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, 14. August 2008

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen, als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Aachen, 14. August 2008

Geheimhaltung

Diese Diplomarbeit darf weder vollständig, noch auszugsweise ohne schriftliche Zustimmung des Autors, des betreuenden Referenten bzw. der Fachhochschule, Aachen vervielfältigt, veröffentlicht oder Dritten zugänglich gemacht werden.

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
- Claudia Steudter
- Florian Bütow
- Jörn Bütow
- Jochen Demmer
- 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 wählten für die Beschreibung eines verteilten Systems die folgende lockere Charakterisierung:

[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 befindlichen Computer auseinandersetzen, während die Software des lokalen Computers die reibungslose Kommunikation mit den anderen beteiligten Computern des verteilten Systems sicherstellt.

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

Um dieses Ziel zu erreichen soll, insbesondere für Lehr- und Lernzwecke an der Fachhochschule Aachen, ein Simulator entwickelt werden. Mit dem Simulator sollen Protokolle aus den verteilten Systemen mit ihren wichtigsten Einflussfaktoren anhand von Simulationen nachgeblidet werden können. Gleichzeitig muss für eigene Experimente ein großer Spielraum zur Verfügung stehen, wobei es keine Beschränkung auf eine feste Anzahl von Protokollen geben darf. Es ist also wichtig, dass es dem Anwender ermöglicht wird eigene Protokolle zu entwerfen.

1.2. Grundlagen

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

Client/Server Modell

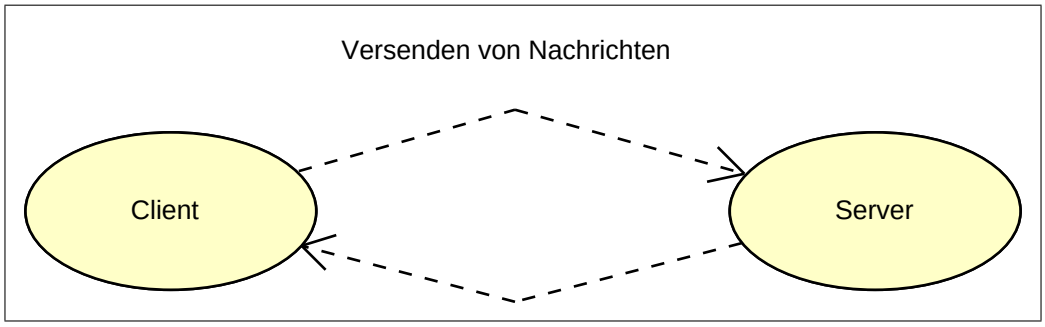

Abbildung 1.1.: Client/Server Modell

Der Simulator basiert auf dem Client/Server-Prinzip. Jede Simulation besteht in der Regel aus einen teilnehmenden Client und einen Server, die miteinander über Nachrichten kommunizieren (s. Abb. 1.1.). Bei komplexen Simulationen können auch mehrere Clients und/oder Server mitwirken.

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. Die Möglichkeit einem Prozess die Client- und Serverrolle gleichzeitig zuzuweisen ist ebenso gegeben. Ein Prozess könnte auch die Rollen mehrerer Server und Clients gleichzeitig einnehmen. Um einen Prozess zu kennzeichnen, besitzt jeder eine **eindeutige** Prozess-Identifikationsnummer (PID).

Nachrichten

In einem verteilten 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 beteiligte 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, bzw. 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.

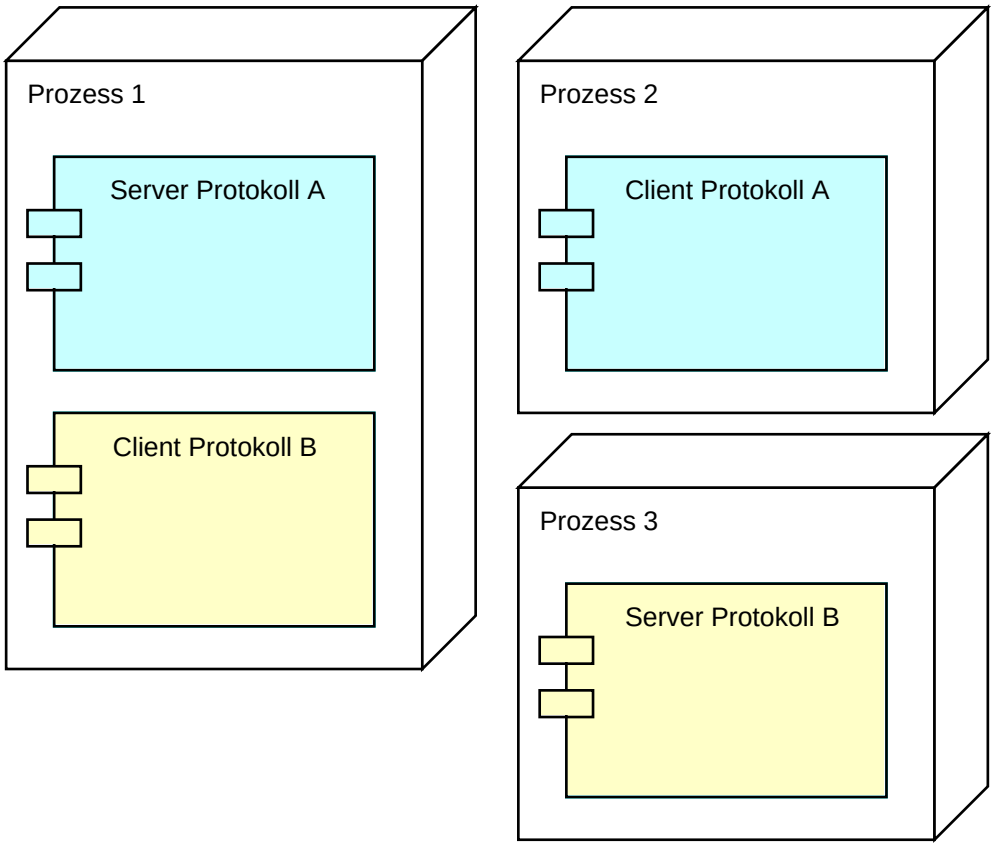

Abbildung 1.2.: Client/Server Protokolle

Neben den normalen Uhren sind auch die Vektor-Zeitstempel sowie die logischen Uhren von Lamport von Interesse. 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 in Kapitel 3.11.1. 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. Ereignisse mit selber Eintrittszeit werden vom Simulator direkt hintereinander ausgeführt. Den Anwender des Simulators hindert dies jedoch nicht, da Ereignisse aus ihrer 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 1.2. 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 Servern 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 (vgl. Broadcast Protokoll in Kap. 3.3.).

Kapitel 2.

Grafische Benutzeroberfläche (GUI)

2.1. Einfacher Modus

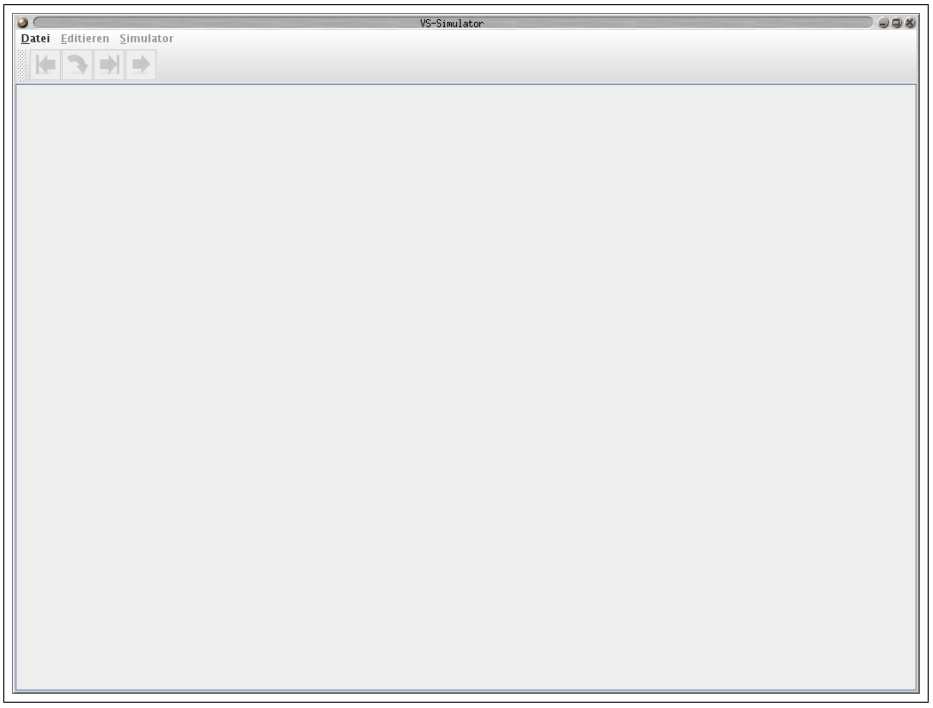

Abbildung 2.1.: Der Simulator nach dem ersten Starten

Der Simulator benötigt die JRE 6.0 (1.6) und lässt sich mit dem Befehl *java -jar VS-Sim.jar* starten. Der Simulator präsentiert sich danach so wie in Abbildung 2.1. zu sehen ist. Für die Erstellung einer neuen Simulation wird im Menü “Datei” (s. Abb. 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 in Abbildung 2.3.

Standardmäßig wird der Simulator im “einfachen Modus” gestartet. Daneben gibt es noch einen “Expertenmodus”, auf welchen später eingegangen wird.

Die Menüzeile

Im Datei-Menü (s. Abb. 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.

Abbildung 2.2.: Datei-Menü

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.

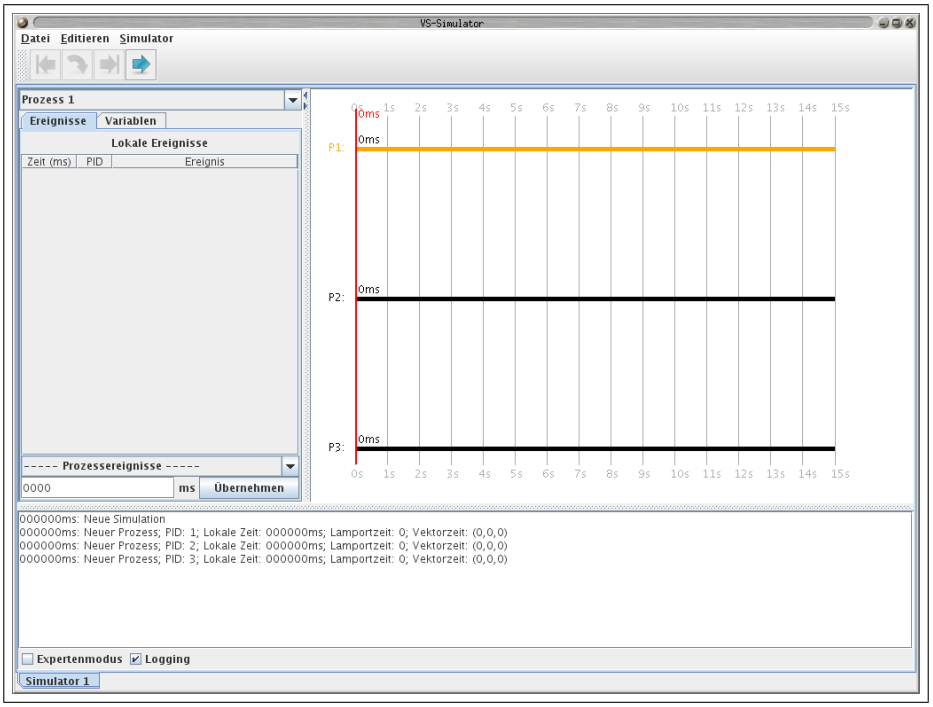

Abbildung 2.3.: Eine neue Simulation

Die Toolbar

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

Die Toolbar bietet vier verschiedene Funktionen an:

Abbildung 2.4.: Die Menüzeile inklusive Toolbar

- 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 und noch nicht abgelaufen ist.

Die Visualisierung

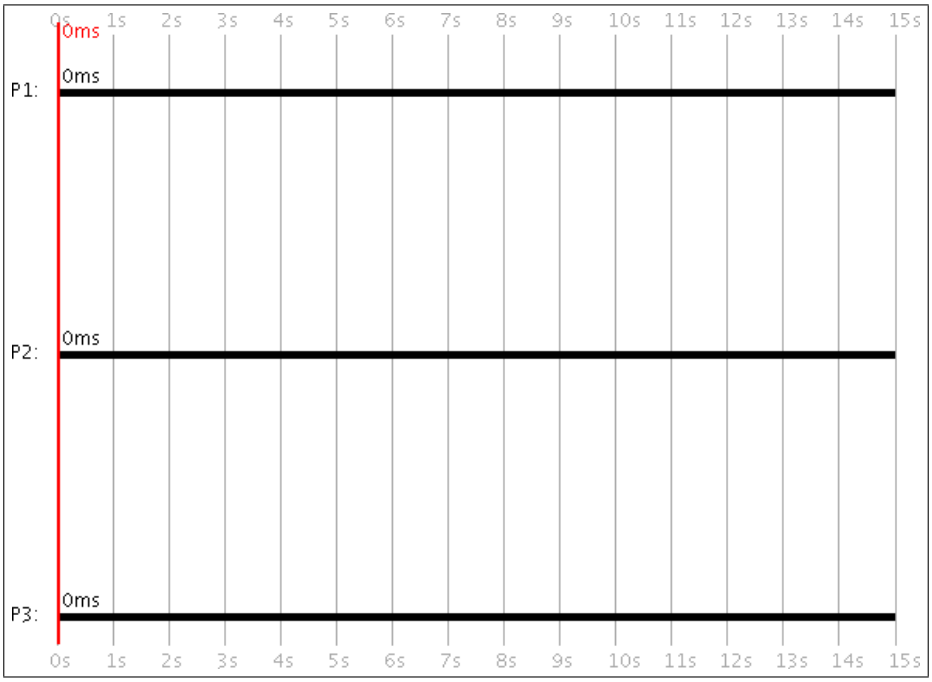

Abbildung 2.5.: Visualisierung einer noch nicht gestarteten Simulation

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. Die Demo-Simulation endet nach genau 15 Sekunden. In 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.

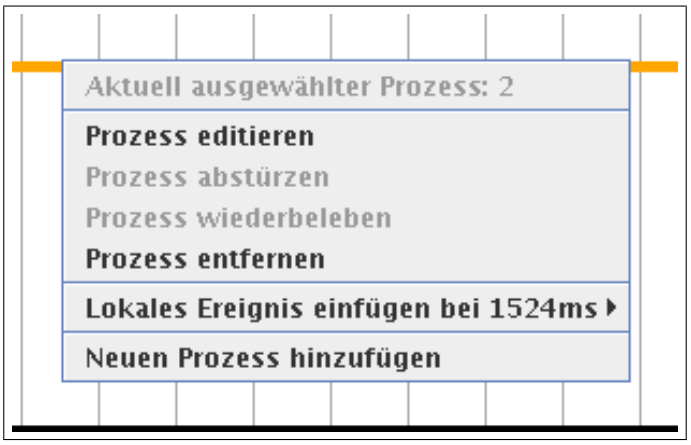

Abbildung 2.6.: Rechtsklick auf einen Prozessbalken

Die Prozessbalken dienen auch für Start- und Zielpunkte von Nachrichten. Wenn beispielsweise Prozess 1 eine

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

Nachricht an Prozess 2 verschickt, so wird eine Linie vom einen Prozessbalken zum anderen gezeichnet. Nachrichten, die ein Prozess an sich selbst verschicken, werden nicht visualisiert. Sie werden aber im Logfenster (mehr dazu später) protokolliert.

Eine andere Möglichkeit einen Prozesseditor aufzurufen ist ein Linksklick auf den zum Prozess gehörigen Prozessbalken. Ein Rechtsklick hingegen öffnet ein Popup-Fenster mit weiteren Auswahlmöglichkeiten (s. Abb. 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

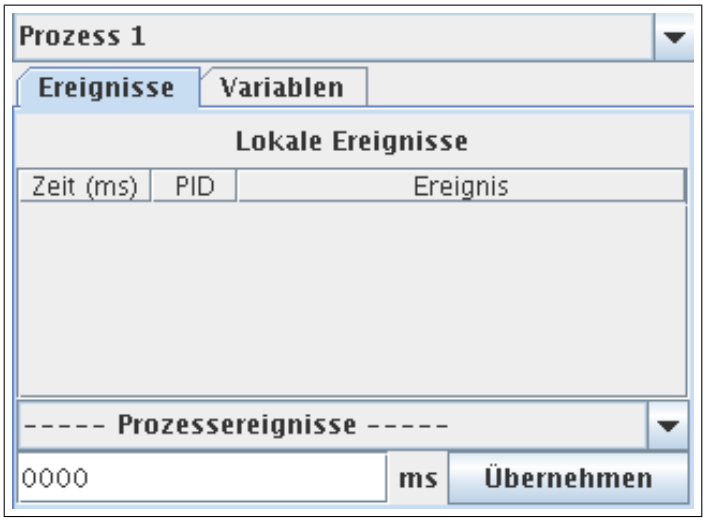

Abbildung 2.7.: Die Sidebar mit leerem Ereigniseditor

Mit Hilfe der Sidebar lassen sich Prozessereignisse programmieren. Oben in Abbildung 2.7. ist der zu verwaltende Prozess selektiert (hier mit der PID 1). In dieser Prozessauswahl gibt es auch die Möglichkeit “Alle Prozesse” auszuwählen, womit alle programmierten Ereignisse aller Prozesse gleichzeitig dargestellt werden. Unter “Lokale Ereignisse” versteht man diejenigen Ereignisse, die auftreten, wenn eine bestimmte lokale Zeit des dazugehörigen Prozesses eingetreten ist. Die darunter liegende Ereignistabelle listet alle programmierten Ereignisse (hier noch keine vorhanden) mitsamt Eintrittszeiten sowie den PIDs auf.

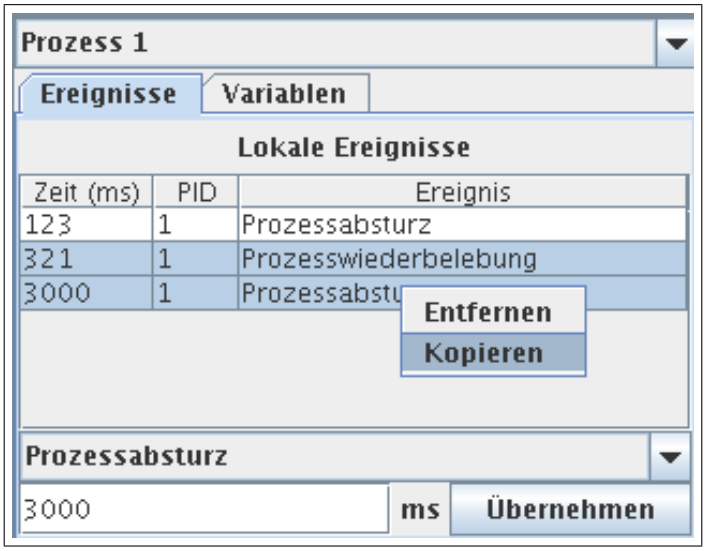

Abbildung 2.8.: Der Ereigniseditor mit 3 programmierten Ereignissen

Für die Erstellung eines neuen Ereignisses kann der Anwender entweder mit einem Rechtsklick auf einen Pro-

zessbalken (s. Abb. 2.6.) klicken und dort “Lokales Ereignis einfügen” wählen, oder unterhalb der Ereignistabelle ein Ereignis auswählen (s. Abb. 2.9.), im darunter liegenden Textfeld die Ereigniseintrittszeit eintragen und auf “Übernehmen” gehen. Beispielsweise wurden in Abbildung 2.8. drei Ereignisse hinzugefügt: Absturz nach *123ms*, Wiederbelebung nach *321ms* und erneuter Absturz nach *3000ms* des Prozesses mit der ID 1.

