



DIPLOMARBEIT

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

Durchgeführt an der

Fachhochschule Aachen

Fachbereich Elektrotechnik und Informationstechnik

Eupener Str. 70

D-52066 Aachen

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

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

durch

Paul C. Bütow

Matr.Nr.: 266617

Matthiashofstr. 15

D-52064 Aachen

Aachen, den 25. Juli 2008

Danksagungen

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

- Martin Oßmann für die Betreuung der Diplomarbeit und der Bereitstellung des für mich sehr interessanten Themas
- Andre Herbst, für das Testen des Simulators; durch seine Hilfe wurden viele Mängel und Bugs aufgedeckt
- Mein Bruder Florian Bütow, für Tipps und Tricks rund um Java, für die Bereitstellung eines Buches sowie für das Testen des Simulators und Probelesen
- Meine Eltern Jörn und Leslie Bütow, die mir das Studium ermöglichten und stets für alle Dinge ein offenes Ohr hatten sowie für das Sponsoring eines weiteren Buches
- Die Open Source Gemeinde; diese Diplomarbeit wurde ausschließlich mithilfe von Open Source Software angefertigt

Inhaltsverzeichnis

1. Einleitung	7
1.1. Motivation	7
1.2. Grundlagen	8
2. Der Simulator	12
2.1. Die grafische Benutzerschnittstelle	12
2.2. Der Expertenmodus	20
2.3. Ereignisse	22
2.4. Einstellungen	25
2.4.1. Simulationseinstellungen	25
2.4.2. Prozess- und Protokolleinstellungen	27
2.4.3. Einstellungen im Expertenmodus	29
2.5. Protokolle	29
2.5.1. Beispiel (Dummy) Protokoll	30
2.5.2. Das Ping-Pong Protokoll	30
2.5.3. Das Broadcast-Sturm Protokoll	32
2.5.4. Das Protokoll zur internen Synchronisierung in einem syn- stem	32
2.5.5. Christians Methode zur externen Synchronisierung	34
2.5.6. Der Berkeley Algorithmus zur internen Synchronisierung	36
2.5.7. Das Ein-Phasen Commit Protokoll	38
2.5.8. Das Zwei-Phasen Commit Protokoll	39
2.5.9. Der ungenügende (Basic) Multicast	43
2.5.10. Der zuverlässige (Reliable) Multicast	44
2.6. Weitere Beispiele	46
2.6.1. Vektor- und Lamportzeitstempel	46

3. Die Implementierung	49
3.1. Gliederung der Pakete	50
3.2. Editoren	50
3.3. Ereignisse	50
3.3.1. Interne Ereignisse	50
3.4. Protokolle	50
3.4.1. Protokoll-API	50
3.5. Serialisierung von Simulationen	50
3.5.1. Rückwärtskompatibel	50
3.6. Programmierrichtlinien	50
3.7. Entwicklungsumgebung	50
4. Ausblick	51
A. Akronyms	52
B. Literaturverzeichnis	53

Abbildungsverzeichnis

1.1. Ein verteiltes System bestehend aus 4 Computern	7
1.2. Client/Server Modell	8
1.3. Client/Server Protokolle	10
2.1. Der Simulator nach dem ersten Starten	12
2.2. Datei-Menü	13
2.3. Eine neue Simulation	14
2.4. Die Menüleiste inklusive Toolbar	14
2.5. Visualisierung einer noch nicht gestarteten Simulation	15
2.6. Rechtsklick auf einen Prozessbalken	16
2.7. Die Sidebar mit leerem Ereigniseditor	17
2.8. Die Ereignisauswahl via Sidebar	18
2.9. Der Ereigniseditor mit 3 programmierten Ereignissen	19
2.10. Das Loggfenster	19
2.11. Der Simulator im Expertenmodus	20
2.12. Die Sidebar im Expertenmodus	21
2.13. Das Fenster zu den Simulationseinstellungen	25
2.14. Weitere Simulationseinstellungen im Expertenmodus	28
2.15. Das Ping-Pong Protokoll	30
2.16. Das Ping-Pong Protokoll (Sturm)	31
2.17. Das Broadcast-Sturm Protokoll	32
2.18. Das Protokoll zur internen Synchronisierung	34
2.19. Interne Synchronisierung und Christians Methode im Vergleich	35
2.20. Der Berkeley Algorithmus zur internen Synchronisierung	36
2.21. Das Ein-Phasen Commit Protokoll	38
2.22. Das Zwei-Phasen Commit Protokoll	40
2.23. Das Basic-Multicast Protokoll	43
2.24. Das Reliable-Multicast Protokoll	45

Tabellenverzeichnis

2.1. Farbliche Differenzierung von Prozessen und Nachrichten	17
2.2. Farbeinstellungen	29
2.3. Programmierte Ping-Pong Ereignisse	30
2.4. Programmierte Ping-Pong Ereignisse (Sturm)	31
2.5. Programmierte Broadcast-Sturm Ereignisse	33
2.6. Programmierte Ereignisse zur internen Synchronisierung	33
2.7. Programmierte Ereignisse, Vergleich interne und externe Synchronisierung . .	36
2.8. Programmierte Ereignisse zum Berkeley Algorithmus	37
2.9. Programmierte Ein-Phasen Commit Ereignisse	39
2.10. Programmierte Zwei-Phasen Commit Ereignisse	40
2.11. Auszug aus der Loggausgabe des Zwei-Phasen Commit Beispiels	41
2.12. Auszug aus der Loggausgabe des Zwei-Phasen Commit Beispiels (2)	42
2.13. Programmierte Basic-Multicast Ereignisse	44
2.14. Programmierte Reliable-Multicast Ereignisse	46
2.15. Auszug aus der Loggausgabe des Reliable-Multicast Beispiels	47
2.16. Auszug aus der Loggausgabe des Reliable-Multicast Beispiels (2)	48

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 schwerfällt 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 Benutzer wie ein einzelnes, kohärentes System erscheinen”*

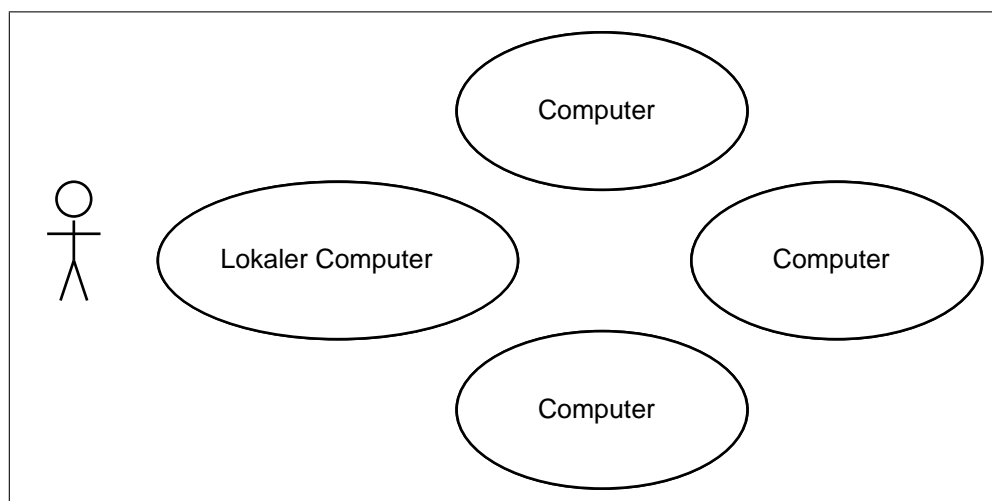


Abbildung 1.1.: Ein verteiltes System bestehend aus 4 Computern

Demnach erscheint dem Benutzer ein aus mehreren Computern bestehendes verteiltes System wie ein System, welches lediglich aus einem einzigen Computer besteht. Der Benutzer muss sich nur mit dem lokalen vor ihm befindenden Computer auseinandersetzen (Abbildung 1.1). Die Software des lokalen Computer stellt die reibungslose Kommunikation mit anderen Computern des verteilten Systems sicher und der Benutzer muss sich darum nicht selbst kümmern.

Hier betrachten wir ein verteiltes System von einer anderen Perspektive. Wir nehmen nicht die Sichtweise eines Endbenutzers ein, sondern wollen die grundlegenden Funktionsweisen, wie unabhängige Computer in einem verteilten System miteinander agieren, verstehen. Es sollen alle relevanten Ereignisse eines verteilten Systems sichtbar und verständlich repräsentiert werden.

Um dieses Ziel zu erreichen wurde ein Simulator entwickelt, der dies ermöglicht. Der Simulator wurde insbesondere für Lehr- und Lernzwecke entwickelt. Er inspiriert auch die Entwicklung eigener verteilter Systeme.

1.2. Grundlagen

Für das Verständnis wie die Simulation von verteilten Systemen funktioniert, werden hier einige Grundbegriffe erläutert. Eine Vertiefung findet erst in den nachfolgenden Kapiteln statt.

