



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, 11. 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, 11. August 2008

Geheimhaltung

Diese Diplomarbeit darf weder vollständig noch auszugsweise ohne schriftliche Zustimmung des Autors, des betreuenden Referenzen 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
- Florian Bütow
- Jörn Bütow
- Leslie Bütow

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

Inhaltsverzeichnis

1. Einleitung	12
1.1. Motivation	12
1.2. Grundlagen	13
2. Der Simulator	16
2.1. Grafische Benutzeroberfläche (GUI)	16
2.2. Expertenmodus	23
2.3. Ereignisse	26
2.4. Einstellungen	28
2.4.1. Variablendatentypen	29
2.4.2. Simulationseinstellungen	29
2.4.3. Prozess- und Protokolleinstellungen	32
2.4.4. Einstellungen im Expertenmodus	34
2.5. Protokolle	35
2.5.1. Beispiel (Dummy) Protokoll	35
2.5.2. Das Ping-Pong Protokoll (<i>ping-pong.dat</i> , <i>ping-pong-sturm.dat</i>)	36
2.5.3. Das Broadcast Protokoll (<i>broadcast.dat</i>)	38
2.5.4. Das Protokoll zur internen Synchronisierung in einem synchronen System (<i>int-sync.dat</i>)	40
2.5.5. Christians Methode zur externen Synchronisierung (<i>ext-vs-int-sync.dat</i>)	42
2.5.6. Der Berkeley Algorithmus zur internen Synchronisierung (<i>berkeley.dat</i>)	44
2.5.7. Das Ein-Phasen Commit Protokoll (<i>one-phase-commit.dat</i>)	46
2.5.8. Das Zwei-Phasen Commit Protokoll (<i>two-phase-commit.dat</i>)	48
2.5.9. Der ungenügende (Basic) Multicast (<i>basic-multicast.dat</i>)	52
2.5.10. Der zuverlässige (Reliable) Multicast (<i>reliable-multicast.dat</i>)	53
2.6. Weitere Beispiele	57
2.6.1. Simulation von Vektor- und Lamportzeitstempel	57
2.6.2. Simulation langsamer Verbindungen (<i>slow-connection.dat</i>)	60

3. Die Implementierung	62
3.1. Einstellungen und Editoren	62
3.1.1. Einstellungsobjekte	62
3.1.2. Editorobjekte	65
3.2. Ereignisse	67
3.2.1. Funktionsweise von Ereignissen	67
3.2.2. Beispielimplementierung eines Ereignisses	70
3.3. Zeitformate, Prozesse, Nachrichten sowie Task-Manager	71
3.3.1. Funktionsweise	71
3.3.2. Beispiel für die Erstellung von Prozessereignissen	74
3.4. Protokolle	75
3.4.1. Funktionsweise des Protokoll-APIs	75
3.4.2. Beispielimplementierung eines Protokolls	79
3.4.3. Erstellung eigener Protokolle (Schnelldurchlauf)	84
3.5. GUI sowie Simulationsvisualisierung	84
3.5.1. Funktionsweise	84
3.5.2. Threads und Zeitsynchronisierung	86
3.6. Serialisierung und Deserialisierung von Simulationen	87
3.6.1. Funktionsweise	87
3.6.2. Beispielimplementierung einer <i>serialize</i> -Methode	89
3.7. Helferklassen und Klassen für Ausnahmebehandlungen	90
3.8. Programmierrichtlinien	91
3.9. Entwicklungsumgebung	93
4. Ausblick	96
A. Akronyme	98
B. Literaturverzeichnis	99

Abbildungsverzeichnis

1.1. Client/Server Modell	13
1.2. Client/Server Protokolle	14
2.1. Der Simulator nach dem ersten Starten	16
2.2. Datei-Menü	17
2.3. Eine neue Simulation	18
2.4. Die Menüzeile inklusive Toolbar	18
2.5. Visualisierung einer noch nicht gestarteten Simulation	19
2.6. Rechtsklick auf einen Prozessbalken	19
2.7. Die Sidebar mit leerem Ereigniseditor	21
2.8. Der Ereigniseditor mit 3 programmierten Ereignissen	21
2.9. Die Ereignisauswahl via Sidebar	22
2.10. Das Loggfenster	23
2.11. Der Simulator im Expertenmodus	24
2.12. Die Sidebar im Expertenmodus	25
2.13. Der Prozesseditor in der Sidebar	26
2.14. Das Fenster zu den Simulationseinstellungen	30
2.15. Weitere Simulationseinstellungen im Expertenmodus	31
2.16. Das Ping-Pong Protokoll	36
2.17. Das Ping-Pong Protokoll (Sturm)	36
2.18. Das Broadcast Protokoll	38
2.19. Das Protokoll zur internen Synchronisierung	40
2.20. Interne Synchronisierung und Christians Methode im Vergleich	42
2.21. Der Berkeley Algorithmus zur internen Synchronisierung	44
2.22. Das Ein-Phasen Commit Protokoll	46
2.23. Das Zwei-Phasen Commit Protokoll	48
2.24. Das Basic-Multicast Protokoll	52
2.25. Das Reliable-Multicast Protokoll	53

2.26. Lamportzeitstempel	57
2.27. Vektorzeitstempel	58
2.28. Simulation einer langsamen Verbindung	60
3.1. Das Paket <i>prefs</i>	63
3.2. Das Paket <i>prefs.editors</i>	66
3.3. Die Pakete <i>events</i> und <i>events.*</i>	67
3.4. Das Paket <i>core.time</i>	71
3.5. Das Paket <i>core</i>	72
3.6. Gekapseltes <i>VSMMessage</i> im <i>VSMMessageReceiveEvent</i> -Objekt	73
3.7. Die Pakete <i>protocols</i> und <i>protocols.*</i>	75
3.8. Protokollvariablen im Prozesseditor	76
3.9. Das Paket <i>simulator</i>	85
3.10. Das Paket <i>serialize</i> und serialisierbare Klassen	88
3.11. Das Paket <i>utils</i>	90
3.12. Das Paket <i>exceptions</i>	91
3.13. Serialisierungssequenz	92

Tabellenverzeichnis

2.1. Farbliche Differenzierung von Prozessen und Nachrichten	20
2.2. Verfügbare Datentypen für editierbare Variablen	29
2.3. Farbeinstellungen	34
2.4. Programmierte Ping-Pong Ereignisse	37
2.5. Programmierte Ping-Pong Ereignisse (Sturm)	37
2.6. Programmierte Broadcast Ereignisse	38
2.7. Programmierte Ereignisse zur internen Synchronisierung	41
2.8. Programmierte Ereignisse, Vergleich interne und externe Synchronisierung	43
2.9. Programmierte Ereignisse zum Berkeley Algorithmus	45
2.10. Programmierte Ein-Phasen Commit Ereignisse	47
2.11. Programmierte Zwei-Phasen Commit Ereignisse	49
2.12. Auszug aus der Loggausgabe des Zwei-Phasen Commit Beispiels	50
2.13. Auszug aus der Loggausgabe des Zwei-Phasen Commit Beispiels (2)	51
2.14. Programmierte Basic-Multicast Ereignisse	53
2.15. Programmierte Reliable-Multicast Ereignisse	54
2.16. Auszug aus der Loggausgabe des Reliable-Multicast Beispiels	55
2.17. Auszug aus der Loggausgabe des Reliable-Multicast Beispiels (2)	56
3.1. Die Paketstruktur	63
3.2. Konventionen für Variablennamen-Präfixe in <i>VSDefaultPrefs</i>	64

Kapitel 1.

Einleitung

1.1. Motivation

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

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

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

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

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

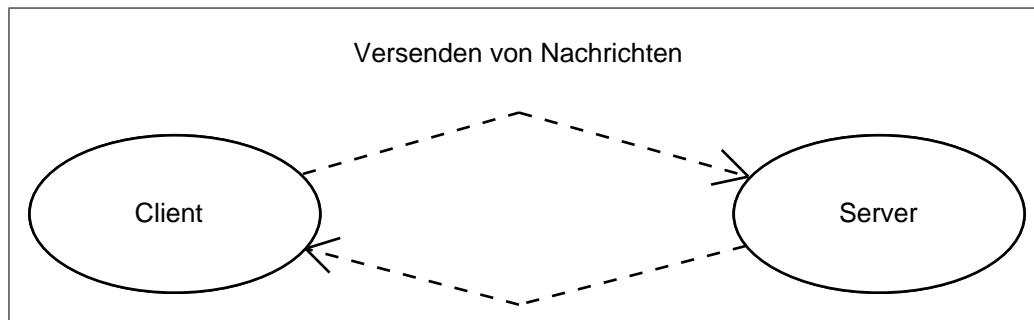


Abbildung 1.1.: Client/Server Modell

Der Simulator basiert auf dem Client/Server-Prinzip. Jeder Simulation besteht in der Regel aus einem teilnehmenden Client und einem Server, die miteinander über Nachrichten kommunizieren (Abbildung 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. Ein Prozess kann auch Client und Server gleichzeitig sein. Es besteht auch die Möglichkeit, dass ein Prozess die Rollen mehrerer Server und Clients gleichzeitig einnimmt. Ob das sinnvoll ist hängt vom simulierten Szenario ab. Um einen Prozess zu kennzeichnen besitzt jeder Prozess eine **eindeutige** Prozess-Identifikationsnummer (PID).

Nachrichten

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

Lokale und globale Uhren

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

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

Neben den normalen Uhren sind auch die Vektor-Zeitstempel sowie die logischen Uhren von Lamport von Interesse. Jeder Prozess besitzt zusätzlich einen Vektor-Zeitstempel für seine Vektorzeit, sowie einen Lamportzeitstempel für seine Lamportzeit. Für die Vektor- und Lamportzeiten gibt es hier, im Gegensatz zu der normalen Zeit, keine globalen äquivalente. Konkrete Beispiele zu den Lamport- und Vektorzeiten werden später anhand einer Simulation behandelt.

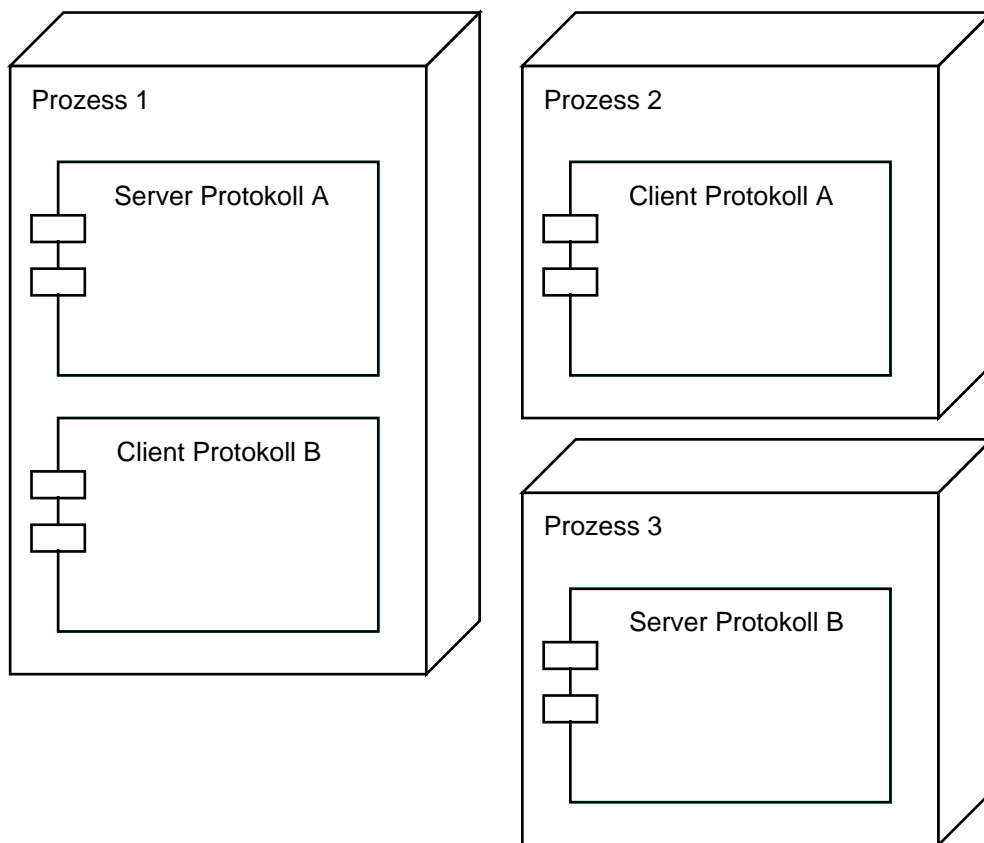


Abbildung 1.2.: Client/Server Protokolle

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

Kapitel 2.

Der Simulator

2.1. Grafische Benutzeroberfläche (GUI)

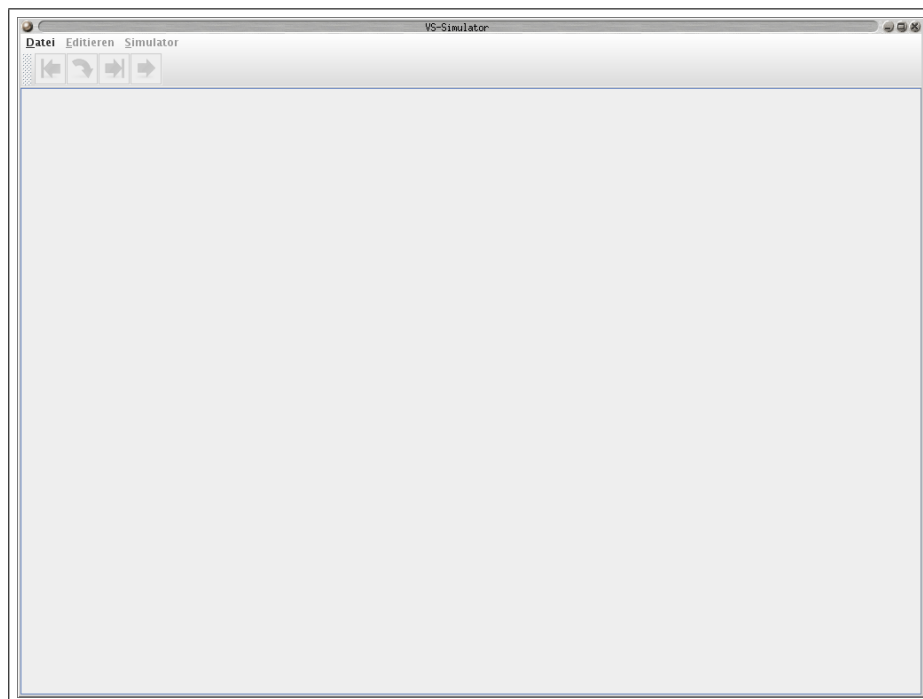


Abbildung 2.1.: Der Simulator nach dem ersten Starten

Der Simulator lässt sich mit dem Befehl `java -jar VS-Sim.jar` starten und präsentiert sich danach wie auf [Abbildung 2.1](#). Für die Erstellung einer neuen Simulation wird im Menü "Datei" ([Abbildung 2.2](#)) der Punkt "Neue Simulation" ausgewählt, wo anschließend das Einstellungsfenster für die neue Simulation erscheint. Auf die einzelnen Optionen wird später genauer eingegangen und es werden nun nur die Standardeinstellungen übernommen. Die GUI mit einer frischen Simulation sieht aus wie auf [Abbildung 2.3](#).

Die Menüzeile

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

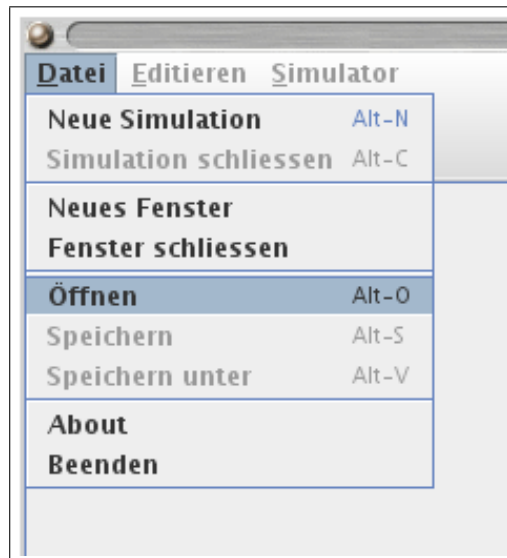


Abbildung 2.2.: Datei-Menü

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.