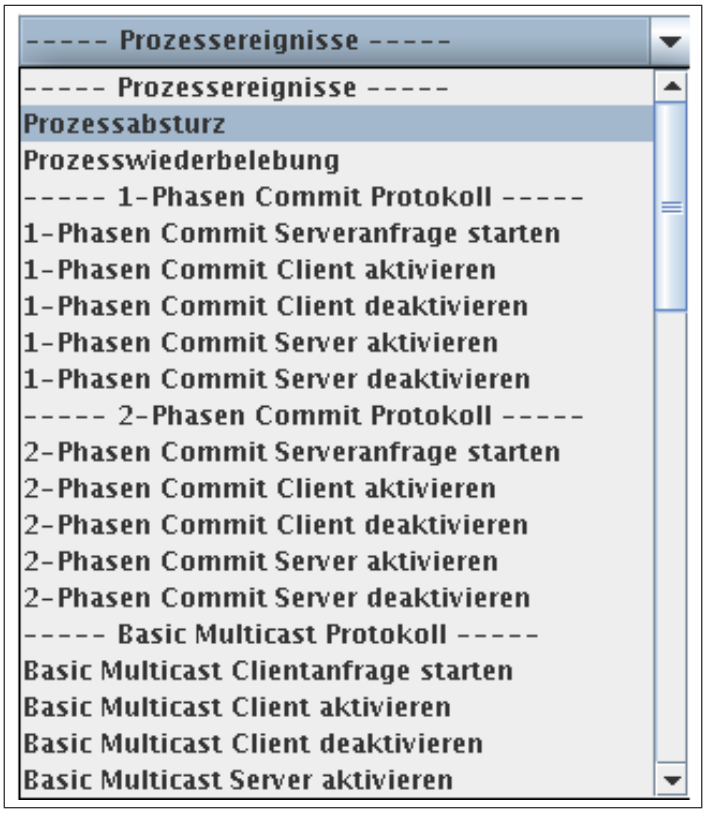

Abbildung 2.9.: Die Ereignisauswahl via Sidebar

Mit einem Rechtsklick auf den Ereigniseditor lassen sich alle selektierten Ereignisse entweder kopieren oder löschen. Mit Hilfe 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 (s. Abb. 2.13. links). Dort können alle Variablen des Prozesses editiert werden und ist somit eine weitere Möglichkeit einen Prozesseditor aufzurufen.

Das Logfenster

Das Logfenster (s. Abb. 2.3., unten) protokolliert in chronologischer Reihenfolge alle eingetroffenen Ereignisse. In Abbildung 2.10. ist das Logfenster nach Erstellung der Demo-Simulation zu sehen, an welcher 3 Prozesse beteiligt sind. Am Anfang eines Logeintrages wird stets die globale Zeit in Millisekunden protokolliert. Bei jedem Prozess werden ebenso seine lokalen Zeiten sowie die Lamport- und die Vektor-Zeitstempel aufgeführt. 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.

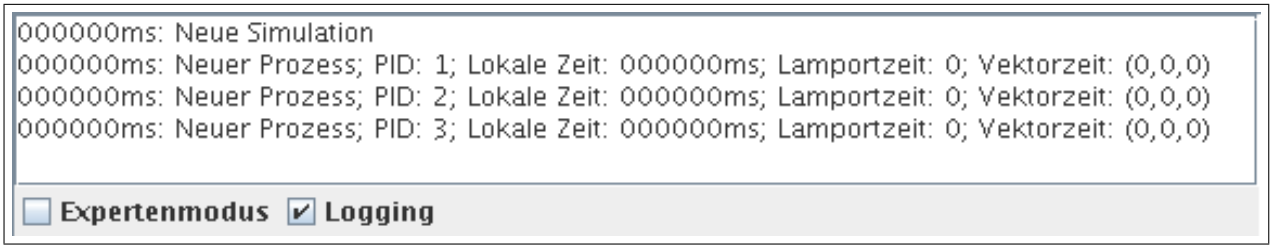

Abbildung 2.10.: Das Logfenster

Mit dem Deaktivieren des Logging-Schalters lässt sich das Loggen von Nachrichten temporär ausstellen. Mit deaktiviertem Loggen werden keine neuen Nachrichten mehr ins Logfenster geschrieben. Nach Reaktivieren des Schalters werden alle ausgelassenen Nachrichten nachträglich in das Fenster geschrieben. Ein deaktiviertes Loggen kann zu verbessertem Leistungsverhalten des Simulators führen. Dieser Umstand ist der sehr langsamen Java-Implementierung der JTextArea-Klasse zu verdanken, die Updates nur sehr träge durchführt (s. [Loy02]).

Über den Schalter “Expertenmodus” wird der Expertenmodus aktiviert, bzw. deaktiviert.

2.2. Expertenmodus

Der Simulator kann in zwei verschiedenen Modi betrieben werden. Es gibt einen einfachen und einen Expertenmodus. Der Simulator startet standardmäßig im einfachen Modus, so dass 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 den gleichnamigen Schalter unterhalb des Logfensters oder über die Simulationseinstellungen aktiviert oder deaktiviert werden. In Abbildung 2.11. ist der Simulator im Expertenmodus zu sehen. Wenn der Expertenmodus mit dem einfachen Modus verglichen wird, so fallen einige Unterschiede auf:

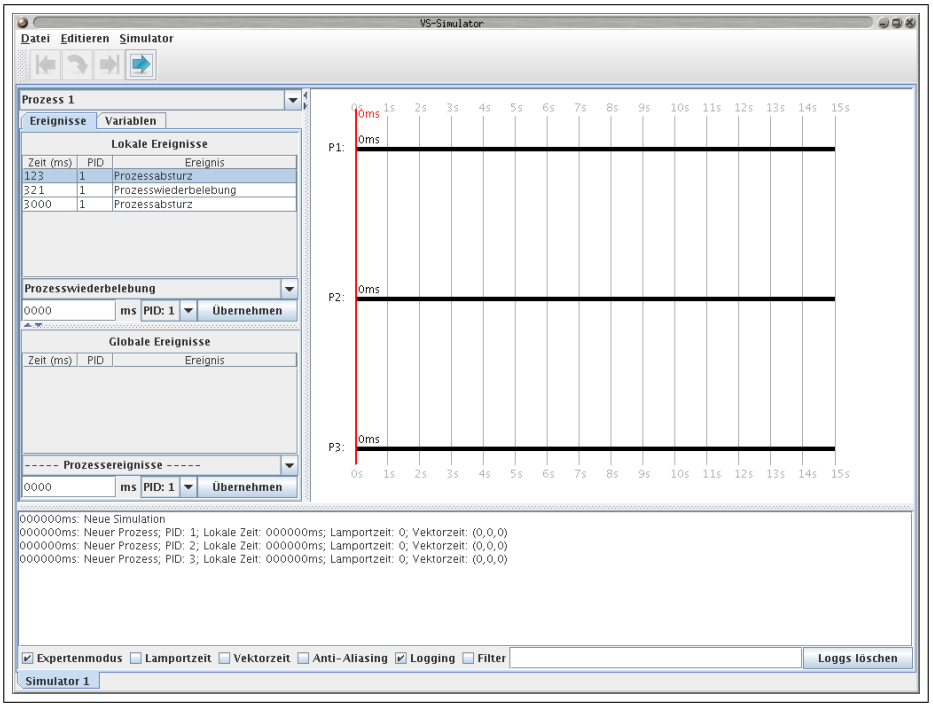

Abbildung 2.11.: Der Simulator im Expertenmodus

Neue Funktionen in der Sidebar

Der erste Unterschied ist in der Sidebar erkennbar (s. Abb. 2.12.). Dort sind nun, zusätzlich zu 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 Simulationszeit 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).

Lamportzeit-, Vektorzeit- und Anti-Aliasing Schalter

Weitere Unterschiede machen sich unterhalb des Logfensters bemerkbar. Dort gibt es unter anderem zwei neue Schalter “Lamportzeit” und “Vektorzeit”. Aktiviert der Anwender einen dieser beiden Schalter, so werden die

Lamport- bzw. die Vektor-Zeitstempel in der Visualisierung dargestellt. Damit die Übersichtlichkeit nicht leidet, kann der Anwender nur jeweils einen dieser beiden Schalter zur gleichen Zeit aktiviert haben.

Abbildung 2.12.: Die Sidebar im Expertenmodus

Der Anti-Aliasing-Schalter ermöglicht dem Anwender Anti-Aliasing zu aktivieren bzw. zu deaktivieren. Mit Anti-Aliasing werden alle Grafiken der Visualisierung gerundet dargestellt (s. [Bra03]). Aus Performance-Gründen ist Anti-Aliasing standardmäßig nicht aktiv.

Der Logfilter

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

Der Logfilter wird anhand des dazugehörigen Schalters “Filter” aktiviert und deaktiviert. In der dahinterliegenden Eingabezeile kann ein regulärer Ausdruck in Java-Syntax, angegeben werden. Die Verwendung regulärer Ausdrücke mittels Java wird in [Fri06] behandelt. Beispielsweise werden mit “*PID: (1|2)*” nur Logzeilen angezeigt, die entweder “*PID: 1*” oder “*PID: 2*” beinhalten. Alle anderen Zeilen, die z.B. nur “*PID: 3*” beinhalten, werden

dabei nicht angezeigt. Mit Logfilter werden nur die Logzeilen angezeigt, auf die der angegebene reguläre Ausdruck passt. Der Logfilter kann auch nachträglich aktiviert werden, da bereits protokollierte Ereignisse nach jeder Filteränderung erneut gefiltert werden.

Der Logfilter kann auch während einer laufenden Simulation verwendet werden. Bei Filterdeaktivierung werden alle Nachrichten wieder dargestellt. Lognachrichten, die aufgrund des Filters noch nie angezeigt wurden, werden dann nachträglich angezeigt.

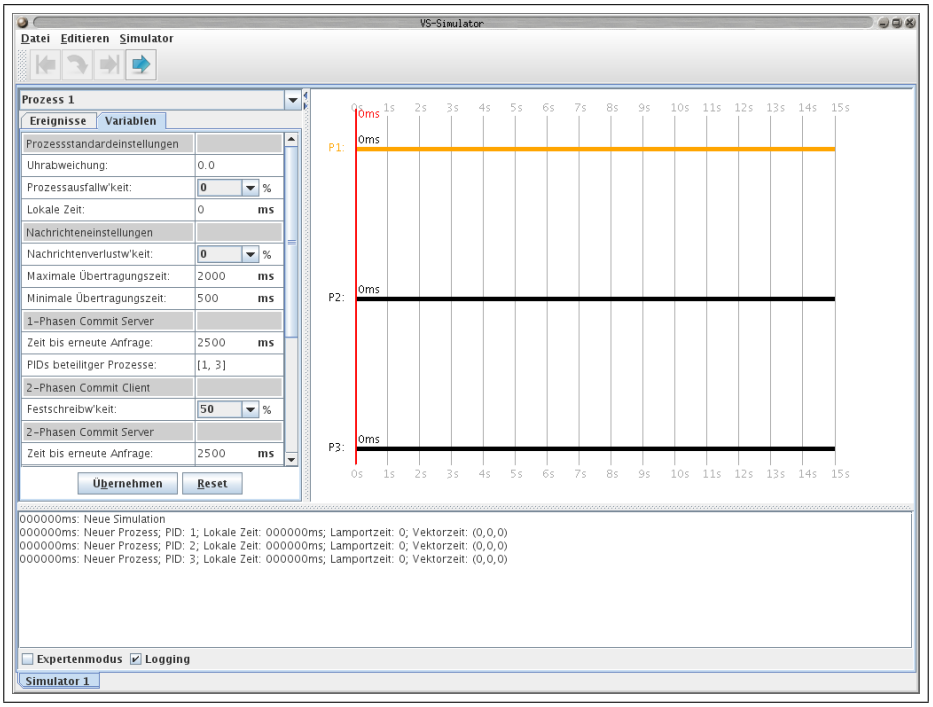

Abbildung 2.13.: Der Prozesseditor in der Sidebar

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 z.B. das Eintreffen einer Nachricht oder das Ausführen einer Aktion aufgrund eines Weckers (mehr dazu später).

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 geht sie verloren. Die einzige Ausnahme bietet ein Wiederbelebungsereignis. Ein abgestürzter Prozess kann nichts, außer wiederbelebt werden. Während eines Prozessabsturzes läuft die lokale Prozessuhr, abgesehen von den Lamport- und Vektor-Zeitstempel, normal weiter. Das heißt, es besteht die Möglichkeit, dass ein Prozess einige seiner Ereignisse gar nicht ausführt, da er zu den Ereigniseintrittszeiten abgestürzt ist.

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 ggf. ein anderes Protokoll ablöst. Jedes Protokoll kann entweder server- oder clientseitig aktiviert bzw. deaktiviert werden. Der Anwender hat somit 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 Anfragen. Es gibt kein Protokoll, wo der Client und der Server jeweils Anfragen starten können.

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 vice 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 wiedereintreffende Ereignisse realisieren. Beispielsweise verwenden die sog. Commit-Protokolle (s. Kap. 3.3.7, 3.3.8) 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 (s. Kapitel 2.4.3.).

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 (Client- sowie Serverseite) für jeden Prozess separat eingestellt werden.

Typ	Beschreibung
Boolean	Boolescher Wert, z.B. *true* oder *false*
Color	Java-Farbobjekt
Float	32-Bit Fließkommazahl
Integer[]	Vektor aus 32-Bit Integer
Integer	32-Bit Integer
Long	64-Bit Long
String	Java-Stringobjekt

Tabelle 2.2.: Verfügbare Datentypen für editierbare Variablen

2.4.1. Variablendatentypen

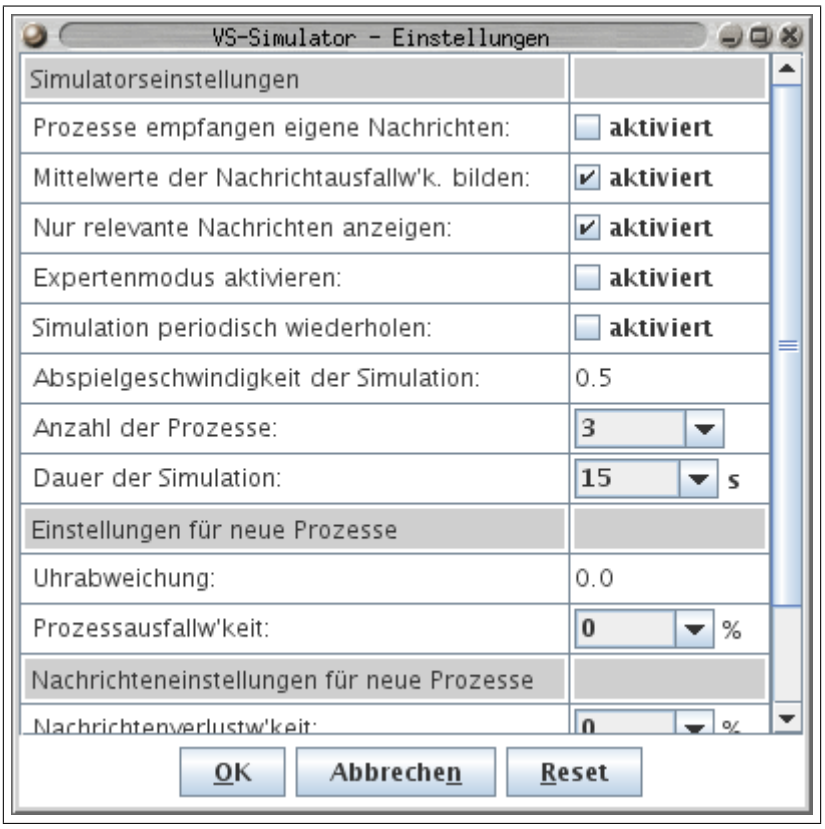

Abbildung 2.14.: Das Fenster zu den Simulationseinstellungen

Der Simulator unterscheidet zwischen mehreren Datentypen, in denen die einstellbaren Variablen vorliegen können (s. Tabelle 2.2.). Jede Variable besitzt einen Namen, einen Wert und eine optionale Beschreibung. Wenn eine Variablenbeschreibung vorhanden ist, so wird sie anstelle des Variablennamen in einem Editor angezeigt. Der Variablenname wird vom Simulator lediglich für die interne Verwendung benötigt. Im folgenden bedeutet *Typ: varname = wert*, dass die Variable vom Typ *Typ* ist, der interne Variablenname *varname* lautet, und standard-

mäßig den Wert *wert* zugewiesen hat. Vom Anwender lassen sich lediglich die Variablenwerte, jedoch nicht die Variablentypen, Variablennamen und Beschreibungen ändern.

2.4.2. Simulationseinstellungen

Beim Erstellen einer neuen Simulation erscheint zunächst das dazugehörige Einstellungsfenster (s. Abb. 2.14.). 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, Namen und die Standardwerte an, in denen die Variablen vorliegen.

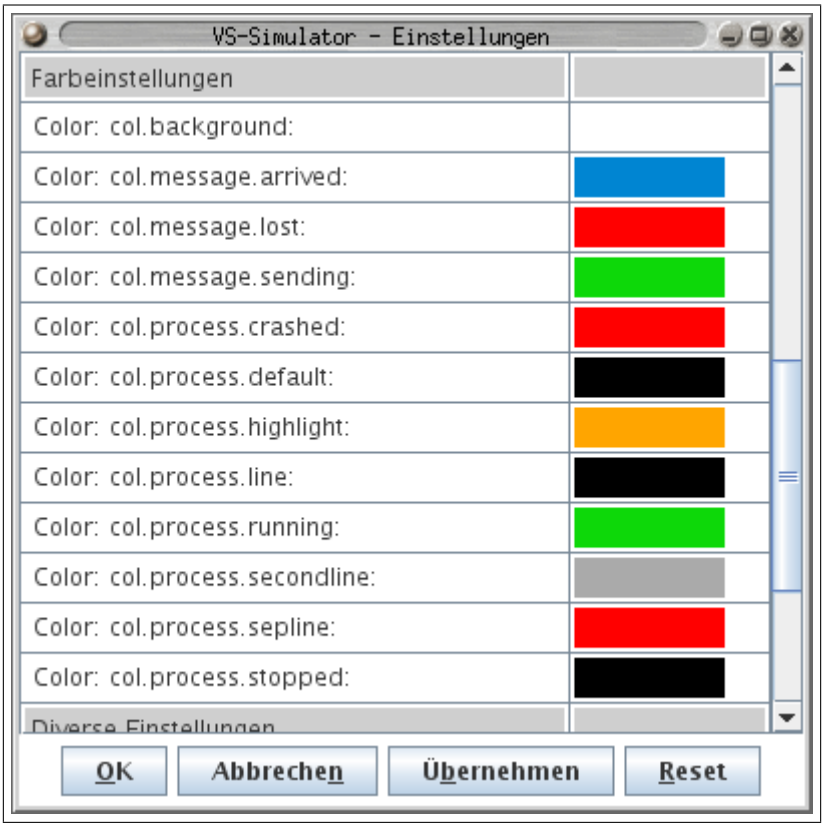

Abbildung 2.15.: Weitere Simulationseinstellungen im Expertenmodus

- **Prozesse empfangen eigene Nachrichten** *(Boolean: sim.message.own.recv = 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 Nachrichten empfangen 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 bedingt durch den *0ms* Endlosschleifen entstehen können.

- **Mittelwerte der Nachrichtenverlustwahrscheinlichkeiten bilden** *(Boolean: sim.message.prob.mean = true)*: Jede Nachricht, die verschickt wird, hat, je nach Einstellungen, eine vom verschickenden Prozess abhängige zufällige Verlustwahrscheinlichkeit. Wenn diese Option aktiviert ist, wird der Mittelwert aus den Verlustwahrscheinlichkeiten des Senders- und Empfängerprozesses gebildet. Ansonsten wird die Verlustwahrscheinlichkeit, die beim Senderprozesses eingestellt wurde, verwendet.

- **Mittelwerte der Übertragungszeiten bilden** *(Boolean: sim.message.sendingtime.mean = true)*: Jede Nachricht, die verschickt wird, hat, je nach Einstellungen, eine vom verschickenden Prozess abhängige zufällige Übertragungszeit bis sie ihr Ziel erreicht (siehe Prozesseinstellungen später). Wenn diese Option aktiviert ist, wird der Mittelwert vom Sender- und Empfängerprozess gebildet. Ansonsten wird die Übertragungszeit, die beim Senderprozesses angegeben wurde, verwendet.

- **Nur relevante Nachrichten anzeigen** *(Boolean: sim.messages.relevant = true)*: Wenn nur alle relevanten Nachrichten angezeigt werden, dann werden Nachrichten an einen Prozess die er selbst nicht verarbeiten kann nicht angezeigt, da er das dazugehörige Protokoll nicht unterstützt. Dadurch wird die Übersichtlichkeit verbessert.