Client/Server Modell

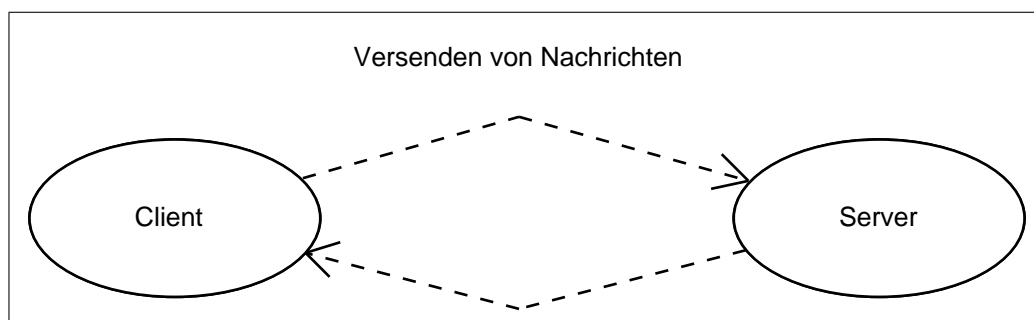


Abbildung 1.2.: Client/Server Modell

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

Prozesse und deren Rollen

Ein verteiltes System wird anhand von Prozessen simuliert. Jeder Prozess nimmt hierbei eine oder mehrere Rollen ein. Beispielsweise kann ein Prozess die Rolle eines Clients einnehmen und ein weiterer Prozess die Rolle eines Servers. Ein Prozess kann auch Client und Server gleichzeitig sein. Es ist auch möglich, dass ein Prozess die Rollen mehrerer Server und Clients auf einmal einnimmt. Ob das sinnvoll ist hängt vom 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, jedoch nicht zwangsmäßig global-korrekte, Zeit des jeweiligen Prozesses dar. Wenn die Prozesszeit nicht global-korrekt ist (nicht der globalen Zeit gleicht), dann wurde die Prozessuhr entweder im Laufe einer Simulation geändert, oder sie geht wegen einer Uhrabweichung falsch. Die Uhrabweichung gibt an, um welchen Faktor die Uhr falsch geht. Wenn eine lokale Uhr nicht geändert wird und auch keine Uhrabweichung hat, dann geht sie stets global-korrekt.

Neben diesen “normalen” Uhren sind auch die **Vektor-Zeitstempel** sowie die **logische Uhr von Lamport** von Interesse. Jeder Prozess besitzt zusätzlich einen Vektor-Zeitstempel für die Vektorzeit, sowie einen Lamportzeitstempel für die 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.

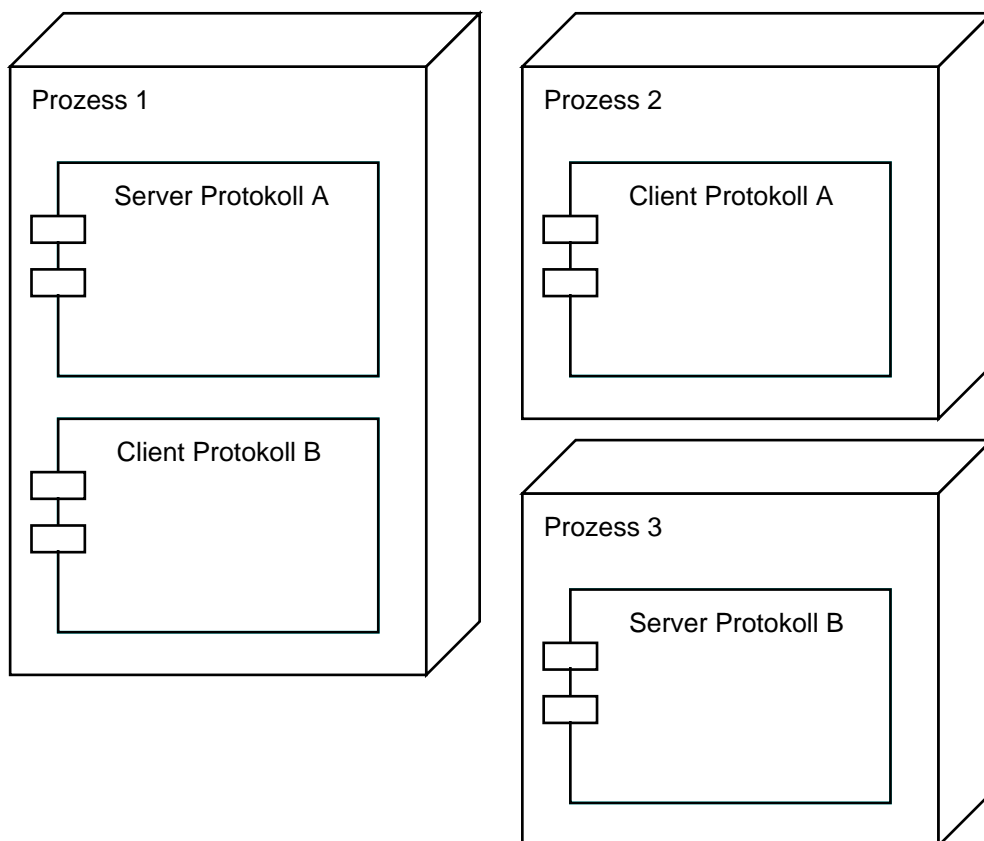


Abbildung 1.3.: Client/Server Protokolle

Ereignisse

Eine Simulation besteht aus der Hintereinanderausführung von endlich vielen Ereignissen. Beispielsweise kann es ein Ereignis geben, welches einen Prozess eine Nachricht verschicken- oder den Prozess selbst abstürzen lässt. Jedes Ereignis tritt zu einem bestimmten Zeitpunkt ein. Wenn es zeitgleiche Ereignisse gibt, so werden sie in Wirklichkeit ebenso hintereinander ausgeführt, erscheinen aber in der Simulation als ob sie parallel ausgeführt würden. Dieser

Umstand ist auf die Implementierung des Simulators zurückzuführen, worauf später noch genauer eingegangen wird. Dem Benutzer des Simulators stört dies jedoch nicht, da Ereignisse aus seiner Sicht parallel ausgeführt werden.

Protokolle

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

In Abbildung 1.3 sind 3 Prozesse dargestellt. Prozess 1 unterstützt serverseitig das Protokoll "A" und clientseitig das Protokoll "B". Prozess 2 unterstützt clientseitig das Protokoll "A" und Prozess 3 serverseitig das Protokoll "B". D.h., Prozess 1 kann mit Prozess 2 via Protokoll "A" und mit Prozess 3 via Protokoll "B" kommunizieren. 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. Dieser Einschränkung kann aber umgangen werden, indem Prozesse ein gegebenes Protokoll sowohl server- als auch clientseitig unterstützt (siehe Broadcast-Sturm Protokoll später). Alle vom Simulator verfügbaren Protokolle werden später genauer behandelt.

Kapitel 2.

Der Simulator

2.1. Die grafische Benutzerschnittstelle

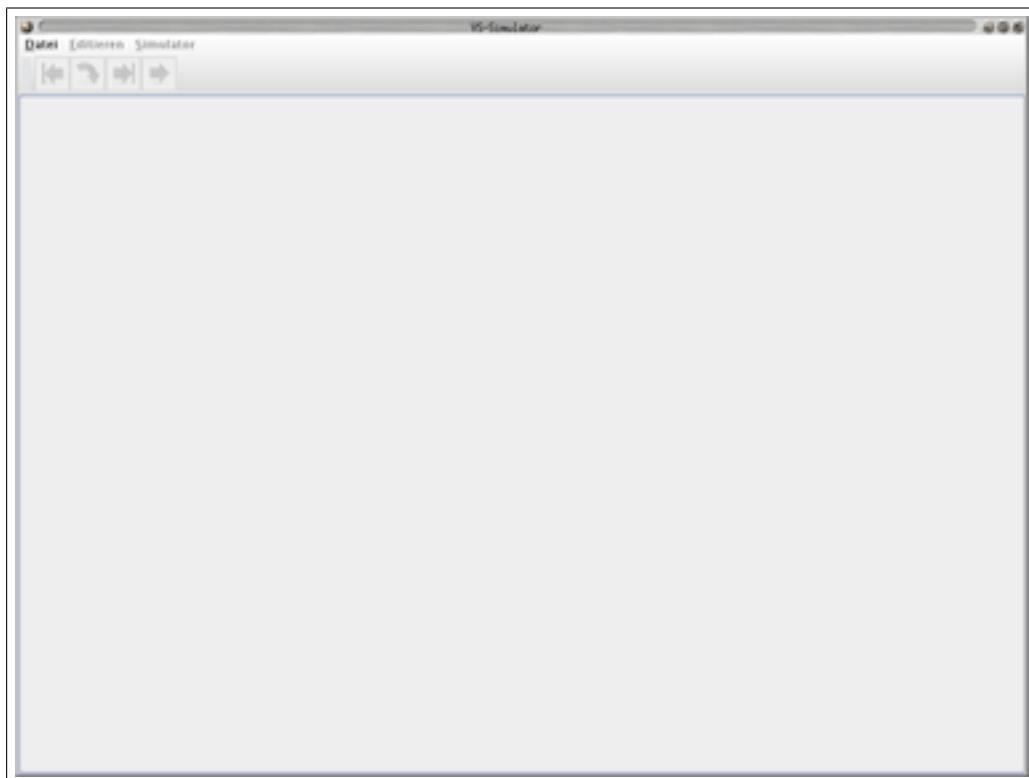


Abbildung 2.1.: Der Simulator nach dem ersten Starten

Der Simulator präsentiert sich nach dem ersten Starten wie in [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 dann wie in Abbildung 2.3 aus.

