die geplanten lokalen Ereignisse (auch gekapselt in einem *VSTask*-Objekt) abgelegt. Der Task-Manager greift über eine *java.util.ArrayList* auf alle Prozessobjekte zu und kann somit auch auf alle lokalen Warteschlangen zugreifen und diese verwalten.

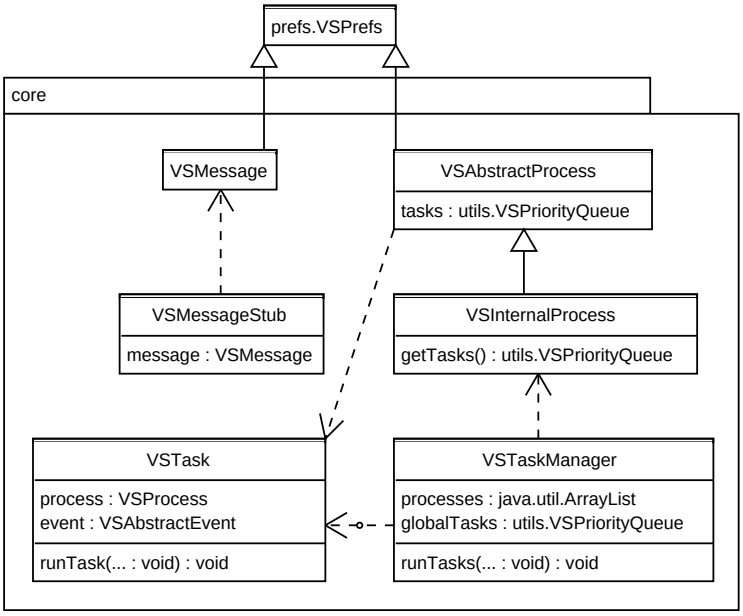

Abbildung 3.5.: Das Paket *core*

Eine Instanz von *VSMessage* stellt eine Nachricht dar, die von einem Prozess verschickt wird. Für jedes Versenden einer Nachricht wird hiervon eine Instanz gebildet, wo der Senderprozess die zu verschickende Daten ablegt. Da *VSMessage* von *VSPrefs* erbt, können zwischen zwei Prozessen beliebige Datentypen (Tabelle 2.2) über eine Nachricht verschickt werden. Anschließend wird für jeden Empfängerprozess das neues Ereignisobjekt der Klasse *VSMessageReceiveEvent* angelegt, welche eine Referenz der verschickten Nachricht besitzt. Danach wird ein *VSTask*-Objekt instanziert, wo die Referenz auf das Ereignisobjekt und das dazugehörige Prozessobjekt sowie die Ereigniseintrittszeit als Attribute gespeichert werden. Das *VSTask*-Objekt wird dann dem Task-Manager

übergeben, der das dazugehörige Ereignis ausführt, wenn die Ereigniseintrittszeit eingetroffen ist.

Via Java-Polymorphie wird das *VSMessageReceiveEvent*-Objekt in ein *VSAbstractEvent* umgewandelt. Die dazugehörige Ereigniseintrittszeit t_e wird jeweils wie folgt bestimmt:

$$t_e := TODO$$

Erwähnentswert ist auch die Klasse *VSMessageStub*, welche ein *VSMessage* kapselt. Ihr Zweck ist das Verstecken einiger Methoden von *VSMessage* im Protokoll-API, welches für die Erstellung eigener Protokolle dient. Der Protokoll-Entwickler soll möglichst nichts falsch machen können und deswegen soll den Protokoll-API ein eingeschränkter Funktionsumpfang zur Verfürung gestellt werden. Da sich *VSMessageStub* im selben Paket wie *VSMessage* befindet, kann *VSMessageStub* auf paket-private Methoden (diejenigen Methoden, die weder als *private*, *protected* noch *public* gekennzeichnet sind) von *VSMessage* zugreifen. Protokolle hingegen werden in einem anderen Paket implementiert und haben somit keinen Zugriff auf diese paket-privaten Methoden. Zwar kann der Protokollentwickler ein eigenes *VSMessageStub*-Objekt anlegen, jedoch kann er auf diese Weise besser unterscheiden, auf welche Mehhoden er zugreifen sollte und auf welche nicht. Das Protokoll-API wird später genauer behandelt.