- **Expertenmodus aktivieren** *(Boolean: sim.mode.expert = false)*: Hier lässt sich der Expertenmodus aktivieren und deaktivieren. Alternativ kann dies über den gleichnamigen Schalter unterhalb des Logfensters geschehen.

- **Simulation periodisch wiederholen** *(Boolean: sim.periodic = false)*: Wenn diese Variable auf *true* gesetzt ist, dann wird die Simulation jedes Mal nach Ablauf automatisch neu gestartet.

- **Lamportzeiten betreffen alle Ereignisse** *(Boolean: sim.update.lamporttime.all = false)*: Wenn diese Variable auf *true* gesetzt ist, dann werden bei jedem Ereignis alle Lamport-Zeitstempel aller Prozesse inkrementiert. Bei einem Wert *false* werden die Lamport-Zeitstempel jeweils nur inkrementiert, wenn eine Nachricht empfangen oder verschickt wurde.

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

- **Abspielgeschwindigkeit der Simulation** *(Float: sim.clock.speed = 0.5)*: Gibt den Faktor der Simulationsabspielgeschindigkeit an. Wenn als Faktor *1* gewählt wird, dann dauert eine Sekunde in einer Simulation

Schlüssel	Beschreibung
col.background	Hintergrundfarbe des Simulationsfensters
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 "Prozessfahne" 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 der Sekunden-Zeitgitter
col.process.sepline	Farbe der globalen Zeitachse
col.process.stopped	Prozessfarbe wenn die Simulation pausiert wurde

Tabelle 2.3.: Farbeinstellungen

so lange wie eine echte Zeitsekunde. Ein Wert von *0.5* gibt somit an, dass die Simulation mit halber Echtzeitgschwindigkeit abgespielt werden soll.

- **Anzahl der Prozesse** *(Integer: sim.process.num = 3)*: Gibt die Anzahl beteiligter Prozesse an. Der Anwender kann auch später während der Simulation mit einem Rechtsklick auf den Prozessbalken Prozesse aus der aktuellen Simulation entfernen oder weitere Prozesse hinzufügen.

- **Dauer der Simulation** *(Integer: sim.seconds = 15)*: Gibt die Dauer der Simulation in Sekunden vor.

Alle weiteren Simulationeinstellungen unter "Einstellungen für neue Prozesse" sowie "Nachrichteneinstellungen für neue Prozesse" definieren das Verhalten des jeden neu erzeugten Prozesses.

2.4.3. Prozess- und Protokolleinstellungen

Jeder Prozess besitzt folgende Variablen:

- **Uhrabweichung** *(Float: process.clock.variance = 0.0)*: Gibt den Wert t_v an, um den die lokale Prozessuhr t abweicht. Wenn t_n die neu verstrichene Zeit ist, dann wird die lokale Prozessuhr wie folgt neu berechnet:

$$t := t + t_n * t_v$$

Der Wert *0.0* besagt beispielsweise, dass die Uhr keine Abweichung hat und somit global-korrekt läuft. Ein Wert von *1.0* hingegen bedeuten, dass die Uhr mit doppelter Geschwindigkeit läuft. Ein Wert von *-0.5* bedeutet, dass die lokale Prozessuhr mit halber globaler Geschwindigkeit fortschreitet. Es sind nur Werte > *-1.0* erlaubt, da sonst die Prozessuhr rückwärts laufen könnte. Bei allen anderen Werten wird die Einstellung wieder automatisch auf *0.0* gesetzt. Der Simulator arbeitet intern mit Fließkommazahlen doppelter Genauigkeit arbeitet, so dass es es zu kleinen, jedoch vernachlässigbaren Rundungsfehlern kommen.

- **Prozessausfallwahrscheinlichkeit** *(Integer: process.prob.crash = 0)*: Gibt eine Wahrscheinlichkeit in Prozent an, mit der der 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 mehr 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: process.localtime = 0)*: Gibt die lokale Prozesszeit in Millisekunden an.

- **Nachrichtenverlustwahrscheinlichkeit** *(Integer: message.prob.crash = 0)*: Gibt eine Wahrscheinlichkeit in Prozent an, mit der eine vom aktuell ausgewählten Prozess verschickte Nachricht unterwegs verloren geht. An welcher Stelle die Nachricht zwischen dem Sende- und Empfängerprozess verloren geht wird zufällig bestimmt.

- **Maximale Übertragungszeit** *(Long: message.sendingtime.max = 2000)*: Gibt die Dauer in Millisekunden an, die eine vom Prozess verschickte Nachricht maximal benötigt, um einen Empfängerprozess zu erreichen. Im weiteren Verlauf wird dieser Wert mit t_{max} bezeichnet.

- **Minimale Übertragungszeit** *(Long: message.sendingtime.min = 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.

  Wenn die Übertragungszeiten von Nachrichten immer exakt die selbe Zeit in Anspruch nehmen sollen, dann müssen alle Prozesseinstellungen mit $t_{min} = t_{max}$ konfiguriert werden. Im Folgenden wird die globale Zeit mit t_g bezeichnet. Ist die Simulationseinstellung “Mittelwerte der Übertragungszeiten bilden” nicht aktiv, dann wird die Ereigniseintrittszeit t_e für den Empfang der Nachricht wie folgt berechnet:

$$t_e := t_g + rand(t_{min}, t_{max})$$

Das heißt, dass die Nachricht nach einer zufälligen Zeit zwischen t_{min} und t_{max} beim Empfänger eintrifft. Für jeden Empfänger wird hierbei ein neuer Zufallswert gewählt. Für den Fall, dass die Einstellung “Mittelwerte der Übertragungszeiten bilden” aktiviert ist, und wenn t'_{min} und t'_{max} die beim Empfängerprozess eingestellten Werte entsprechen, dann wird die Nachrichtenempfangszeit wie folgt berechnet:

$$t_e := t_g + \frac{1}{2}(rand(t_{min}, t_{max}) + rand(t'_{min}, t'_{max}))$$

Das bedeutet, dass der Mittelwert der Nachrichtenübertragungszeiten vom Sender- und Empfängerprozesses verwendet wird.

Im Prozesseditor lassen sich ebenfalls die Protokollvariablen editieren. Die Protokollvariablen werden in Kapitel 3. beschrieben.

2.4.4. Einstellungen im Expertenmodus

Im Expertenmodus lassen sich zusätzliche Variablen, wie Farbwerte und Fenstergrößen, editieren. Abbildung 2.15. zeigt alle einstellbaren Farben. Die fett gedruckten Schlüssel in Tabelle 2.3. sind mit Standardwerten für die neu zu erstellenden Prozesse belegt. Alle Werte sind in den Prozesseinstellungen des jeweiligen Prozesses individuell einstellbar.

Kapitel 3.

Protokolle und Beispiele

Im Folgenden werden alle verfügbaren Protokolle behandelt. Wie bereits beschrieben wird bei Protokollen zwischen Server- und Clientseite unterschieden. Server können auf Clientnachrichten, und Clients 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 der simulatoreigenen API (Application Programming Interface) erstellen (s. Kap. 4.4.4.). Im Programmverzeichnis des Simulators befindet sich das Verzeichnis *saved-simulations* mit Beispielsimulationen. Diese liegen jeweils als serialisierter plattformunabhängiger Java-Bytecode in *.dat*-Dateien vor. Alle Protokolle, bis auf das Beispiel-, Ping Pong- sowie das Broadcast-Protokoll, orientieren sich an den in [Tan03] und [Oßm07] behandelten Protokollen.

3.1. Beispiel (Dummy) Protokoll

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

3.2. Das Ping-Pong Protokoll *(ping-pong.dat, ping-pong-sturm.dat)*

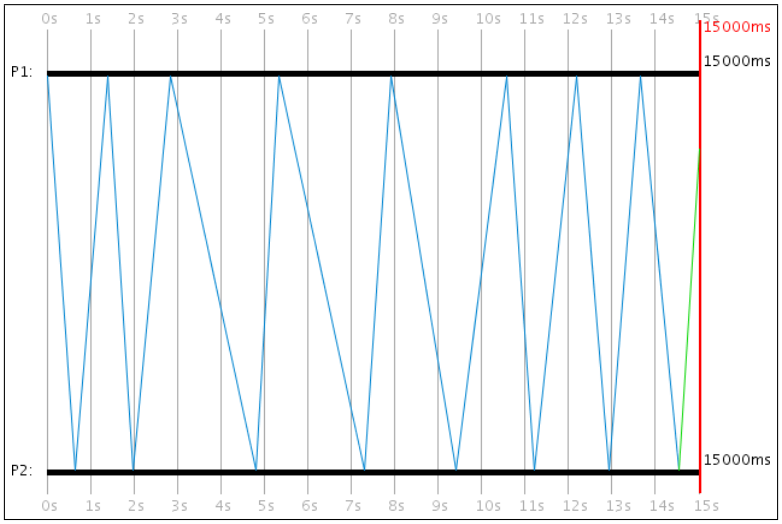

Abbildung 3.1.: Das Ping-Pong Protokoll

Bei dem Ping-Pong Protokoll (s. Abb. 3.1.) 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 Logfenster protokolliert wird. In Tabelle 3.1. sind alle für dieses Beispiel programmierten Ereignisse aufgeführt.

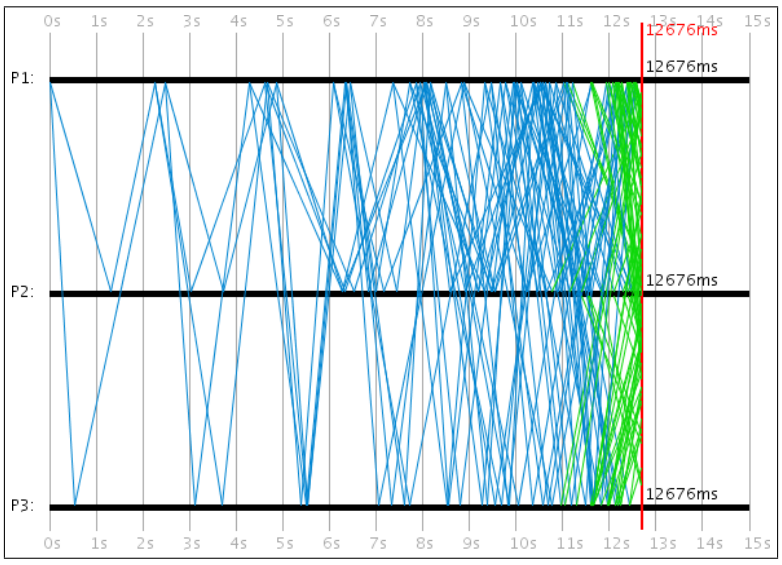

Abbildung 3.2.: Das Ping-Pong Protokoll (Sturm)

Wichtig ist, dass Prozess 1 seinen Ping-Pong Client aktiviert, bevor er eine Ping-Pong Clientanfrage startet.

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

Tabelle 3.1.: Programmierte Ping-Pong Ereignisse

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 3.2.: Programmierte Ping-Pong Ereignisse (Sturm)

Wenn die Eintrittszeiten für die Aktivierung des Protokolls und das Starten der Anfrage identisch sind, ordnet der Task-Manager (s. Kap. 4.4.3.) diese Ereignisse automatisch in der richtigen Reihenfolge an. Bei nicht aktiviertem Ping-Pong Client kann P1 keine Ping-Pong Anfrage starten. Bevor ein Prozess eine Anfrage starten kann, muss er das dazugehörige Protokoll aktiviert haben. Entsprechend gilt dies auch für alle anderen Protokolle. Anhand dieses Beispiels ist erkennbar, dass die noch nicht ausgelieferte Nachrichte grün eingefärbt ist während alle ausgelieferten Nachrichten bereits die Farbe Blau tragen (s. Tabelle 2.1.).

Werden die Ereignisse wie in Tabelle 3.2. vorgegeben, so lässt sich ein Ping-Pong Sturm realisieren. Hier wird ein neuer Prozess P3 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 Nachrichten. In Abbildung 3.2. ist der dazugehörige Simulationsverlauf bis zum Zeitpunkt *12676ms* dargestellt.

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 3.3.: Programmierte Broadcast Ereignisse

3.3. Das Broadcast Protokoll *(broadcast.dat)*

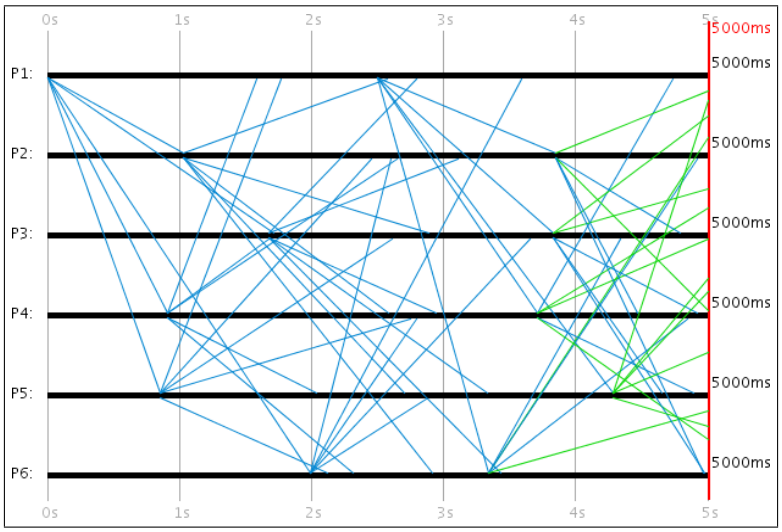

Abbildung 3.3.: Das Broadcast Protokoll

Das Broadcast Protokoll verhält sich ähnlich wie das Ping-Pong Protokoll. Der Unterschied ist, dass sich das Protokoll anhand einer eindeutigen Broadcast-ID merkt, welche Nachrichten bereits verschickt wurden. Jeder Prozess verschickt beim Broadcast Protokoll alle erhaltenen Nachrichten erneut, sofern er sie noch nicht schon einmal verschickt hat.

In diesem Fall wird nicht zwischen Client und Server unterschieden, so dass bei der Ankunft einer Nachricht

jeweils die gleiche Aktion durchgeführt wird. Somit lässt sich, unter Verwendung mehrerer Prozesse (s. Abb. 3.3.) 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 (s. Tabelle 3.3) gleichzeitig Server und Client.

3.4. Das Protokoll zur internen Synchronisierung in einem synchronen System *(int-sync.dat)*

Bisher wurden nur Protokolle dargestellt, in denen die beteiligten Prozesse keine Uhrabweichungen 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 Prozesszeit 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 eine neue und genauere Prozesszeit für sich berechnen kann.

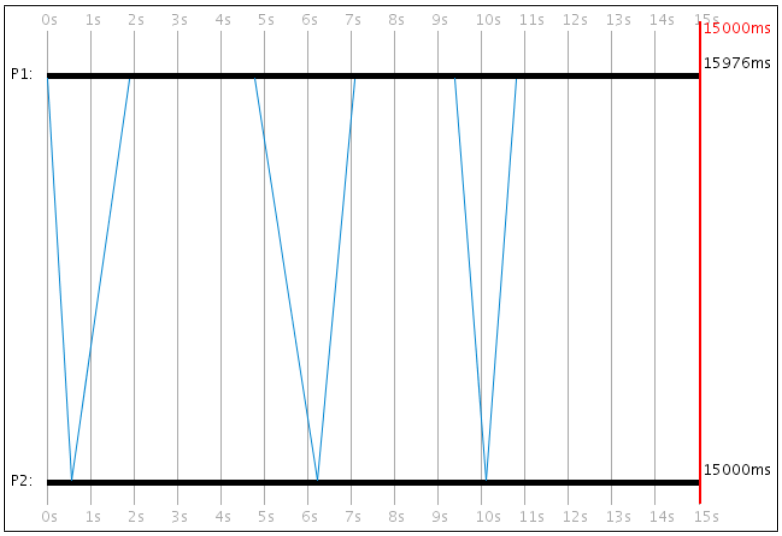

Abbildung 3.4.: Das Protokoll zur internen Synchronisierung

Hier (s. Abb. 3.4.) stellt P1 den Client und P2 den Server dar. Da die Übertragungszeit t_u einer Nachricht zwischen den vermuteten Werten t'_{min} und t'_{max} liegt, berechnet der Client P1 nach Empfang der Serverantwort seine neue lokale Prozesszeit mit:

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

Somit wird die lokale Zeit von P1, mit einem Fehler von $< \frac{1}{2}(t'_{max} - t'_{min})$, mit der Serverzeit synchronisiert (siehe [Oßm07]).

Im Beispiel hat der Clientprozess als Uhrabweichung den Wert *0.1* und der Server hat als Uhrabweichung den Wert *0.0* konfiguriert. Der Client startet, wie in Tabelle 3.4. angegeben, nach *0ms*, *5000ms* und *10000ms* seiner lokalen Prozesszeit jeweils eine Clientanfrage. In der Abbildung lässt sich erkennen, dass die zweite und die dritte

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 3.4.: Programmierte Ereignisse zur internen Synchronisierung

Anfrage nicht synchron zu der globalen Zeit gestartet wurden, 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} (s. Kap. 2.4.3.) unterscheiden. Somit lassen sich auch Szenarien simulieren, in denen das Protokoll falsch eingestellt wurde und wo bei der Zeitsynchronisierung große Fehler auftreten können.

3.5. Christians Methode zur externen Synchronisierung *(ext-vs-int-sync.dat)*

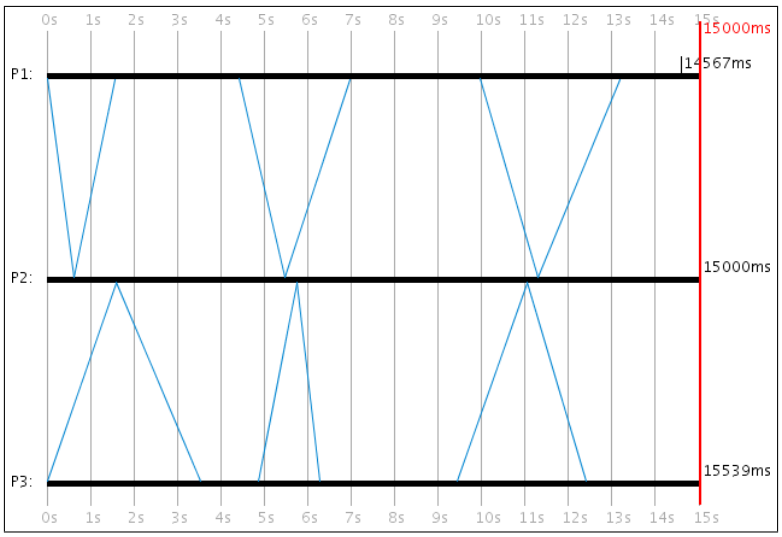

Abbildung 3.5.: Interne Synchronisierung und Christians Methode im Vergleich

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), um die Übertragungszeit 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 des Servers von dem Zeitpunkt an dem der Server die Antwort verschickte. Der Client berechnet dann seine lokale Zeit neu mit:

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

Die Genauigkeit beträgt $\pm(\frac{1}{2}t_{rtt} - u_{min})$ wobei u_{min} mit $t_{rtt} < u_{min}$ eine Schranke für eine Nachrichtenübertragung darstellt(s. [Oß m07]).

Im Prinzip sieht der Verlauf einer Christians-Simulation so aus wie in Abbildung 3.4., 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 (s. Abb. 3.5.). 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 (s. Tabelle 3.5.). P1 und P3 haben als Uhrabweichung einen Wert von *0.1* eingestellt und die Simulationsdauer beträgt insgesamt *15000ms*.

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 3.5.: Programmierte Ereignisse, Vergleich interner und externer Synchronisierung

Aus Abbildung 3.5. ist ersichtlich, 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 immer zwingend der Fall, da nach einer erneuten Simulationsausführung alle Nachrichten jeweils eine neue zufällige Übertragungszeit zwischen t_{min} und t_{max} haben werden, die auf die Zeitsynchronisation mit den einem oder anderem Protokoll jeweils andere Auswirkungen haben können.