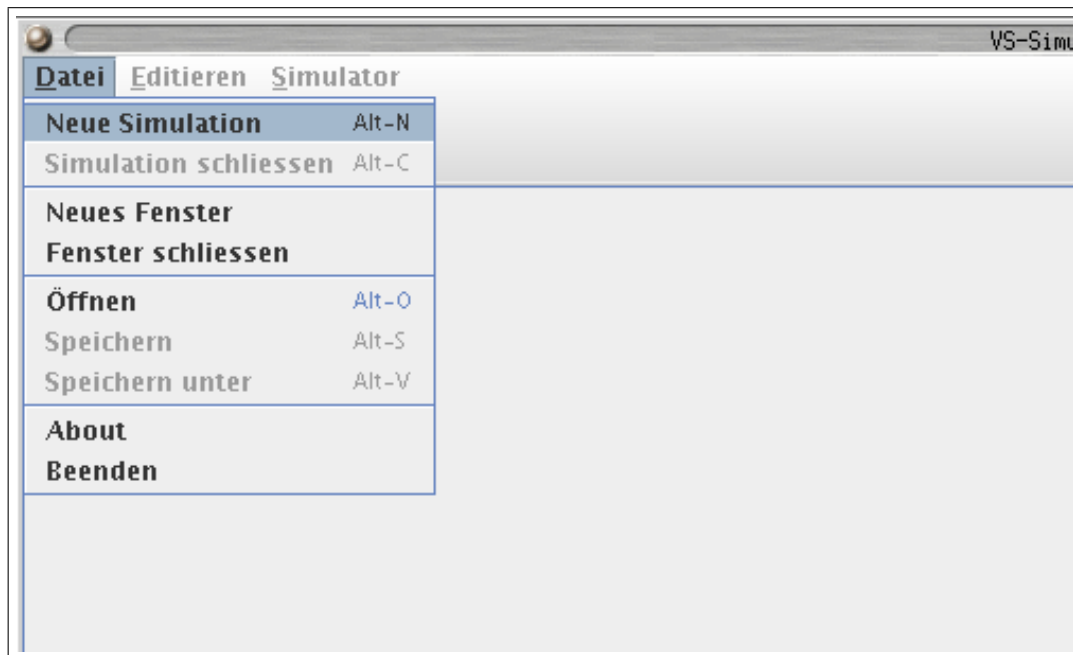


Abbildung 2.2.: Datei-Menü

Die Menüzeile

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

Über das Editieren-Menü gelangt der Benutzer zu den Simulationseinstellungen, worauf später genauer eingegangen wird. Es werden in diesem Menü auch alle beteiligten Prozesse zum Editieren aufgelistet. Wählt der Benutzer 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.

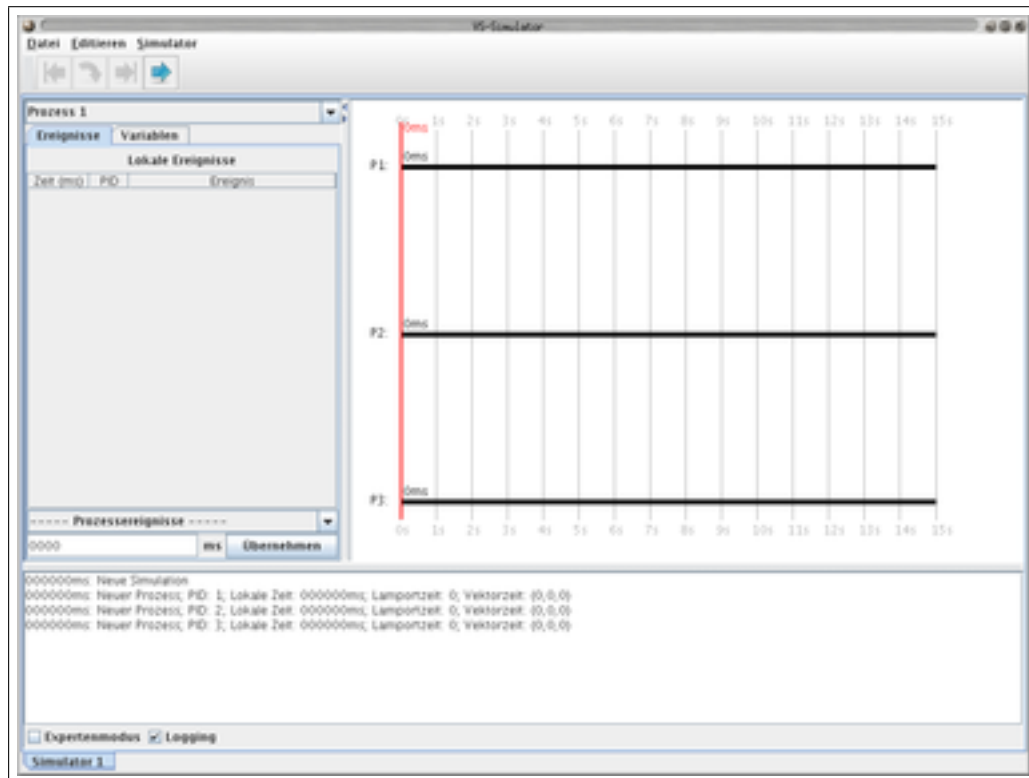


Abbildung 2.3.: Eine neue Simulation

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 Benutzer am häufigsten verwendet werden.



Abbildung 2.4.: Die Menüleiste inklusive Toolbar

Die Toolbar bietet vier verschiedene Funktionalitäten an:

- Starten der Simulation; kann nur betätigt werden, wenn die Simulation derzeit nicht läuft.

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

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

Die Visualisierung

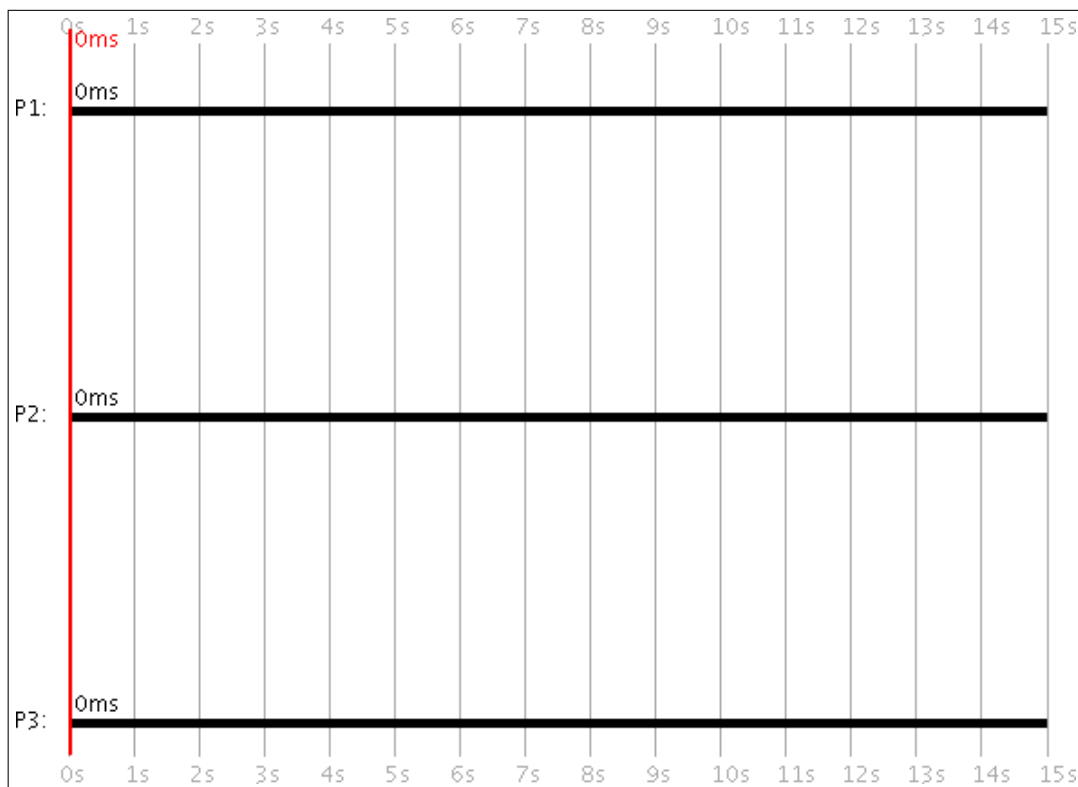


Abbildung 2.5.: Visualisierung einer noch nicht gestarteten Simulation

Mittig rechts in Abbildung 2.3 befindet sich die grafische Repräsentation der Simulation. Die X-Achse gibt die Zeit in Millisekunden an. Unsere Demo-Simulation endet nach genau 15 Sekunden. In Abbildung 2.5 sind 3 Prozesse (mit den PIDs 1, 2 und 3) dargestellt, die jeweils einen eigenen horizontalen schwarzen Balken besitzen. Auf diesen Prozessbalken kann der

Benutzer die jeweilige lokale Prozesszeit ablesen. Die vertikale rote Linie stellt die globale Simulationszeit dar.