Der Task-Manager speichert anschließend die Empfangsereignisse in den lokalen Warteschlangen der Empfängerprozesse. Die Nachricht kommt bei einem Empfängerprozess an, sobald das Ereignis für den Empfang eintritt. Für die korrekte Implementierung der Lamport- und Vektor-Zeitstempel wird jeder Nachricht automatisch eine Referenz auf die Lamport- sowie auf die Vektorzeit des sendenden Prozesses als Attribut beigefügt. Für die Überprüfung des Protokolls wird in jeder Nachricht auch der Klassenname des jeweiligen Protokolls abgespeichert.

Eine Instanz von *VSInternalProcess* repräsentiert einen simulierten Prozess. Ein *VSInternalProcess* stellt alle vom Simulator intern verwendeten Methoden zur Verfügung, während ein *VSAbstractProcess* lediglich Methoden hat, die man im Protokoll-API für die Erstellung eigener Protokolle verwenden darf. Da *VSAbstractProcess* abstrakt ist und hiervon keine Instanz gebildet werden darf, muss für einen neuen Prozesses stets ein *VSInternalProcess*-Objekt erstellt werden. Via Polymorphie wird dieses Objekt nach *VSAbstractProcess* umgewandelt und so dem Protokoll-API zur Verfügung gestellt. Beispielsweise darf mit *getTasks()* nur vom Simulator intern auf die Prioritäts-Warteschlangen zugegriffen werden, während man im Protokoll-API selbiges vermeiden sollte und auch gar nicht direkt möglich ist. Hierfür hätte man auch ein Stub-Objekt *VSProcessStub* verwenden können. Da aber so gut wie alle paar Millisekunden auf die Methoden von *VSInternalProcess* zugegriffen wird, wurde hier aus Performancegründen der Weg über eine Vererbungungsstufe preferiert.

Alle einstellbaren Prozessvariablen werden von der Klasse *VSPrefs* vererbt. Damit bei Neuberechnungen die Variablen nicht dauernd über eine *HashMap* von *VSPrefs* zugregriffen werden muß, speichert *VSInternalProcess*

aus Performancegründen einige Variablen als lokale Kopie ab. Zum Beispiel wird für die lokale Prozesszeit nicht auf das *HashMap<String,Long>*-Objekt von *VSPrefs*, sondern auf das Klassenattribut *private long localTime* zugegriffen. Vor- und nach dem Editieren über den Prozesseditor werden die *VSPrefs* beziehungsweise die lokalen Kopien auf den neusten Stand gebracht. Selbiges gilt für weitere Variablen wie zum Beispiel der Uhrabweichung eines Prozesses.

3.4.2. Beispiel für die Erstellung von Prozessereignissen

Anhand der Prozessabsturz- und Wiederbelebungsereignisse läßt sich wie folgt sehr gut demonstieren, wie intern Ereignisse angelegt werden können:

```
void createCrashAndRecoverExample(VSTaskManager taskManager,
                                  VSInternalProcess process) {
    VSAbstractEvent crashEvent = new VSProcessCrashEvent();
    VSTask localTask = new VSTask(process.getTime()+500, process,
                                  crashEvent, VSTask.LOCAL);
    taksManager.addTask(localTask);

    VSAbstractEvent recoverEvent = new VSProcessRecoverEvent();
    VSTask globalTask = new VSTask(2000, process,
                                  recoverEvent, VSTask.GLOBAL);
    taksManager.addTask(globalTask);
}
```

In diesem Beispiel wurden zwei Ereignisse (Absturz- und Wiederbelebung eines gegebenen Prozesses) angelegt. Das Absturzereignis tritt bei der aktuellen lokalen Prozesszeit plus *500ms* ein, während das Wiederbelebungsereignis bei einer globalen Zeit von *2000ms* stattfindet. Für den Fall, dass das Wiederbelebungsereignis vor dem Absturzereignis eintritt, wird es nicht ausgeführt, da der Prozess noch nicht abgestürzt ist.