Die Toolbar

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

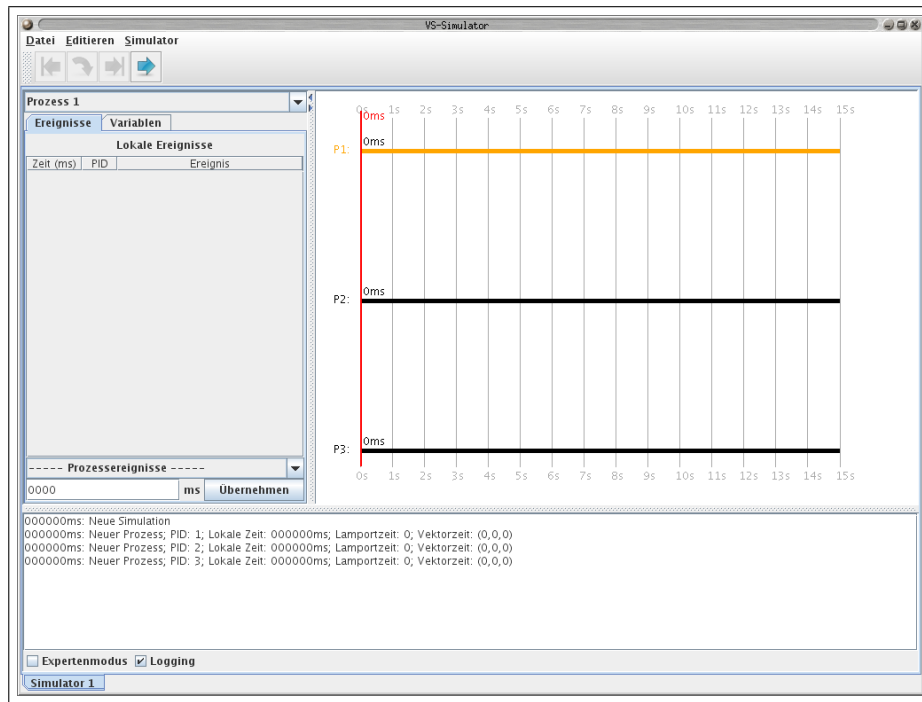


Abbildung 2.3.: Eine neue Simulation

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. Auf [Abbildung 2.5](#) sind 3 Prozesse (mit den PIDs 1, 2 und 3) dargestellt, die jeweils einen eigenen horizontalen schwarzen Balken besitzen. Auf diesen Prozessbalken kann der Anwender die jeweilige lokale Prozesszeit ablesen. Die vertikale rote Linie stellt die globale Simulationszeit dar.

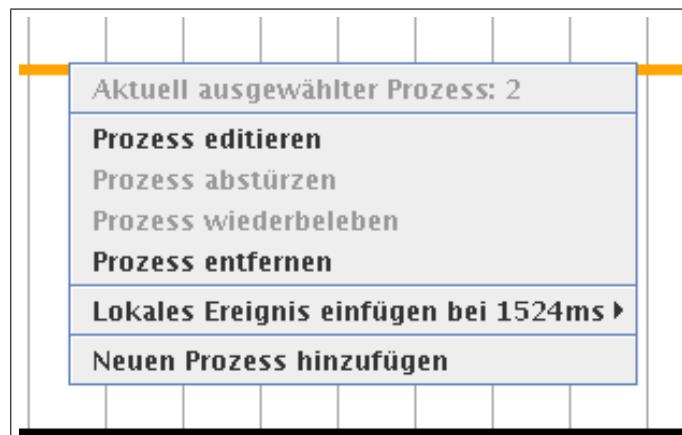


Abbildung 2.6.: Rechtsklick auf einen Prozessbalken

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 verschickt, werden nicht visualisiert. Sie werden aber im Loggfenster (mehr dazu später) protokolliert.

Eine andere Möglichkeit einen Prozesseditor aufzurufen ist ein Linksklick auf den zum Prozess gehörigen Prozessbalken. Dies muss also nicht immer über das Simulator-Menü geschehen. Ein Rechtsklick hingegen öffnet ein Popup-Fenster mit weiteren Auswahlmöglichkeiten (Abbildung 2.6). Ein Prozess kann über das Popup-Menü nur während einer laufenden Simulation zu einem Absturz oder einer Wiederbelebung bewegt werden.

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

Farbliche Differenzierung

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

Die Sidebar

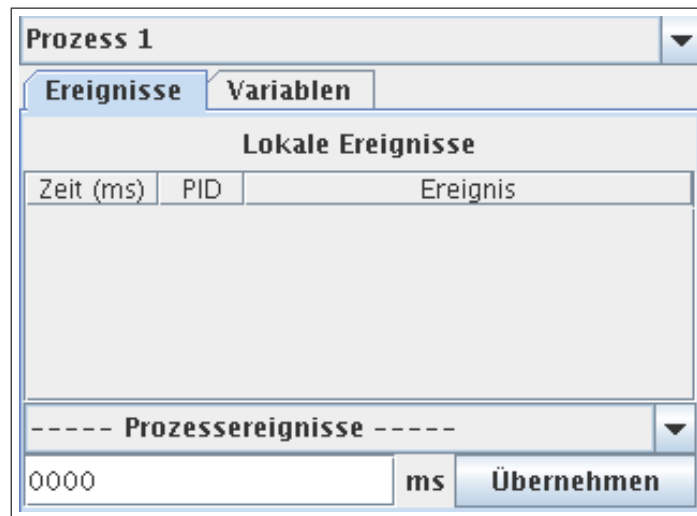


Abbildung 2.7.: Die Sidebar mit leerem Ereigniseditor

Mithilfe der Sidebar lassen sich Prozessereignisse programmieren. Oben auf Abbildung 2.7 ist der zu verwaltende Prozess selektiert (hier mit der PID 1). In dieser Prozessauswahl gibt es auch die Möglichkeit "Alle Prozesse" auszuwählen, womit die Ereignisse aller Prozesse gleichzeitig verwaltet werden können. Unter "Lokale Ereignisse" versteht man diejenigen Ereignisse, die auftreten, wenn eine bestimmte lokale Zeit des dazugehörigen Prozesses eingetreten ist. Die 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 (Abbildung 2.6) klicken und dort “Lokales Ereignis einfügen” wählen, oder unterhalb der Ereignistabelle ein Ereignis auswählen (Abbildung 2.9), im darunter liegenden Textfeld die Ereigniseintrittszeit eintragen und auf “übernehmen” gehen. Beispielsweise wurden auf Abbildung 2.8 drei Ereignisse hinzugefügt: Absturz nach 123ms, Wiederbelebung nach 321ms und erneuter Absturz nach 3000ms des Prozesses mit der ID 1.

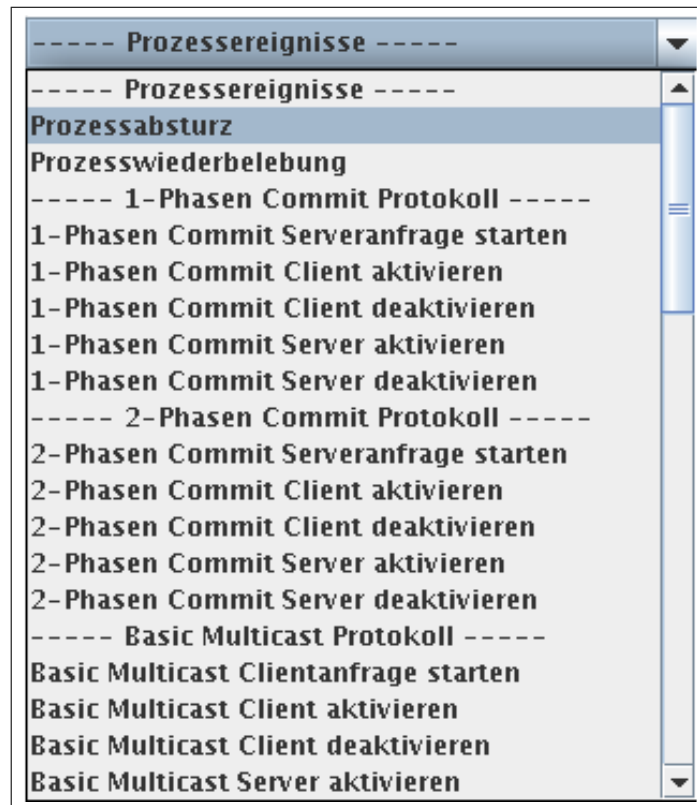


Abbildung 2.9.: Die Ereignisauswahl via Sidebar

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

In der Sidebar gibt es neben dem Ereignis-Tab einen weiteren Tab “Variablen”. Hinter diesem Tab verbirgt sich der Prozesseditor des aktuell ausgewählten Prozesses (Abbildung 2.13 links). Dort können alle Variablen des Prozesses editiert werden und ist somit eine weitere Möglichkeit einen Prozesseditor aufzurufen.

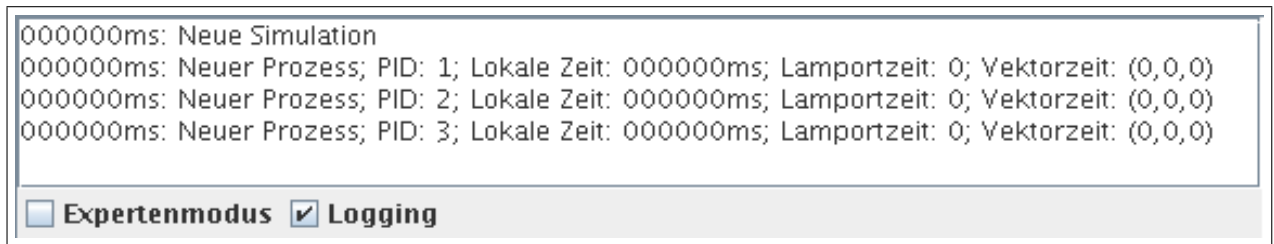


Abbildung 2.10.: Das Loggfenster

Das Loggfenster

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

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

über den Schalter "Expertenmodus" wird der Expertenmodus aktiviert beziehungsweise 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, sodass sich der Anwender nicht mit der vollen Funktionalität des Simulators auf einmal auseinandersetzen muss. Der einfache Modus ist übersichtlicher, bietet jedoch weniger Funktionen an. Der Expertenmodus eignet sich mehr für erfahrene Anwender und bietet dementsprechend auch mehr Flexibilität. Der Expertenmodus kann über den gleichnamigen Schalter unterhalb des Loggfensters oder über die Simulationseinstellungen aktiviert oder deaktiviert werden. Auf Abbildung 2.11 ist der Simulator im Expertenmodus zu sehen. Wenn der Expertenmodus mit dem normalen Modus verglichen wird, dann fallen einige Unterschiede auf:

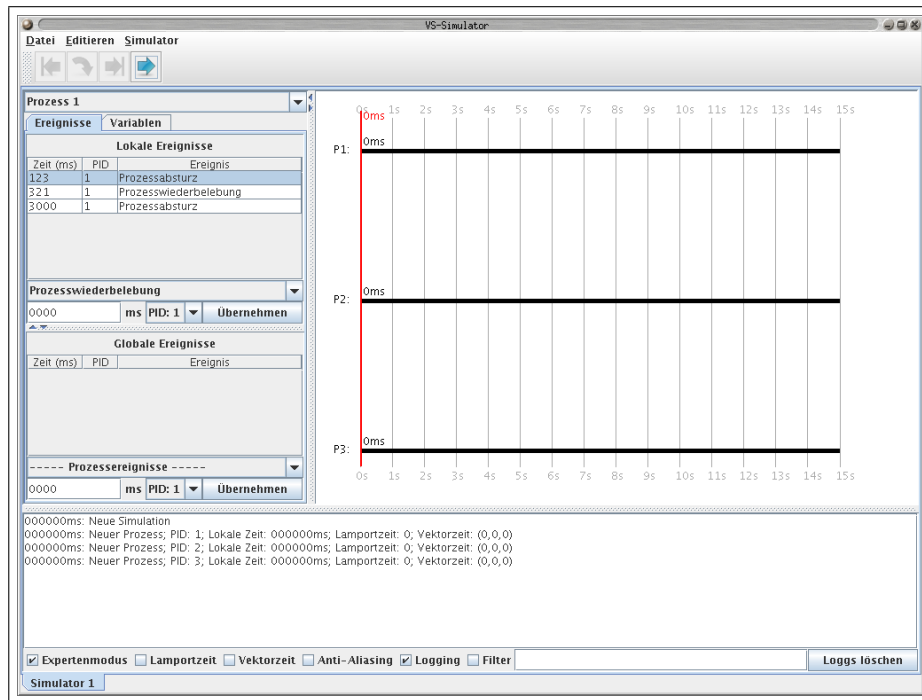


Abbildung 2.11.: Der Simulator im Expertenmodus

Neue Funktionen in der Sidebar

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

Des Weiteren kann der Anwender bei der Programmierung eines neuen Ereignisses direkt die dazugehörige PID selektieren. Im einfachen Modus wurde hier immer standardmäßig die PID des aktuell (in der obersten Combo-Box) ausgewählten Prozesses verwendet (hier mit PID 1).

Lamportzeit-, Vektorzeit- und Anti-Aliasing Schalter

Weitere Unterschiede machen sich unterhalb des Loggfensters bemerkbar. Dort gibt es unter Anderem zwei neue Schalter "Lamportzeit" und "Vektorzeit". Aktiviert der Anwender einen dieser beiden Schalter, so wird die

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

The screenshot shows a sidebar window titled 'Prozess 1'. It has two tabs: 'Ereignisse' (selected) and 'Variablen'. Under 'Ereignisse', there are two sections: 'Lokale Ereignisse' and 'Globale Ereignisse'. The 'Lokale Ereignisse' section contains a table with three rows of events. Below this is a section for 'Prozesswiederbelebung' with input fields for time (0000 ms), PID (1), and a button 'Übernehmen'. The 'Globale Ereignisse' section is currently empty. At the bottom, there is a section for 'Prozessereignisse' with similar input fields and a button 'Übernehmen'.

Zeit (ms)	PID	Ereignis
123	1	Prozessabsturz
321	1	Prozesswiederbelebung
3000	1	Prozessabsturz

Abbildung 2.12.: Die Sidebar im Expertenmodus

Der Anti-Aliasing-Schalter ermöglicht dem Anwender Anti-Aliasing zu aktivieren beziehungsweise zu deaktivieren. Mit Anti-Aliasing werden alle Grafiken der Visualisierung gerundet dargestellt. Aus Performance-gründen ist Anti-Aliasing standardmäßig nicht aktiv.

Der Loggfilter

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

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

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

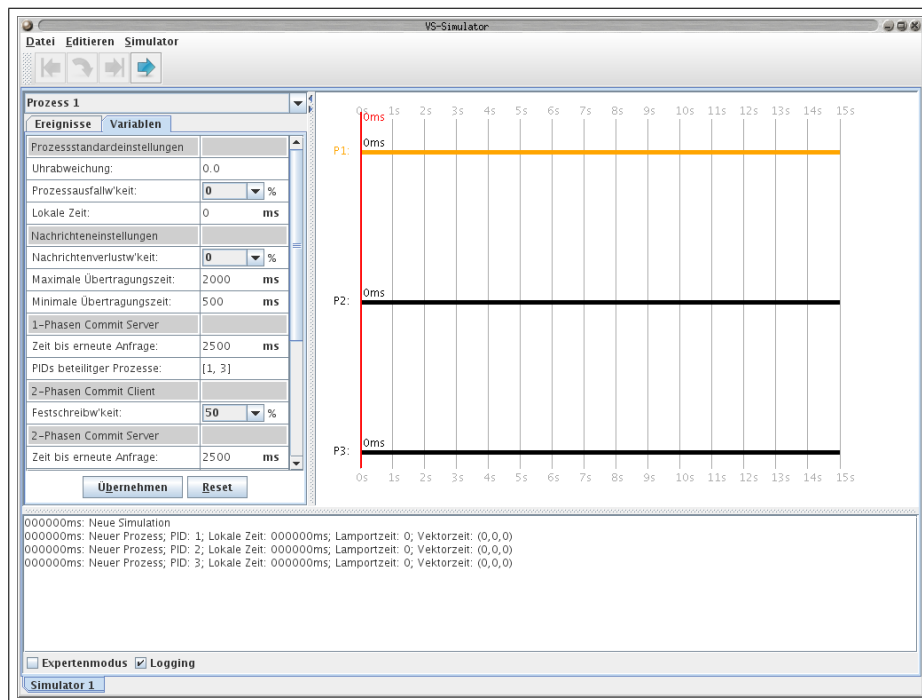


Abbildung 2.13.: Der Prozesseditor in der Sidebar

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 zum Beispiel das Eintreffen einer Nachricht oder das Ausführen einer Aktion aufgrund eines Weckers, worauf später nochmal genauer eingegangen wird.

Prozessabsturz- und Wiederbelebung (programmierbar)

Die beiden einfachsten Ereignisse sind "Prozessabsturz" sowie "Prozesswiederbelebung". Wenn ein Prozess abgestürzt ist, so wird sein Prozessbalken in rot dargestellt. Ein abgestürzter Prozess kann keine weiteren Ereignisse mehr verarbeiten und wenn bei ihm eine Nachricht eintrifft, dann kann sie nicht verarbeitet werden und

geht deshalb verloren. Die einzige Ausnahme bildet ein Wiederbelebungseignis. Ein abgestürzter Prozess kann nichts, außer wiederbelebt werden. Während eines Prozessabsturzes läuft die lokale Prozessuhr, abgesehen der Lamport- und Vektor-Uhren, normal weiter. Das heißt es besteht die Möglichkeit, dass ein Prozess einige seiner Ereignisse gar nicht ausführt, da er zu den Ereigniseintrittszeiten abgestürzt ist. Wenn im echten Leben ein Computer abstürzt oder abgeschaltet wird, dann läuft seine Hardware-Uhr unabhängig vom Betriebssystem auch weiter.