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

Eine andere Möglichkeit einen Prozesseditor aufzurufen ist ein Linksklick auf den zum Prozess gehörigen Prozessbalken. Dies muss also nicht zwingend 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 dann abstürzen oder wiederbelebt werden, wenn die Simulation aktuell läuft.

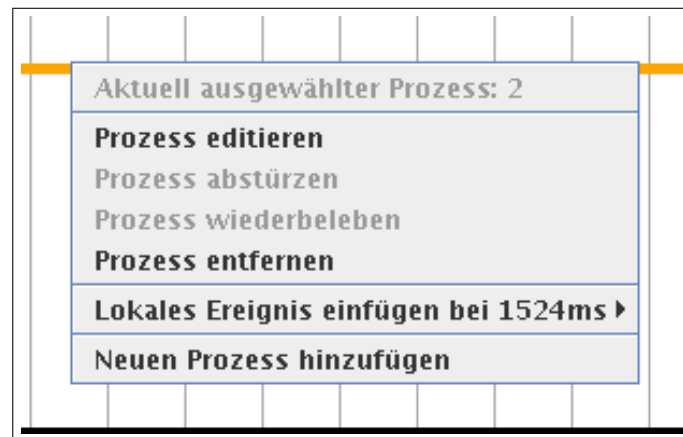


Abbildung 2.6.: Rechtsklick auf einen Prozessbalken

Generell kann die Anzahl der Prozesse nach belieben variieren. Die Dauer der Simulation beträgt mindestens 5 -und maximal 120 Sekunden. Die Simulation endet erst, wenn die globale Zeit 15 Sekunden erreicht hat, und nicht, wenn eine lokale Prozesszeit die 15 Sekunden erreicht.

Farbliche Differenzierung

Farben helfen dabei die Vorgänge einer Simulation 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 der Benutzer über die Einstellungen umkonfigurieren kann.

Prozessfarbe	Bedeutung
Schwarz	Simulation läuft derzeit nicht (z.B. noch nicht gestartet, abgelaufen oder pausiert)
Orange	Die Maus befindet sich über den Prozessbalken
Rot	Der Prozess ist abgestürzt
Nachrichtfarbe	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 (entweder weil der Zielprozess abgestürzt ist oder weil sie unterwegs verloren ging)

Tabelle 2.1.: Farbliche Differenzierung von Prozessen und Nachrichten

Die Sidebar

Mithilfe der Sidebar lassen sich Ereignisse von Prozessen verwalten. Ganz oben in Abbildung 2.7 ist der zu verwaltende Prozess selektiert (hier mit der PID 1). In dieser Prozessauswahl gibt es auch die Möglichkeit "Alle Prozesse" auszuwählen, womit die Ereignisse aller Prozesse gleichzeitig verwaltet werden können. Unter "Lokale Ereignisse" versteht der Benutzer diejenigen Ereignisse, die auftreten, wenn eine bestimmte lokale Zeit des dazugehörigen Prozesses eingetreten ist. Die darunterliegende Ereignistabelle listet alle programmierten Ereignisse (hier noch keine vorhanden) mitsamt Eintrittszeiten sowie den PIDs auf.

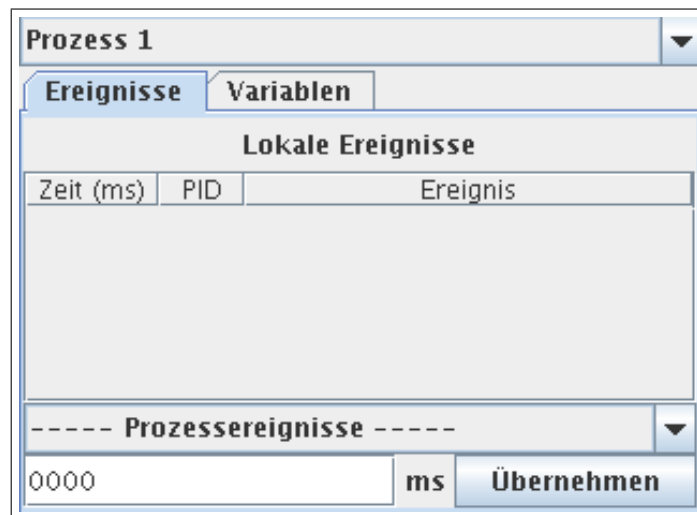


Abbildung 2.7.: Die Sidebar mit leerem Ereigniseditor

Für die Erstellung eines neuen Ereignisses kann der Benutzer entweder mit einem Rechtsklick

auf einen Prozessbalken (Abbildung 2.6) klicken, oder unterhalb der Ereignistabelle ein Ereignis auswählen (Abbildung 2.8), im darunterliegenden Textfeld die Zeit eintragen und auf “Übernehmen” klicken. Beispielsweise wurden auf Abbildung 2.9 drei Ereignisse hinzugefügt: Absturz nach 123ms, Wiederbelebung nach 321ms und erneuerter Absturz nach 3000ms des Prozesses mit der ID 1.

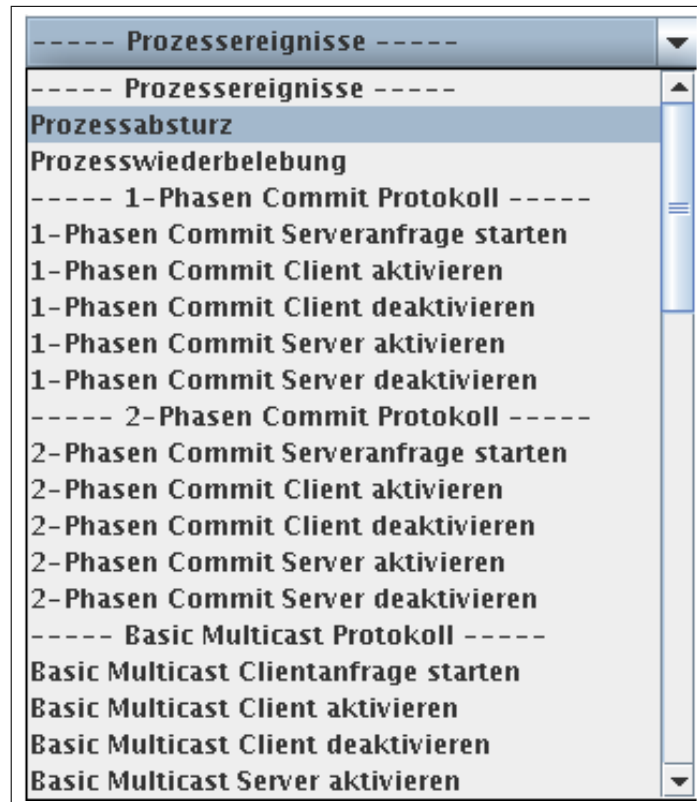


Abbildung 2.8.: Die Ereignisauswahl via Sidebar

Mit einem Rechtsklick auf den Ereigniseditor lassen sich alle selektierten Ereignisse entweder kopieren oder löschen. 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 einem anderen Prozess zuzuweisen.

In der Sidebar gibt es neben dem Ereignis-Tab einen weiteren Tab “Variablen”. Hinter diesem Tab verbirgt sich der Prozesseditor des aktuell ausgewählten Prozesses. Dort können alle Variablen des Prozesses editiert werden. Dies wird später genauer behandelt.