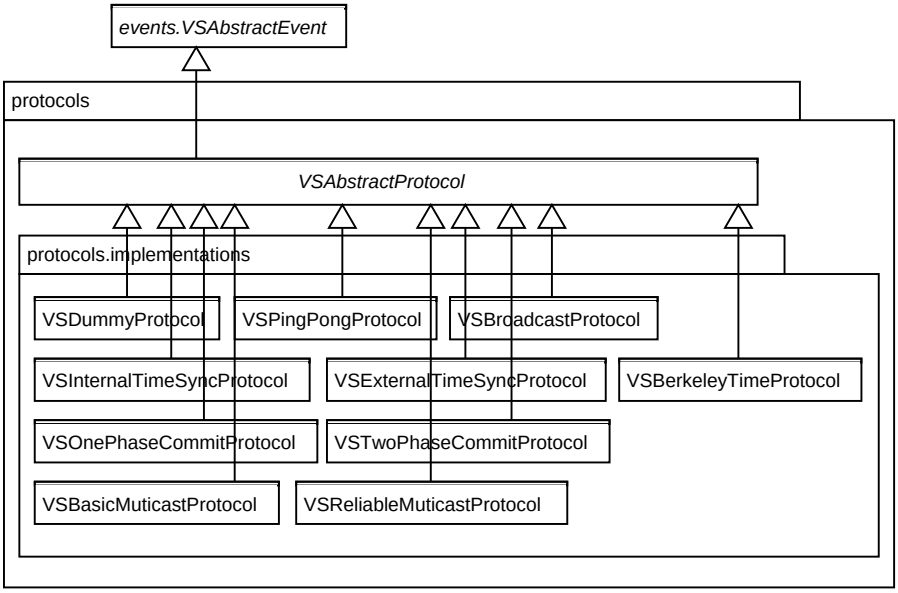

Abbildung 3.6.: Die Pakete *protocols* und *protocols.**

3.5. Protokolle

3.5.1. Die Funktionsweise des Protokoll-APIs

In diesem Abschnitt wird auf die Implementierung der Protokolle eingegangen. Auf Abbildung 3.6 sind die Pakete *protocols* und *protocols.implementations* dargestellt, welche für die Protokollimplementierungen zuständig sind. *VSAbstractProtocol* stellt lediglich gemeinsame Methoden und Attribute zur Verfügung, die von allen Protokollen verwendet werden können. Jedes Protokoll hat im Paket *protocols.implementations* seine eigene Klasse, die von *VSAbstractProtocol* erbt. Wie bereits erwähnt erbt *VSAbstractProtocol* von *VSAbstractEvent*, sodass jedes Protokoll auch als Ereignis (Server- beziehnungsweise Clientanfrage starten) eingesetzt werden kann.

Jede Protokollklasse muß die folgenden Methoden implementieren:

- Einen öffentlichen (*public*) Konstruktor. Der Konstruktor muß angeben, ob bei dem gegebene Protokoll der Client oder der Server die Anfragen startet.

- *abstract public void onClientInit()*: Bevor das Protokollobjekt benutzt werden kann, muß es initialisiert werden. Diese Methode wird vor dem ersten Verwenden des Protokolls innerhalb einer Simulation ausgeführt. In der Regel werden hier Protokollvariablen unter Verwendung von *VSPrefs* und Attribute der Protokollklasse initialisiert. Die hier initialisierten Protokollvariablen lassen sich vom Benutzer im Prozesseditor des jeweiligen Prozesses editieren.