3.6. Der Berkeley Algorithmus zur internen Synchronisierung *(berkeley.dat)*

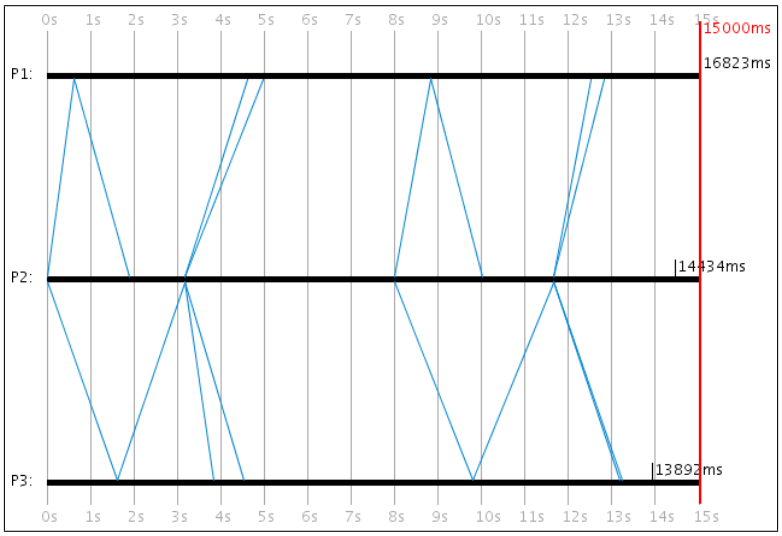

Abbildung 3.6.: Der Berkeley Algorithmus zur internen Synchronisierung

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 fungiert gewissermaßen als Koordinator des Protokolls. Die Clientprozesse 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 lokale Zeit t_s und auch die 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 in der Berechnung von t_{avg} wie folgt 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 an die einzelnen Clients zurückschickt. Diese setzten dann jeweils die lokalen Prozesszeiten

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 3.6.: Programmierte Ereignisse zum Berkeley Algorithmus

auf $t_i' := t_i' + k_i$. Wobei t_i' die lokale, aktuelle Prozesszeit des jeweiligen Clients ist.

Im Beispiel in Abbildung 3.6. gibt es die 2 Clientprozesse P1 und P3 sowie den Serverprozess P2. Der Server startet nach jeweils *0ms* und *7500ms* eine Synchronisierungsanfrage (s. Tabelle 3.6.). 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 überprüft 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 hierbei keine Einstellungsmöglichkeiten.

- **PIDs beteiligter 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 nicht existiert oder die das Berkeley Protokoll clientseitig nicht unterstützt. In diesem Fall würde ewig auf eine (fehlende) Clientantwort gewartet werden.

3.7. Das Ein-Phasen Commit Protokoll *(one-phase-commit.dat)*

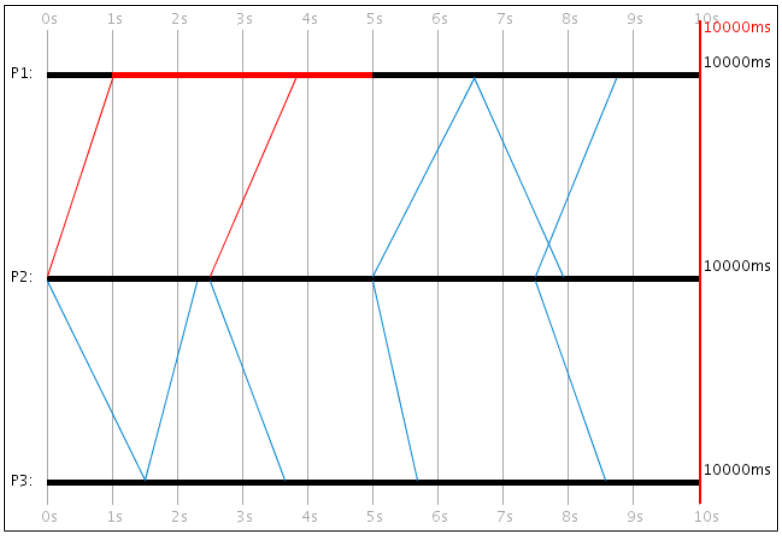

Abbildung 3.7.: Das Ein-Phasen Commit Protokoll

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. Hierfür müssen für den Server die PIDs aller beteiligten Clientprozesse sowie ein Wecker für erneutes Versenden des Festschreibewunsches konfiguriert werden.

Die programmierten Ereignisse des Beispiels auf (s. Abb. 3.7.) sind in Tabelle 3.7. 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 beiden 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 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 (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 3.7.: Programmierte Ein-Phasen Commit Ereignisse

- **Zeit bis erneute Anfrage** *(Long: timeout = 2500)*: Gibt die Anzahl von Millisekunden an, die gewartet werden sollen, bis der Festschreibewunsch erneut verschickt wird.
- **PIDs beteiligter Prozesse** *(Integer[]: pids = [1,3])*: Dieser Vektor aus Integerwerten beinhaltet alle PIDs der Clientprozesse, die festschreiben sollen.

3.8. Das Zwei-Phasen Commit Protokoll *(two-phase-commit.dat)*

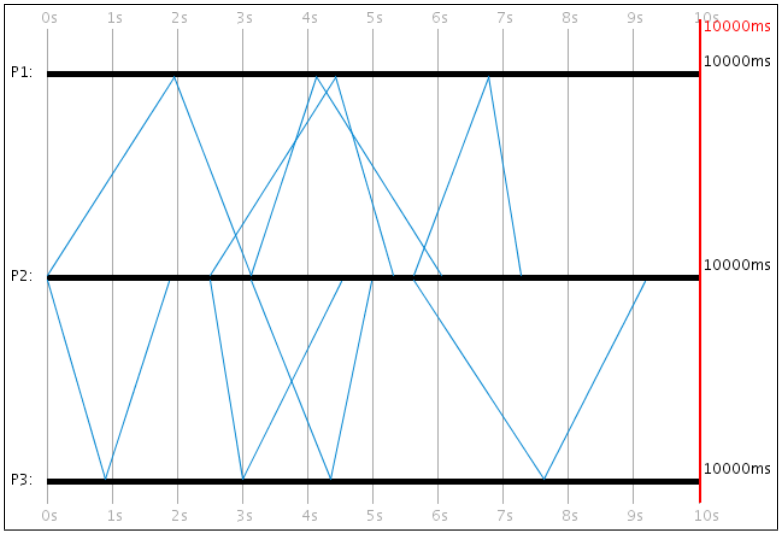

Abbildung 3.8.: Das Zwei-Phasen Commit Protokoll

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 periodisch so oft 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* an alle Clients 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 (s. Abb. 3.8.) sind P1 und P3 Clients und P2 der Server. Der Server verschickt nach *0ms* seine erste Anfrage (s. Tabelle 3.8.). Da diese Simulation recht unübersichtlich ist, liegen in den Tabellen 3.9. und 3.10. Auszüge aus dem Logfenster vor. Die Lamport- und Vektor-Zeitstempel sowie die lokalen Prozesszeiten sind hier nicht aufgeführt. Da keine Uhrabweichungen konfiguriert wurden, sind die lokalen Prozesszeiten stets identisch mit der globalen Zeit, weswegen in den Tabellen pro Logeintrag jeweils nur eine Zeit angegeben ist. Anhand der Nachrichten IDs lassen sich dort die einzelnen Sendungen zuordnen. In den Logs wird auch der Inhalt der verschickten Nachricht sowie die dazugehörigen Datentypen aufgeführt. Hier stimmen P1 und P3 jeweils mit *true* ab, das heißt es soll festgeschrieben werden.

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 3.8.: Programmierte Zwei-Phasen Commit Ereignisse

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

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

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

Zeit (ms)	PID	Lognachricht
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 3.9.: Auszug aus dem Logfenster des Zwei-Phasen Commit Beispiels

Zeit (ms)	PID	Lognachricht
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 3.10.: Auszug aus dem Logfenster 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 3.11.: Programmierte Basic-Multicast Ereignisse

3.9. Der ungenügende (Basic) Multicast *(basic-multicast.dat)*

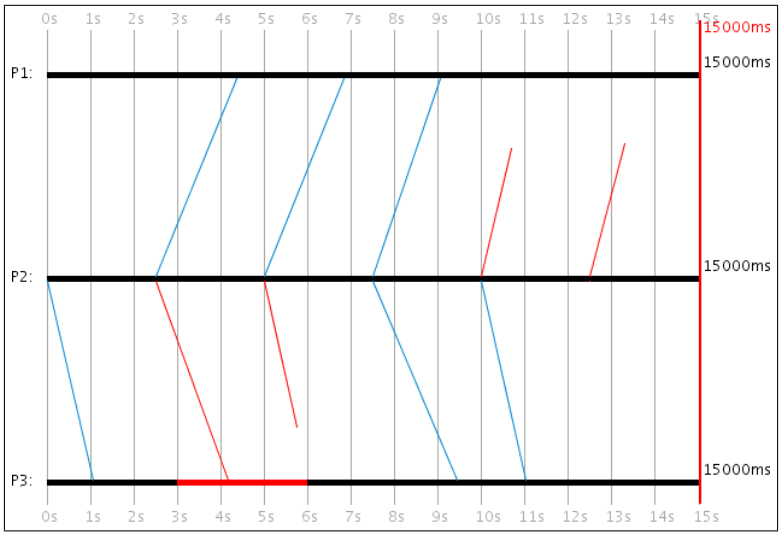

Abbildung 3.9.: Das Basic-Multicast Protokoll

Das Basic-Multicast Protokoll ist sehr einfach aufgebaut. Im Beispiel in Abbildung 3.9. 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 darstellt. Die Basic-Multicast Server dienen dabei lediglich zum Empfang einer Nachricht. Es werden keine Bestätigungen verschickt. Wie in Tabelle 3.11. 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.

In diesem Beispiel simuliert P1 ein Server, 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, so dass 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 beide Ziele erreicht.

3.10. Das zuverlässige (Reliable) Multicast Protokoll *(reliable-multicast.dat)*

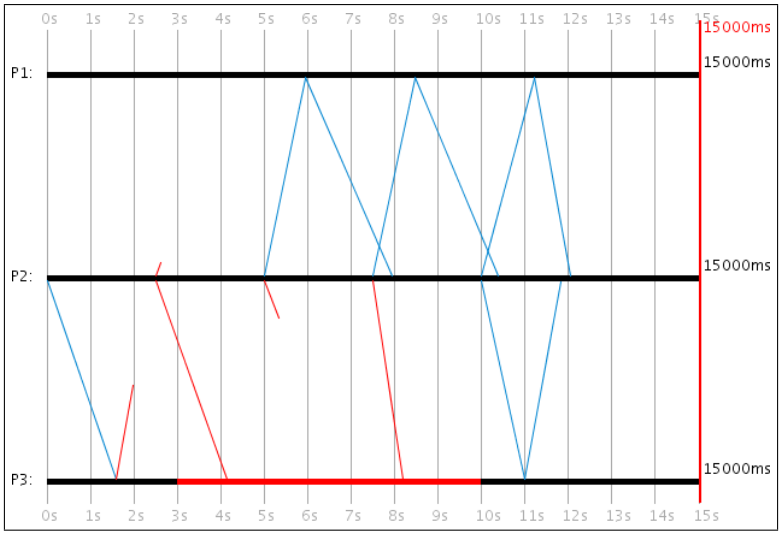

Abbildung 3.10.: Das Reliable-Multicast Protokoll

Bei dem zuverlässigen (Reliable) Multicast verschickt der Client periodisch so oft seine Multicast-Nachricht, 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, wodurch jede erneute Anfrage von allen Teilnehmern aufs Neue bestätigt werden muss. In dem Beispiel (s. Abb. 3.10., Tabelle 3.12., 3.13. und 3.14.) ist P2 der Multicast-verschickende Client, während P1 und P3 die Server darstellen. 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:

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 3.12.: Programmierte Reliable-Multicast Ereignisse

- 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.

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

Zeit (ms)	PID	Lognachricht
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

Tabelle 3.13.: Auszug aus dem Logfenster des Reliable-Multicast Beispiels

3.11. Weitere Beispiele

Bisher wurden alle verfügbaren Protokolle anhand von Beispielen aufgeführt. Mit dem Simulator lassen sich jedoch noch weitere Szenarien simulieren. Aus diesem Grund soll hier auf weitere Anwendungsbeispiele eingegangen werden.

3.11.1. Simulation von Lamport- und Vektor-Zeitstempel

Die Vektor- und Lamport-Zeitstempel lassen sich sehr gut am bereits behandeltem Beispiel des Berkeley-Protokoll's (vgl. Kap. 3.6.) demonstrieren. Nach Aktivierung des Lamportzeit-Schalters erscheint bei jedem Ereignis eines Prozesses der aktuelle Lamport-Zeitstempel (s. Abb. 3.11.). Jeder Prozess besitzt einen eigenen Lamport-Zeitstempel, der beim Versenden und Erhalten einer Nachricht inkrementiert wird. Jeder Nachricht wird die aktuelle Lamportzeit $t_l(i)$ des Senderprozesses i beigefügt. Wenn ein weiterer Prozess j diese Nachricht erhält, so wird der aktuelle Lamport-Zeitstempel $t_l(j)$ von Prozess j wie folgt neu berechnet:

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

Es wird also stets die größere Lamportzeit vom Sender- und Empfängerprozess verwendet und anschließend um

Zeit (ms)	PID	Lognachricht
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
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 3.14.: Auszug aus dem Logfenster des Reliable-Multicast Beispiels (2)

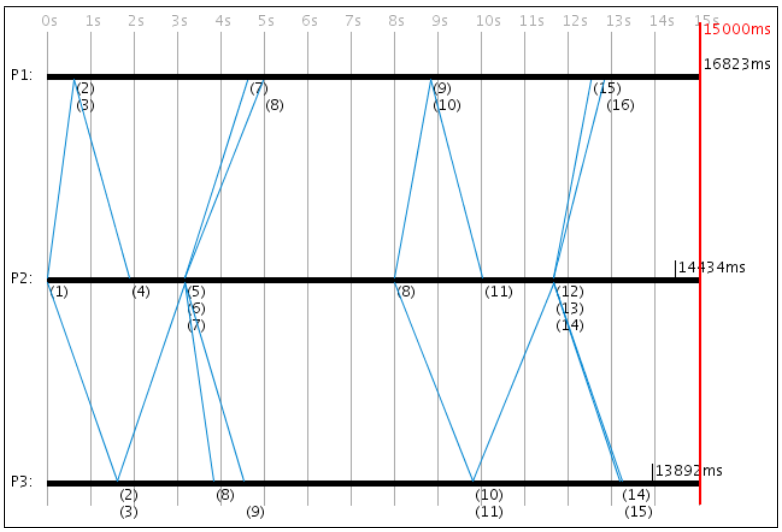

Abbildung 3.11.: Lamport-Zeitstempel

1 inkrementiert. Nach Ablauf der Berkeley-Simulation hat P1 *(16)*, P2 (*14*) und P3 (*15*) als Lamport-Zeitstempel abgespeichert.

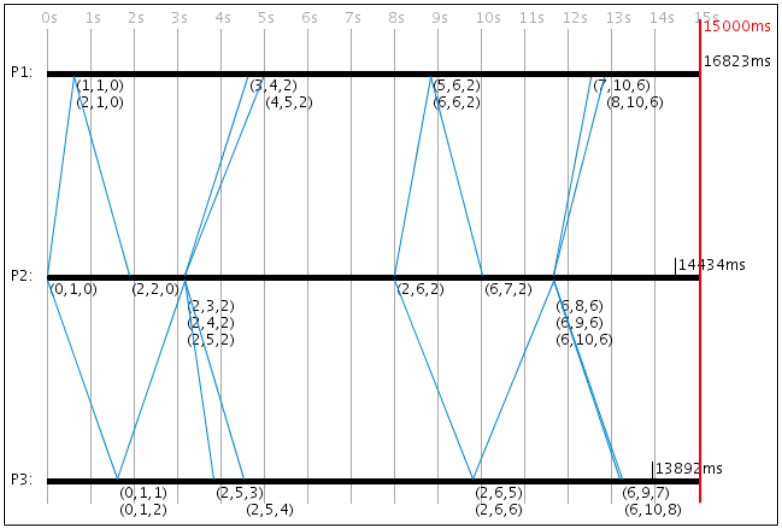

Abbildung 3.12.: Vektor-Zeitstempel

Mit aktivem Vektorzeit-Schalter werden alle Vektor-Zeitstempel angezeigt (s. Abb. 3.12.). Wie beim Lamport-Zeitstempel wird auch hier jeder Nachricht der aktuelle Vektor-Zeitstempel des Senderprozesses beigefügt. Bei n beteiligten Prozessen hat der Vektor-Zeitstempel v die Größe n. Somit gibt es für jeden beteiligten Prozess i einen eigenen Index i. Mit $v(i)$ kann jeder Prozess auf seinen lokalen Eintrag zugreifen. Wenn v der Vektor-Zeitstempel des Empfängerprozesses j ist und w der Vektor-Zeitstempel des Senderprozesses ist, dann wird der neue lokale Vektor-Zeitstempel vom Prozess j wie folgt neu berechnet:

Pseudo-Code

```
for (i := 0; i < n; i++) {
    if (i = j) {
        v(i)++;
    } else if (v(i) < w(i)) {
        v(i) := w(i);
    }
}
```

Standardmäßig wird der Vektor-Zeitstempel nur inkrementiert, wenn eine Nachricht verschickt- oder erhalten wird. Bei beiden Fällen inkrementiert der Sender- und Empfängerprozess jeweils seinen eigenen Index im Vektor-Zeitstempel um *1*. Beim Empfang einer Nachricht wird anschließend der lokale Vektor-Zeitstempel mit dem des Senderprozesses verglichen und für alle Indizes stets der größere Wert in den lokalen Vektor-Zeitstempel übernommen.

Im Beispiel (vgl. Abbildung 3.12.) hat P1 *(8,10,6)*, P2 *(6,10,6)*. und P3 *(6,10,8)* als Vektor-Zeitstempel abgespeichert.

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

Wie bereits beschrieben (s. Kap. 2.4.2.) 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].

3.11.2. Simulation langsamer Verbindungen *(slow-connection.dat)*

Mit dem Simulator lassen sich auch langsame Verbindungen zu einem bestimmten Prozess simulieren. Für die Demonstration wird das Beispiel aus Kapitel 3.5. wieder aufgegriffen, wo das Protokoll zur internen Synchronisation (durch P1) mit der Christians-Methode (durch P3) parallel simuliert wurden. P2 stellt den Server beider Protokolle zur Verfügung. In diesem Szenario soll P3 eine schlechte Netzwerkverbindung besitzen, so dass Nachrichten von- und an P3 stets eine längere Übertragungszeit benötigen.

Die Ereignisse sind so wie bereits in Tabelle 3.5. dargestellt wurde programmiert. In den Simulationseinstellungen ist hier die Einstellung “Mittelwerte der Übertragungszeiten bilden” aktiviert. In den Prozesseinstellungen von P3 wurde die “Minimale Übertragungszeit” auf *2000ms* und die “Maximale Übertragungszeit” auf *8000ms* gestellt. P1 und P2 behalten als Standardeinstellungen für die minimale und maximale Übertragungszeiten die Werte *500ms* und *2000ms*, die Simulationsdauer beträgt nun *20000ms*.

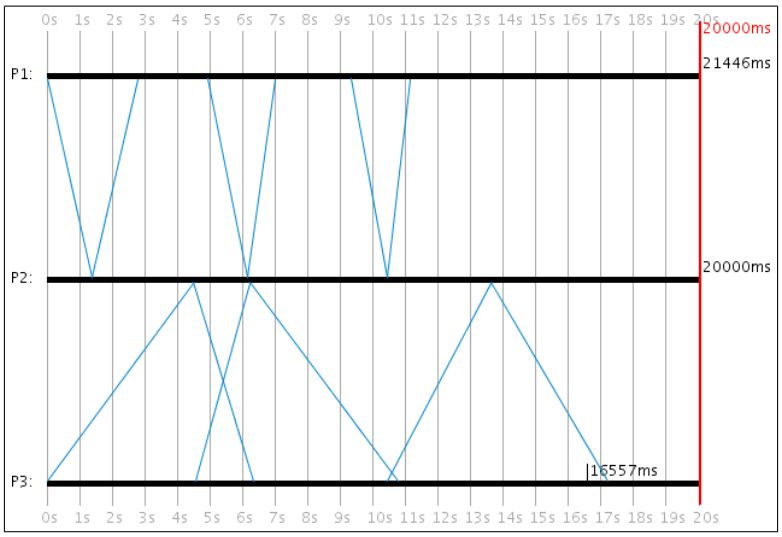

Abbildung 3.13.: Simulation einer langsamen Verbindung

Als Folge (s. Abb. 3.13.) benötigen Nachrichten, die von- und an P3 verschickt werden, für eine Übertragung immer mehr Zeit. Bevor P3 eine Antwort auf seine vorherige Anfrage bekommt, verschickt er eine erneute Anfrage. Da P3 die Serverantworten immer stets seiner letzten verschickten Anfrage zuordnet, berechnet er alle RTTs inkorrekt und seine lokale Zeit wird dadurch bei jedem Durchgang erneut falsch bestimmt. Die Berechnungsformeln der Übertragungszeiten wurde bereits in Kapitel 2.4.3. bei den Prozesseinstellungen behandelt. Konkret bedeutet dies hier für die Übertragungszeiten alle Nachrichten von- und an P3 jeweils:

$$\frac{1}{2}(rand(500,2000)+rand(2000,8000))=\frac{1}{2}rand(2500,10000)=rand(1250,5000)ms$$

In dem Beispiel in Abbildung 3.13. ist die lokale Prozesszeit von P1 bis auf *20000 - 21446 = - 1446ms* synchronisiert, während die Prozesszeit von P3 ganze *20000 - 16557 = 3443ms* falsch geht.