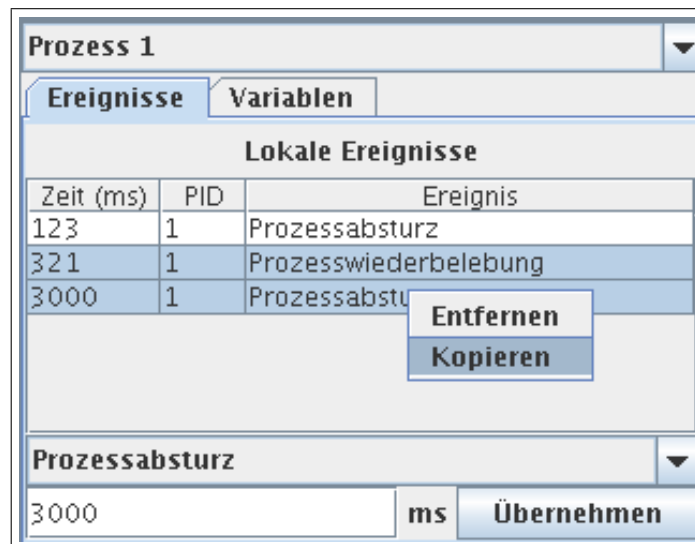


Abbildung 2.9.: Der Ereigniseditor mit 3 programmierten Ereignissen

Das Loggfenster

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

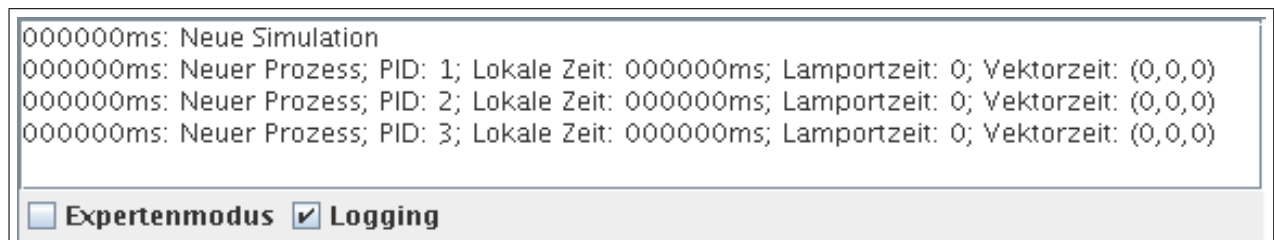


Abbildung 2.10.: Das Loggfenster

Mit dem Deaktivieren der Checkbox "Logging" läßt sich das direkte Loggen von Nachrichten temporär deaktivieren. Ohne aktivierter Checkbox erscheinen keine neuen Nachrichten mehr im Loggfenster. Nach Reaktivieren der Checkbox werden alle ausgelassenen Nachrichten nachträglich in das Fenster geschrieben. Eine Deaktivierung des Loggings kann zu verbessertem Leistungsverhalten des Simulators führen (z.B. kein Ruckeln; ist vom verwendeten Computer, auf dem der Simulator läuft, abhängig). Dieser Umstand ist der sehr langsamen

Java-Implementierung der JTextArea-Klasse zu verdanken.

Über die Checkbox "Expertenmodus" wird der Expertenmodus aktiviert beziehungsweise deaktiviert.

2.2. Der Expertenmodus

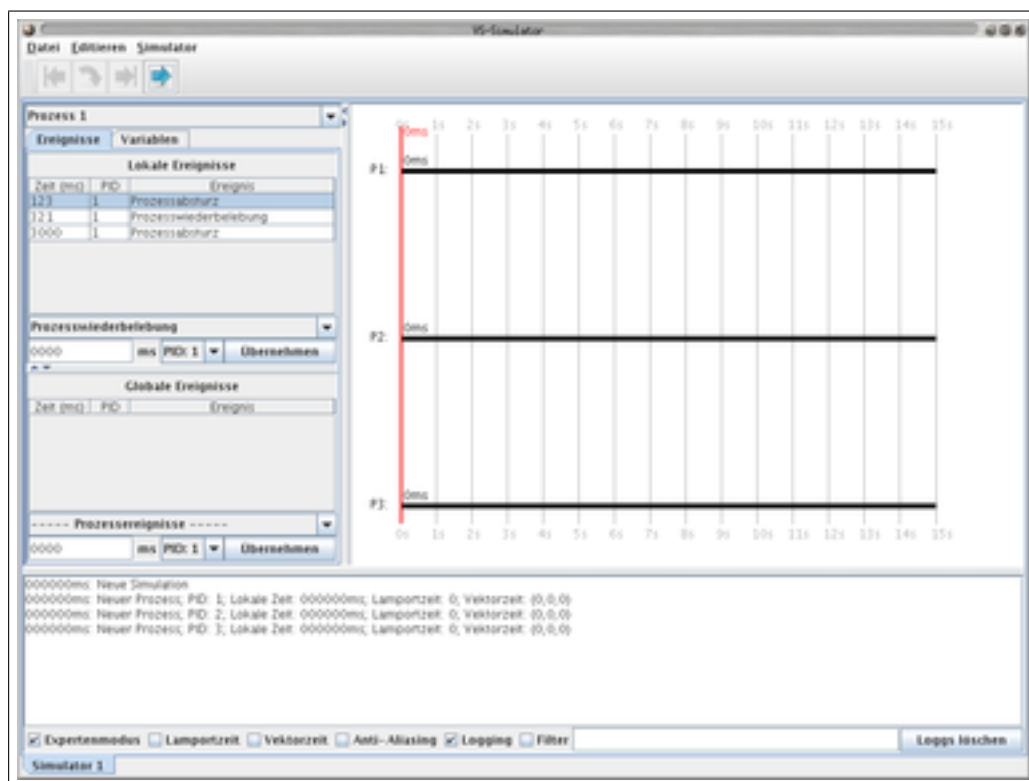


Abbildung 2.11.: Der Simulator im Expertenmodus

Der Simulator kann in zwei verschiedenen Modi betrieben werden. Es gibt einen einfachen und einen Expertenmodus. Der Simulator startet standardmäßig im einfachen Modus, so dass sich der Benutzer 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 für mehr erfahrene Anwender und bietet dementsprechend auch mehr Flexibilität. Der Expertenmodus kann über die gleichnamige Checkbox unterhalb des Loggfensters oder über die Simulationseinstellungen aktiviert oder deaktiviert werden. Auf Abbildung 2.11 ist der Simulator im Expertenmodus zu sehen. Wenn der Benutzer den

Simulator im Expertenmodus mit Abbildung 2.3 vergleicht, dann fallen einige Unterschiede auf, die nun aufs Weitere behandelt werden.

Prozess 1

Ereignisse | Variablen

Lokale Ereignisse

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

Prozesswiederbelebung

0000 ms PID: 1 Übernehmen

Globale Ereignisse

Zeit (ms)	PID	Ereignis
-----------	-----	----------

----- **Prozessereignisse** -----

0000 ms PID: 1 Übernehmen

Abbildung 2.12.: Die Sidebar im Expertenmodus

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.

Eine weitere neue Funktionalität ist die Möglichkeit einem neuzuerstellenden Ereignis direkt die PID zuzuweisen. Im einfachen Modus wurde, wenn der Benutzer ein neues Ereignis erstellte, standardmäßig immer die PID des aktuell ausgewählten Prozesses (in der obersten Combo-Box) verwendet. In dieser Combo-Box sollte der Benutzer gegebenenfalls "Alle Prozesse" selektieren, damit im Ereigniseditor stets die Ereignisse aller Prozesse aufgelistet werden.

Weitere Unterschiede machen sich unterhalb des Loggfensters bemerkbar. Dort gibt es unter Anderem zwei neue Checkboxes “Lamportzeit” und “Vektorzeit”. Aktiviert der Benutzer eine dieser beiden Checkboxes, dann wird die Lamport- beziehungsweise Vektorzeit in die Visualisierung dargestellt. Übersichtshalber kann der Benutzer nur jeweils eine dieser beiden Checkboxes aktivieren. Wenn die Lamportzeit-Checkbox bereits aktiviert ist und der Benutzer versucht die Vektorzeit-Checkbox zusätzlich zu aktivieren, so wird die Lamportzeit-Checkbox automatisch deaktiviert und vice versa.

Die Anti-Aliasing-Checkbox ermöglicht dem Benutzer Anti-Aliasing zu aktivieren und deaktivieren. Mit aktiviertem Anti-Aliasing werden alle Grafiken der Visualisierung gerundet dargestellt. Aus Performancegründen ist Anti-Aliasing standardmäßig deaktiviert.