- *abstract public void onClientReset()*: Dese Methode wird jedes Mal ausgeführt, wenn die Simulation zurückgesetzt wird.
- *abstract public void onClientStart()*: Diese Methode wird nur benötigt, wenn der Client immer die Anfragen startet. Diese Methode generiert in der Regel immer eine Clientanfrage, die via *VSMessage*-Objekt an alle beteiligten Prozesse verschickt wird.
- *abstract public void onClientRecv(VSMessage message)*: Diese Methode wird jedes Mal Ausgeführt, wenn eine Servernachricht *message* bei dem Client eintrifft.
- *abstract public void onClientSchedule()*: Diese Methode wird jedes Mal ausgeführt, wenn ein Wecker-Ereignis eintritt.
- *public String toString()*: Diese Methode ist nur optional. Hiermit lassen sich die Loggnachrichten eines Protokolls anpassen. Wenn diese Methode in einer Protokollimplementierung ausgelassen wird, so wird stets die *toString*-Methode der Mutterklasse *VSAbstractProtocol* verwendet.

Für alle hier aufgelisteten Client-Methoden sind auch die korespondierenen Server-Methoden anzugeben. Die Server-Methoden sind analog zu den Client-Methoden aufgebaut.

3.5.2. Beispielimplementierung eines Protokolls

Im Folgenden wird die Implementierung des zuverlässigen Multicast-Protokolls *VSReliableMulticastProtocol.java* als Beispiel aufgeführt. Die Funktionsweise des Protokolls wurde bereits in Kapitel 2.5.10 beschrieben. Client- und Serverseite werden in der selben Klasse implementiert.

Im Konstruktor muß stets angegeben werden, ob beim gegebenen Protokoll der Client oder der Server die Anfragen startet. Mit *HAS_ON_CLIENT_START* wird dem API mitgeteilt, dass der Client die Anfragen startet. Für *HAS_ON_SERVER_START* und Serveranfragen gilt Selbiges analog:

```
package protocols.implementations;

import java.util.ArrayList;
import java.util.Vector;
import protocols.VSAbstractProtocol;
import core.VSMessage;

public class VSReliableMulticastProtocol
```

```
extends VSAbstractProtocol {
    public VSReliableMulticastProtocol() {
        super(VSAbstractProtocol.HAS_ON_CLIENT_START);
        setClassname(getClass().toString());
    }
```

Clientseite des Protokolls

Das private Klassenattribut *pids* wird für die Zwischenspeicherung beteiligter PIDs benötigt. Hier sind alle PIDs abgelegt, von denen noch Bestätigungsnachrichten erwartet werden. Hier werden als Standard-PIDs *1* und *3* verwendet. Die Methoden *initVector* und *initLong* wurden von *VSPrefs* vererbt und initialisieren die Protokollvariablen *pids* und *timeout*, welche vom Benutzer im Prozesseditor editiert werden können:

```
    private ArrayList<Integer> pids;

    public void onClientInit() {
        Vector<Integer> vec = new Vector<Integer>();
        vec.add(1); vec.add(3);
        initVector("pids", vec, "PIDs beteilitger Prozesse");
        initLong("timeout", 2500,
                 "Zeit bis erneute Anfrage", "ms");
    }
```

Wenn die Simulation zurückgesetzt wird, dann wird auch *pids* reinitialisiert:

```
    public void onClientReset() {
        pids.clear();
        pids.addAll(getVector("pids"));
    }
```

In *onClientStart* wird geprüft, ob eine Clientanfrage gestartet werden soll. Wenn dies der Fall ist (wenn von mindestens einem beteiligten Prozess noch keine Bestätigung erhalten wurde), wird ein neues Nachrichtenobjekt erstellt, und mit dem Inhalt *Boolean: isMulticast=true* verschickt (intern wird hier für jeden Empfängerprozess ein

VSMessageReceiveEvent erzeugt). Mit *scheduleAt* wird ein Wecker festgelegt, zur welchen lokalen Prozesszeit die Methode *onClientSchedule* aufgerufen werden soll (intern wird hier ein *VSProtocolScheduleEvent* erzeugt):

```
public void onClientStart() {
    if (pids.size() != 0) {
        long timeout = getLong("timeout") + process.getTime();
        scheduleAt(timeout);

        VSMessage message = new VSMessage();
        message.setBoolean("isMulticast", true);
        sendMessage(message);
    }
}
```