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

Es ist bereits bekannt, dass ein Prozess mehrere Protokolle client- und auch serverseitig unterstützen kann. Welches Protokoll von einem Prozess unterstützt wird, kann der Anwender anhand von Protokollaktivierungs- und Protokolldeaktivierungseignissen konfigurieren. Somit besteht die Möglichkeit, dass ein gegebener Prozess ein bestimmtes Protokoll erst zu einem bestimmten Zeitpunkt unterstützt und gegebenenfalls ein anderes Protokoll ablöst. Jedes Protokoll kann entweder server- oder clientseitig aktiviert beziehungsweise deaktiviert werden. Welche Protokolle es gibt wird später behandelt. Der Anwender hat 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 wieder-eintreffende Ereignisse realisieren. Beispielsweise verwenden die "Commit-Protokolle" (mehr dazu später) Callback-Ereignisse, indem solange Anfragen verschickt werden, bis alle benötigten Antworten vorliegen.

Zufallseignisse (nicht-programmierbar)

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

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
<i>Boolean</i>	Boolescher Wert, z.B. <i>true</i> oder <i>false</i>
<i>Color</i>	Java-Farbobjekt
<i>Float</i>	32-Bit Fließkommazahl
<i>Integer[]</i>	Vektor aus 32-Bit Integer
<i>Integer</i>	32-Bit Integer
<i>Long</i>	64-Bit Long
<i>String</i>	Java-Stringobjekt

Tabelle 2.2.: Verfügbare Datentypen für editierbare Variablen

2.4.1. Variablendatentypen

Der Simulator unterscheidet zwischen mehreren Datentypen, in denen die einstellbaren Variablen vorliegen können (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 (mehr zu Editoren später) 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 standardmäßig 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 (Abbildung 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.

- **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.



Abbildung 2.14.: Das Fenster zu den Simulationseinstellungen

- **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, so wird hier der Mittelwert aus den Verlustwahrscheinlichkeiten vom Sender- und Empfängerprozess gebildet. Ansonsten wird stets die Verlustwahrscheinlichkeit, die beim Senderprozesses angegeben 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, so wird der Mittelwert vom Sender- und Empfängerprozess gebildet. Ansonsten wird stets 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, weil er das dazugehörige Protokoll nicht unterstützt, nicht angezeigt. Dies verbessert die Übersicht.
- **Expertenmodus aktivieren** (*Boolean: sim.mode.expert = false*): Hier lässt sich der Expertenmodus akti-



Abbildung 2.15.: Weitere Simulationseinstellungen im Expertenmodus

vieren beziehungsweise deaktivieren. Alternativ kann dies über den gleichnamigen Schalter unterhalb des Loggfensters geschehen.

- **Simulation periodisch wiederholen** (Boolean: *sim.periodic = false*): Wenn diese Variable auf *true* gesetzt ist, dann wird die Simulation jedes Mal nach Ablauf automatisch erneut gestartet.
- **Lamportzeiten betreffen alle Ereignisse** (Boolean: *sim.update.lamporttime.all = false*): Wenn diese Variable auf *true* gesetzt ist, dann werden bei jedem Ereignis alle Lamportzeitstempel aller Prozesse jeweils inkrementiert. Bei einem Wert *false* inkrementieren sich die Lamportzeitstempel jeweils nur, wenn eine Nachricht empfangen oder verschickt wurde.
- **Vektorzeiten betreffen alle Ereignisse** (Boolean: *sim.update.vectortime.all = false*): Wenn diese Variable auf *true* gesetzt ist, dann werden bei jedem Ereignis alle Vektor-Zeitstempel aller Prozesse jeweils inkrementiert. Bei einem Wert *false* inkrementieren sich die Vektor-Zeitstempel jeweils nur, wenn eine Nachricht empfangen oder verschickt wurde.

Lamport- und Vektorzeitstempel werden später anhand eines Beispiels verdeutlicht.

- **Abspielgeschwindigkeit der Simulation** (*Float: sim.clock.speed = 0.5*): Gibt den Faktor der Simulationsabspielgeschwindigkeit an. Wenn als Faktor 1 gewählt wird, dann dauert eine simulierte Sekunde so lange wie eine echte Sekunde. Der Faktor 0.5 gibt somit an, dass die Simulation mit halber Echtzeitgeschwindigkeit abgespielt wird.
- **Anzahl der Prozesse** (*Integer: sim.process.num = 3*): Gibt die Anzahl beteiligter Prozesse an. Der Anwender kann auch nachträglich via Rechtsklick auf den Prozessbalken den jeweiligen Prozess aus der Simulation entfernen oder weitere Prozesse hinzufügen.
- **Dauer der Simulation** (*Integer: sim.seconds = 15*): Gibt die Dauer der Simulation in Sekunden an.

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

2.4.3. Prozess- und Protokolleinstellungen

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

- **Uhrabweichung** (*Float: process.clock.variance = 0.0*): Gibt den Wert an, um den die lokale Prozessuhr abweicht. Der Wert 0.0 besagt beispielsweise, dass die Uhr keine Abweichung hat und somit global-korrekt läuft. Ein Wert von 1.0 würde hingegen bedeuten, dass die Uhr mit doppelter Geschwindigkeit- und ein Wert von -0.5, dass die lokale Prozessuhr mit halber Geschwindigkeit der globalen Uhr fortschreitet. Es sind nur Werte > -1.0 erlaubt, da sonst die Prozessuhr rückwärts laufen könnte. Bei allen anderen Werten wird die Einstellung wieder automatisch auf 0.0 gesetzt. Da der Simulator intern mit Fließkommazahlen doppelter Genauigkeit arbeitet, kann es zu kleinen, jedoch vernachlässigbaren, Rundungsfehlern kommen.
- **Prozessausfallwahrscheinlichkeit** (*Integer: process.prob.crash = 0*): Gibt eine Wahrscheinlichkeit in Prozent an, ob der gegebene Prozess während der Simulation zufällig abstürzt. Die Wahrscheinlichkeit bezieht sich auf die komplette Simulationsdauer. Bei einer Einstellung von 100 Prozent und der Simulationsdauer von 15 Sekunden stürzt der Prozess auf jeden Fall zwischen 0ms und 15000ms ab. An welcher Stelle dies geschieht wird zufällig bestimmt. Wenn der Prozess nach seinem Absturz wiederbelebt wird, stürzt er nicht noch einmal zufällig ab. Dies gilt allerdings nicht, wenn die Prozesseinstellungen nach dem Zufallsabsturz erneut geändert und übernommen werden, da dann das Zufallsabsturzereignis 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, ob eine vom aktuell ausgewählten Prozess verschickte Nachricht unterwegs verloren geht. An welcher Stelle die Nachricht zwischen dem Sende- und Empfängerprozess verloren geht wird vom Simulator zufällig gewählt.
- **Maximale Übertragungszeit** (*Long: message.sendingtime.max = 2000*): Gibt die Dauer in Millisekunden an, die eine vom Prozess verschickte Nachricht maximal benötigt, bis sie einen Empfängerprozess erreicht. Im weiteren Verlauf wird dieser Wert mit t_{max} bezeichnet.
- **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. Wenn die aktuelle globale Zeit t_g ist und die Simulationseinstellung "Mittelwerte der Übertragungszeiten bilden" nicht aktiv ist, dann wird die Ereignisseintrittszeit 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 Zufalls-wert 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 heißt, dass stets der Mittelwert der Nachrichtenübertragungszeiten des Sender- und Empfängerprozesses verwendet wird.

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

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

Tabelle 2.3.: Farbeinstellungen

2.4.4. Einstellungen im Expertenmodus

Im Expertenmodus lassen sich zusätzliche Variablen, wie beispielsweise diverse Farbwerte und Anzahl oder Pixel verschiedener der GUI-Elemente, editieren. Auf Abbildung 2.15 sieht der Anwender alle einstellbaren Farben. Die fett-gedruckten Schlüssel in Tabelle 2.3 dienen nur als Standardwerte für die neu zu erstellenden Prozesse und sind auch jeweils in den Prozesseinstellungen für jeden Prozess separat editierbar.

2.5. Protokolle

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

Im mitgelieferten Verzeichnis *saved-simulations* befinden sich alle Beispielsimulationen zum Selbst probieren als *.dat* (Java-Serialisierungsobjekt) abgespeichert.

2.5.1. Beispiel (Dummy) Protokoll

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

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

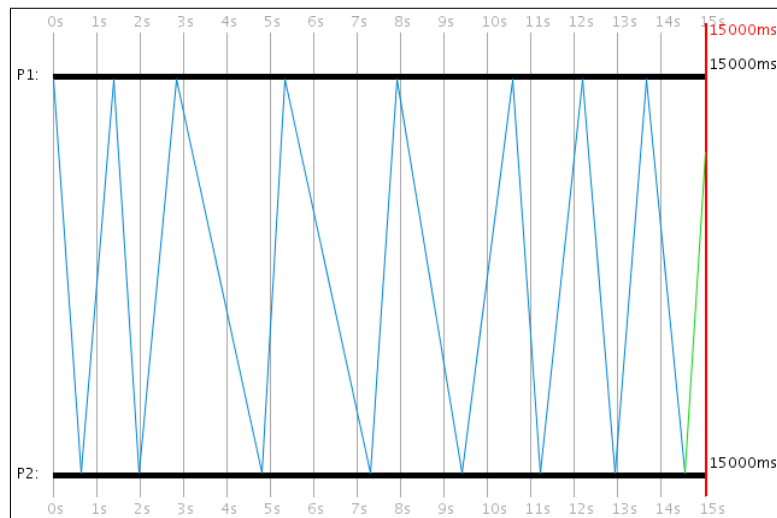


Abbildung 2.16.: Das Ping-Pong Protokoll

Bei dem Ping-Pong Protokoll (Abbildung 2.16) werden zwischen zwei Prozessen, Client P1 und Server P2, ständig Nachrichten hin- und hergeschickt. Der Ping-Pong Client startet die erste Anfrage, worauf der Server dem Client antwortet. Auf diese Antwort wird vom Client ebenfalls geantwortet und so weiter. Jeder Nachricht wird ein Zähler mitgeschickt, der bei jeder Station um eins inkrementiert- und jeweils im Loggfenster protokolliert wird. In Tabelle 2.4 sind alle für dieses Beispiel programmierten Ereignisse aufgeführt.

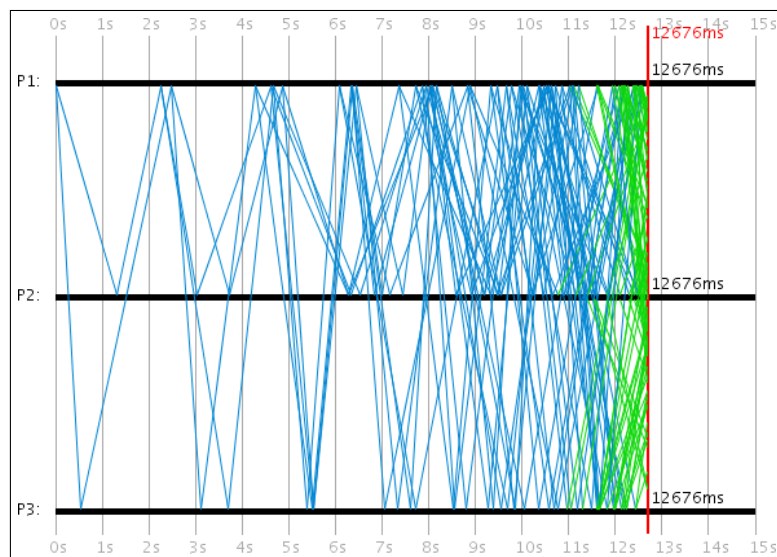


Abbildung 2.17.: Das Ping-Pong Protokoll (Sturm)

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

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

Tabelle 2.4.: Programmierte Ping-Pong Ereignisse

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

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

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

Werden die Ereignisse wie in Tabelle 2.5 abgeändert, so lässt sich ein Ping-Pong Sturm realisieren. Dort wurde ein neuer Prozess 3 eingeführt, der als zusätzlicher Ping-Pong Server agiert. Da auf jede Clientnachricht stets zwei Serverantworten folgen, verdoppelt sich bei jedem Ping-Pong Durchgang die Anzahl der kursierenden Nachrichten. Auf Abbildung 2.17 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 2.6.: Programmierte Broadcast Ereignisse

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

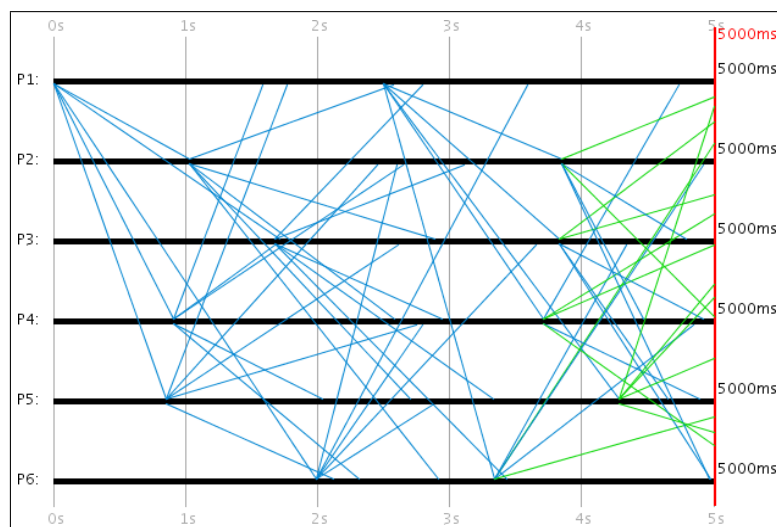


Abbildung 2.18.: Das Broadcast Protokoll

Das Broadcast Protokoll verhält sich ähnlich wie das Ping-Pong Protokoll. Der Unterschied besteht darin, dass sich das Protokoll anhand einer eindeutigen Broadcast-ID merkt, welche Nachrichten bereits verschickt wurden. Das Broadcast Protokoll (server- und clientseitig) verschickt alle erhaltenen Nachrichten, sofern sie vom jeweiligen Prozess noch nicht schon einmal verschickt wurden, erneut.

Der Server und der Client unterscheiden sich in diesem Fall nicht und führen bei Ankunft einer Nachricht jeweils

die selben Aktionen durch. Somit lässt sich, unter Verwendung mehrerer Prozesse (hier 6), wie auf Abbildung [2.18](#), ein Broadcast erzeugen. P1 ist der Client und startet je eine Anfrage nach *0ms* und *2500ms*. Die Simulationsdauer beträgt hier genau *5000ms*. Da ein Client nur Servernachrichten und ein Server nur Clientnachrichten empfangen kann, ist in dieser Simulation jeder Prozess, wie in Tabelle [2.6](#) angegeben, gleichzeitig Server und Client.

2.5.4. Das Protokoll zur internen Synchronisierung in einem synchronen System

(*int-sync.dat*)

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

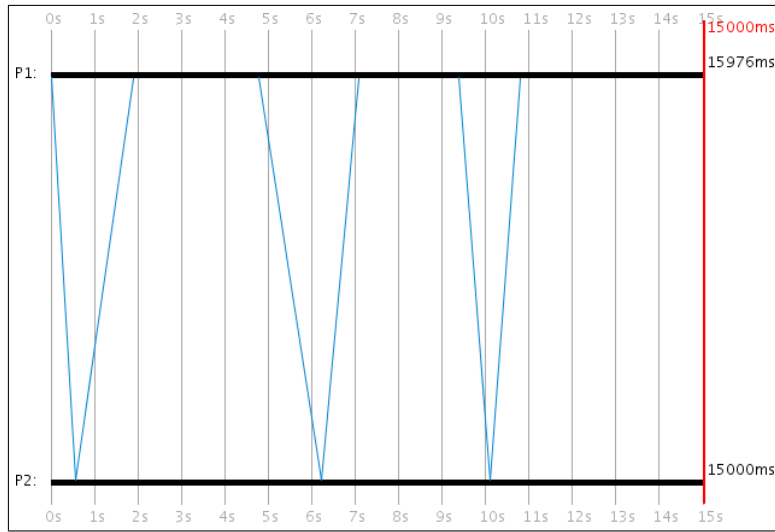


Abbildung 2.19.: Das Protokoll zur internen Synchronisierung

Hier (Abbildung 2.19) stellt P1 den Client und P2 den Server dar. Da die Übertragungszeit t_u einer Nachricht angenommen zwischen t'_{min} und t'_{max} liegt, setzt der Client P1 nach Empfang der Serverantwort seine lokale Prozesszeit auf

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

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

Der Clientprozess hat in der Abbildung 2.19 als Uhrabweichung den Wert 0.1 und der Server hat als Uhrabweichung den Wert 0.0 konfiguriert. Der Client startet, wie in Tabelle 2.7 angegeben, nach 0ms, 5000ms und 10000ms seiner lokalen Prozesszeit jeweils eine Clientanfrage. In der Abbildung lässt sich erkennen, dass die 2. und die 3. Anfrage nicht synchron zu der globalen Zeit (siehe Sekunden-Gatter) gestartet wurden, was auf die

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

Tabelle 2.7.: Programmierte Ereignisse zur internen Synchronisierung

Uhrabweichung von P1 zurückzuführen ist. Nach Simulationsende ist die Zeit von P1 bis auf $15000ms - 15976ms = -976ms$ synchronisiert.