Je komplexer eine Simulation wird, desto unübersichtlicher werden die Einträge im Loggfenster. Hier fällt es zunehmend schwerer die Übersicht aller Ereignisse zu behalten. Um dem entgegenzuwirken gibt es im Expertenmodus einen Loggfilter, welcher es ermöglicht nur die wesentlichen Daten aus den Loggs zu filtern. Der Loggfilter wird anhand der dazugehörigen Checkbox “Filter” aktiviert beziehungsweise deaktiviert. In der dahinterliegenden Eingabezeile kann ein regulärer Ausdruck in Java-Syntax angegeben werden. Beispielsweise werden mit “PID: (1|2)” nur Loggzeilen angezeigt, die entweder “PID: 1” oder “PID: 2” beinhalten. Alle anderen Zeilen, beispielsweise mit “PID: 3”, werden dabei nicht angezeigt. Mit aktiviertem Loggfilter werden nur die Loggzeilen angezeigt, auf die der reguläre Ausdruck passt. Der Loggfilter kann auch nachträglich aktiviert werden. Bereits protokollierte Ereignisse werden jedes Mal erneuert gefiltert. Der Loggfilter kann auch während einer laufenden Simulation verwendet werden. Wenn der Loggfilter deaktiviert wird, dann werden wieder alle Nachrichten (auch nachträglich) im Loggfenster angezeigt.

2.3. Ereignisse

Es wird zwischen zwei verschiedenen Haupttypen von Ereignissen unterschieden: Programmierbare Ereignisse und nicht-programmierbare Ereignisse. Programmierbare Ereignisse lassen sich im Ereigniseditor editieren und deren Eintrittszeiten hängen von den lokalen Prozessuhren oder der globalen Uhr ab. Nicht-programmierbare Ereignisse lassen sich hingegen nicht im Ereigniseditor angeben und treten nicht aufgrund einer Uhrzeit auf sondern beispielsweise wenn eine Nachricht eintrifft.

Prozessabsturz- und Wiederbelebung (programmierbar)

Die beiden grundlegendsten 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 er eine Nachricht empfangen sollte, geht diese 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, wie gewohnt weiter. D.h. es könnte sein, dass ein Prozess einige seiner lokalen Ereignisse gar nicht ausführt, da er zu den Ereigniseintrittszeiten abgestürzt ist. Selbiges trifft natürlich auch auf globale Ereignisse zu. Wenn im echten Leben ein Computer abstürzt oder abgeschaltet wird, dann läuft dort die Hardwareuhr, unabhängig vom Betriebssystem, auch weiter.

Aktivierung und Deaktivierung von Protokollen (programmierbar)

Wir wissen bereits, dass ein Prozess mehrere Protokolle Client- und auch Serverseitig unterstützen kann. Welches Protokoll von einem Prozess unterstützt wird, kann der Benutzer 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.

Weitere Protokollereignisse (programmierbar)

Der Benutzer hat die Auswahl zwischen fünf weiteren Protokollereignissen:

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

Ob sich das Ereignis für das Starten einer Anfrage auf den Client oder Server bezieht, hängt vom verwendeten Protokoll ab. der Benutzer unterscheidet von Protokollen die Clientseitig- oder Serverseitig eine initiale Anfrage starten. Beispielsweise startet bei dem "Ping-Pong Protokoll" der Client- und bei dem "Commit-Protokollen" der Server immer die erste Anfrage. Es gibt kein Protokoll, wo Client und Server jeweils eine initiale Anfragen starten können.

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

Nachrichtenempfang sowie Antwortnachrichten (nicht-programmierbar)

Nachdem ein Prozess eine Anfragenachricht versendet hat, und ein weiterer Prozess diese Nachricht erhält, so überprüft der Empfängerprozess zunächst ob er das dazugehörige Protokoll versteht. Wenn es sich um eine Clientnachricht handelt, so muß der Empfänger ein Server sein und vice versa. Passt alles, so führt der Empfängerprozess die vom Protokoll definierten Aktionen aus. In der Regel berechnet der Prozess einen Wert und schickt eine Antwortnachricht zurück. Es können aber auch beliebig andere Aktionen ausgeführt werden.

Callback-Ereignisse (nicht-programmierbar)

Ein Callback-Ereignis kann von einem Protokoll ausgelöst werden. Das Protokoll setzt einen Wecker 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. Nach dem Timeout kann dann eine Anfrage erneuert 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, können sie vom Protokoll auch wieder entfernt werden. Wenn ein Callback-Ereignis ausgeführt wird, kann es sich selbst wieder für eine weitere Ausführung erneuert planen. So lassen sich periodisch wiedereintreffende Ereignisse realisieren. Beispielsweise verwenden die "Commit-Protokolle" (mehr dazu später) Callback-Ereignisse, indem solange Anfragen verschickt werden, bis alle benötigten Antworten vorliegen.

2.4. Einstellungen

In diesem Abschnitt wird auf die möglichen Simulationseinstellungen genauer eingegangen. Es wird zwischen drei verschiedenen Typen von Einstellungen unterschieden. Zunächst gibt es globale Simulationseinstellungen. Diese beinhalten Variablen die für die gesamte Simulation gelten. Zudem hat jeder Prozess eigene lokale Einstellungen. Selbst jedes Protokoll hat für jeden Prozess eigene Einstellungen die editiert werden können.

2.4.1. Simulationseinstellungen

Beim Erstellen einer neuen Simulation erscheint zunächst das dazugehörige Einstellungsfenster (Abbildung 2.13). In der Regel reicht es, wenn der Benutzer hier die Standardwerte übernimmt. Es besteht auch die Möglichkeiten nach Erstellen einer Simulation die Einstellungen nachträglich zu ändern, indem der Benutzer das Einstellungsfenster erneuert unter "Editieren → Einstellungen" aufruft.

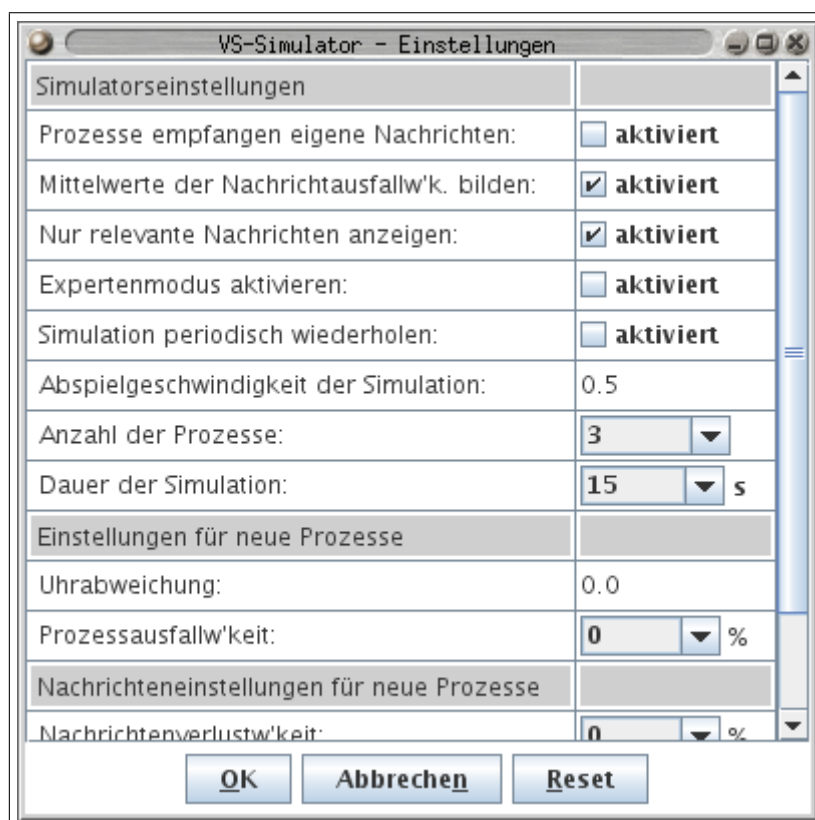


Abbildung 2.13.: Das Fenster zu den Simulationseinstellungen

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

- **Prozesse empfangen eigene Nachrichten** (*Boolean, false*): Standardmäßig können Prozesse übersichtshalber keine Nachrichten empfangen, die sie selbst verschickt haben. Wenn diese Variable jedoch auf true gesetzt wird, dann kann ein Prozess auch auf selbst verschickte Nachrichten antworten. Die Zeit für das Versenden und Empfangen einer Nachricht an sich selbst beträgt jedoch stets 0ms. Diese Variable sollte mit Vorsicht verwendet werden, da hierdurch, bedingt aus den 0ms, Endlosschleifen entstehen können.
- **Mittelwerte der Nachrichtenverlustwahrscheinlichkeiten bilden** (*Boolean, true*): Jede Nachricht die verschickt wird hat, je nach Einstellungen, eine vom verschickendem Prozess abhängige zufällige Übertragungszeit. Wenn diese Option aktiviert ist, so wird der Mittelwert vom Sende- und Empfangsprozess gebildet. Ansonsten wird stets die Übertragungszeit, die beim Senderprozesses angegeben wurde, verwendet.
- **Nur relevante Nachrichten anzeigen** (*Boolean, true*): Wenn nur alle relevanten Nachrichten angezeigt werden, so werden Nachrichten an einen Prozess die er selbst nicht verarbeiten kann, weil er das dazugehörige Protokoll nicht unterstützt, nicht angezeigt. Hierdurch wird eine Simulation viel übersichtlicher.
- **Expertenmodus aktivieren** (*Boolean, false*): Hier lässt sich der Expertenmodus ebenso aktivieren oder deaktivieren.
- **Simulation periodisch wiederholen** (*Boolean, false*): Wenn diese Variable auf true gesetzt ist, so wird die Simulation jedes Mal nach Ablauf automatisch erneuert gestartet.
- **Abspielgeschwindigkeit der Simulation** (*Float, 0.5*): Gibt den Faktor der Simulationsabspielgeschwindigkeit an. Wenn als Faktor 1 gewählt wird, dann dauert eine simulierte Sekunde auch in echt eine Sekunde. Der Faktor 0.5 gibt somit an, dass die Simulation mit halber Echtzeitgeschwindigkeit simuliert wird.
- **Anzahl der Prozesse** (*Integer, 3*): Gibt an, wieviele Prozesse an der Simulation teilnehmen sollen. Wie schon erwähnt kann der Benutzer auch nachträglich via Rechtsklick auf den Prozessbalken den jeweiligen Prozess aus der Simulation entfernen oder weitere Prozesse hinzufügen.