Wenn eine Serverantwort eintrifft, dann wird *onClientRecv* aufgerufen. Hier wird überprüft, ob überhaupt noch Multicast-Bestätigungen benötigt werden. Wenn dies der Fall ist, dann wird geschaut, ob es sich bei der Antwort um eine noch nicht eingetroffene Bestätigung handelt. Gegebenenfalls wird die jeweilige PID aus *pids* entfernt. Wenn *pids* leer ist, dann wurde von allen beteiligten Prozessen eine Bestätigung erhalten und der Client entfernt mit *removeSchedules* alle seine derzeit programmierten Wecker.

```
public void onClientRecv(VSMessage recvMessage) {
    if (pids.size() != 0 && recvMessage.getBoolean("isAck")) {
        Integer pid = recvMessage.getIntegerObj("pid");

        if (pids.contains(pid))
            pids.remove(pid);
        else
            return;

        logg("ACK von Prozess " + pid + " erhalten!");

        if (pids.size() == 0) {
            logg("ACKs von allen beteiligten Prozessen " +
                 "erhalten!");

            removeSchedules();
```

```
            }
        }
    }
```

Für das erneute Verschicken einer Clientanfrage ruft *onClientSchedule* lediglich die Methode *onClientStart* auf, die wiederum einen neuen Wecker planen kann:

```
    public void onClientSchedule() {
        onClientStart();
    }
```

Serverseite des Protokolls

Die Serverseite des Protokolls speichert im Attribut *ackSent* ab, ob es bereits eine Bestätigung des Multicasts verschickt hat oder nicht. In diesem Protokoll werden in *onServerInit* keine Initialisierungen vorgenommen. Demach gibt es für den Benutzer auch keine serverseitigen Protokollvariablen zu editieren. Beim Zurücksetzen der Simulation wird lediglich *ackSent* auf den Ursprungswert gesetzt:

```
    private boolean ackSent;

    public void onServerInit() { }

    public void onServerReset() {
        ackSent = false;
    }
```

Wenn der Server eine Clientanfrage erhalten hat, so überprüft der Server, ob es sich um eine Multicast-Nachricht handelte. Anschließend wird gegebenenfalls die Bestätigungsnachricht mit *Boolean: isAck=true* und der Server-PID verschickt. Jenachdem ob bereits eine Bestätigung verschickt wurde oder nicht wird eine andere Nachricht geloggt:

```
    public void onServerRecv(VSMessage recvMessage) {
        if (recvMessage.getBoolean("isMulticast")) {
```

```
            VSMessage message = new VSMessage();
            message.setBoolean("isAck", true);
            message.setInteger("pid", process.getProcessID());
            sendMessage(message);

            if (ackSent) {
                logg("ACK erneut versendet");
            } else {
                logg("ACK versendet");
                ackSent = true;
            }
        }
    }
```

Der Server benutzt in diesem Beispiel keinen Wecker. Dementsprechend hat die Methode *onServerSchedule* auch einen leeren Rumpf:

```
    public void onServerSchedule() { }
}
```

3.5.3. Erstellung eigener Protokolle

3.6. GUI

3.7. Serialisierung von Simulationen

3.7.1. Rückwärtskompatibel

3.8. Weiteres

exceptions, utils

3.9. Entwicklungsumgebung

In diesem Teilkapitel soll ein kleiner Einblick in die Umgebung, in der der Simulator entwickelt wurde, gewährt werden. Für diese Diplomarbeit wurde ausschließlich Open Source Software verwendet. Die einzige Ausnahme stellt Microsoft Windows XP dar, worauf der Simulator zusätzlich getestet wurde. Der Simulator wurde jedoch hauptsächlich unter dem Betriebssystem FreeBSD 7.0, was ein open source Unix-Derivat ist, programmiert.