Protokollvariablen

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

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

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

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

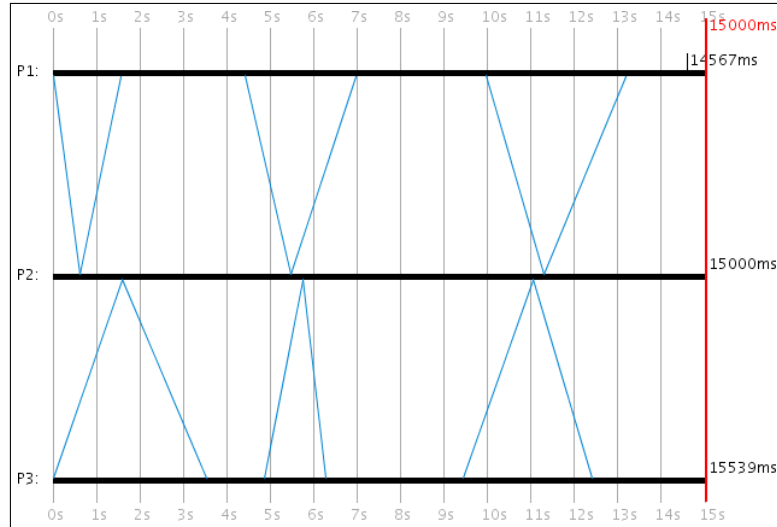


Abbildung 2.20.: Interne Synchronisierung und Christians Methode im Vergleich

Ein weiteres Protokoll für die Synchronisierung von Uhrzeiten funktioniert nach der Christians Methode zur externen Synchronisierung. Die Christians Methode benutzt die RTT (Round Trip Time) t_{rtt} , um die 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 vom Server von dem Zeitpunkt, als der Server die Antwort verschickte. Der Client setzt dann seine lokale Zeit neu auf:

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

und zwar mit einer Genauigkeit von $\pm(\frac{1}{2}t_{rtt} - u_{min})$ wenn u_{min} eine Schranke für eine Nachrichtenübertragung mit $t_{rtt} < u_{min}$ ist (siehe [OBm07]).

Im Prinzip sieht ein Verlauf einer Christians-Simulation so aus wie auf Abbildung 2.19, daher wird hier auf eine einfache Abbildung vom Christians-Protokoll verzichtet. Viel Interessanter ist der direkte Vergleich zwischen dem Protokoll zur internen Synchronisierung und der Christians Methode der externen Synchronisierung (Abbildung 2.20). Hier stellt P1 den Client zur internen Synchronisierung und P3 den Client zur externen Synchronisierung dar. P2 fungiert für beide Protokolle gleichzeitig als Server. P1 und P3 starten jeweils zu den lokalen Prozesszeiten 0ms, 5000ms und 10000ms eine Clientanfrage (Tabelle 2.8). P1 und P3 haben als Uhrabweichung 0.1 eingestellt und die Simulationsdauer beträgt insgesamt 15000ms.

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

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

Auf der Abbildung 2.20 ist ablesbar, dass nach Ablauf der Simulation P1 seine Zeit bis auf $15000ms - 14567ms = 433ms$ und P3 seine Zeit bis auf $15000ms - 15539ms = -539ms$ synchronisiert hat. In diesem Beispiel hat also das Protokoll zur internen Synchronisierung ein besseres Ergebnis geliefert. Dies ist allerdings nicht zwingend immer der Fall, da nach einer erneuten Simulationsausführung alle Nachrichten jeweils eine neue zufällige Übertragungszeit zwischen t_{min} und t_{max} haben werden, die auf das eine oder andere Protokoll wieder andere Auswirkungen haben können.

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

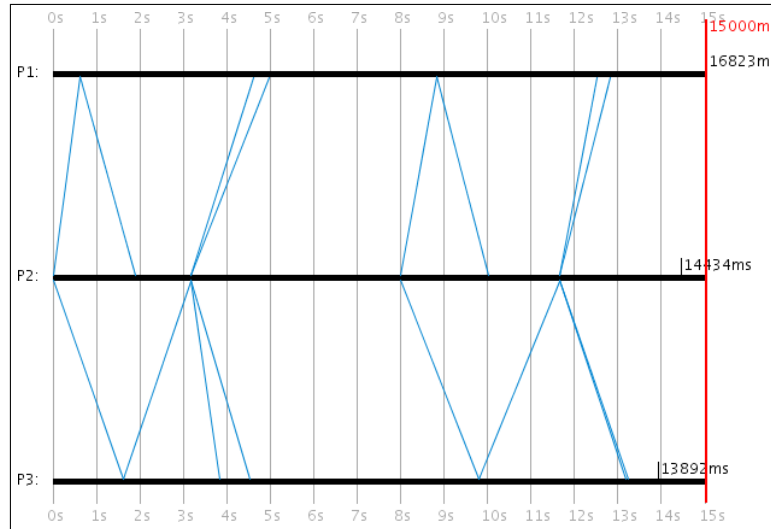


Abbildung 2.21.: Der Berkeley Algorithmus zur internen Synchronisierung

Der Berkeley Algorithmus zur internen Synchronisierung ist eine weitere Möglichkeit lokale Uhrzeiten abzugleichen. Dies ist das erste Protokoll, wo der Server die Anfragen startet. Der Server stellt den Koordinator des Protokolls dar. Die Clients sind somit passiv und müssen warten, bis eine Serveranfrage eintrifft. Hierbei muss der Server wissen, welche Clientprozesse an dem Protokoll teilnehmen, was sich in den Protokolleinstellungen des Servers einstellen lässt.

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

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

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

$$t_s := t_{avg}$$

Anschließend berechnet der Server für jeden Client einen Korrekturwert $k_i := t_{avg} - t_i$, den er jeweils in einer separaten Nachricht zurückschickt. Die Clients setzen dann jeweils die lokale Prozesszeit auf $t'_i := t_i + k_i$. Hierbei

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

Tabelle 2.9.: Programmierte Ereignisse zum Berkeley Algorithmus

stellt t_i' die derzeit aktuelle Prozesszeit des jeweiligen Clients dar. Denn bis zum Eintreffen des Korrekturwertes ist inzwischen wieder neue Zeit verstrichen.

Im Beispiel auf Abbildung 2.21 gibt es die 2 Clientprozesse P1 und P3 sowie den Serverprozess P2. Der Server startet nach jeweils 0ms und 7500ms eine Synchronisierungsanfrage (Tabelle 2.9). Hier fällt auf, dass der Server stets 2 Korrekturwerte verschickt, die jeweils P1 und P3 erreichen. Es werden hier also pro Synchronisierungsvorgang insgesamt 4 Korrekturwerte ausgeliefert. Eine Korrekturnachricht enthält neben dem Korrekturwert k_i auch die PID des Prozesses, für den die Nachricht bestimmt ist. Indem das Protokoll die PID 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 hier keine Variablen.

- **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 gar nicht existiert oder nicht das Berkeley Protokoll clientseitig gar nicht unterstützt. In diesem Fall würde ewig auf eine fehlende Clientantwort gewartet werden.

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

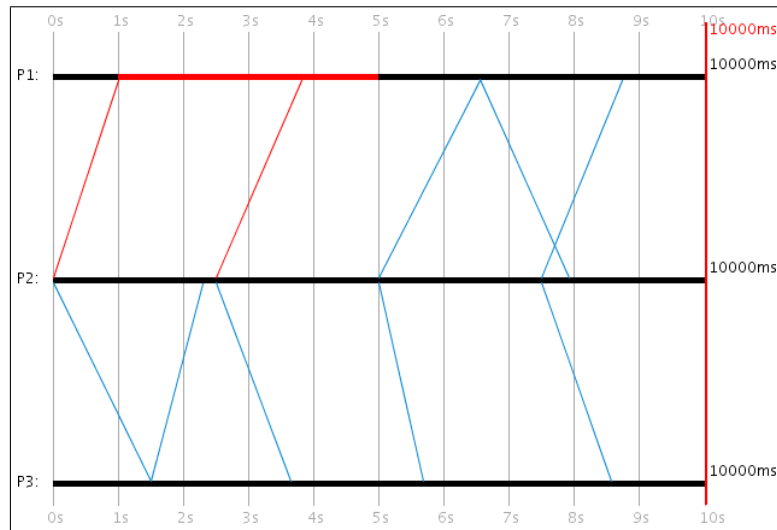


Abbildung 2.22.: Das Ein-Phasen Commit Protokoll

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

Die programmierten Ereignisse des Beispiels auf Abbildung 2.22 sind in Tabelle 2.10 aufgelistet. P1 und P3 simulieren jeweils einen Client und P2 den Server. Damit die Simulation mehrere Festschreibewünsche verschickt, stürzt in der Simulation P1 nach 1000ms ab und nach 5000ms steht er wieder zur Verfügung. Die ersten 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, sodass ein weiterer Festschreibewunsch versendet wird. Da P1 und P3 jeweils schon eine Bestätigung verschickt haben, wird diese Festschreibewunschsnachricht 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 2.10.: 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.

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

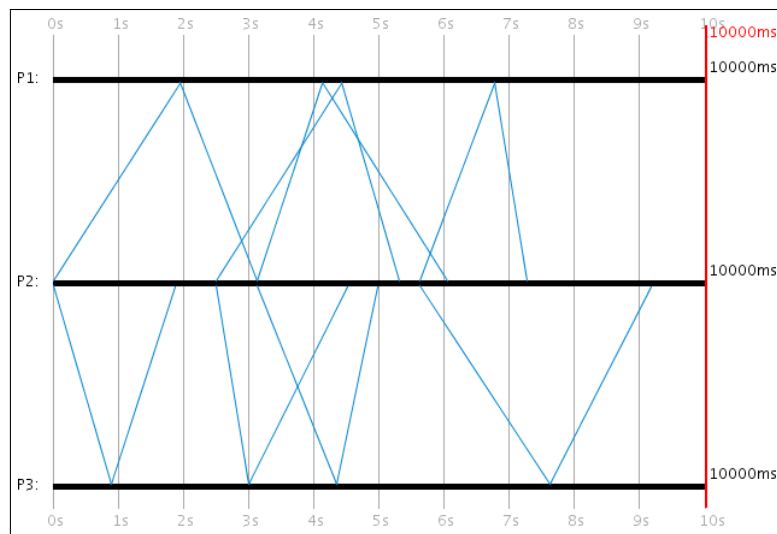


Abbildung 2.23.: Das Zwei-Phasen Commit Protokoll

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

In dem Beispiel (Abbildung 2.23) sind P1 und P3 Clients und P2 der Server. Der Server verschickt nach 0ms seine erste Anfrage (Tabelle 2.11). Da diese Simulation recht unübersichtlich ist, liegen in den Tabellen 2.12 und 2.13 Auszüge aus dem Loggfenster vor. Auf die Lamport- und Vektorzeitstempel sowie die lokalen Prozesszeiten wurde hier wegen Irrelevanz verzichtet. Da keine Uhrabweichungen konfiguriert wurden, sind die lokalen Prozesszeiten stets gleich der globalen Zeit und deswegen wird in den Tabellen pro Loggeintrag jeweils nur eine Zeit angegeben. Anhand der Nachrichten IDs lassen sich dort die einzelnen Sendungen zuordnen. In den Logs wird auch ständig der Inhalt der verschickten Nachricht sowie die dazugehörigen Datentypen aufgeführt. Hier stimmen P1 und P3 jeweils mit *true*, d.h. es soll festgeschrieben werden, ab.

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

Tabelle 2.11.: Programmierte Zwei-Phasen Commit Ereignisse

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.

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

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

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

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

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

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

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

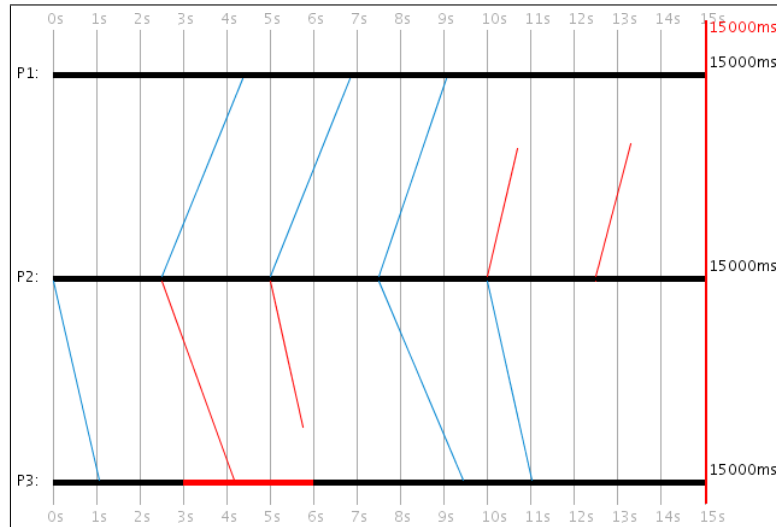


Abbildung 2.24.: Das Basic-Multicast Protokoll

Das Basic-Multicast Protokoll ist sehr einfach aufgebaut. Im Beispiel auf Abbildung 2.24 sind P1 und P3 Server und P2 der Client. Bei diesem Protokoll startet der Client immer die Anfrage, welche bei diesem Protokoll eine einfache Multicast-Nachricht darstellen soll. Die Basic-Multicast Server dienen lediglich für den Empfang einer Nachricht. Es werden keine Bestätigungen verschickt. Wie in Tabelle 2.14 aufgeführt verschickt P2 alle 2500ms jeweils eine Multicast-Nachricht, die alle voneinander völlig unabhängig sind.

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

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

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

Tabelle 2.14.: Programmierte Basic-Multicast Ereignisse

2.5.10. Der zuverlässige (Reliable) Multicast (*reliable-multicast.dat*)

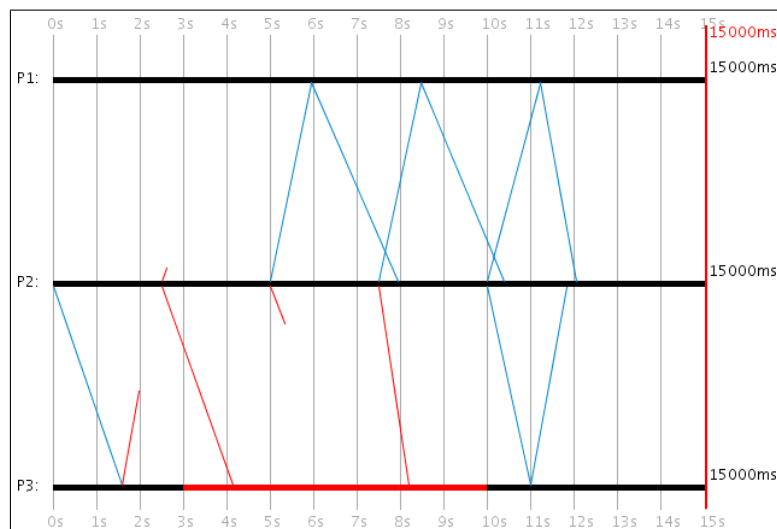


Abbildung 2.25.: Das Reliable-Multicast Protokoll

Bei dem zuverlässigen (Reliable) Multicast verschickt der Client so oft periodisch seine Multicast-Nachricht erneut, bis er von allen beteiligten Servern eine Bestätigung erhalten hat. Nach jedem erneuten Versuch vergisst der Client, von welchen Servern er bereits eine Bestätigung erhalten hat, womit jeder erneuter Versuch von allen Teilnehmern aufs Neue bestätigt werden muss. In dem Beispiel (Abbildung 2.25, Tabelle 2.15, sowie den Loggs in den Tabellen 2.16 und 2.17) ist P2 der Multicast-verschickende Client, während P1 und P3 die Server darstellen. 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

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

Tabelle 2.15.: Programmierte Reliable-Multicast Ereignisse

Versuche:

1. Versuch:

- P1 unterstützt das Reliable-Multicast Protokoll noch nicht, und kann somit weder Multicast-Nachricht erhalten noch eine Bestätigung verschicken.
- P3 empfängt die Multicast-Nachricht, jedoch geht seine Bestätigungsnachricht verloren.

2. Versuch:

- P1: Die Multicast-Nachricht geht unterwegs zu P1 verloren.
- P3: Die Multicast-Nachricht erreicht P3, aber P3 ist abgestürzt und kann somit keine Nachricht verarbeiten.

3. Versuch:

- P1 empfängt die Multicast-Nachricht und seine Bestätigung kommt wie geplant bei P2 an.
- P3: Die Multicast-Nachricht geht unterwegs zu P3 verloren.

4. Versuch:

- P1 empfängt die Multicast-Nachricht und seine Bestätigung kommt wie geplant bei P2 an.
- P3: Die Multicast-Nachricht erreicht P3, aber P3 ist abgestürzt und kann somit keine Nachricht verarbeiten.

5. Versuch:

- P1 empfängt die Multicast-Nachricht und seine Bestätigung kommt wie geplant bei P2 an.
- P3 empfängt die Multicast-Nachricht und seine Bestätigung kommt wie geplant bei P2 an.

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

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

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

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

Protokollvariablen

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

- **Zeit bis erneute Anfrage** (*Long: 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.