Kapitel 4.

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 geeignetste Programmiersprache ist. Der Simulator ist somit auf jedem Betriebssystem ausführbar, für das es eine JRE (Java Runtime Environment) gibt. Da an der Fachhochschule Aachen die Programmiersprache Java gelehrt wird, sollten die Studenten die Möglichkeit haben durch eigene Erweiterungen des Simulators Protokolle entwerfen zu können. Der Simulator wurde mit dem derzeit aktuellsten Java SDK (Software Development Kit) in der Version 6 (1.6) entwickelt.

Im Folgenden wird der Quelltext auszugsweise behandelt. Der Quelltext erstreckt sich, einschließlich Kommentare, auf ca. 15.000 Zeilen Text und 61 Dateien. Der Umfang der generierten Quelltext-Dokumentationen im Javadoc-Format ist ca. 2MB groß. Alle folgenden UML-Diagramme stellen zwecks Übersichtlichkeit lediglich die wesentlichen Sachverhalte dar. Alle weiteren Details können im Quelltext und der dazugehörigen Dokumentation eingesehen werden. Die Paketstruktur des Quelltextes ist in Tabelle 4.1. in alphanumerischer Reihenfolge aufgeführt.

4.1. Einstellungen und Editoren

Der Verlauf einer Simulation ist von einer Vielzahl von Einstellungen abhängig. Da auf diese Einstellungen in Weiteren stets zurückgegriffen wird, werden die dazugehörigen Klassen zuerst betrachtet.

Einstellungsobjekte

In Abbildung 4.1. 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, damit stellt eine solche

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 4.1.: Die Paketstruktur

Instanz einen Container für diese Daten dar. In einem *VSPrefs*-Objekt speichert der Simulator alle Einstellungen ab. Zudem besitzt jedes Prozessobjekt und jedes Ereignisobjekt für lokale Einstellungen seine eigene Instanz von *VSPrefs*. Später wird gezeigt, wie Protokollobjekte auch als Ereignisse eingesetzt werden, womit Protokolleinstellungen auch in einem *VSPrefs*-Objekt abgespeichert werden können. Auch Nachrichtenobjekte besitzt hiervon eine eigene Instanz dieser Klasse, um die zu verschickenden Daten zwischen zu speichern.

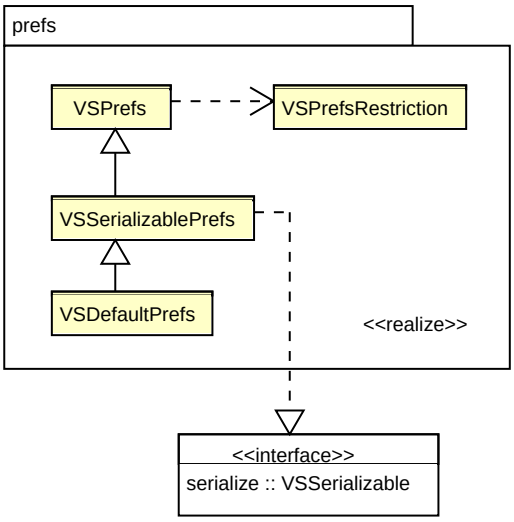

Abbildung 4.1.: Das Paket *prefs*

Jede Variable hat einen Datentypen, einen Variablennamen, eine optionale Beschreibung sowie einen Variablenwert. Einige Datentypen unterstützen auch die Angabe von Minimal- und Maximalwerten (z.B. besteht eine Prozentangabe aus einen Integerwert zwischen *0* und *100*), was mit Hilfe der Klasse *VSPrefsRestriction* imple-

mentiert wird. Da der Anwender 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.

Eine Variablenbeschreibung wird für die Darstellung im GUI verwendet, während der Variablenname für die interne Verwendung vom Simulator verwendet wird. Zum Beispiel hat die Variable *message.prob.outage* (Verlustwahrscheinlichkeit einer Nachricht) als Variablenbeschreibung “Nachrichtenverlustw’keit”. Wenn für eine Variable keine Beschreibung existiert so werden für die Anzeige einer Variable der Datentyp und der Variablenname verwendet. Variablennamen verwenden die in Tabelle 4.2. angegebenen Präfixkonventionen. Alle verfügbaren Datentypen wurden bereits in Tabelle 2.2. aufgelistet. Die Klasse *VSPrefs* stellt für alle Variablentypen entsprechende Selektoren zur Verfügung.

Im Folgenden werden einige der existierenden Methoden aufgelistet, eine komplette Liste kann in der Quelltext-Dokumentation eingesehen werden. Die Methoden werden anhand des Integer-Datentyps verdeutlicht. 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 steht *key* für den Variablennamen- und *val* für den Variablenwert. *descr* ist die optionale Variablenbeschreibung. Es können sowohl Java’s Integer-Objekte, als auch Java’s primitiver Integer-Typ *int* verwendet werden. Ein *int*-Wert wird intern zwecks Serialisierbarkeit als Integer-Objekt abgespeichert. Die Methode *getIntegerKeySet* gibt alle vorhandenen Integer-Variablennamen (*key*s) als *Set* (s. [Javb]) zurück.

Die Klasse *VSPrefs* bietet auch eine Reihe von *initInteger*-Methoden an, welche sich von den *setInteger*-Methoden dadurch unterscheiden, dass sie einer Variable nur einen Wert zuweisen, wenn sie vorher noch nicht initialisiert

Variablen-Präfix	Beschreibung	Beispiel
col	Farbvariablen	*Color: col.background = Color-Objekt*
div	Diverse versteckte Variablen	*Integer: div.window.logsize = 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 4.2.: Konventionen für Präfixe von Variablennamen

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.

Die Klasse *VSPrefs* speichert alle Integervariablen in einem *HashMap<String,Integer>*-Objekt ab, wobei der String-Wert den Variablennamen *key* angibt. Für die Beschreibung *descr*, den Einheiten-String *unit* sowie möglichen Minimal- und Maximalwerte werden separate Instanzen von *HashMap* verwendet. Da die Selektoren von *VSPrefs* synchronisiert sind, können alle *HashMap*s aus verschiednenen Threads gleichzeitig verwendet werden.

Die Klasse *VSSerializablePrefs* implementiert das Interface *VSSerializable* und kann somit durch Serialisierung alle enthaltenen Daten in eine Datei abspeichern bzw. wieder in den Speicher laden (s. Kap. 4.4.6.).

Die Klasse *VSDefaultPrefs* erweitert *VSSerializablePrefs* und initialisiert bei ihrer Instantiierung automatisch alle verfügbaren Simulationsvariablen mit Standardwerten. Hier sind auch alle Spracheinstellungen abgelegt. Möchte man den Simulator in eine andere Sprache (z.B. Englisch) übersetzen, so muss lediglich diese Datei und die Protokoll-Klassen editiert werden. Die Spracheinstellungen sind einem referenzierten *VSPrefs*-Objekt als versteckte String-Variablen abgespeichert. Spracheinstellungen für Protokolle wurden in den Protokollklassen direkt angegeben, da dies mehr Komfort für Protokollentwickler bedeutet und damit für jede neu programmierte Textausgabe nicht *VSDefaultPrefs.java* editiert werden muss.

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

Editorobjekte

Wie Variablen intern abgespeichert werden, ist nun bekannt. Für das Editieren von Variablen werden Editor-Objekte verwendet. In Abbildung 4.2. ist die Klassenstruktur des dazugehörigen Paketes *prefs.editors* angegeben.

Die Basis eines solchen Editors ist die abstrakte Klasse *VSAbstractEditor*. Jedem Objekt dieser Klasse wird jeweils ein *VSPrefs*-Objekt zum Editieren übergeben. Ein Editor stellt alle verfügbaren nicht-versteckten Variablen des *VSPrefs*-Objektes im GUI dar und bietet die Möglichkeit diese Variablen zu editieren. Für das Editieren von Farbwerten wird auf die Klasse *VSColorChooser* zurückgegriffen. Die Klasse *VSEditorTable* ist für das *JTable*-Objekt aus Java's Swing-Bibliothek (s. [Loy02]) zuständig, welches zur graphischen Darstellung aller Variablen eingesetzt wird. Die abstrakte Klasse *VSAbstractBetterEditor* wurde zur Verbesserung der Wartbarkeit des Quelltextes als Zwischenklasse eingeführt.

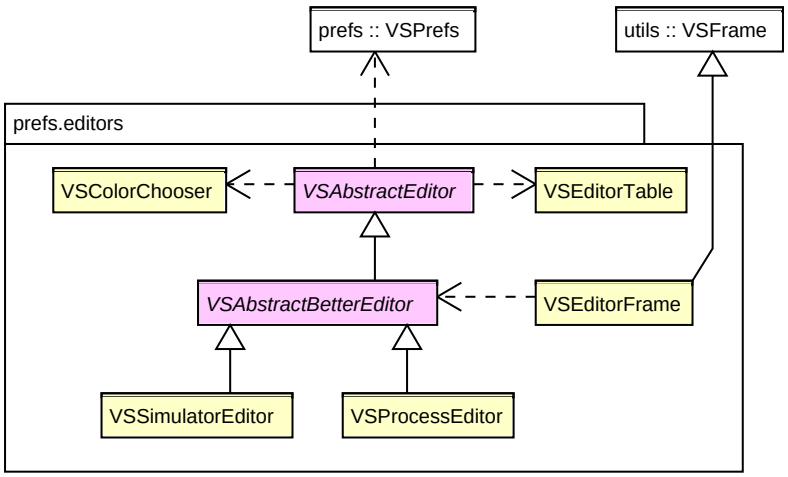

Abbildung 4.2.: Das Paket *prefs.editors*

Die Klasse *VSSimulatorEditor* erlaubt das Editieren der globalen Simulationseinstellungen und der *VSProcessEditor* das Editieren der Prozesseinstellungen sowie der dazugehörigen Protokollvariablen. Da diese beiden Klassen die abstrakte Klasse *VSAbstractBetterEditor* erweitern, können sie mit Hilfe von *VSEditorFrame* in einem separaten Fenster angezeigt werden. Alternativ können die Editoren auch in der Sidebar im Tab "Variablen" angezeigt werden. In Abbildung 2.14. wurde bereits ein *VSEditorFrame* in Aktion gesehen. In Abbildung 2.13. hingegen wurde ein Prozesseditor in der Sidebar geöffnet. Für Protokolle gibt es keine separate Editor-Klasse, da diese bereits im Prozesseditor editiert werden können. Hierbei iteriert der Prozesseditor über alle dem jeweiligen Prozess verfügbaren Protokollobjekte und fügt deren Variablen in den Prozesseditor ein. Somit erscheinen die Prozess- und die dazugehörigen Protokollvariablen im selben Editor und bieten dem Benutzer eine bessere Übersicht.

4.2. Ereignisse

Für jedes Ereignis existiert eine dazugehörige Klasse, welche die auszuführenden Aktionen implementiert. Eine Instanz einer solchen Klasse wird für eine spätere Ausführung dem Task-Manager (s. Kap. 4.4.3.) übergeben.

Jedes programmierbare Ereignis muss, bevor es vom Simulator verwendet werden kann, in der statischen Klasse *VSRegisteredEvents* registriert werden. Der Simulator bezieht die Liste aller verfügbaren Ereignisse aus *VSRegisterEvents*, wodurch der Entwickler bei der Entwicklung eines neuen Ereignisses keine andere Stelle im Quelltext des Simulators ändern muss. Da sich die Anzahl der verfügbaren Ereignisklassen des Simulators bei Laufzeit nicht ändert, gibt es keine Instanzen von *VSRegisteredEvents*. Alle Methoden und Klassenattribute sind statisch. Wenn beispielsweise eigene Ereignisse implementiert werden, dann müssen alle neuen Ereignisse per Hand in der Quelltext-Datei *VSRegisteredEvents.java* übernommen werden, und der Simulator muss neu kompiliert werden.

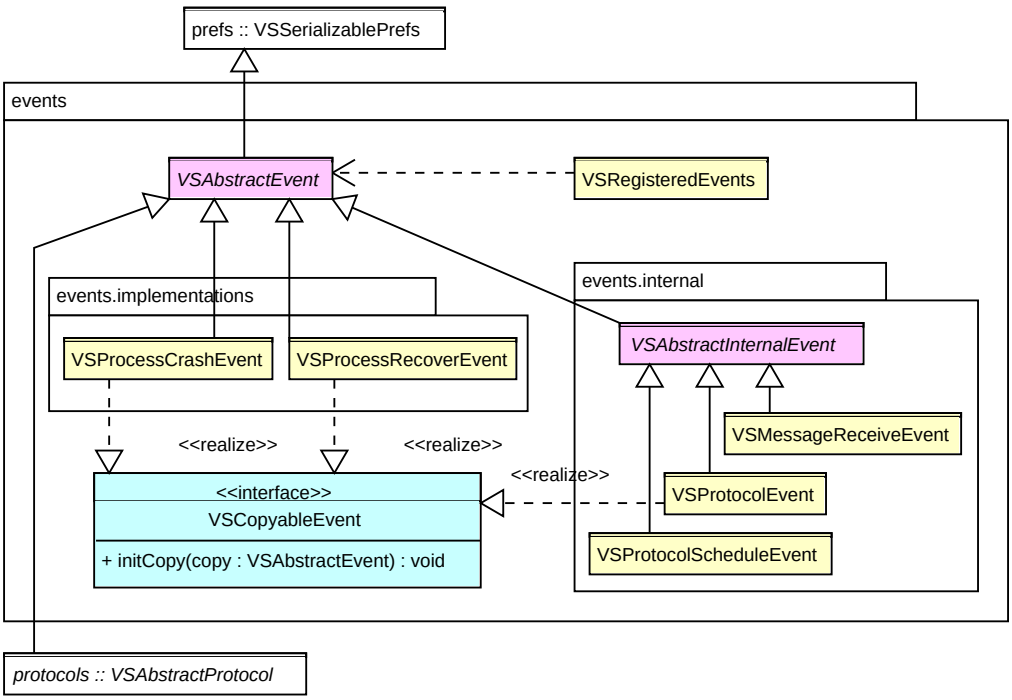

Abbildung 4.3.: Das Paket *events.**

In der Implementierung wird zwischen drei Haupttypen von Ereignissen unterschieden, die jeweils in einem unterschiedlichen Paket liegen (s. Abb. 4.3.):

1. *events.implementations*: In diesem Paket befinden sich alle Ereignisse, die ohne weitere Spezialbehandlung vom 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.

- *VSAbstractInternalEvent*: Diese Klasse stellt weitere Methoden zur Verfügung, die von allen internen Ereignissen zur Serialisierung benötigt werden.
- *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 gehörige Protokoll versteht. Diese Klasse überprüft auch die Simulationseinstellung “Nur relevante Nachrichten anzeigen” und entscheidet, ob die Nachricht nach Eintreffen in der Visualisierung und im Logfenster berücksichtigt werden soll oder nicht.
- *VSProtocolEvent*: Diese Klasse implementiert gleichzeitig vier verschiedene Ereignisse: Das Aktivieren bzw. Deaktivieren eines Servers oder 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, jeweils mit individuellen Attributwerten.
- *VSProtocolScheduleEvent*: Diese Klasse wird für die Wecker-Ereignisse benötigt. Wecker-Ereignisse können nur von Protokollen erstellt werden. *VSProtocolScheduleEvent* besitzt eine Referenz auf das verwendete 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 seine eigene Klasse. Alle Protokolle erben hierbei von der in Abbildung 4.3. 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- bzw. einer Clientanfrage entspricht (s. Kap. 4.4.4.).

Alle Ereignisse, die das Interface *VSCopyableEvent* implementieren, können vom Anwender im Ereigniseditor mit einem Rechtsklick kopiert werden. Des Weiteren müssen sie die Methode *initCopy(VSAbstractEvent copy)* implementieren. Es werden dann 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. Diese Methode wird pro Ereignisobjekt nach dessen Erzeugung 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 der Klasse *VSAbstractEvent* geerbt:

- *public void log(String message)*: Diese Methode schreibt eine Lognachricht in das Simulationslogfenster.
- *public VSAbstractEvent getCopy()*: Diese Methode erstellt vom aktuellen Ereignis eine Kopie, auf die eine Referenz zurückgegeben wird. Alle Ereignisse, die kopiert werden können, müssen auch das Interface *VSCopyableEvent* implementieren. Wenn ein Ereignis dies nicht tut und *getCopy()* aufgerufen wird, wird die Ausnahme *exceptions.VSEventNotCopyable* ausgelöst.
- *public VSAbstractEvent getCopy(VSAbstractProcess process)*: Diese Methode erstellt vom aktuellen Ereignis ebenfalls eine Kopie, mit dem Unterschied, dass das Ereignis einem anderen Prozess zugewiesen wird.

Jede Ereignisklasse hat außerdem Zugriff auf folgende Attribute, welche von *VSAbstractEvent* geerbt werden:

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

Da *VSAbstractEvent* die Klasse *VSSerializablePrefs* erweitert, sind alle Ereignisse mit allen ihren Variablen serialisierbar.

Beispielimplementierung eines Ereignisses

Im Folgenden wird als Beispiel die Implementierung des Prozessabsturzereignisses *VSProcessCrashEvent* behandelt. Da die dazugehörige Klasse keine Attribute besitzt, verbleibt auch hier die *initCopy*-Methode mit leerem Methodenrumpf. Aufgrund der Serializierbarkeit von Ereignisobjekten muss jede Ereignisklasse in *onInit()* mit *setClassname* den eigenen Klassennamen mitteilen. Bei der Deserialisierung von Ereignissen werden nämlich Objekte anhand der Klassennamen dynamisch neu erstellt, wobei der Klassenname stets bekannt sein muss. In *onStart()* wird das eigentliche Ereignis ausgeführt. Hier wird überprüft, ob der Prozess bereits abgestürzt ist und ggf. zum Absturz gebracht.

```
package events.implementations;

import events.*;
```

```
public class VSProcessCrashEvent
extends VSAbstractEvent implements VSCopyableEvent {
    public void initCopy(VSAbstractEvent copy) {
    }

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

    public void onStart() {
        if (!process.isCrashed()) {
            process.isCrashed(true);
            super.log(prefs.getString("lang.crashed"));
        }
    }
}
```

Der Task-Manager (s. Kap. 4.4.3.) überprüft bereits, ob der Prozess abgestürzt ist oder nicht. Das führt dazu, dass ein Ereignis bei einem abgestürztem Prozess gar nicht erst ausgeführt wird. Die einzige Ausnahme bildet ein Wiederbelebungsereignis (VSProcessRecover), welches auch dann ausgeführt wird, auch wenn der Prozess abgestürzt ist. Mit *log* wird eine Nachricht (die über *prefs* bezogen wird) in das Logfenster geschrieben.

In der Datei *events/VSRegisteredEvents.java* muss in der *init*-Methode für jedes Ereignis ein Eintrag existieren. Die *init*-Methode wird einmal beim Starten des Simulators ausgeführt:

```
public static void init(VSPrefs prefs_) {
    .
    .
    .
    registerEvent("events.implementations.VSProcessCrashEvent",
                  "Prozessabsturz");
    .
    .
    .
}
```

4.3. Zeitformate, Prozesse, Nachrichten sowie Task-Manager

Das Paket *core.time* in Abbildung 4.4. stellt lediglich die Klassen für die Vektor- und Lamport-Zeitstempel zur Verfügung. Für die normale lokale Prozesszeit wird, aus Performance-Gründen, keine eigene Klasse, sondern ein einfaches *long*-Attribut des Prozessobjektes verwendet.