Wie bereits bekannt ist, wurde Sun's Java, was mittlerweile auch Open Source Software ist, in der Version 6 (1.6) als die Implementierungssprache gewählt und für die Quelltextdokumentation kam Javadoc und für die automatische Quelltexteinrückung astyle zum Einsatz. Als Built-Tool wurde hier auf Apache Ant gesetzt. Für die Erstellung dieses PDF-Dokumentes wurde LaTeX in Verbindung mit dem Built-Tool GNU Make und Rubber verwendet. Eine Rechtschreibüberprüfung wurde mit aspell durchgeführt. xPDF diente als PDF-Anzeigeprogramm.

Als Versionierungssystem wurde SVN (Subversion) verwendet. Für den Zugriff auf das SVN-Repository mittels HTTPS (Hypertext Transfer Protocol Secure) wurde der Apache-Websever mit WebDAV-Plugin verwendet. Zudem kam WebSVN als Webschnitstelle des SVN-Repositories zum Einsatz. Als SSL Zertifikat diente ein Kostenloses von CaCert (`http://www.CaCert.org`). Mozilla Firefox diente für das Betrachten der Javadocs und der WebSVN-Oberfläche.

Für schreiben von Java-Quelltext wurde GVim (Graphical Vi IMproved) sowie Eclipse verwendet. Eclipse unterstützt bessere Code-Refactoring-Methoden, während GVim mit seiner Flexibilität und schnelleren Editiermöglichkeiten und mit Vim-Script, der eigenen Script-Engine, glänzt. Es wurden ausserdem das JAutoDoc- (für die Erstellung von Javadoc-Kommentate) und das Subversion-Eclipse-Plugin verwendet. Je nach Zweck wurde zwischen diesen beiden Umgebungen gewechselt. Für das Verfassen des LaTeX-Dokumentes wurde GVim verwendet.

Sämtliche UML-Diagramme wurden mit ArgoUML angefertigt und die Screenshots mit The GIMP (GNU Image Manipulation Program) sowie ImageMagick nachbearbeitet. Mit zip wurden alle VS-Simulator Distributionen eingepackt.

Linkliste der verwendeten Software

- Apache Webserver - `http://httpd.apache.org`
- ArgoUML - `http://argouml.tigris.org`
- Eclipse - `http://www.eclipse.org`
- FreeBSD - `http://www.FreeBSD.org`
- GNU Make - `http://www.gnu.org/software/make`
- GVim - `http://www.vim.org`

- ImageMagick - http://www.imagemagick.org
- Javadoc - http://java.sun.com/j2s2/javadoc
- Mozilla Firefox - http://www.mozilla.com
- Rubber - http://www.pps.jussieu.fr/~beffara/soft/rubber
- Sun Java - http://java.sun.com
- The GIMP - http://www.gimp.org
- WebDAV - http://httpd.apache.org/docs/2.0/mod/mod_dav.html
- WebSVN - http://websvn.tigris.org
- aspell - http://aspell.sourceforge.net
- astyle - http://astyle.sourceforge.net
- xPDF - http://www.foolabs.com/xpdf
- zip - http://www.info-zip.org/Zip.html

Kapitel 4.

Ausblick

Anhang A.

Akronyms

API Application Programming Interface

BSD Berkeley Software Distribution

GIMP GNU Image Manipulation Program

GNU GNU's Not UNIX

GUI Graphical User Interface

GVim Graphical Vi IMproved

HTTPS Hypertext Transfer Protocol Secure

JRE Java Runtime Environment

NID Nachrichten-Identifikationsnummer

PDF Protable Document Format

PID Prozess-Identifikationsnummer

RTT Round Trip Time

SDK Software Development Kit

SVN Subversion

VS Verteilte Systeme

Anhang B.

Literaturverzeichnis

[Fas06] Heinrich Fassbender. Vorlesung “objektorientierte softwareentwicklung” an der fh aachen. Vorlesung, 2006.

[Oßm07] Martin Oßmann. Vorlesung “verteilte systeme” an der fh aachen. Vorlesung, 2007.

[Tan03] Andrew Tanenbaum. Verteilte systeme - grundlagen und paradigmen. Buch, 2003. 2. Autor Marten van Steen; ISBN: 3-8273-7057-4.