2.6. Weitere Beispiele

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

2.6.1. Simulation von Vektor- und Lamportzeitstempel

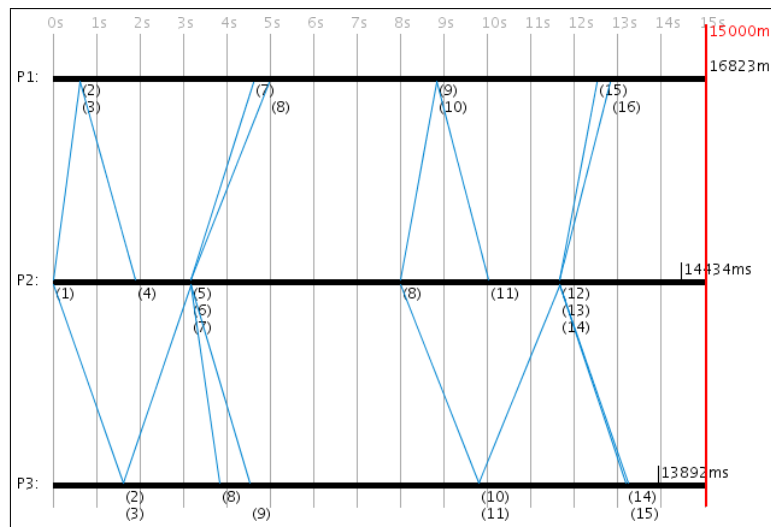


Abbildung 2.26.: Lamportzeitstempel

Die Vektor- und Lamportzeitstempel lassen sich sehr gut am bereits behandeltem Beispiel des Berkeley-Protokoll's demonstrieren. Nach Aktivierung des Lamportzeit-Schalters erscheint bei jedem Ereignis eines Prozesses der aktuelle Lamportzeitstempel (Abbildung 2.26). Jeder Prozess besitzt einen eigenen Lamportzeitstempel, der bei jedem Versenden oder Erhalten einer Nachricht inkrementiert wird. Jeder Nachricht wird die aktuelle Lamportzeit $t_l(i)$ des Senderprozesses i beigefügt. Wenn ein weiterer Prozess j diese Nachricht erhält, so wird der aktuelle Lamportzeitstempel $t_l(j)$ von Prozess j wie folgt neu berechnet:

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

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

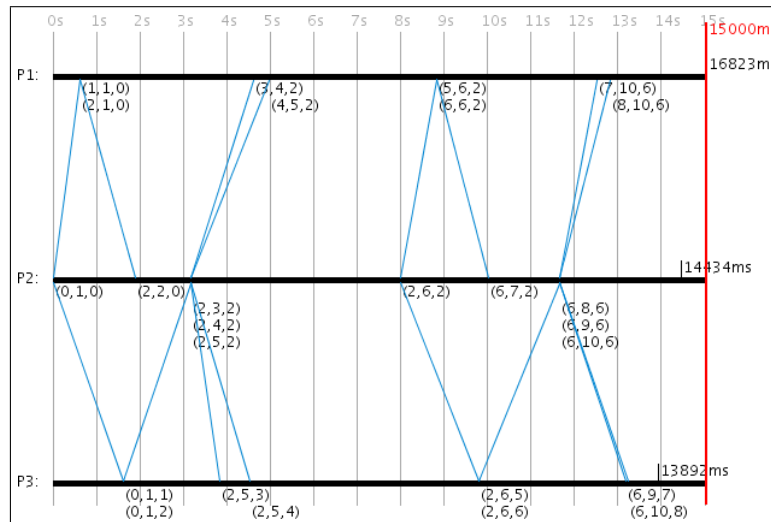


Abbildung 2.27.: Vektorzeitstempel

Mit aktivem Vektorzeit-Schalter werden alle Vektor-Zeitstempel angezeigt (Abbildung 2.27). Wie bei den Lamportzeitstempel wird auch hier jeder Nachricht der aktuelle Vektor-Zeitstempel des Senderprozesses beigelegt. Bei n beteiligten Prozessen hat der Vektor-Zeitstempel v die Größe n . Somit gibt es für jeden beteiligten Prozess i einen eigenen Index i . über $v(i)$ kann jeder Prozess auf 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 Vektorzeitstempel wie folgt (hier in Pseudo-Code angegeben) neu berechnet:

```
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 mit $v(i) = v(i) + 1$. 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 auf Abbildung 2.27 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 totalen Anzahl der Prozesse an.

Wie bereits beschrieben gibt es in den Simulationseinstellungen die booleschen 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].

2.6.2. 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 2.5.5 wieder aufgegriffen, wo das Protokoll zur internen Synchronisation (P1) mit der Christians-Methode (P3) parallel simuliert wurden. P2 stellt den Server beider Protokolle zur Verfügung. In diesem Szenario soll P3 eine schlechte Netzwerkverbindung besitzen, sodass Nachrichten von- und an P3 stets eine längere Übertragungszeit benötigen.

Die Ereignisse sind so wie bereits auf Tabelle 2.8 dargestellt wurde programmiert. In den Simulationseinstellungen ist hier die Einstellung "Mittelwerte der Übertragungszeiten bilden" aktiviert. In den Prozesseinstellungen von P3 wurde "Minimale Übertragungszeit" auf *2000ms* und "Maximale Übertragungszeit" auf *8000ms* gestellt. P1 und P2 behalten als Standardeinstellungen für die minimale und maximale Übertragungszeiten jeweils *500ms* und *2000ms* eingestellt. Die Simulationsdauer wurde auf *20000ms* gestellt.

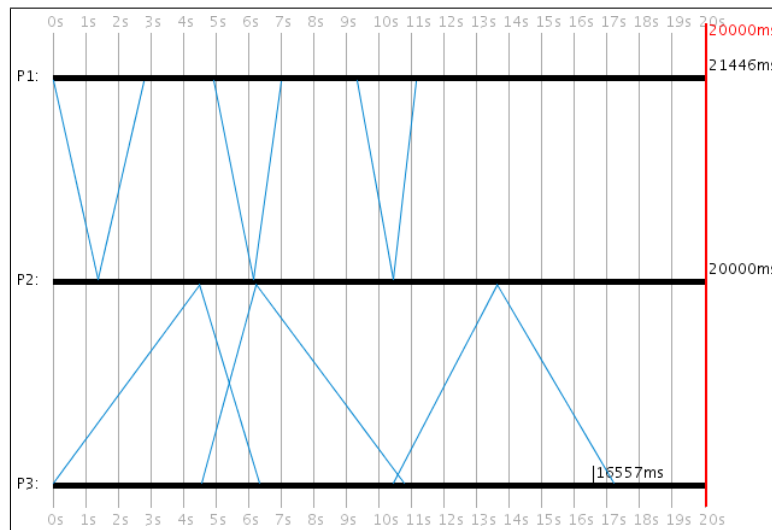


Abbildung 2.28.: Simulation einer langsamen Verbindung

Als Folge (Abbildung 2.28) 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 die RTTs allesamt falsch und seine lokale Zeit wird bei jedem Durchgang zusätzlich verfälscht. Die Berechnungsformeln der Übertragungszeiten wurde bereits in Kapitel 2.4.3 bei den Prozesseinstellungen behandelt. Konkret bedeutet dies für die Übertragungszeiten alle Nachrichten von- und an P3 jeweils:

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

In dem Beispiel auf Abbildung 2.28 ist die lokale Prozesszeit von P1 bis auf $20000 - 21446 = -1446\text{ms}$ synchronisiert, während die Prozesszeit von P3 satte $20000 - 16557 = 3443\text{ms}$ falsch geht.

Kapitel 3.

Die Implementierung

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

Da es sonst den Rahmen sprengen würde, soll im Folgenden der komplette Quelltext nicht bis in das letzte Detail behandelt werden. Der Quelltext erstreckt sich nämlich, einschließlich Kommentare, auf über 15.000 Zeilen und über 61 Dateien. Zudem ist die generierte Quelltext-Dokumentation (Javadoc) über 2MB groß. Alle folgenden UML-Diagramme stellen aufgrund der Übersichtlichkeit lediglich die wesentlichen Dinge dar. Alle Details lassen sich im Quelltext und der dazugehörigen Dokumentation einsehen. Die Paketstruktur des Quelltextes ist in Tabelle 3.1 in alphanumerischer Reihenfolge aufgeführt.

3.1. Einstellungen und Editoren

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

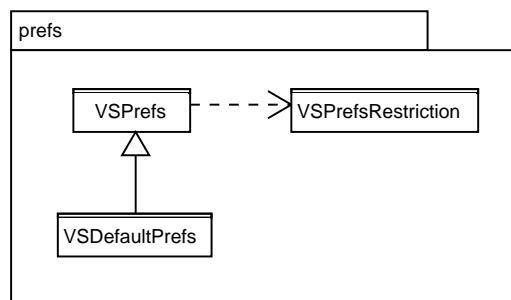
3.1.1. Einstellungsobjekte

Auf Abbildung 3.1 ist der Aufbau des Pakets *prefs* zu sehen. In einer Instanz der Klasse *VSPrefs* lassen sich viele verschiedene Daten als Variablen für eine spätere Verwendung dynamisch ablegen und stellt somit einen Con-

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

Tabelle 3.1.: Die Paketstruktur

tainer für diese Daten dar. In einem *VSPrefs*-Objekt speichert der Simulator alle seine Einstellungen ab. Zudem besitzt jedes Prozessobjekt und jedes Ereignisobjekt für lokale Einstellungen seine eigene Instanz von *VSPrefs*. Später wird noch erklärt, dass Protokollobjekte auch als Ereignisse eingesetzt werden. Somit können Protokolleinstellungen auch in ein *VSPrefs*-Objekt abgespeichert werden. Selbst Nachrichtenobjekte besitzt hiervon eine eigene Instanz, wobei hier die zu verschickenden Daten abgelegt werden können.


Abbildung 3.1.: Das Paket *prefs*

Jede Variable besteht aus einem Datentypen, einem Variablennamen und einer optionalen Beschreibung sowie einem Wert. Einige Datentypen unterstützen auch die Angabe von Minimal- und Maximalwerten (zum Beispiel besteht eine Prozentangabe aus einem Integerwert zwischen 0 und 100), was mithilfe der *VSPrefsRestriction*-Klasse implementiert wird. Da man beispielsweise bei Prozent ein % und bei Millisekunden ein *ms* hinter der Variable sehen möchte, kann für jede Variable auch ein optionaler Einheiten-String abgespeichert werden.

Eine Variablenbeschreibung wird für die Darstellung im GUI verwendet, während der Variablenname eher für die interne Verwendung vom Simulator verwendet wird. Zum Beispiel hat die Variable *message.prob.outage* (Verlust-

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

Tabelle 3.2.: Konventionen für Variablennamen-Präfixe in *VSDefaultPrefs*

wahrscheinlichkeit einer Nachricht) als Variablenbeschreibung “Nachrichtenverlustw’keit”. Wenn für eine Variable keine Beschreibung existiert so wird, wie auf Abbildung 2.15 anhand der Farbvariablen schon gesehen wurde, für die Anzeige einer Variable der Datentyp und der Variablenname verwendet. Variablennamen verwenden die auf Tabelle 3.2 angegebenen Prefixkonventionen. Alle verfügbaren Typen wurden bereits in Tabelle 2.2 aufgelistet. *VSPrefs* stellt für alle Variablentypen entsprechende Zugriffsmethoden zur Verfügung.

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

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

Hierbei stellt *key* den Variablennamen- und *val* den Variablenwert dar. *descr* ist eine optionale Variablenbeschreibung. Es können sowohl Java's Integer-Objekte, als auch Java's primitiver Integer-Typ *int* verwendet werden. Ein *int*-Wert wird intern allerdings als Integer-Objekt abgespeichert (für eine spätere Serialisierung, mehr dazu aber später) und macht somit keinen großen Unterschied. Die Methode *getIntegerKeySet* gibt alle vorhandenen Integer-Variablennamen (*keys*) als *Set* zurück.

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

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

VSSerializablePrefs implementiert das Interface *VSSerializable* und kann somit alle enthaltenen Daten in eine Datei abspeichern beziehungsweise laden. Auf die Serialisierung und Deserialisierung von Simulationen wird später genauer eingegangen.

Die Klasse *VSDefaultPrefs* erweitert *VSSerializablePrefs* und initialisiert bei Instantiierung automatisch alle verfügbaren Simulationsvariablen (bereits schon über 160) mit ihren Standardwerten. Dort sind auch alle Spracheinstellungen abgelegt. Sollte jemand den Simulator in eine andere Sprache, zum Beispiel ins Englische, übersetzen wollen, so muss er lediglich diese Datei und die Protokoll-Klassen (mehr dazu später) editieren. Die Spracheinstellungen sind nämlich in einem *VSPrefs*-Objekt als versteckte String-Variablen abgespeichert. Spracheinstellungen für Protokolle wurden in den Protokollklassen direkt angegeben, da dies mehr Komfort für den Protokollentwickler bietet und für jede neue Textausgabe nicht ständig *VSDefaultPrefs.java* editiert werden muss.

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

3.1.2. Editorobjekte

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

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

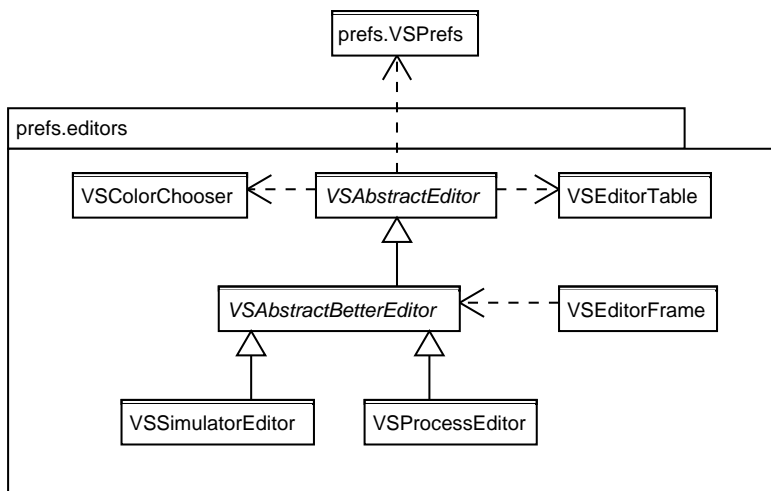


Abbildung 3.2.: Das Paket *prefs.editors*

Die Klasse *VSSimulatorEditor* dient für das Editieren der globalen Simulationseinstellungen und *VSProcessEditor* für das Editieren der Prozesseinstellungen sowie der dazugehörigen Protokollvariablen. Da diese beiden Klassen von *VSAbstractBetterEditor* erben, können sie mithilfe von *VSEditorFrame* in einem separaten Fenster angezeigt werden. Alternativ können die Editoren auch in der Sidebar im Tab "Variablen" angezeigt werden. Auf [Abbildung 2.14](#) wurde bereits ein *VSEditorFrame* in Aktion gesehen. Auf [Abbildung 2.13](#) wurde hingegen ein Prozesseditor in der Sidebar geöffnet. Für Protokolle gibt es keine separate Editor-Klasse, da sie bereits vom Prozesseditor aus editiert werden können. Dabei iteriert der Prozesseditor über alle für den jeweiligen Prozess verfügbaren Protokollobjekte und fügt deren Variablen zusätzlich in den Prozesseditor ein. Somit erscheinen die Prozess- und die dazugehörigen Protokollvariablen im selben Editor und bieten dem Benutzer so eine bessere Übersicht.

3.2. Ereignisse

3.2.1. Funktionsweise von Ereignissen

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

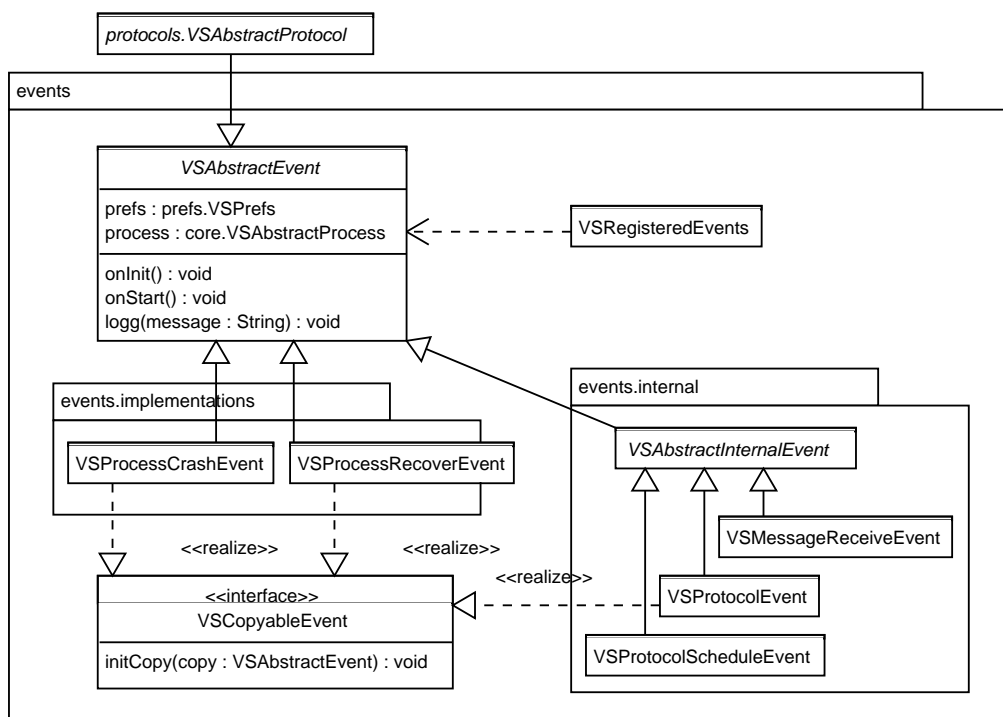


Abbildung 3.3.: Die Pakete *events* und *events.internal*. *

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

In der Implementierung wird zwischen drei Haupttypen von Ereignissen unterschieden, die jeweils in verschiedenen Paketen liegen (Abbildung 3.3):

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.
 - *VSPProcessCrashEvent*: Dieses Ereignis lässt den dazugehörigen Prozess abstürzen.
 - *VSPProcessRecoverEvent*: Dieses Ereignis lässt den dazugehörigen Prozess wiederbeleben.
2. *events.internal*: In diesem Paket befinden sich alle Ereignisse, die vom Simulator intern verwendet werden. Bevor der Simulator von diesen Ereignissen Gebrauch machen kann, muss er vorher einige Spezialbehandlungen durchführen.
 - *VSAbstractInternalEvent*: Diese Klasse stellt weitere Methoden zur Verfügung, die von allen internen Ereignissen benötigt werden. Derzeit betrifft dies nur Methoden zur Serialisierung der gegebenen Objekte. Auf die Serialisierung (Abspeichern/Laden) von Simulationen wird später noch genauer eingegangen.
 - *VSMessageReceiveEvent*: Diese Klasse wird für die Ankunft einer Nachricht bei einem Empfängerprozess benötigt. Sie kapselt die eigentliche Nachricht und überprüft, ob der Empfängerprozess das zur Nachricht dazugehörige Protokoll versteht. Diese Klasse überprüft auch die Simulationseinstellung "Nur relevante Nachrichten anzeigen" und entscheidet, ob die Nachricht nach Eintreffen in der Visualisierung und im Loggfenster berücksichtigt werden soll oder nicht.
 - *VSProtocolEvent*: Diese Klasse implementiert gleichzeitig vier verschiedene Ereignisse: Das Aktivieren/Deaktivieren eines Servers/Clients eines gegebenen Protokolls. Der Ereigniseditor berechnet anhand der verfügbaren Protokolle automatisch alle möglichen Kombinationen und bietet sie dem Anwender in seiner Auswahl an. Für alle dieser vier Ereignisse wird jeweils ein Objekt von *VSProtocolEvent* verwendet, jedoch mit jeweils anderen Attributwerten.
 - *VSProtocolScheduleEvent*: Diese Klasse wird für die Wecker-Ereignisse benötigt. Wecker-Ereignisse können nur von Protokollen (mehr dazu später) erstellt werden. *VSProtocolScheduleEvent* besitzt eine Referenz auf das gegebene Protokoll und ruft bei Ereigniseintrittszeit entweder die Methode *onServerScheduleStart* bei einem Server- oder *onClientScheduleStart* bei einem Clientprotokoll auf.
3. *protocols.implementations*: In diesem Paket befinden sich alle Protokollimplementierung. Jedes Protokoll besitzt hier seine eigene Klasse. Alle Protokolle erben hierbei von der auf Abbildung 3.3 zu sehenden Klasse *protocols.VSAbstractProtocol*. Da *protocols.VSAbstractProtocol* von *events.VSAbstractEvent* erbt, kann ein Protokollobjekt auch als Ereignis eingesetzt werden. Ein solches Ereignis ruft bei Eintritt entweder die Methode *onServerStart* oder die Methode *onClientStart* des Protokolls auf, was einer Server-

beziehungsweise einer Clientanfrage entspricht. Die Implementierung von Protokollen wird später genauer behandelt.

Alle Ereignisse, die das Interface *VSCopyableEvent* implementieren, können vom Anwender im Ereigniseditor mit einem Rechtsklick kopiert werden und müssen die Methode *initCopy(VSAbstractEvent copy)* implementieren. Dort werden 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. Je nach Ereignis können hier verschiedene Werte initialisiert werden. Diese Methode wird pro Ereignisobjekt nach Erstellung nur ein einziges Mal ausgeführt.
- *abstract public void onStart()*: Diese Methode wird jedes Mal ausgeführt, wenn das Ereignis eintritt. Sie stellt somit das Kernstück eines Ereignisses dar.

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

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

Jede Ereignisklasse hat außerdem Zugriff auf folgende Attribute, die von *VSAbstractEvent* vererbt werden:

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

Da *VSAbstractEvent* die Klasse *VSSerializablePrefs* erweitert, können alle Ereignisse mit allen ihren Variablen serialisiert werden. Hierauf wird später nochmal eingegangen.

3.2.2. Beispielimplementierung eines Ereignisses

Im Folgenden wird als Beispiel die Implementierung des Prozessabsturzereignisses *VSPProcessCrashEvent* behandelt. Da die dazugehörige Klasse keine Attribute besitzt, verbleibt hier auch die *initCopy*-Methode mit leerem Rumpf. Wegen der Serialisierung und Deserialisierung 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, wo der Klassenname stets bekannt sein muss. In *onStart()* wird das eigentliche Ereignis ausgeführt. Hier wird obligatorisch überprüft, ob der Prozess bereits abgestürzt (hier eigentlich nicht notwendig, verbessert hier aber die Lesbarkeit) ist und gegebenenfalls wird der Prozess dann zum Absturz bewegt.