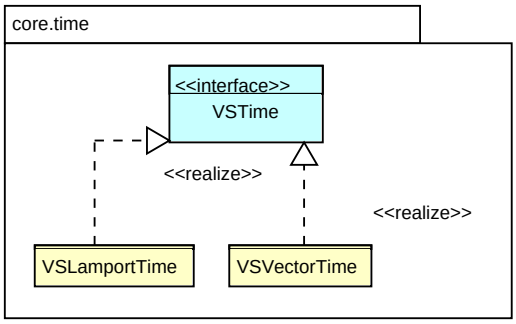

Abbildung 4.4.: Das Paket *core.time*

In Abbildung 4.5. ist das Paket *core* dargestellt. Für jedes auszuführende Ereignis wird eine Instanz von *VSTask* benötigt, welche die Ereigniseintrittszeit als Attribut abgespeichert. Die Instanz besitzt eine Referenz auf das Ereignisobjekt (*VSAbstractEvent*) sowie das Prozessobjekt (*VSInternalProcess*). Geplante *VSTask*-Instanzen werden dem Task-Manager für eine spätere Ausführung übergeben.

Die Kapselung eines *VSAbstractEvent*-Objektes in einem *VSTask*-Objekt erlaubt es, dass die gleiche *VSAbstractEvent*-Instanz mehrmals im Task-Manager geplant werden kann. Ohne dieser Kapselung gäbe es für jedes Ereignis nur eine einzige mögliche Eintrittszeit. Von dieser Möglichkeit wird z.B. 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 (vgl. [Cor01], [Sed99]) 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.

Eine Instanz von *VSMessage* stellt eine Nachricht dar, die von einem Prozess verschickt wird. Da *VSMessage* von *VSPrefs* erbt, können zwischen zwei Prozessen beliebige Datentypen (s. Tabelle 2.2.) über eine Nachricht verschickt werden. Anschließend wird für jeden Empfängerprozess ein neues Ereignisobjekt der Klasse *VSMessageReceiveEvent* angelegt, welches eine Referenz der verschickten Nachricht besitzt (s. Abb. 4.6.). Danach wird ein *VSTask*-Objekt instantiiert, in dem 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 hier das *VSMessageReceiveEvent*-Objekt in ein *VSAbstractEvent* umgewandelt und so in *VSTask* abgelegt.

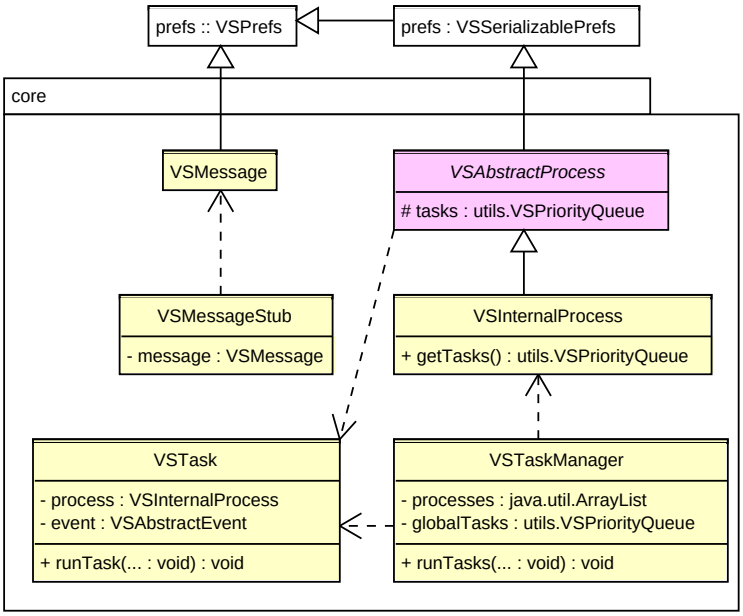

Abbildung 4.5.: Das Paket *core*

Erwähnenswert ist auch die Klasse *VSMessageStub*, welche ein *VSMessage* kapselt. Ihr Zweck ist das Verstecken einiger Methoden vor dem Protokoll-API, welches für die Erstellung eigener Protokolle dient. Der Protokoll-Entwickler soll möglichst nichts falsch machen können und deswegen soll dem Protokoll-API ein eingeschränkter Funktionsumfang zur Verfügung gestellt werden. Da sich *VSMessageStub* im selben Paket wie *VSMessage* befindet, kann *VSMessageStub* auf paket-private Methoden 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 Methoden er zugreifen sollte, und auf welche nicht.

Der Task-Manager speichert anschließend die Nachrichtenempfangsereignisse in seiner globalen Warteschlange. 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 Vektor-Zeitstempel des sendenden Prozesses als Attribut beigefügt. Für die Überprüfung des Protokolls wird in jeder Nachricht auch der Klassenname des jeweiligen Protokolls abgespeichert.

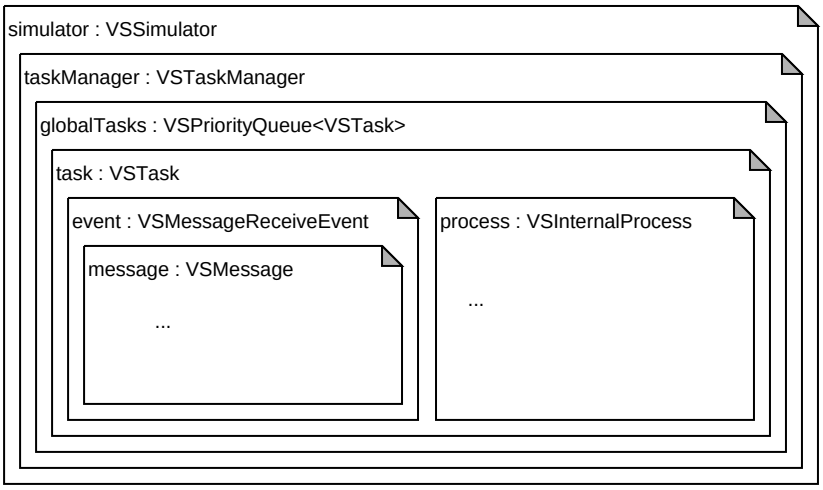

Abbildung 4.6.: Gekapseltes *VSMessage*-Objekt

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 der Protokollentwickler 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 dies auch gar nicht direkt möglich ist. Hier wäre auch ein Stub-Objekt *VSProcessStub* denkbar gewesen. Da aber fast jede Millisekunde auf die Methoden von *VSInternalProcess* zugegriffen wird, wurde hier aus Performance-Gründen der Weg über eine Vererbungsstufe preferiert.

Alle einstellbaren Prozessvariablen werden von der Klasse *VSPrefs* geerbt. Damit bei Neuberechnungen die Variablen nicht dauernd über eine *HashMap* von *VSPrefs* zugegriffen werden muss, speichert *VSInternalProcess* aus Performance-Gründen 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*, bzw. die lokalen Kopien, auf den neuesten Stand gebracht. Selbiges gilt für weitere Variablen, wie z.B. der Uhrabweichung eines Prozesses.

Beispiel für die Erstellung von Prozessereignissen

Anhand der Prozessabsturz- und Wiederbelebungsereignisse lässt sich wie folgt sehr gut demonstrieren, 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);
    taskManager.addTask(localTask);

    VSAbstractEvent recoverEvent = new VSProcessRecoverEvent();
    VSTask globalTask = new VSTask(2000, process,
                                   recoverEvent, VSTask.GLOBAL);
    taskManager.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.

4.4. Protokoll-API

In diesem Abschnitt wird auf die Implementierung der Protokolle und das Protokoll-API eingegangen. Im Protokoll-API wird in der Regel nicht direkt auf den Task-Manager und auf die explizite Instantiierung von Ereignisobjekten zurückgegriffen, da dies vom API automatisch durchgeführt wird.

In Abbildung 4.9. 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. Im Prinzip besitzt jedes Prozessobjekt von jedem Protokoll seine eigene Instanz. Bei *10* Protokollen und *3* beteiligten Prozessen werden also *30* Protokollobjekte verwendet. Jedes Protokollobjekt verwaltet sowohl die Server- als auch die Clientseite eines Protokolls auf einmal. Dabei merkt sich *VSAbstractProtocol* anhand einer Flagge, ob der aktuelle Kontext server-

oder clientbezogen ist, und führt dementsprechend beim Eintreffen von Ereignissen die Server- bzw. Clientmethoden des Protokolls auf. *VSAbstractProtocol* überprüft auch, ob ein Client oder ein Server überhaupt aktiviert ist.

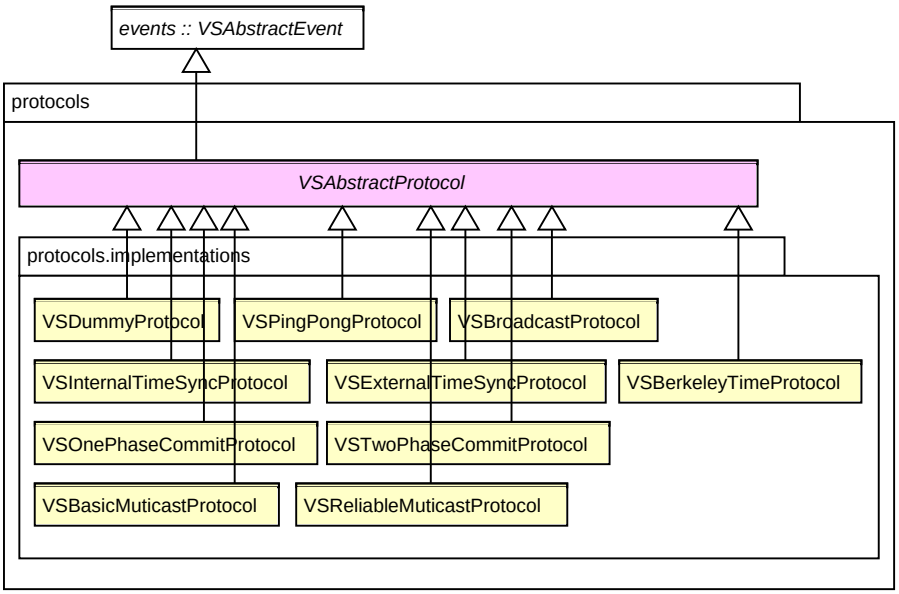

Abbildung 4.7.: Das Paket *protocols.**

Es ist bereits bekannt, dass Protokolle im Prozesseditor editierbare Variablen haben können. Da *VSAbstractProtocol* von *VSAbstractEvent* erbt, was wiederum von *VSSerializablePrefs* erbt (und *VSSerializablePrefs* erweitert *VSPrefs*), werden alle Protokollvariablen einfach in die Mutterklasse *VSPrefs* abgelegt. Zum Beispiel kann mit *super.setBoolean("test", true);* eine neue Protokollvariable *test* mit dem Standardwert *true* angelegt werden. Diese Variable erscheint dann automatisch im Prozesseditor und kann so vom Anwender konfiguriert werden.

Da der Simulator darauf ausgelegt wurde eigene Protokolle zu implementieren, werden im Folgenden alle verfügbaren Protokoll-API-Methoden etwas ausführlicher als gewohnt beschrieben. 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.
- *abstract public void onClientReset()*: Diese Methode wird jedes Mal ausgeführt, wenn die Simulation zurückgesetzt wird.

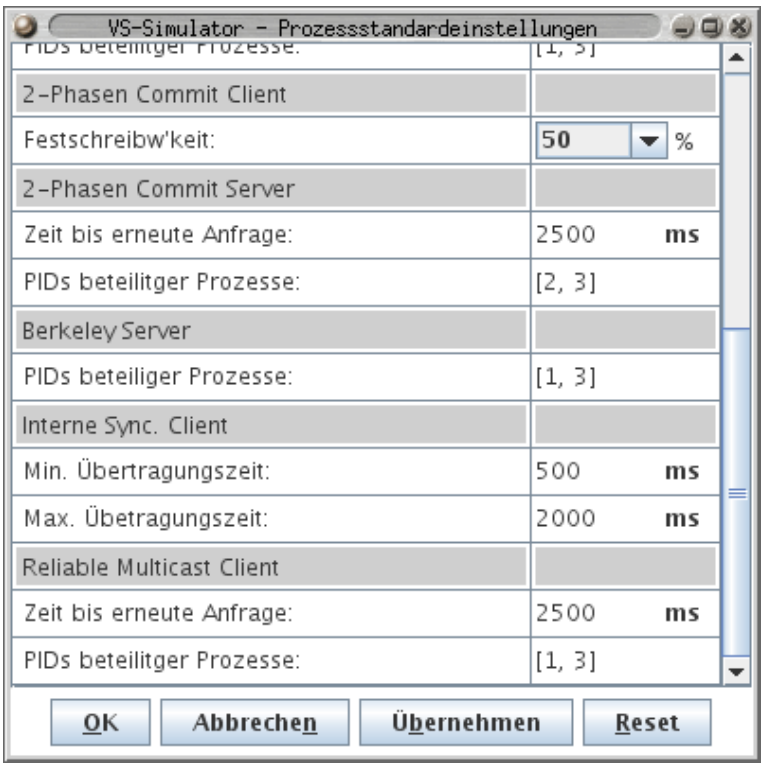

Abbildung 4.8.: Protokollvariablen im Prozesseditor

- *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 anderen beteiligten Prozesse verschickt wird.
- *abstract public void onClientRecv(VSMessage message)*: Diese Methode wird jedes Mal aufgerufen, wenn eine Servernachricht *message* bei dem Client eintrifft.
- *abstract public void onClientSchedule()*: Diese Methode wird jedes Mal ausgeführt, wenn ein Weckerereignis eintritt.
- *public String toString()*: Diese Methode ist nur optional. Hiermit lassen sich die Lognachrichten eines Protokolls anpassen. Wenn diese Methode in einer Protokollimplementierung ausgelassen wird, so wird stets die *toString*-Methode der Mutterklasse *VSAbstractProtocol* verwendet. Bei Verwendung wird empfohlen die Logausgabe lediglich wie folgt zu erweitern:

```
public String toString() {
    return super.toString() + "; Neue Loginformationen";
}
```

Hierbei wird jede Lognachricht, die das aktuelle Protokoll betrifft, um die Ausgabe *; Neue Loginformation* erweitert.

Für alle hier aufgelisteten Client-Methoden sind auch die korrespondierenden Server-Methoden anzugeben. Die Server-Methoden sind analog zu den Client-Methoden aufgebaut, wobei lediglich *Client* durch *Server* ausgetauscht werden muss.

Jede Protokollklasse erbt folgende Methoden von *VSAbstractProtocol*, welche vom Protokollentwickler verwendet werden können:

- *pubic void sendMessage(VSMessage message)*: Hiermit verschickt das Protokoll eine Nachricht.
- *pubic final boolean hasOnServerStart()*: Stellt fest, ob der Server- oder der Client bei dem aktuellen Protokoll die Anfragen startet.
- *pubic final boolean isServer()*: Hiermit lässt sich bestimmen, ob der aktuelle Prozess das aktuelle Protokoll serverseitig aktiviert hat.
- *pubic final boolean isClient()*: Hiermit lässt sich bestimmen, ob der aktuelle Prozess das aktuelle Protokoll clientseitig aktiviert hat.
- *pubic final void scheduleAt(long time)*: Diese Methode erstellt einen Wecker, der zur angegebenen Prozesszeit eintritt. Nach Ablauf des Weckers wird, abhängig vom aktuellen Kontext (server- oder clientseitig), *onClientSchedue* bzw. *onServerSchedule* ausgeführt.
- *pubic final void removeSchedules()*: Entfernt alle gesetzten Wecker im aktuellen Kontext.
- *pubic final int getNumProcesses()*: Gibt die Gesamtzahl an der Simulation beteiligten Prozesse zurück.

Bei der Implementierung von Protokollen kann zusätzlich auf die geerbten Attribute *VSAbstractProcess process* und *VSPrefs prefs* zugegriffen werden. Die verfügbare Methoden von *VSPrefs* wurden bereits behandelt. über *prefs* lassen sich alle globalen Simulationseinstellungen abrufen. Folgende Prozessmethoden dürfen aus dem Protokoll-API auf *process* angewendet werden:

- *public float getClockVariance()*: Gibt die Uhrabweichung zurück.
- *public void setClockVariance(float clockVariance)*: Setzt die Uhrabweichung des Prozesses.
- *public long getGlobalTime()*: Gibt die aktuelle globale Simulationszeit zurück.
- *public long getTime()*: Gibt die aktuelle lokale Prozesszeit zurück.
- *public void setTime(long time)*: Setzt die aktuelle lokale Prozesszeit.
- *public long getLamportTime()*: Gibt den aktuelle Lamport-Zeitstempel des Prozesses zurück.
- *public void setLamportTime(long lamportTime)*: Setzt den aktuellen Lamport-Zeitstempel des Prozesses.
- *public void increaseLamportTime()*: Inkrementiert den Lamport-Zeitstempel um eins.

- *public void updateLamportTime(long lamportTime)*: Erneuert den Lamport-Zeitstempel (vgl. Kap. 3.11.1.).
- *public VSVectorTime getVectorTime()*: Gibt den aktuelle Vektor-Zeitstempel des Prozesses zurück.
- *public VSTime[] getLamportTimeArray()*: Gibt die gesamte Lamport-Zeitstempel-Historie des Prozesses zurück. Kann nach VSLamportTime umgewandelt werden.
- *public VSTime getVectorTimeArray()*: Gibt die gesamte Vektor-Zeitstempel-Historie des Prozesses zurück. Kann nach VSVectorTime umgewandelt werden.
- *public void updateVectorTime(VSVectorTime vectorTimeUpdate)*: Erneuert den Vektor-Zeitstempel (vgl. Kap. 3.11.1.).
- *public void increaseVectorTime()*: Inkrementiert den Vektor-Zeitstempel am lokalen Index um eins.
- *public int getProcessID()*: Gibt die PID zurück.
- *public void setProcessID(int processID)*: Setzt die PID.
- *public int getProcessNum()*: Gibt die Prozessnummer zurück. Die Prozessnummer gibt an, um den wievielten Prozess es sich handelt. Die Prozessnummer ist nicht die PID.
- *public int getRandomPercentage()*: Gibt einen Zufallswert zwischen *0* und *100* zurück.
- *public boolean hasCrashed()*: Gibt *true* zurück, wenn der Prozess während der aktuellen Simulation bereits abgestürzt ist.
- *public boolean isCrashed()*: Gibt *true* zurück, wenn der Prozess zur Zeit abgestürzt ist.
- *public void isCrashed(boolean isCrashed)*: Hiermit kann man den Prozess abstürzen lassen (*isCrashed = true*) sowie wiederbeleben lassen (*isCrashed = false*).

In der Regel werden durch Protokolle Nachrichten (*VSMessage*) verschickt. Davon dürfen folgende Methoden des Protokoll-API verwendet werden:

- *public VSMessage()*: Der Standardkonstruktor für die Erstellung einer neuen Nachricht.
- *public int getMessageID()*: Gibt die Nachrichten-ID zurück.
- *public boolean equals(VSMessage message)*: Überprüft, ob eine weitere Nachricht die selbe NID besitzt.

Folgende weitere Methoden von *VSMessage* können nach Erhalt einer Nachricht verwendet werden:

- *public String getName()*: Gibt den Namen des zur Nachricht gehörenden Protokolls zurück.
- *public String getProtocolClassname()*: Gibt den Klassennamen des zur Nachricht gehörenden Protokolls zurück.
- *public VSAbstractProcess getSendingProcess()*: Gibt eine Referenz auf den Senderprozess zurück.
- *public long getLamportTime()*: Gibt den Lamport-Zeitstempel des Senderprozesses zurück.
- *public VSVectorTime getVectorTime()*: Gibt den Vektor-Zeitstempel des Senderprozesses zurück.

- *public boolean isServerMessage()*: Stellt fest, ob es sich um eine Server- oder eine Clientnachricht handelt.

Wenn über eine Nachricht Daten verschickt werden sollen, so werden die von *VSPrefs* gerbten Methoden verwendet.