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

Die weiteren Einstellungen unter “Einstellungen für neue Prozesse” sowie “Nachrichteneinstellungen für neue Prozesse” geben lediglich Standardwerte an, die für neu zu erstellende Prozesse verwendet werden.

2.4.2. 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:

- **Uhrabweichung** (*Float, 0.0*): Gibt den Faktor f an, um den die lokale Prozessuhr abweicht. Die neue Uhrzeit eines Prozesses wird wie folgt berechnet:
 - f := Der Faktor wie oben beschrieben
 - t := Aktuelle Prozesszeit in ms
 - t' := Die neu verstrichene Zeit in ms

Die Neue Zeit berechnet sich durch $t := t + t'(1 + f)$. Der Faktor 0.0 besagt also, dass die Uhr keine Abweichung hat. Für f sind nur Werte > -1.0 erlaubt, da sonst die Prozessuhr rückwärts laufen könnte. Bei allen anderen Werten wird der Faktor wieder automatisch auf 0.0 gesetzt. Da der Simulator intern mit Fließkommazahlen doppelter Genauigkeit arbeitet, kann es zu kleinen, jedoch vernachlässigbaren, Rundungsfehlern kommen.

- **Prozessausfallwahrscheinlichkeit** (*Integer, 0*): Gibt eine Wahrscheinlichkeit in Prozent an, ob der gegebene Prozess während der Simulation zufällig abstürzt.
- **Lokale Zeit** (*Long, 0*): Gibt die aktuelle lokale Prozesszeit in Millisekunden an. Es empfiehlt sich daher die Simulation, bevor Prozesseinstellungen vorgenommen werden, zu pausieren.
- **Nachrichtenverlustwahrscheinlichkeit** (*Integer, 0*): Gibt eine Wahrscheinlichkeit in Prozent an, ob eine vom aktuell ausgewählten Prozess verschickte Nachricht unterwegs verloren geht.

- **Maximale Übertragungszeit** (Long, 2000): Gibt die Dauer in Millisekunden an, die eine vom Prozess verschickte Nachricht maximal benötigt, bis sie einen Empfängerprozess erreicht. Der verwendete Wert wird zufällig zwischen (inklusive) der minimalen- und der maximalen Zeit gewählt. Im weiteren Verlauf wird dieser Wert mit t_{max} bezeichnet.
- **Minimale Übertragungszeit** (Long, 500): Gibt die Dauer in Millisekunden an, die eine vom Prozess verschickte Nachricht minimal benötigt, bis sie einen Empfängerprozess erreicht. Der verwendete Wert wird zufällig zwischen (inklusive) der minimalen- und der maximalen Zeit gewählt. Im weiteren Verlauf wird dieser Wert mit t_{min} bezeichnet.

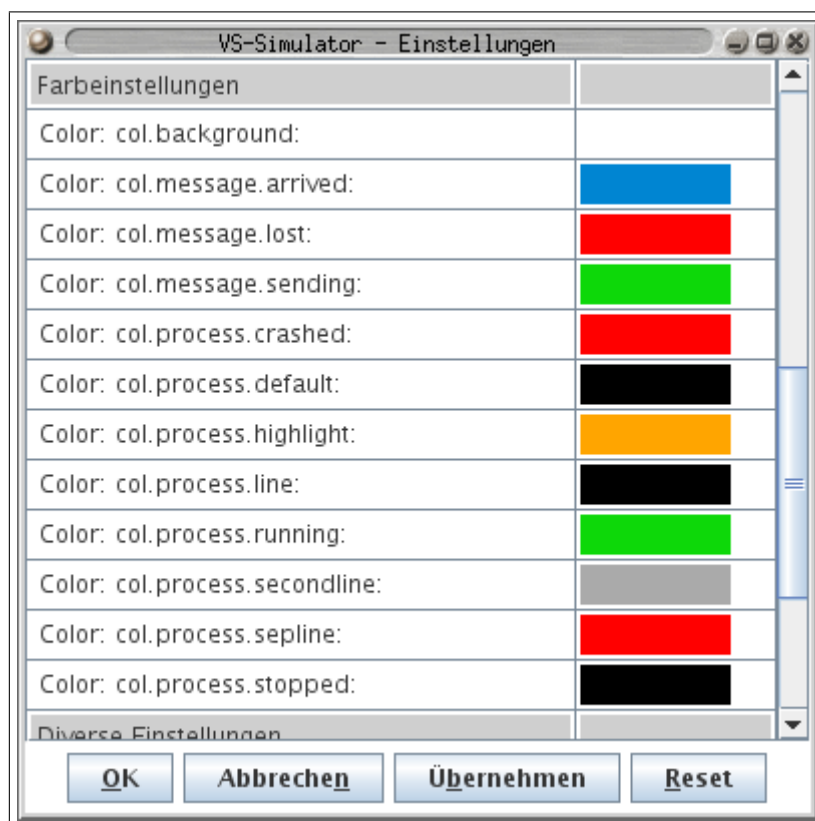


Abbildung 2.14.: Weitere Simulationseinstellungen im Expertenmodus

Im selben Fenster lassen sich auch die Protokollvariablen editieren. Die Protokollvariablen werden jedoch später in der Protokollsektion beschrieben.

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

Tabelle 2.2.: Farbeinstellungen

2.4.3. Einstellungen im Expertenmodus

Mit aktiven Expertenmodus lassen sich zusätzliche Variablen, wie beispielsweise diverse Farbwerte und Anzahl der Pixel verschiedener der GUI-Elemente, editieren. In Abbildung 2.14 sieht der Benutzer alle einstellbaren Farben. Die fettgedruckten Schlüssel in Tabelle 2.2 dienen nur als Standardwerte für neuzuerstellende Prozesse und sind auch jeweils in den Prozesseinstellungen separat editierbar.

2.5. Protokolle

Im Folgenden werden alle bisher 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 Server und Client ist. Der Benutzer 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.

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

Tabelle 2.3.: Programmierte Ping-Pong Ereignisse

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, jedoch keine weiteren Aktionen ausgeführt.

2.5.2. Das Ping-Pong Protokoll

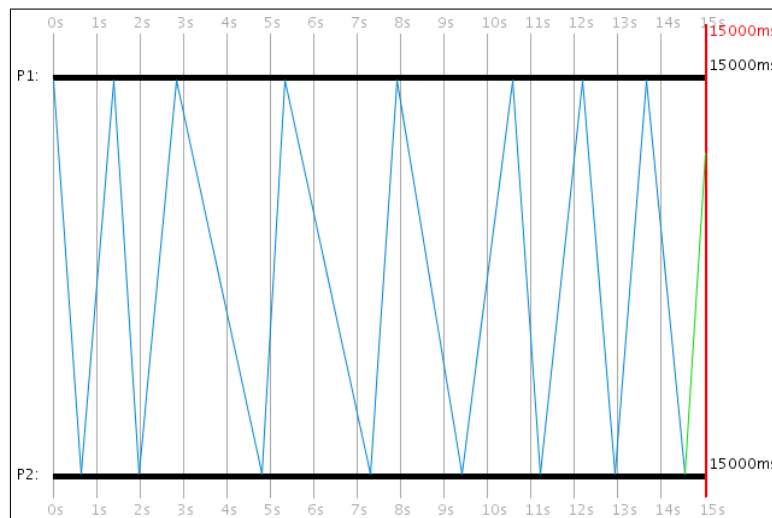


Abbildung 2.15.: Das Ping-Pong Protokoll

Bei dem Ping-Pong Protokoll (Abbildung 2.15) werden zwischen zwei Prozessen, Client P1 und Server P2, ständig Nachrichten hin- und hergeschickt. Der Ping-Pong Client startet die erste Anfrage, worauf der Server dem Client antwortet. Auf diese Antwort wird vom Client wiederum geantwortet und so weiter. Jeder Nachricht wird ein Zähler mitgeschickt, der bei jeder Station um eins inkrementiert- und jeweils im Loggfenster protokolliert wird. In der Simulation werden erst keine Antwortnachrichten mehr verschickt, wenn entweder eine Nachricht verloren geht, oder wenn die Simulationszeit das Ende erreicht hat. In Tabelle 2.3 sind alle für dieses Beispiel programmierten Ereignisse aufgeführt! Wichtig ist, dass Prozess 1

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 2.4.: Programmierte Ping-Pong Ereignisse (Sturm)

seinen Ping-Pong Client aktiviert, bevor er eine Ping-Pong Clientanfrage startet! Wenn die Eintrittszeiten für Aktivierung und das Starten der Anfrage identisch sind, so ordnet der Ereigniseditor diese Ereignisse automatisch in der richtigen Reihenfolge an. Anhand dieses Beispiels ist auch erkennbar, dass die noch nicht ausgelieferte Nachrichten noch grün eingefärbt ist. Alle ausgelieferten Nachrichten tragen bereits die Farbe Blau.