```
package events.implementations;

import events.*;

public class VSPProcessCrashEvent
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.logg(prefs.getString("lang.crashed"));
        }
    }
}
```

Der Task-Manager überprüft bereits, ob der Prozess abgestürzt ist oder nicht. Das heißt, dass ein Ereignis bei einem abgestürzten Prozess gar nicht erst ausgeführt wird. Die einzige Ausnahme bildet ein Wiederbelebungsereignis (*VSPProcessRecover*), welches vom Task-Manager ausgeführt wird, auch wenn der Prozess abgestürzt ist. Mit *logg* wird eine Nachricht (die über *prefs* bezogen wird) in das Loggfenster geschrieben.

In der Datei *events/VRegisteredEvents.java* muss in der *init*-Methode für jedes programmierbare 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");
    .
    .
    .
}
```

Als Resultat kann das Prozessabsturzereignis nach Belieben via GUI programmiert- und eingesetzt werden.

3.3. Zeitformate, Prozesse, Nachrichten sowie Task-Manager

3.3.1. Funktionsweise

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

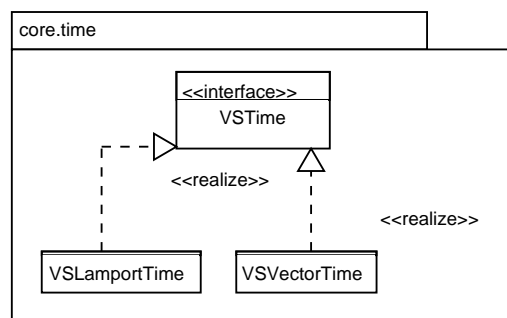


Abbildung 3.4.: Das Paket *core.time*

Auf Abbildung 3.5 ist stark vereinfacht das Paket *core* dargestellt. Für jedes auszuführendes Ereignis wird eine Instanz von *VSTask* benötigt, welche die Ereigniseintrittszeit als Attribut abgespeichert hat sowie eine Referenz

auf das Objekt des auszuführenden Ereignisses (*VSAbstractEvent*) und dem Prozessobjekt (*VSInternalProcess*) besitzt. Geplante *VSTask*-Instanzen werden für eine spätere Ausführung dem Task-Manager übergeben.

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

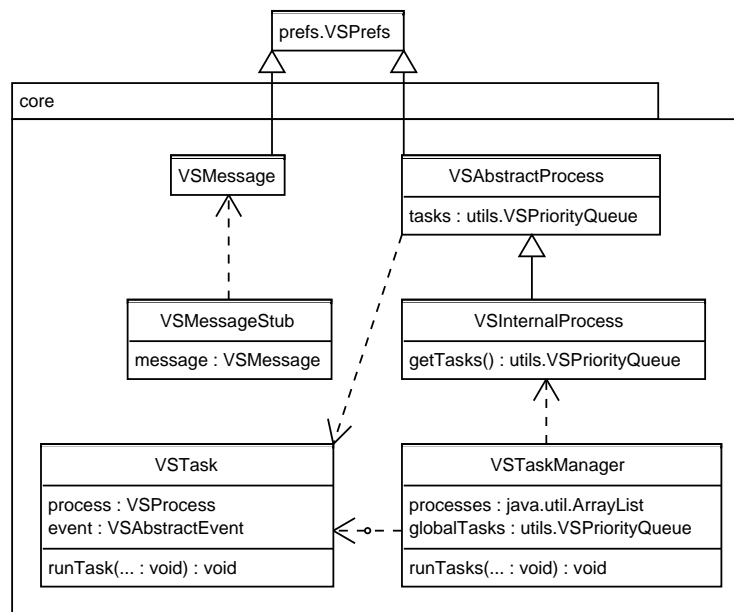


Abbildung 3.5.: Das Paket *core*

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

Eine Instanz von *VSMess* stellt eine Nachricht dar, die von einem Prozess verschickt wird. Für jedes Versen-

den einer Nachricht wird hiervon eine Instanz gebildet, wo der Senderprozess die zu verschickende Daten ablegt. Da *VSMMessage* von *VSPrefs* erbt, können zwischen zwei Prozessen beliebige Datentypen (Tabelle 2.2) über eine Nachricht verschickt werden. Anschließend wird für jeden Empfängerprozess das neue Ereignisobjekt der Klasse *VSMMessageReceiveEvent* angelegt, welches eine Referenz der verschickten Nachricht besitzt (Abbildung 3.6). Danach wird ein *VSTask*-Objekt instantiiert, wo die Referenz auf das Ereignisobjekt und das dazugehörige Prozessobjekt sowie die Ereigniseintrittszeit als Attribute gespeichert werden. Das *VSTask*-Objekt wird dann dem Task-Manager übergeben, der das dazugehörige Ereignis ausführt, wenn die Ereigniseintrittszeit eingetroffen ist. Via Java-Polymorphie wird hier das *VSMMessageReceiveEvent*-Objekt in ein *VSAbstractEvent* umgewandelt.

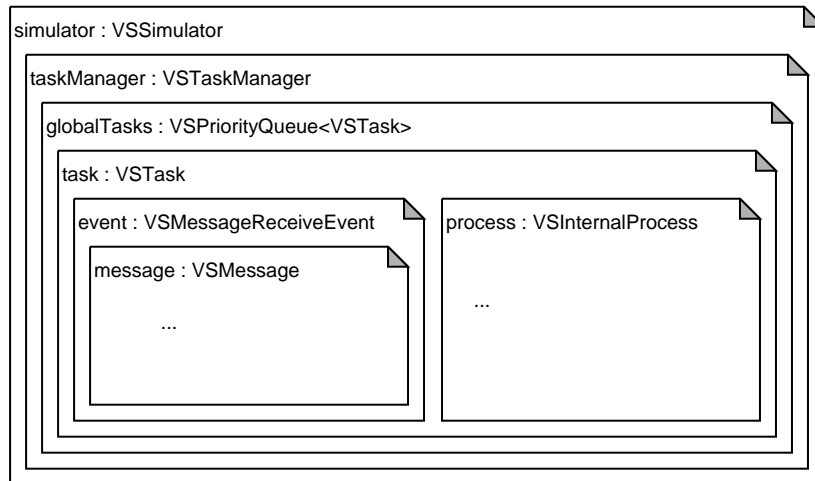


Abbildung 3.6.: Gekapseltes *VSMMessage* im *VSMMessageReceiveEvent*-Objekt

Erwähnenswert ist auch die Klasse *VSMMessageStub*, welche ein *VSMMessage* 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 *VSMMessageStub* im selben Paket wie *VSMMessage* befindet, kann *VSMMessageStub* auf paket-private Methoden von *VSMMessage* 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 *VSMMessageStub*-Objekt anlegen, jedoch kann er auf diese Weise besser unterscheiden auf welche Methoden er zugreifen sollte und auf welche nicht. Das Protokoll-API wird später genauer behandelt.

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

Eine Instanz von *VSInternalProcess* repräsentiert einen simulierten Prozess. Ein *VSInternalProcess* stellt alle vom Simulator intern verwendeten Methoden zur Verfügung, während ein *VSAbstractProcess* lediglich Methoden hat, die 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 auch gar nicht direkt möglich ist. Hierfür hätte man auch ein Stub-Objekt *VSProcessStub* implementieren können. Da aber so gut wie alle paar Millisekunden 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* vererbt. 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* beziehungsweise die lokalen Kopien auf den neusten Stand gebracht. Selbiges gilt für weitere Variablen wie zum Beispiel der Uhrabweichung eines Prozesses.

3.3.2. 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 Wiederbelebungereignis bei einer globalen Zeit von *2000ms* stattfindet. Für den Fall, dass das Wiederbelebungereignis vor dem Absturzereignis eintritt wird es nicht ausgeführt, da der Prozess noch nicht abgestürzt ist.

3.4. Protokolle

3.4.1. Funktionsweise des Protokoll-APIs

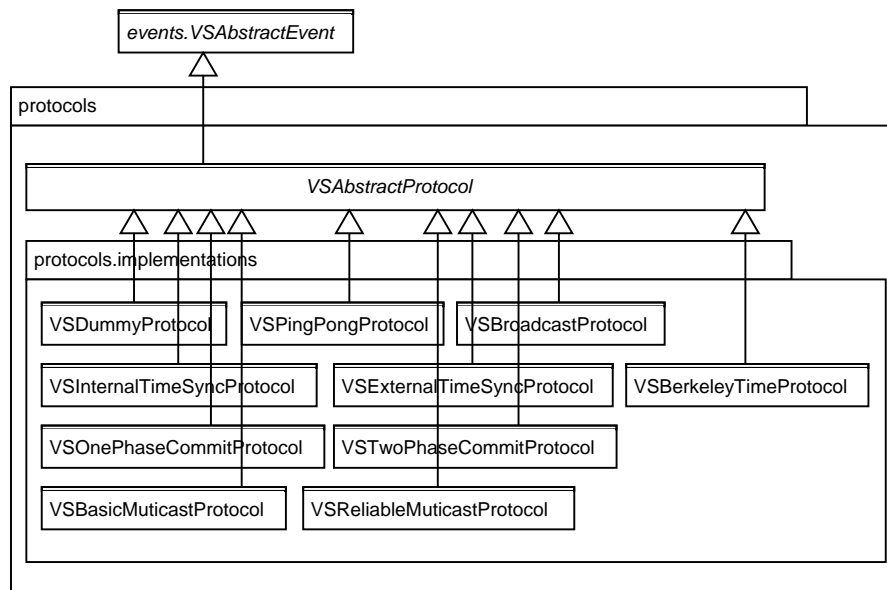


Abbildung 3.7.: Die Pakete *protocols* und *protocols.**

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. Das wird alles vom API automatisch gemacht.

Auf Abbildung 3.9 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- beziehungsweise Clientmethoden des Protokolls auf. *VSAbstractProtocol* überprüft auch, ob ein Client oder ein Server überhaupt aktiviert ist.

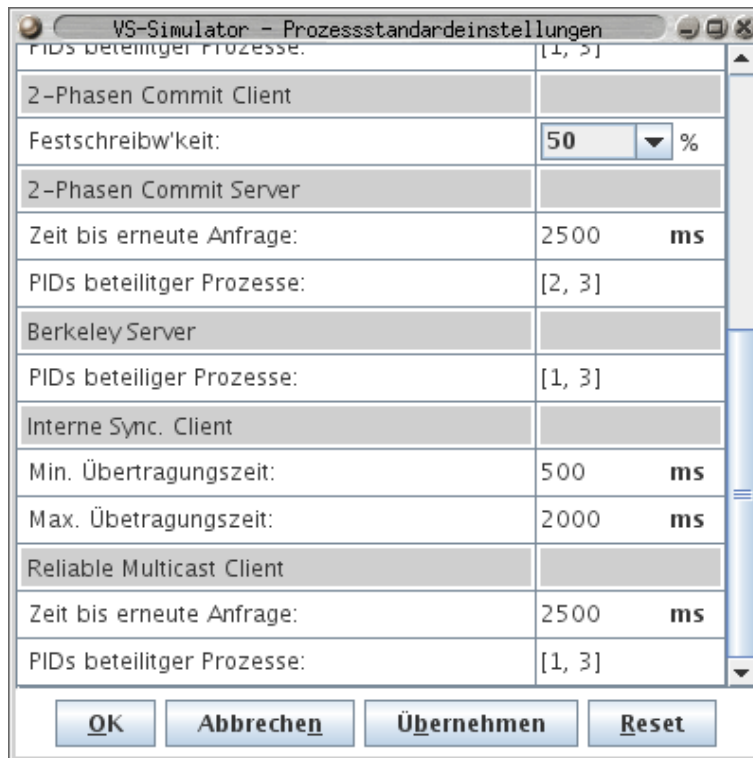


Abbildung 3.8.: Protokollvariablen im Prozesseditor

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 vom Anwender konfiguriert werden.

Da der Simulator dafür 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. Die hier initialisierten Protokollvariablen lassen sich vom Benutzer im Prozesseditor des jeweiligen Prozesses editieren.