Beispielimplementierung eines Protokolls

Im Folgenden wird die Implementierung des zuverlässigen Multicast-Protokolls *VSReliableMulticastProtocol.java* beispielhaft gezeigt. Die Funktionsweise des Protokolls wurde bereits in Kapitel 3.10. beschrieben. Client- und Serverseite werden hier in der selben Klasse implementiert.

Im Konstruktor muss stets angegeben werden, ob beim gegebenen Protokoll der Client oder der Server die Anfragen startet. Mit *VSAbstractProtocol.HAS_ON_CLIENT_START* wird dem Simulator mitgeteilt, dass der Client die Anfragen startet. Für *VSAbstractProtocol.HAS_ON_SERVER_START* und Serveranfragen gilt dies analog. Da ein Protokoll auch ein *VSAbstractEvent* ist, muss auch hier mit *setClassname* der Name der Klasse des aktuellen Protokolls angegeben werden:

```
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);
        super.setClassname(super.getClass().toString());
    }
```

Clientseite des Protokolls

Das private Klassenattribut *pids* wird für die Zwischenspeicherung beteiligter PIDs benötigt. Hier sind alle PIDs der Prozesse abgelegt, von denen noch Bestätigungsnachrichten erwartet werden. als Standard-PIDs werden *1* und *3* verwendet. Die Methoden *initVector* und *initLong* wurden von *VSPrefs* gerbt und initialisieren die Protokollvariablen *pids* und *timeout*, welche vom Benutzer im Prozesseditor editiert werden können (s. Abb. 4.8.):

```
    private ArrayList<Integer> pids;

    public void onClientInit() {
        Vector<Integer> vec = new Vector<Integer>();
        vec.add(1);
        vec.add(3);

        super.initVector("pids", vec,
                         "PIDs beteiligter Prozesse");
        super.initLong("timeout", 2500,
                       "Zeit bis erneute Anfrage", "ms");
    }
```

Wenn die Simulation zurückgesetzt wird, dann wird auch das Klassenattribut *pids* neu initialisiert:

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

In der Methode *onClientStart* wird geprüft, ob eine Clientanfrage gestartet werden soll. Ist dies der Fall (d.h. mindestens von einem beteiligten Prozess wurde noch keine Bestätigung erhalten), so wird ein neues Nachrichtenobjekt erstellt. Dieses Objektt wird mit dem Inhalt *Boolean: isMulticast=true* verschickt (intern wird hier für jeden Empfängerprozess ein *VSMessageReceiveEvent* erzeugt). Mit *scheduleAt* wird ein Wecker festgelegt, der vorgibt zu welcher lokalen Prozesszeit die Methode *onClientSchedule* aufgerufen werden soll (intern wird hier ein *VSProtocolScheduleEvent* erzeugt):

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

            VSMessage message = new VSMessage();
```

```
            message.setBoolean("isMulticast", true);
            super.sendMessage(message);
        }
    }
```

Wenn eine Serverantwort eintrifft, dann wird *onClientRecv* aufgerufen. Hier wird überprüft, ob überhaupt noch Multicast-Bestätigungen benötigt werden. Nach dieser Überprüfung wird geschaut, ob es sich bei der Antwort um eine noch nicht eingetroffene Bestätigung handelt. Gegebenenfalls wird dann 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;

            super.log("ACK von Prozess " + pid + " erhalten!");

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

                super.removeSchedules();
            }
        }
    }
```

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

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

Serverseite des Protokolls

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

```
private boolean ackSent = false;

public void onServerInit() { }

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

Erhält der Server eine Clientanfrage, so überprüft er Server, ob es sich um eine Multicast-Nachricht handelte oder nicht. Daraufhin wird dann ggf. die Bestätigungsnachricht mit *Boolean: isAck=true* und der Server-PID verschickt. Je nachdem ob bereits eine Bestätigung verschickt wurde oder nicht, wird eine andere Nachricht in Log erstellt:

```
public void onServerRecv(VSMessage recvMessage) {
    if (recvMessage.getBoolean("isMulticast")) {
        VSMessage message = new VSMessage();
        message.setBoolean("isAck", true);
        message.setInteger("pid", process.getProcessID());
        super.sendMessage(message);

        if (ackSent) {
            super.log("ACK erneut versendet");
        } else {
            super.log("ACK versendet");
            ackSent = true;
```

```
            }
        }
    }
```

Der Server benutzt in diesem Beispiel keinen Wecker. Dementsprechend hat der Methodenrumpf von *onServerSchedule* keinen Inhalt:

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

Erstellung eigener Protokolle (Schnelldurchlauf)

Hier werden alle Schritte zusammengefasst, die für die Erstellung eines eigenen Protokolls *VSMyProtocol* durchgeführt werden müssen. Ein Protokoll-Entwickler muss hierfür das Java-SDK und Apache Ant installiert- und den Quelltext des Simulators vorliegen haben.

1. VS-Simulator Quelltext beziehen und in das Verzeichnis *vs/sources/protocols/implementations* wechseln.
2. Das Template-Protokoll *VSDummyProtocol.java* nach *VSMyProtocol.java* kopieren.
3. *VSDummyProtocol.java* editieren und den Klassennamen dort anpassen (*VSDummyProtocol* $\longrightarrow$ *VSMyProtocol*).
4. In das oberste Verzeichnis *vs/* wechseln.
5. Die Datei *sources/events/VSRegisteredEvents.java* editieren, und in der *init*-Methode folgende Zeile hinzufügen:

```
    registerEvent("protocols.implementations.VSMyProtocol",
                  "Neues Protokoll"); // Name
```

6. Mit dem Befehl *ant compile* die Änderungen übernehmen und mit *ant test* testen, ob der Simulator das Protokoll übernommen hat. Hierbei wird der Simulator direkt aus dem Quellverzeichnis gestartet. “Neues Protokoll” sollte nun im Ereigniseditor sichtbar sein und programmiert werden können.
7. Mit dem Befehl *ant dist* das Archiv *dist/lib/VS-Sim-Latest.jar* erstellen und verwenden.

Wenn eine Simulatorversion versucht eine abgespeicherte Simulation eines nicht implementierten Protokolls zu laden, dann kommt es unweigerlich zu Laufzeitfehlern. Daher muss mit einem neuen Protokoll also stets auch der neue Simulator ausgeliefert werden.

4.5. GUI sowie Simulationsvisualisierung

Das Paket *simulator* (s. Abb. 4.9.) enthält die Implementierung der graphischen Benutzeroberfläche des Simulators. Einzigste Ausnahmen sind die Editorklassen in *prefs.editors* sowie die Klasse *utils.VSFrame*.

Beim Starten des Simulators wird die *main*-Methode, welche sich in *VSMain* befindet, aufgerufen. Sie instantiiert ein *VSDefaultPrefs*-Objekt, in welchem alle Standardeinstellungen des Simulators definiert sind. Anschließend wird ein *VSSimulatorFrame* erzeugt, welches ein Simulatorfenster (s. Abb. 2.1.) implementiert. Das Simulatorfenster erstellt für jede neue Simulation jeweils ein Objekt der Klasse *VSSimulator*, wobei jede Simulation im Simulationsfenster einen eigenen Tab besitzt (s. Abb. 2.3., unten links). Jede Simulation besitzt dabei eine eigene Simulationsnummer. Jedes *VSSimulator*-Objekt greift auf die Klasse *VSSimulatorVisualization* zurück, welche die Visualisierung der Simulation realisiert (s. Abb. 2.5.) .

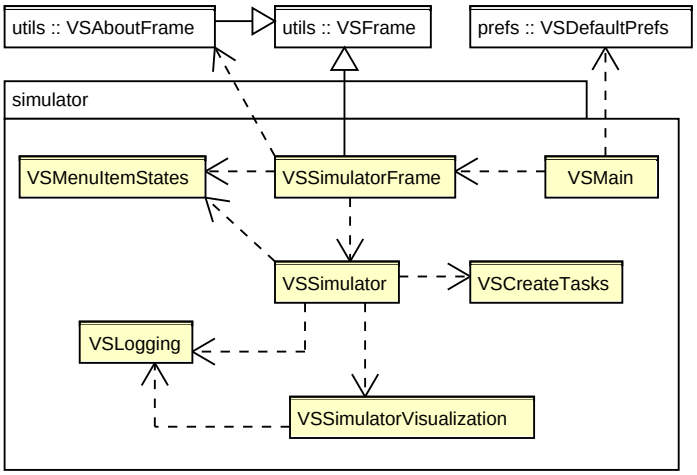

Abbildung 4.9.: Das Paket *simulator*

VSSimulatorVisualization greift auf Java's Grafikbibliothek Java2D (siehe [Javc], [Java], [Bra03]) zu und ist zur Optimierung der Performance mit dem Simulationsverlauf stark verzahnt. Klassenattribute, die von Simulationseinstellungen und den Fenstergrößen abhängigig sind, werden so nur dann neu berechnet wenn dies erforderlich ist.

Die Klasse *VSMenuItemStates* wird für die Synchronisierung der graphischen Elemente des GUI's mit dem Status der Simulation verwendet. Abhängig vom Simulationsstatus kann der Benutzer bestimmte Aktionen entweder

durchführen oder nicht. Zum Beispiel kann eine Simulation nur pausiert werden, wärend sie abgespielt wird. Alle hier möglichen Aktionen sind bereits aus Kapitel 2.1. bekannt.

Die Klasse *VSCreateTask* wird vom Ereigniseditor verwendet. Der Ereigniseditor (s. Abb. 2.8.) wird in der Klasse *VSSimulator* implementiert. Hinter jeder Ereignisauswahl verbirgt sich ein *VSCreateTask*-Objekt, welches vorgibt wie das ein Ereignis anzulegen ist.

Die Klasse *VSLogging* kapselt für das Loggen von Nachrichten die Attribute eines *JTextArea*-Objektes. In dieser Klasse werden alle Logfunktionen implementiert. Die *JTextArea* wird für die Darstellung dem Simulationsobjekt *VSSimulator* übergeben. Für den Logfilter wird auf das Java-Standardpaket *java.util.regex* (s. [Fri06]) zugegriffen. Dadurch können anhand von regulären Ausdrücken in Java-Syntax die Logs gefiltert werden (s. Kap. 2.2.2.).

Threads und Zeitsynchronisierung

Der Simulator soll auf die Millisekunde genau simulieren können und dabei soll jede simulierte Sekunde relativ zur echten Zeit fortschreiten. Die Simulationsabspielgeschwindigkeit lässt sich bei den Simulationseinstellungen unter “Abspielgeschwindigkeit der Simulation” (Float: *sim.clock.speed*) einstellen (s. Kap. 2.4.2.). Hierfür muss folgendes berücksichtigt werden:

- Das Zeichnen der Visualisierung benötigt pro Aktualisierung einige Millisekunden. Hier werden ständig mathematische Berechnungen wie z.B. die Berechnung einer Nachrichtenlinie oder die automatische Skalierung des Diagramms durchgeführt.
- Das Neuberechnen der Simulation benötigt pro Aktualisierung einige Millisekunden. Hier arbeitet insbesondere der Task-Manager, welcher überprüft, ob Ereignisse auszuführen sind.
- Jeder simulierte Prozess sollte mit der selben Geschwindigkeit fortschreiten, und dies auf jedem Betriebssystem und auf jeder Architektur. Da Threads auf Betriebssystemebene implementiert sind sind Java-Threads nicht komplett plattformunabhängig. Dadurch kann das Verhalten je nach Betriebssystem und Architekturen variieren. Insbesondere übernimmt das Betriebssystem die Entscheidung, wann welcher Thread arbeiten darf.
- Die Simulationszeit wird stets in Millisekunden angegeben und sie wird intern in einer *long*-Variable abgespeichert. Somit kann eine Simulationszeit immer nur den Wert einer ganze Zahl betragen. Berechnungsrundungsfehler durch *sim.clock.speed* (s. Kap. 2.4.2.) müssen berücksichtigt werden.
- Der Simulator soll die komplette CPU des Anwender-Computers nicht konstant auslasten.

Es wurde eine Lösung gewählt, bei der lediglich ein einziger Thread für die Visualisierung und die Berechnung der Simulation zuständig ist. Der Algorithmus verläuft in vereinfachter Form wie folgt ab:

1. Die aktuelle simulierte globale Zeit sei t und die globale Zeit wo die Simulation endet sei e.
2. Wenn $t > e$, dann $t := e$ setzen.
3. Neuberechnen und Zeichnen der Visualisierung zum Zeitpunkt t. Die dabei verstrichene Zeit sei v.
4. Wenn $t = e$, dann Simulation beenden.
5. Für einige Millisekunden den Thread pausieren. Hierbei sei p die beim Schlafen verstrichene Zeit.
6. ```
   for (i = t; i < t + v + p && i < e; i++)
       Alle Ereignisse des Zeitpunktes i hintereinander ausführen
   ```
7. Bei Punkt 2 mit neuer Startzeit $t := t + v + p$ weitermachen.

Zusätzlich muss noch die Simulationsvariable *sim.clock.speed* berücksichtigt werden. Sie wurde zur verbesserten Übersicht im obigen Algorithmus nicht extra angegeben. Intern speichert der Simulator jeweils die echte Zeit und die Simulationszeit. Die verstrichenen echten Zeiten werden dabei ständig gemessen und anschließend mit *sim.clock.speed* die neuen tatsächlichen Simulationszeiten berechnet. Die Rundungsfehler werden pro Durchgang in einer *double*-Variable (Fließkommazahl doppelter Genauigkeit) abgespeichert. Wenn der Betrag der Rundungsfehler $>= 1$ ist, dann wird davon der ganzteilige Werteanteil in der Simulationszeit berücksichtigt. Für jede lokale Prozesszeit sowie der dazugehörigen lokalen Uhrabweichung wird ähnlich verfahren.

Jede Simulation besitzt somit seinen eigenen Simulationsthread. Des Weiteren gibt es noch den Java Swing-Thread (s. [Loy02]), der für das GUI und für die Anwenderinteraktion zuständig ist. Der Anwender kann zu jedem Zeitpunkt in die Simulation eingreifen, weswegen die Behandlung der Anwendereingriffe synchronisiert wurde.

4.6. Serialisierung und Deserialisierung von Simulationen

Der Anwender kann eine erstellte Simulation im Datei-Menü speichern oder eine bereits abgespeicherte Simulation laden. Hierbei wird von der aus Java angebotenen Möglichkeit Objekte zu Serialisieren Gebrauch gemacht. Im Paket *serialize* (s. Abb. 4.10.) befinden sich Hilfsklassen, die die Serialisierung einer Simulation unterstützen.

Da nicht alle Daten für die Speicherung einer Simulation relevant sind, wird nur eine Auswahl von Klassenattributen serialisiert. Zum Beispiel werden alle Simulationseinstellungen serialisiert, nicht jedoch GUI-Objekte. Alle serialisierbaren Klassen implementieren das Interface *VSSerializable* mit folgenden zwei Methoden:

- *public void serialize(VSSerialize serialize, ObjectOutputStream oos)*: Diese Methode wird bei jedem Serialisierungsvorgang aufgerufen (Speichern einer Simulation).
   ```

- *public void deserialize(VSSerialize serialize, ObjectInputStream ois)*: Diese Methode wird bei jedem Deserialisierungsvorgang aufgerufen (Laden einer Simulation).

Die Methoden *serialize* und *deserialize* erhalten neben einem Dateistream auch ein *VSSerialize*-Objekt als Übergabeparameter. Für jeden Serialisierungsvorgang wird zuerst ein Objekt der Klasse *VSSerialize* erstellt. Eine zu serialisierende Simulation besteht aus einer Vielzahl von einander abhängigen Objekten. Jedes Objekt kann dabei Referenzen auf andere Objekte besitzen. Würde jedes Objekt komplett serialisiert werden, so würden Objekte, auf denen mehrere Referenzen existierten, in mehrfacher Ausführung behandelt werden. Bei Kreisverweisen (Objekt A referenziert Objekt B welches ebenfalls eine Referenz auf Objetkt A besitzt) würde die Serialisierung sogar in einer Endlosschleife enden. *VSSerialize* hilft hierbei dies zu vermeiden und merkt sich Informationen von allen bereits serialisierten Objekten, so dass jedes Objekt genau einmal serialisiert wird. Bei der Deserialisierung hilft die Klasse *VSSerialize* dabei, alle Objekte wieder mit den richtigen Referenzen auszustatten.

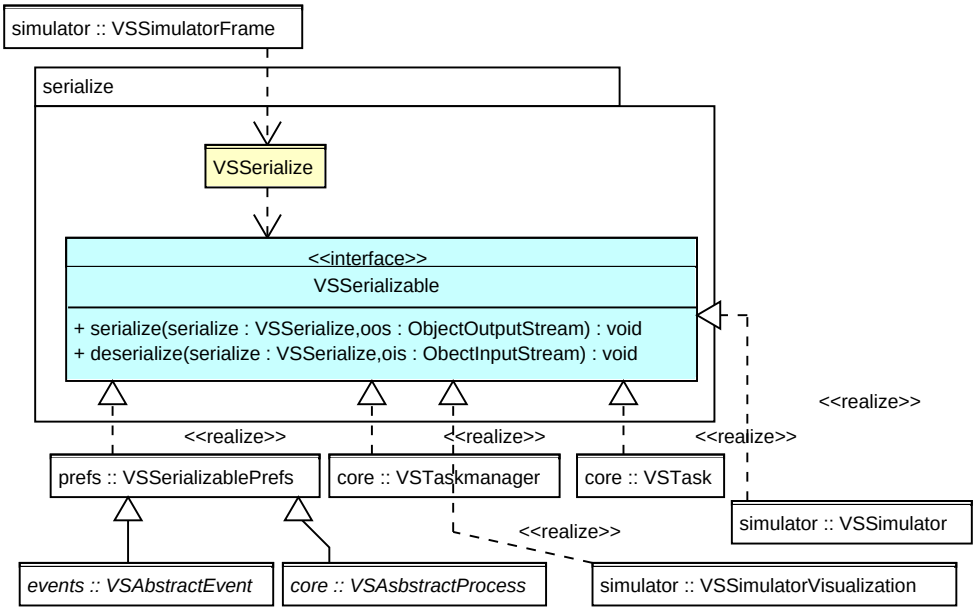

Abbildung 4.10.: Das Paket *serialize* und serialisierbare Klassen

Alle Klassen, die *VSSerializePrefs* erweitern, können komfortabel sämtliche Einstellungen serialisieren. Beispielsweise speichert der Simulator alle seine globalen Simulationseinstellungen bei einer Serialisierung automatisch ab. Bei den Prozessen und den Ereignissen (und somit auch Protokollen) gilt selbiges analog.

Abgespeicherte Simulationen sollen auch mit zukünftigen Versionen des Simulators kompatibel bleiben. Deshalb werden alle Objekte aller Klassen, die *VSSerializable* implementieren, nicht komplett serialisiert. Bei der Serialisierung werden nur relevante Klassenattribute, die der Simulationsprogrammierung, und nicht z.B. der GUI-Komponenten angehören, serialisiert. Eine Erweiterung des GUIs muss somit nicht bei den Serialisierungen berücksichtigt werden.

Beispielimplementierung einer *serialize*-Methode

Der folgende Quelltext-Ausschnitt zeigt eine Beispielimplementierung von *serialize*:

```
public synchronized void serialize(VSSerialize serialize,
                                   ObjectOutputStream oos)
throws IOException {
    oss.writeObject(new Boolean(false)); // flag
    oss.writeObject(attributeOne);
    oss.writeObject(ottributeTwo);
    serialize.setObject("sampleObject", this);
    process.serialize(serialize, oos);
    someOtherSerializableObject.serialize(serialize, oos);
    oss.writeObject(new Boolean(false)); // flag
}
```

Vor und nach der eigentlichen Objektserialisierung wird jeweils eine boolesche Flagge mit dem Standardwert *false* serialisiert. Sobald in einer späteren Simulator-Versionen weitere zu serialisierenden Klassenattribute hinzukommen, dann kann bei der Deserialisierung diese Flagge abgefragt und separat behandelt werden. Somit bleiben ältere bereits abgespeicherte Simulationen stets zu neueren Version des Simulators kompatibel. Wenn eine Flagge auf *true* gesetzt wird, dann kann unter den neuen Attributserialisierungen eine weitere Flagge gesetzt werden, wodurch beliebig viele Erweiterungen in die Serialisierung sukzessiv einbaubar sind.

Das zu serialisierende Objekt besitzt hier lediglich zwei zu serialisierende Attribute. Mit *serialize.setObject* speichert *serialize* eine Referenz auf das aktuelle Objekt ab, worauf folgende Objektserialisierungen zurückgreifen können. Danach wird ein *process* und *someOtherSerializableObject* serialisiert. Die Deserialisierung erfolgt genau in umgekehrter Reihenfolge, wobei ein Objekt von *VSSerialize* hilft die Referenzen auf andere Objekte korrekt zu setzen.

In Abbildung 4.13 ist die komplette Sequenz für die Serialisierung (das Abspeichern) einer Simulation angegeben. Zuerst wird *serialize* auf die globalen Simulationseinstellungen (*VSPrefs*) und dem Simulatorobjekt (*VSSimulator*) ausgeführt. Das Simulator-Objekt führt *serialize* wiederum auf das *VSSimulatorVisualization*-Objekt aus. Dort wird jeder Prozess inklusive alle Protokollobjekte serialisiert. Anschließend folgt der Task-Manager mit allen programmierten Ereignissen.

4.7. Helferklassen und Klassen für Ausnahmebehandlungen

Es wurden noch nicht die Klassen der Pakete *utils* (s. Abb. 4.11.) sowie *exceptions* (s. Abb. 4.12.) vorgestellt. *utils* fasst lediglich einige Helferklassen zusammen, die vom restlichen Quelltext verwendet werden.

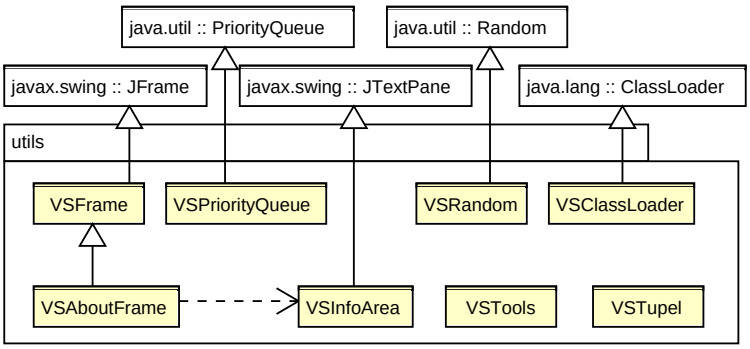

Abbildung 4.11.: Das Paket *utils*

- *VSFrame*: Alle Objekte, die ein eigenes Swing-Fenster besitzen, erben von der Klasse *VSFrame*. Sie stellt sicher, dass neue Fenster an der richtigen Position der Bildfläche platziert werden und dass Unterfenster (Fenster, die aus einem anderen Fenster heraus geöffnet wurden) automatisch mit geschlossen werden, sobald ihr jeweiliges "Erzeugerfenster" geschlossen wird.
- *VSAboutFrame*: Dieses Fenster implementiert die "About-Anzeige" die im Simulator über das Datei-Menü aufgerufen werden kann.
- *VSInfoArea*: Ist für die Textanzeige in *VSAboutFrame* zuständig.
- *VSClassLoader*: Diese Klasse wird für die automatische Instanzierung von Ereignisobjekten benötigt, wenn dem Simulator lediglich die Klassennamen (aus *events.VSRegisteredEvents*) bekannt sind.
- *VSHelper*: In dieser Klasse befinden sich statische Helfermethoden, die in keine andere Klasse gehören.
- *VSPriorityQueue*: Diese Klasse wird für das Verwalten von *core.VSTask*-Objekte im Task-Manager benötigt.
- *VSRandom*: Wird für Zufallsereignisse benötigt. Jedes Prozessobjekt besitzt einen solchen eigenen Pseudozufallsgenerator. Diese Klasse setzt gleichzeitig einen eigenen Seed basierend auf der lokalen Systemzeit und anderer Zahlen fest.
- *VSTupel*: Diese Klasse ist eine Implementierung eines einfach aufgebauten 3-Tupel Datentyps. Alle 3 Elemente können von einem anderen Typ sein, was mit Hilfe der Java-Generics verwirklicht wurde. *VSTupel* wird von den Editorklassen für die Generierung von GUI-Elementen benötigt.

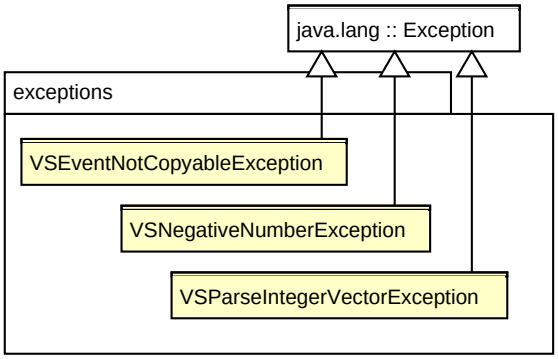

Abbildung 4.12.: Das Paket *exceptions*

Im Paket *exceptions* befinden sich Klassen, die für Ausnahmebehandlungen verwendet werden. *VSNotCopyableException* wird während eines Kopierversuch eines nicht-kopierbaren Ereignisses geworfen. *VSNegatieNumberException* wird geworfen, wenn negative Zahlen dort auftreten, wo sie es nicht sollten. Wenn ein Editorobjekt die Benutzereingabe einer Integer-Vektor-Variable nicht parsen kann, so greift es auf *VSParseIntegerVectorException* zurück.

4.8. Programmierrichtlinien

Die Programmierrichtlinien entsprechen idr. denen aus [Fas06] (vgl. auch [Fas08]).

Die *main*-Methode befindet sich in der Klasse *simulator.VSMain*.

- Es wird kein Gebrauch vom Java-Standardpaket gemacht. Alle Klassen befinden sich somit in explizit angegebenen Paketen (z.B. *events.implementations*).
- Alle Namen von Klassen und Interfaces beginnen mit großen Buchstaben, während alle Variablen-, Methoden- und Attributnamen mit kleinen Buchstaben beginnen. Namen finaler Variablen und Attribute bzw. Konstanten sind komplett in Großbuchstaben gehalten.
- Alle Quelltext-Dateien besitzen einen Header, der Informationen der verwendeten Lizenz angibt.
- Alle Quelltext-Dateien sind vollständig mit Javadoc dokumentiert.
- Der komplette Quelltext inklusive Dokumentation ist in englischer Sprache verfasst.
- Eine Quelltext-Datei hat eine maximale Zeilenlänge von 80 Zeichen, was der Standardbreite eines UNIX-Terminals entspricht. Eine Ausnahme stellt die Klasse *prefs.VSDefaultPrefs* dar, denn hier befinden sich auch längere Texte die in Strings abgespeichert werden müssen.

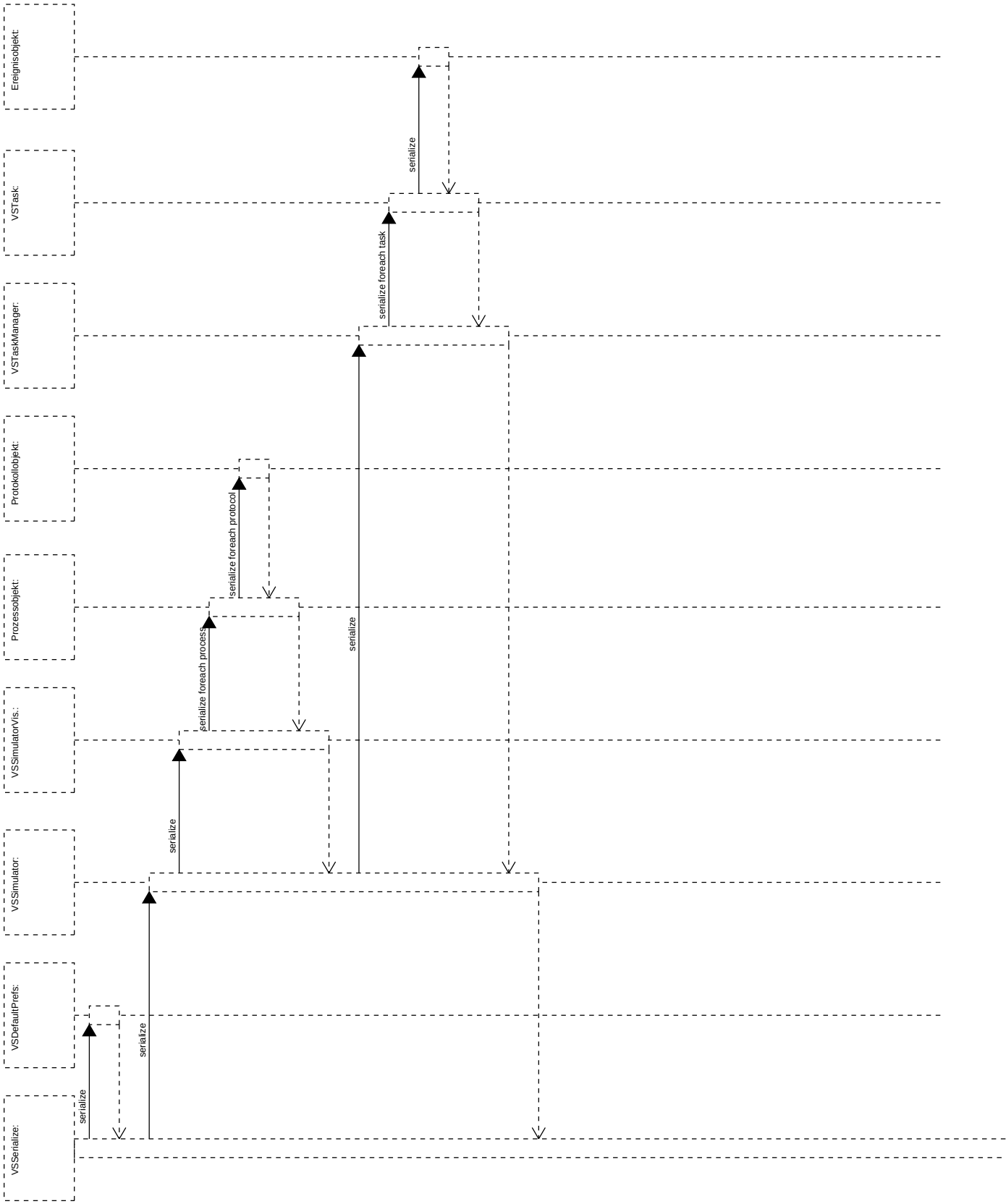

Abbildung 4.13.: Serialisierungssequenz

- 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 aller Klassen und Interfaces tragen als Präfix stets *VS*. Die Abkürzung VS steht hierbei für Verteilte Systeme.
- Namen abstrakter Klassen tragen als Präfix stets *VSAbstract* (z.B. *VSAbstractEditor*).
- Namen aller Protokollklassen tragen als Postfix *Protocol* (z.B. *VSPingPongProtocol*).
- Namen aller Ereignisklassen die keine Protokolle implementieren, tragen als Postfix *Event* (z.B. *VSProcessCrashEvent*).
- Namen aller dejenigen Klassen die ein Fenster implementieren, tragen als Postfix *Frame* (z.B. *VSSimulatorFrame*).
- Überall wo es erforderlich ist werden Java-Generic-Datentypen verwendet (z.B. *java.util.Vector<Integer>* anstelle von *java.util.Vector*).

4.9. 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 das Betriebssystem Microsoft Windows XP dar, auf welchem der Simulator zusätzlich getestet wurde. Der Simulator wurde jedoch hauptsächlich unter dem Betriebssystem FreeBSD 7.0, einem Open Source Unix-Derivat, programmiert.

Wie bereits bekannt ist, wurde die Programmiersprache Java von Sun Microsystems in der Version 6 (1.6) als Implementierungssprache für den Simulator gewählt. Für die Quelltextdokumentation wurde Javadoc, für die automatische Quelltexteinrückung astyle und als Java-Referenz kam [Javb] zum Einsatz. Als Built-Tool wurde hier auf Apache Ant (s. [Antb] und [Anta]) zurückgegriffen.

Als Versionierungssystem wurde SVN (Subversion) verwendet. Für den Zugriff auf das SVN-Repository mittels HTTPS (Hypertext Transfer Protocol Secure) wurde der Apache-Webserver mit WebDAV-Plugin verwendet. Zudem kam WebSVN als Webschnittstelle des SVN-Repository zum Einsatz. Mozilla Firefox diente dabei dem Betrachten der Javadocs und dem Bedienen der WebSVN-Oberfläche.

Für das Schreiben des Java-Quelltextes wurde Graphical Vi IMproved (GVim) sowie die Entwickungsumgebung Eclipse verwendet. Eclipse bietete bessere Code-Refactoring-Methoden, während GVim mit seiner Flexibilität und schnelleren Editiermöglichkeiten in Verbindung mit seiner Script Engine Vim-Script besonders effektiv ist. Es wurden außerdem das JAutoDoc zur Erstellung von Javadoc-Kommentaren und für die Konnektivität mit dem SVN Server das Subversion-Eclipse-Plugin verwendet. Je nach Anforderung wurde zwischen diesen beiden Umgebungen gewechselt. Für das Verfassen des LaTeX-Dokumentes wurde ebenfalls GVim verwendet.

Für die Erstellung dieses PDF-Dokumentes wurde LaTeX in Verbindung mit dem Built-Tool GNU Make und Rubber verwendet. Eine Rechtschreibüberprüfung wurde mit aspell sowie OpenOffice.org durchgeführt. xPDF diente dabei als PDF-Anzeigeprogramm.

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

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`
- OpenOffice.org - `http://www.OpenOffice.org`
- 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 5.

Ausblick

Es wurde erfolgreich ein Simulator für die Simulation verteilter Systeme entwickelt. Der Simulator hat bereits 10 implementierte Protokolle zur Auswahl eingebaut. Zudem steht dem Anwender ein sehr komfortables Protokoll-API zur Verfügung, womit der Entwicklung neuer Protokolle quasi keine Grenzen gesetzt sind.

Darüber hinaus verfügt der Simulator über eine Vielzahl von sehr flexiblen Einstellungsmöglichkeiten. Für jede Simulation lassen sich somit komplett andere Konfigurationen verwenden. Jeder beteiligte Prozess hat wiederum eigene lokale Einstellungen, wo sich auch jedes Protokoll für jeden Prozess separat einstellen lässt. Die Anzahl und Flexibilität der möglichen Szenarien wird dadurch um einen sehr großen Faktor erweitert.

Mit dem Ereigniseditor gibt es eine komfortable Möglichkeit eigene Szenarien zu programmieren um sie anschließend zu Simulieren. Hierbei kann entweder auf die bereits enthaltenen Protokolle oder auf selbst implementierte Protokolle zugegriffen werden. Alle dazugehörigen Einstellungen und programmierten Ereignisse lassen sich vom Anwender für eine spätere Wiederverwendung plattformunabhängig abspeichern. Somit können auch abgespeicherte Szenarien beispielsweise an Kommilitonen weitergegeben werden oder für eine spätere Präsentierung zwischengespeichert werden. Durch den Logfilter lassen sich mit Hilfe von regulären Ausdrücken nur die relevanten Lognachrichten anzeigen, was die Analyse einer Simulation erheblich vereinfacht. Weitere Funktionen wie Lamport- und Vektor-Zeitstempel sowie Anti-Aliasing runden den Simulator ab.

Durch den objektorientierten Aufbau ist der Simulator relativ einfach erweiterbar, was nicht nur das Protokoll-API betrifft. Insgesamt wurde an den meisten Stellen darauf geachtet, dass zu einem späteren Zeitpunkt Erweiterungen einfließen könnten. Insbesondere soll die Serialisierung von Objekten rückwärtskompatibel bleiben, da sonst bei jeder neuen Simulatorversion alle Simulationen erneut angelegt und abgespeichert werden müssten.

Hätte für diese Diplomarbeit noch mehr Zeit zur Verfügung gestanden, dann hätten einige der folgenden Funktionen (hier in alphanumerisch sortierter Reihenfolge aufgelistet) auch Einzug halten können:

- Die Möglichkeit Protokolle zu entwickeln ohne den kompletten Quelltext des Simulators vorliegen zu haben. Protokollklassen als separate Bibliothek einzubinden, die dynamisch geladen werden können.
- Die Simulationsdauer beliebig lang machen zu können. Dazu müsste die Klasse *VSSimulatorVisualisation* entlang der Zeitachse scrollbar gemacht werden, so dass der Benutzer für eine nachträgliche Betrachtung des Simulationsverlaufes zu jeder beliebigen Position zurückspringen kann.
- Eine Zoomfunktion für die Simulationsvisualisierung einzubauen.
- Im Ereigniseditor selbst auch periodische Ereignisse programmierbar zu machen. Bisher kann nur jeder Ereigniseintritt separat programmiert werden oder auf Protokoll-Interne Wecker zurückgegriffen werden.
- Lamport- und Vektor-Zeitstempel als Ereigniseintrittskriterien verwenden zu können.
- Tiefere Schichten des OSI-Referenzmodells simulieren können, wie z.B. TCP, UDP, IP, ...
- Weitere Funktionen einzubauen, wie z.B. das Anklicken einer Nachrichtenlinie, was zu der jeweiligen Nachricht alle verfügbaren Informationen anzeigt und welche gegebenenfalls vom Benutzer editiert werden können.

Da der Simulator höchstwahrscheinlich unter einer Open Source Lizenz freigegeben wird, werden die einen oder anderen Funktionen nachträglich eingebaut werden. Kommilitonen werden auch herzlich dazu eingeladen sein sich an diesem Software-Projekt zu beteiligen. Als Vorbild sei hier der CPU-Simulator M32 [Oßm], der von Prof. Oßmann an der Fachhochschule Aachen entwickelt wurde, genannt. Hier existieren bereits einige Erweiterungen und Verbesserungen der Ursprungsversion, die von den Studenten angefertigt wurden. Für die Entwicklung des VS-Simulators wurde keine proprietäre Software verwendet, so dass jeder kostenlosen Zugriff auf die dazugehörigen Tools hat.

Anhang A.

Akronyme

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

IP Internet Protocol

JRE Java Runtime Environment

NID Nachrichten-Identifikationsnummer

PDF Protable Document Format

PID Prozess-Identifikationsnummer

RTT Round Trip Time

SDK Software Development Kit

SVN Subversion

TCP Transmission Control Protocol

UDP User Datagram Protocol

VS Verteilte Systeme

Anhang B.

Literaturverzeichnis

[Anta] *Apache Ant Introduction*. http://www.developer.com/tech/article.php/989631.

[Antb] *Apache Ant Manual*. http://ant.apache.org/manual/index.html.

[Bra03] David Brackeen. *Developing Games in Java*. 2003. ISBN-13: 978-1592730056.

[Cor01] Thomas H. Cormen. *Introduction to Algorithms*. 2001. ISBN-13: 978-0262032933.

[Fas06] Prof. Heinrich Fassbender. *Vorlesung Objektorientierte Softwareentwicklung an der Fachhochschule Aachen*. 2006. http://www.fassbender.fh-aachen.de.

[Fas08] Prof. Heinrich Fassbender. *Programmierrichtlinien an der Fachhochschule Aachen*. 2008. http://www.fassbender.fh-aachen.de/Downloads/OOS/Programmierrichtlinien.pdf.

[Fri06] Jeffrey Friedl. *Mastering Regular Expressions*. 2006. ISBN-13: 978-0596528126.

[Java] *Java 2D API*. http://java.sun.com/j2se/1.4.2/docs/guide/2d/spec.html.

[Javb] *Java Platform Standard Edition 6 Javadoc*. http://java.sun.com/javase/6/docs/api/.

[Javc] *Sun's Java 2D*. http://java.sun.com/products/java-media/2D/.

[Loy02] Marc Loy. *Java Swing*. 2002. ISBN-13: 978-0596004088.

[Oßm] Prof. Martin Oßmann. *M32 CPU Simulator*. http://www.ossmann.fh-aachen.de.

[Oßm07] Prof. Martin Oßmann. *Vorlesung Verteilte Systeme an der Fachhochschule Aachen*. 2007. Mitschrift: ftp://ftp.buetow.org/pub/studium/FHAC_VS-SS07/Mitschrift/verteilte-systeme.pdf.

[Sed99] Robert Sedgewick. *Algorithms in C*. 1999. SBN-13: 978-0201314526.

[Tan03] Andrew Tanenbaum. *Verteilte Systeme - Grundlagen und Paradigmen*. 2003. ISBN: 3-8273-7057-4.