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

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

Jede Protokollklasse bekommt folgende Methoden von *VSAbstractProtocol* vererbt, welche allesamt vom Protokollentwickler verwendet werden können:

- *public void sendMessage(VSMMessage message)*: Hiermit verschickt das Protokoll eine Nachricht.
- *public final boolean hasOnServerStart()*: Hiermit lässt sich bestimmen, ob der Server- oder der Client bei dem aktuellen Protokoll die Anfragen startet.
- *public final boolean isServer()*: Hiermit lässt sich bestimmen, ob der aktuelle Prozess das aktuelle Protokoll serverseitig aktiviert hat.
- *public final boolean isClient()*: Hiermit lässt sich bestimmen, ob der aktuelle Prozess das aktuelle Protokoll clientseitig aktiviert hat.
- *public final void scheduleAt(long time)*: Diese Methode stellt einen Wecker dar, der zur angegebenen lokalen Prozesszeit eintritt. Nach Ablauf des Weckers wird, abhängig ob der aktuelle Kontext client- oder serverseitig ist, *onClientSchedule* beziehungsweise *onServerSchedule* ausgeführt.
- *public final void removeSchedules()*: Entfernt alle gesetzten Wecker des aktuellen Kontextes (Server oder Client).
- *public final int getNumProcesses()*: Gibt die totale Anzahl an der Simulation beteiligten Prozesse zurück.

Bei der Implementierung von Protokollen können zusätzlich auf die vererbten Attribute *VSAbstractProcess process* und *VSPrefs prefs* zugegriffen werden. Verfügbare Methoden von *VSPrefs* wurden bereits behandelt. über *prefs* lassen sich alle globalen Simulationseinstellungen abrufen (zum Beispiel die Simulationsvariable die Angibt, ob Prozesse eigene Nachrichten empfangen: *bool recvOwn = prefs.getBoolean("sim.message.own.recv")*). Folgende Prozessmethoden dürfen auf *process* aus dem Protokoll-API verwendet 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 die aktuelle Lamportzeit des Prozesses zurück.
- *public void setLamportTime(long lamportTime)*: Setzt die aktuelle Lamportzeit des Prozesses.
- *public void increaseLamportTime()*: Inkrementiert die Lamportzeit um eins.
- *public void updateLamportTime(long lamportTime)*: Erneuert die Lamportzeit. Siehe Kapitel 2.6.1 wie die Lamportzeiten erneuert werden.
- *public VSVectorTime getVectorTime()*: Gibt die aktuelle Vektor-Zeit des Prozesses zurück.
- *public VSTime[] getLamportTimeArray()*: Gibt die gesamte Lamportzeitstempelhistorie des Prozesses zurück. Kann jeweils nach *VSLamportTime* umgewandelt werden.
- *public VSTime getVectorTimeArray()*: Gibt die gesamte Vektor-Zeitstempel-Historie des Prozesses zurück. Kann jeweils nach *VSVectorTime* umgewandelt werden.
- *public void updateVectorTime(VSVectorTime vectorTimeUpdate)*: Erneuert die Vektorzeit. Siehe Kapitel 2.6.1 wie die Vektorzeiten erneuert werden.
- *public void increaseVectorTime()*: Inkrementiert die Vektorzeit 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. Dieser Wert unterscheidet sich von der PID. Die Prozessnummer gibt an, um den wievielten Prozess, beginnend bei 0, es sich handelt. PIDs hingegen können alle möglichen ganzzahligen positiven Zahlen sein.
- *public int getRandomPercentage()*: Gibt einen Zufallswert zwischen 0 und 100 zurück.
- *public boolean hasCrashed()*: Gibt *true* zurück, wenn der Prozess während der aktuellen Simulation schon mal abgestürzt ist.
- *public boolean isCrashed()*: Gibt *true* zurück, wenn der Prozess aktuell abgestürzt ist.

- *public void isCrashed(boolean isCrashed)*: Hiermit kann man den Prozess abstürzen (*isCrashed = true*) und wiederbeleben (*isCrashed = false*) lassen.

In der Regel werden in Protokollen auch Nachrichten (*VSMMessage*) verschickt. Folgende Methoden dürfen davon im Protokoll-API verwendet werden:

- *public VSMMessage()*: Der Standardkonstruktor für die Erstellung einer neuen Nachricht.
- *public int getMessageID()*: Gibt die Nachrichten-ID zurück.
- *public boolean equals(VSMMessage message)*: Hiermit lässt sich überprüfen, ob eine weitere Nachricht die selbe NID besitzt (wobei es sich dann um die selbe Nachricht handeln würde).

Folgende weitere Methoden von *VSMMessage* können bei Erhalt einer Nachricht verwendet werden:

- *public String getName()*: Gibt den Namen des zur Nachricht dazugehörigen Protokolls zurück. Diese Methode funktioniert nur bei empfangenen Nachrichten richtig.
- *public String getProtocolClassname()*: Gibt den Klassennamen des zur Nachricht dazugehörigen Protokolls zurück. Diese Methode funktioniert nur bei empfangenen Nachrichten richtig.
- *public VSAbstractProcess getSendingProcess()*: Gibt eine Referenz auf den Senderprozess zurück. Diese Methode funktioniert nur bei empfangenen Nachrichten richtig.
- *public long getLamportTime()*: Gibt die Lamportzeit des Senderprozesses zurück. Diese Methode funktioniert nur bei empfangenen Nachrichten richtig.
- *public VSVectorTime getLamportTime()*: Gibt die Vektorzeit des Senderprozesses zurück. Diese Methode funktioniert nur bei empfangenen Nachrichten richtig.
- *public boolean isServerMessage()*: Hiermit lässt sich entscheiden, ob es sich um eine Server- oder eine Clientnachricht handelt. Diese Methode funktioniert nur bei empfangenen Nachrichten richtig.

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

3.4.2. Beispielimplementierung eines Protokolls

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

Im Konstruktor muss stets angegeben werden, ob beim gegebenen Protokoll der Client oder der Server die Anfragen startet. Mit *VSAbstractProtocol.HAS_ON_CLIENT_START* wird dem API mitgeteilt, dass der Client die

Anfragen startet. Für *VSAbstractProtocol.HAS_ON_SERVER_START* und Serveranfragen gilt Selbiges analog. Da ein Protokoll auch ein *VSAbstractEvent* ist, muss auch hier mit *setClassname* der Klassenname 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 abgelegt, von denen noch Bestätigungsnachrichten erwartet werden. Hier werden als Standard-PIDs 1 und 3 verwendet. Die Methoden *initVector* und *initLong* wurden von *VSPrefs* vererbt und initialisieren die Protokollvariablen *pids* und *timeout*, welche vom Benutzer im Prozesseditor editiert werden können (siehe Abbildung 3.8 unter "Reliable Multicast Client" ganz unten):

```
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 *pids* reinitialisiert:

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

In *onClientStart* wird geprüft, ob eine Clientanfrage gestartet werden soll. Wenn dies der Fall ist (wenn von mindestens einem beteiligten Prozess noch keine Bestätigung erhalten wurde), wird ein neues Nachrichtenobjekt erstellt, und mit dem Inhalt *Boolean: isMulticast=true* verschickt (intern wird hier für jeden Empfängerprozess ein *VSMMessageReceiveEvent* erzeugt). Mit *scheduleAt* wird ein Wecker festgelegt, zur welchen lokalen Prozesszeit die Methode *onClientSchedule* aufgerufen werden soll (intern wird hier ein *VSProtocolScheduleEvent* erzeugt):

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

        VSMMessage message = new VSMMessage();
        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. Wenn dies der Fall ist, dann wird geschaut, ob es sich bei der Antwort um eine noch nicht eingetroffene Bestätigung handelt. Gegebenenfalls wird die jeweilige PID aus *pids* entfernt. Wenn *pids* leer ist, dann wurde von allen beteiligten Prozessen eine Bestätigung erhalten und der Client entfernt mit *removeSchedules* alle seine derzeit programmierten Wecker.

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

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

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

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

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

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

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

Serverseite des Protokolls

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

```
private boolean ackSent = false;

public void onServerInit() { }

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

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

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

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

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

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

3.4.3. Erstellung eigener Protokolle (Schnelldurchlauf)

Hier werden alle Schritte zusammengefasst, die für die Erstellung eines eigenen Protokolls *VSMYProtocol* durchgeführt werden müssen. Hierfür muss der Protokoll-Entwickler das Java-SDK sowie 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* → *VSMYProtocol*).
4. In das oberste Verzeichnis *vs/* wechseln.
5. Die Datei *sources/events/VRegisteredEvents.java* editieren, und in der *init*-Methode folgende Zeile hinzufügen:

```
registerEvent("protocols.implementations.VSMYProtocol",  
             "Langer Name des Protokolls", // Langer Name  
             "Neues Protokoll"); // Kurzer 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 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 zu Fehlern. Mit einem neuen Protokoll muss also stets auch immer ein neuer Simulator ausgeliefert werden.

3.5. GUI sowie Simulationsvisualisierung

3.5.1. Funktionsweise

Das Paket *simulator* (vereinfacht auf Abbildung 3.9 dargestellt) implementiert die eigentliche graphische Benutzeroberfläche des Simulators. Ausnahmen sind die Editorklassen in *prefs.editors* sowie *utils.VSFrame*.

Beim Starten des Simulators wird auf die Main-Methode, welche sich in *VSMMain* befindet, aufgerufen. Sie instantiiert ein *VSDefaultPrefs*-Objekt, wo alle Standardeinstellungen des Simulators abgelegt sind. Anschließend wird ein *VSSimulatorFrame* erzeugt, welches ein Simulatorfenster (wie es schon auf Abbildung 2.1 zu sehen war) implementiert. Das Simulatorfenster erstellt für jede neue Simulation jeweils ein Objekt von *VSSimulator*. Jede Simulation hat im Simulationsfenster einen eigenen Tab. Auf Abbildung 2.3 wurde bereits eine neue Simulation erstellt, wo auch unten links der dazugehörige Tab mit der Beschriftung “Simulator 1” zu sehen ist. Jede Simulation besitzt dabei eine eigene Simulationsnummer, die bei jeder neuen Simulation um eins inkrementiert wird. Jedes *VSSimulator*-Objekt greift auf *VSSimulatorVisualization* zurück, was die Simulationsvisualisierung (Abbildung 2.5) implementiert.

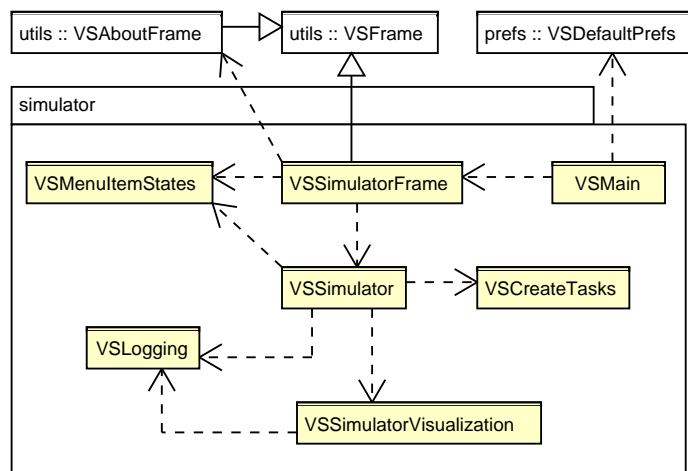


Abbildung 3.9.: Das Paket *simulator*

VSSimulatorVisualization greift auf Java’s Grafikbibliothek Java2D zurück und ist aus Performance-gründen mit dem Simulationsverlauf stark verzahnt [Bra03]. Klassenattribute, dessen Wert sich nie ändert, wurden stets als *final* deklariert. Attribute, die von Konfigurationen oder Einstellungen abhängig sind, die sich nur nach Konfigurationsänderung oder Vergrößern beziehungsweise Verkleinern des Simulationsfensters ändern (Werte, die für die Berechnung des Sekunden-Gatters notwendig sind), werden nur wenn es nötig ist neu berechnet.

Die Klasse *VSMenultemStates* wird für die Synchronisierung des Simulationsstatusses, der Toolbar und des Simulations-Menüs (beide Letztere auf Abbildung 2.4 zu sehen) verwendet. Abhängig davon kann der Benutzer bestimmte Aktionen durchführen oder nicht (beispielsweise kann eine Simulation nur pausiert werden, wenn sie aktuell abgespielt wird). Alle hier möglichen Aktionen wurden bereits in Kapitel 2.1 im Abschnitt “Die Toolbar” behandelt.

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

VSLogging kapselt ein *javax.swing.JTextArea*-Objekt, wo alle Nachrichten geloggt werden. Hier werden alle Logg-funktionen (inklusive Loggfilter sowie temporäre Deaktivierung des Loggen) implementiert. Die *JTextArea* wird dem *VSSimulator*-Objekt übergeben und dort dargestellt. Für den Loggfilter wird intern auf das Java-Standardpaket *java.util.regex* zugegriffen, womit anhand von regulären Ausdrücken in Java-Syntax die Loggs gefiltert werden können.

3.5.2. Threads und Zeitsynchronisierung

Ziel vom Simulator ist es bis auf jede Millisekunde genau simulieren zu können. Jede simulierte Sekunde soll relativ zur echten Zeit fortschreiten. Die Simulationsabspielgeschwindigkeit lässt sich bei den Simulationseinstellungen unter "Abspielgeschwindigkeit der Simulation" (Float: *sim.clock.speed*) einstellen. Damit dies gewährleistet wird, muss folgendes berücksichtigt werden:

- Das Zeichnen der Visualisierung benötigt pro Aktualisierung einige Millisekunden. Dies ist der rechenintensivste Teil des Simulators. Hier werden ständig mathematische Berechnungen (wie zum Beispiel die Gerade einer Nachrichtenlinie, die automatische Skalierung des Diagramms die sich automatisch an die Fenstergröße und der Simulationsdauer anpasst und vieles mehr).
- Das Neuberechnen der Simulation benötigt pro Aktualisierung einige Millisekunden. Hier wird insbesondere der Task-Manager beansprucht, der überprüft, ob Ereignisse auszuführen sind und sie gegebenenfalls dann auch ausführt.
- Jeder simulierte Prozess sollte mit selber Geschwindigkeit fortschreiten, und dies auf jedem Rechner wo der Simulator ausgeführt wird. Da Java-Threads nicht komplett plattformunabhängig sind (Threads sind im Betriebssystem implementiert), kann das Verhalten auf verschiedenen Rechnern minimal variieren. Außerdem übernimmt das Betriebssystem die Entscheidung, wann welcher Thread arbeiten darf. Außer man synchronisiert Threads manuell so, dass sie den eigenen Ansprüchen entsprechen. Letzteres bedeutet aber auch mehr Programmieraufwand.
- Die Simulationszeit ist stets in Millisekunden angegeben, welche in einer *long*-Variable abgespeichert wird. Somit kann eine Simulationszeit immer nur eine ganze Zahl sein. Berechnungsrundungsfehler wegen *sim.clock.speed* müssen berücksichtigt werden.
- Der Simulator soll nicht ständig die komplette CPU des Anwender-Computers voll beanspruchen.

Es wurde folgende relativ einfache Lösung gewählt, bei der lediglich ein einziger Thread für die Visualisierung und die Berechnung der Simulation zuständig ist (alle Zeitangaben sind in Millisekunden). Der Algorithmus verläuft leicht vereinfacht in folgender Form ab:

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

Hinzu kommt noch die Berücksichtigung der Simulationsvariable *sim.clock.speed*, die wegen der Übersicht im Algorithmus nicht dargestellt wurde. Intern hat der Simulator die echte Zeit und die Simulationszeit abgespeichert. Es werden ständig die verstrichenen echten Zeiten gemessen und anschließend anhand von *sim.clock.speed* die neuen tatsächlichen Simulationszeiten berechnet. Rundungsfehler werden pro Durchgang in eine *double*-Variable (Fließkommazahl doppelter Genauigkeit) abgespeichert und wenn der Betrag der Rundungsfehler ≥ 1 ist, dann werden davon die ganzen Wertanteile in der Simulationszeit berücksichtigt. Für jede lokale Prozesszeit sowie der dazugehörigen lokalen Uhrabweichung und den lokale Ereignisse wird ähnlich verfahren.

Jede Simulation besitzt somit seinen eigenen Simulationsthread. Bei mehreren parallel laufenden Simulationen laufen auch mehrere parallele voneinander unabhängige Threads. Des Weiteren gibt es noch den Java Swing-Thread, der für die GUI und der Anwenderinteraktion zuständig ist. Der Anwender kann zu jedem Zeitpunkt in die Simulation eingreifen. Anwendereingriffe werden deswegen synchronisiert, da es ansonsten zu zeitgleichen Zugriffen/Änderungen gleicher Objekte kommen kann und somit eine Java-Ausnahme geworfen wird die das Stoppen eines Threads verursacht.

3.6. Serialisierung und Deserialisierung von Simulationen

3.6.1. Funktionsweise

Der Anwender kann eine erstellte Simulation im Datei-Menü speichern und/oder eine bereits abgespeicherte Simulation laden. Hierbei wird von den aus Java angebotenen Möglichkeit Objekte zu Serialisieren Gebrauch gemacht. Im Paket *serialize* (Abbildung 3.10) befinden sich Helfer, die bei einer Serialisierung und einer Deserialisierung einer Simulation unter die Arme greifen.

Der Simulator serialisiert nur notwendige Daten, und nicht jedes existierende Objekt. 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 (beim Speichern einer Simulation).
- *public void deserialize(VSSerialize serialize, ObjectInputStream ois)*: Diese Methode wird bei jedem Deserialisierungsvorgang aufgerufen (beim Laden einer Simulation).

Die Methoden *serialize* und *deserialize* erhalten neben einen Dateistream auch ein *VSSerialize*-Objekt. Für jeden (De)serialisierungsvorgang wird ein *VSSerialize*-Objekt erzeugt, welches dabei Hilft die benötigten Aktionen durchzuführen. Eine zu serialisierende Simulation besteht aus vielen voneinander abhängigen Objekten. Jedes Objekt kann dabei Referenzen auf andere Objekte besitzen. Würde jedes Objekt komplett serialisiert werden, so würden Objekte, auf denen mehrere Referenzen existieren, in mehrfacher Ausführung behandelt (in eine Datei abgespeichert) werden. Bei Kreisverweisen (Objekt A hat eine Referenz auf Objekt B und Objekt B hat eine Referenz auf Objekt A als Attribut gespeichert) würde die Serialisierung sogar in einer Endlosschleife enden. *VSSerialize* hilft hierbei dies zu vermeiden und merkt sich Informationen von allen bereits serialisierten Objekten, sodass jedes Objekt nur genau einmal serialisiert wird. Bei der Deserialisierung werden alle Objekte wieder automatisch mit den richtigen Referenzen ausgestattet, wobei kein Objekt doppelt deserialisiert wird.

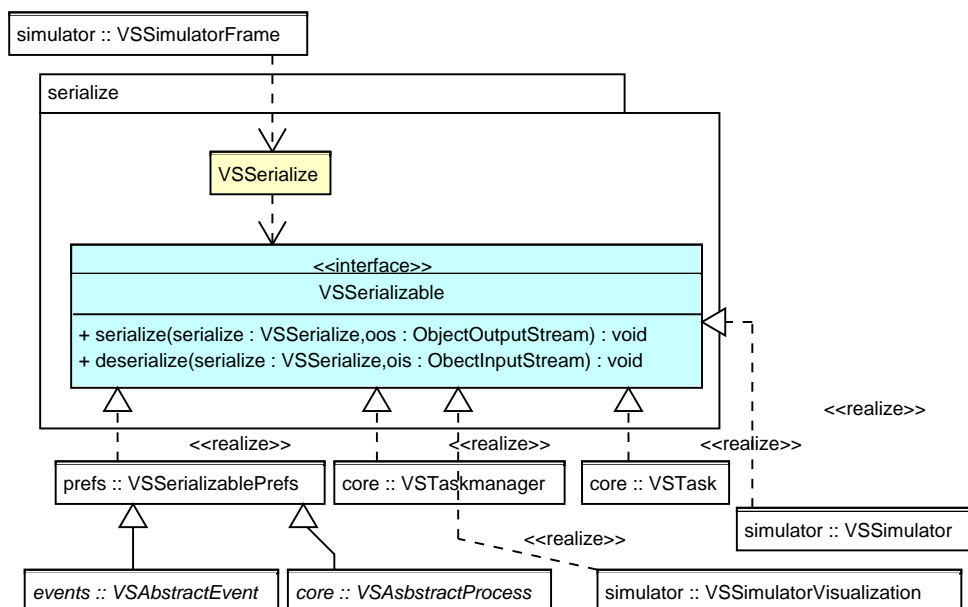


Abbildung 3.10.: Das Paket *serialize* und serialisierbare Klassen

Alle Klassen, die *VSSerializablePrefs* erweitern, können automatisch sämtliche Einstellungen komfortabel serialisieren und deserialisieren. Beispielsweise speichert ein Simulator (*VSSimulator*) alle seine globalen Simulations-

einstellungen 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 derjenigen Klassen, die *VSSerializable* implementieren, nicht komplett serialisiert. Bei der Serialisierung werden nur relevante Klassenattribute, die der Simulationsprogrammierung- und nicht beispielsweise GUI-Komponenten angehören, serialisiert.

3.6.2. Beispielimplementierung einer *serialize*-Methode

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

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

Vor- und nach der eigentlichen Objektserialisierung wird jeweils eine boolesche Flagge serialisiert, welche auf *true* gesetzt wird, sobald in späteren Simulator-Versionen (was relativ unwahrscheinlich, aber möglich ist) weitere zu serialisierende Klassenattribute hinzukommen. Eine Deserialisierung kann die Flaggen dann abfragen und separat behandeln. Somit bleiben ältere bereits abgespeicherte Simulationen stets zur neusten Version des Simulators kompatibel. Wenn eine Flagge auf *true* gesetzt wird, dann kann unter den neuen Attributserialisierungen eine weitere Flagge gesetzt werden. Somit können beliebig viele Erweiterungen in die Serialisierung Einzug halten.

Das zu serialisierende Objekt besitzt hier lediglich zwei Attribute, die serialisiert werden sollen. Alle anderen Klassenattribute können vernachlässigt werden. Mit *serialize.setObject* speichert *serialize* eine Referenz auf das aktuelle Objekt ab, worauf andere Objektserialisierungen zurückgreifen können. Danach wird ein *prefs* und *someOtherSerializableObject* serialisiert. Die Deserialisierung folgt genau den Umgekehrten weg. Wobei hier zuerst

die Instanzen der Klassen auf normalen Weg erstellt werden und dann nachträglich die relevanten Attribute deserialisiert und den Objekten zugewiesen werden. Hierbei werden auch mithilfe von *VSSerialize* mehrere Referenzen auf das selbe Objekt korrekt behandelt.

Wenn der Anwender *Datei* → *Simulation speichern* wählt, dann wird zunächst ein *VSSerialize*-Objekt erstellt. Ausgehend davon wird *serialize* auf *VSSimulator* ausgeführt (siehe Serialisierungssequenz auf Abbildung 3.13). Das Simulator-Objekt führt *serialize* wiederum auf das *VSSimulatorVisualization*-Objekt aus. Dort wird jeder Prozess inklusive alle Protokollobjekte serialisiert. Anschließend folgt der Task-Manager inklusive allen programmierten Ereignissen.

3.7. Helferklassen und Klassen für Ausnahmebehandlungen

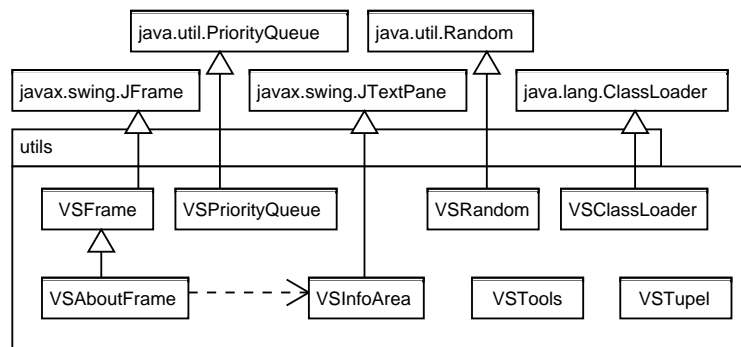


Abbildung 3.11.: Das Paket *utils*

Es wurden noch nicht die Klassen der Pakete *utils* (Abbildung 3.11) sowie *exceptions* (Abbildung 3.12) vorgestellt. *utils* fasst lediglich einige Helferklassen zusammen, die vom restlichen Quelltext verwendet werden.

- *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 aus geöffnet wurden) automatisch mit-geschlossen werden, sobald eines ihrer "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.
- *VSCClassLoader*: Diese Klasse wird für die automatische Instantiierung 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 passen.
- *VSPriorityQueue*: Diese Klasse wird für das Verwalten von *core.VSTask*-Objekte im Task-Manager benötigt. *VSPriorityQueue* passt die Prioritäts-Warteschlange aus der Java-Standardbibliothek den Anforderungen des Simulators an.
- *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 Berechnungen fest.
- *VSTupel*: Diese Klasse ist eine Implementierung eines sehr einfach aufgebauten 3-Tupel Datentyps. Alle 3 Elemente können von einem anderen Typ sein, was mithilfe der Java-Generics verwirklicht wurde. *VSTupel* wird von den Editorklassen für die Generierung von GUI-Elementen benötigt.

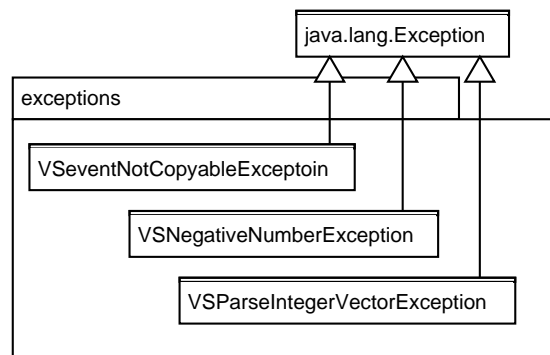


Abbildung 3.12.: Das Paket *exceptions*

Im Paket *exceptions* befinden sich lediglich einige Klassen die für Ausnahmebehandlungen verwendet werden. *VNotCopyableException* wird während einem Kopierversuch eines nicht-kopierbaren Ereignis geworfen. *VSNegativeNumberException* 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 greifen es auf *VSParseIntegerVectorException* zurück.

3.8. Programmierrichtlinien

Die Programmierrichtlinien [Fas08] entsprechen in den meisten Fällen denen aus der Vorlesung [Fas06].

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 (zum Beispiel *events.implementations*).

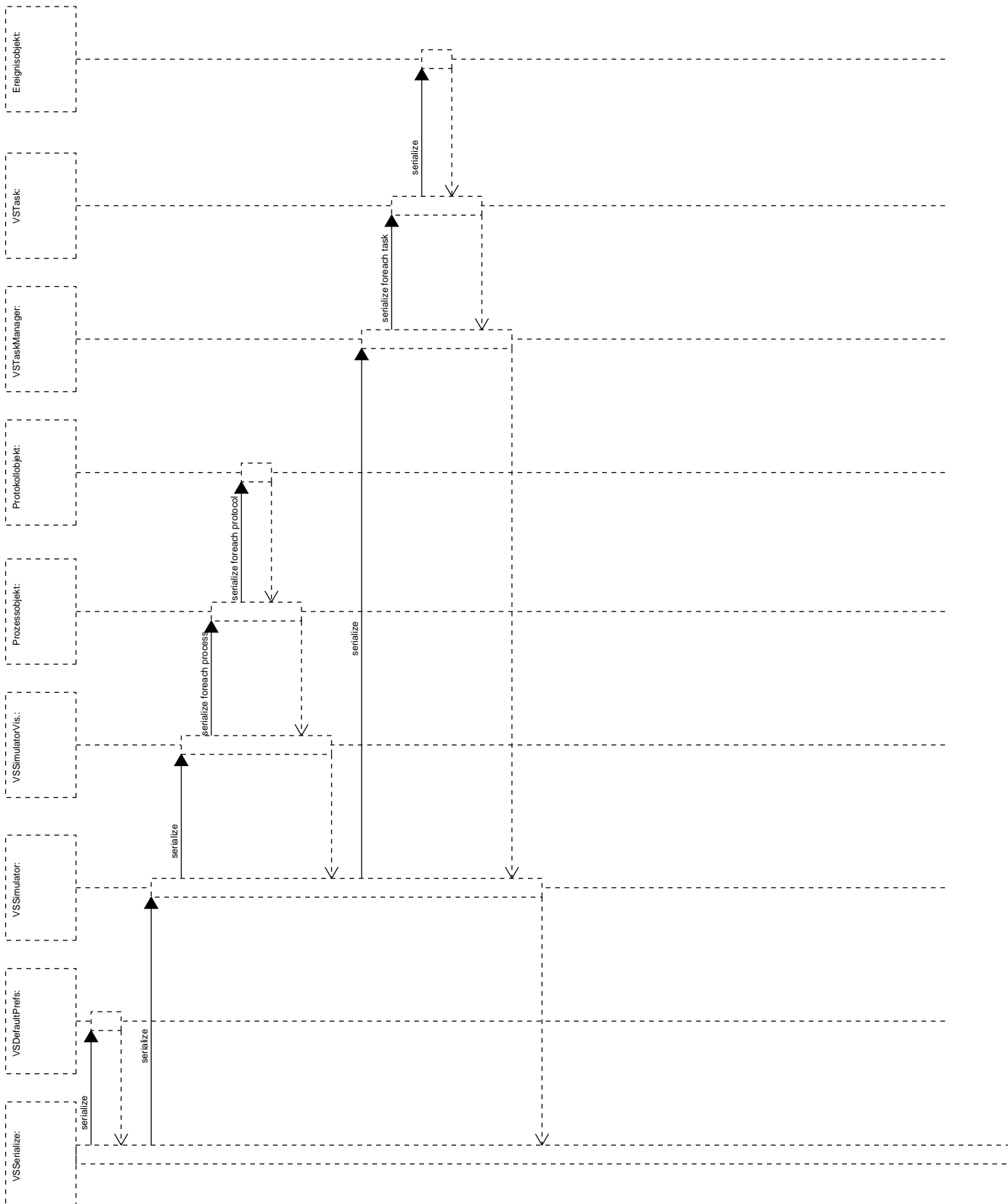


Abbildung 3.13.: Serialisierungssequenz

- Alle Klassen- und Interfacenamen beginnen mit großen Buchstaben, während alle Variablen-, Methoden- und Attributnamen mit kleinen Buchstaben beginnen. Namen finaler Variablen und Attribute sind komplett in Großbuchstaben gehalten.
- Alle Quelltext-Dateien besitzen einen Header, der Informationen der verwendeten Lizenz angibt.
- Alle Quelltext-Dateien werden vollständig mit Javadoc dokumentiert.
- Der komplette Quelltext inklusive Dokumentation werden 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, wo manuelle Zeilenumbrüche wenig Sinn ergeben.
- Es werden zuerst Klassen aus der Java-Standardbibliothek importiert, bevor Klassen aus dem VS-Simulator selbst importiert werden.
- Für die Einrückung des Quelltextes wird das Tool *astyle* mit den Aufrufparametern `-style=java -mode=java` verwendet. Hierbei wird eine Einrückungslänge von 4 Zeichen verwendet.
- Namen aller Klassen und Interfaces tragen als Prefix stets *VS*, was für Verteilte Systeme steht.
- Namen abstrakter Klassen tragen als Prefix stets *VSAbstract*.
- Namen aller Protokollklassen tragen als Postfix *Protocol* (zum Beispiel *VSPingPongProtocol*).
- Namen aller Ereignisklassen, die keine Protokolle implementieren, tragen als Postfix *Event* (zum Beispiel *VSProcessCrashEvent*).
- Namen aller diejenigen Klassen die ein Fenster implementieren tragen als Postfix *Frame* (zum Beispiel *VSSimulatorFrame*).
- überall wo es Sinn ergibt werden Java-Generic-Datentypen verwendet (z.B. *java.util.Vector<Integer>* anstelle von *java.util.Vector*).

3.9. Entwicklungsumgebung

In diesem Teilkapitel soll ein kleiner Einblick in die Umgebung, in der der Simulator entwickelt wurde, gewährt werden. Für diese Diplomarbeit wurde ausschließlich Open Source Software verwendet. Die einzige Ausnahme

stellt Microsoft Windows XP dar, worauf der Simulator zusätzlich getestet wurde. Der Simulator wurde jedoch hauptsächlich unter dem Betriebssystem FreeBSD 7.0, was ein open source Unix-Derivat ist, programmiert.

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

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

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

Sämtliche UML-Diagramme wurden mit ArgoUML angefertigt und die Screenshots mit The GIMP (GNU Image Manipulation Program) sowie ImageMagick nachbearbeitet. Mit 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 4.

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 Gebraucher 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 Gebraucher für eine spätere Wiederverwendung plattformunabhängig abspeichern. Somit können auch abgespeicherte Szenarien beispielsweise an Kommilitonen weitergegeben werden oder für eine spätere Präsentation zwischengespeichert werden. Mit dem Loggfilter lassen sich mithilfe von regulären Ausdrücken nur die relevanten Loggnachrichten anzeigen, was die Analyse einer Simulation erheblich vereinfacht. Weitere Funktionalitäten 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 könnten einige der folgenden Funktionen (hier in alphanumerisch sortierten Reihenfolge aufgelistet) auch eingebaut worden sein:

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

Da der Simulator höchstwahrscheinlich unter einer Open Source Lizenz freigegeben wird, und ich mich selbst sehr für die Entwicklung und Anwendung von Open Source Software interessiere, 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, 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/Erweiterung wurde keine proprietäre Software verwendet, sodass jeder kostenlosen Zugriff auf die dazugehörigen Tools hätte.

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

JRE Java Runtime Environment

NID Nachrichten-Identifikationsnummer

PDF Portable Document Format

PID Prozess-Identifikationsnummer

RTT Round Trip Time

SDK Software Development Kit

SVN Subversion

VS Verteilte Systeme

Anhang B.

Literaturverzeichnis

- [Bra03] David Brackeen. Developing games in java. Buch, 2003. ISBN-13: 978-1592730056.
- [Fas06] Heinrich Fassbender. "objektorientierte softwareentwicklung" an der fh aachen. Vorlesung, 2006.
- [Fas08] Heinrich Fassbender. Programmierrichtlinien "objektorientierte softwareentwicklung" an der fh aachen. Internet, 2008. <http://www.fassbender.fh-aachen.de/Downloads/OOS/Programmierrichtlinien.pdf>.
- [Oßm07] Martin Oßmann. Vorlesung "verteilte systeme" an der fh aachen. Vorlesung, 2007. Mitschrift: ftp://ftp.buetow.org/pub/studium/FHAC_VS-SS07/Mitschrift/verteilte-systeme.pdf.
- [Tan03] Andrew Tanenbaum. Verteilte systeme - grundlagen und paradigmen. Buch, 2003. 2. Autor Marten van Steen; ISBN: 3-8273-7057-4.