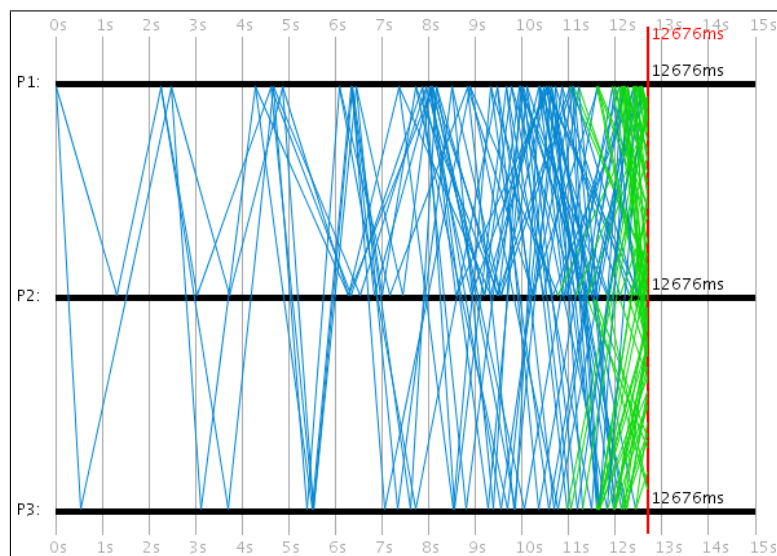


Abbildung 2.16.: Das Ping-Pong Protokoll (Sturm)

Werden die Ereignisse wie in Tabelle 2.4 abgeändert, so lässt sich ein Ping-Pong Sturm realisieren. Dort wurde ein neuer Prozess 3 eingeführt, der als Ping-Pong Server fungiert. Als Ursache verdoppelt sich die Anzahl der kursierenden Nachrichten bei jedem Client-Server Roundtrip, da auf jede Clientnachricht stets 2 Serverantworten verschickt werden. In Abbildung 2.16 ist der resultierende Simulationsverlauf dargestellt.

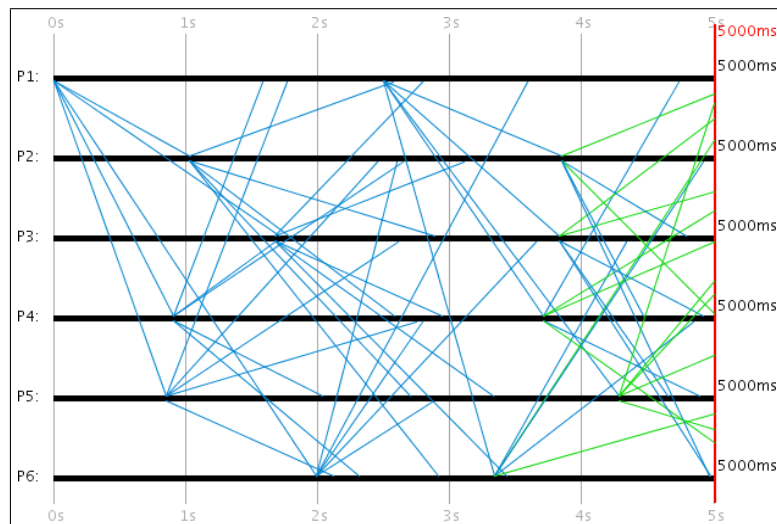


Abbildung 2.17.: Das Broadcast-Sturm Protokoll

2.5.3. Das Broadcast-Sturm Protokoll

Das Broadcast-Sturm 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-Sturm Protokoll (Server- und Clientseitig) verschickt alle erhaltenen Nachrichten, sofern sie vom jeweiligen Prozess noch nicht schon einmal verschickt wurden, erneuert. Somit lässt sich, unter Verwendung mehrerer Prozesse (hier 6), wie auf Abbildung 2.17, ein Broadcast-Sturm erzeugen. P1 ist der Client und startet je eine Anfrage nach 0ms und 2500ms. Die Simulationsdauer beträgt hier genau 5000ms. Da Client nur Servernachrichten und Server nur Clientnachrichten empfangen können, ist in dieser Simulation jeder Prozess, wie in Tabelle 2.5 angegeben, gleichzeitig Server und Client.

2.5.4. Das Protokoll zur internen Synchronisierung in einem synchronen System

Bisher haben wir uns nur mit Protokollen beschäftigt, in denen die beteiligten Prozesse keine Uhrabweichung hatten. Das Protokoll zur internen Synchronisierung ist ein Protokoll zur Synchronisierung der lokalen Prozesszeit, welches beispielsweise angewandt 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.

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

Tabelle 2.5.: Programmierte Broadcast-Sturm Ereignisse

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 2.6.: Programmierte Ereignisse zur internen Synchronisierung

Der Server schickt als Antwort seine eigene lokale Prozesszeit t_s zurück, womit der Client seine neue und genauere Prozesszeit berechnen kann. Wie genau die neue Prozesszeit berechnet wird, wird im Folgenden beschrieben.

Hier (Abbildung 2.18) stellt P1 den Client und P2 den Server dar. Da die Übertragungszeit t_u einer Nachricht angenommen zwischen t'_{min} und t'_{max} liegt, setzt der Client P1 nach Empfang der Serverantwort seine lokale Prozesszeit auf $t_c := t_s + \frac{1}{2}(t'_{min} + t'_{max})$. Somit wurde die lokale Zeit von P1, bis auf einen Fehler f mit $f < \frac{t'_{max} - t'_{min}}{2}$, synchronisiert.

Der Clientprozess hat in der Abbildung 2.18 als Uhrabweichung den Wert 0.1 und der Server hat als Uhrabweichung den Wert 0.0 konfiguriert. Der Client startet, wie in Tabelle 2.6 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 (Sekunden-Gatter) gestartet werden. Dies ist auf die Uhrabweichung von

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

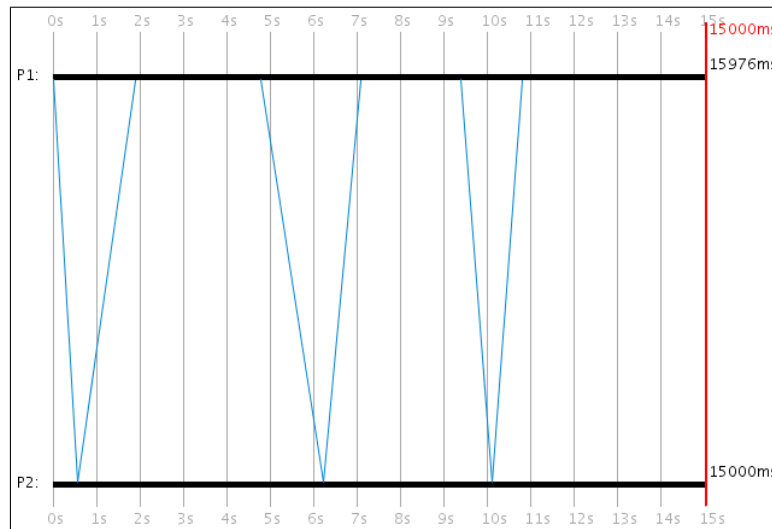


Abbildung 2.18.: Das Protokoll zur internen Synchronisierung

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) abweichen. Somit lassen sich auch Szenarien simulieren, in denen das Protokoll falsch konfiguriert wurde und die Zeitsynchronisation grössere Ungenauigkeiten aufweisen kann.

2.5.5. Christians Methode zur externen Synchronisierung

Ein weiteres Protokoll für die Synchronisierung von Uhrzeiten funktioniert nach der Christians Methode zur externen Synchronisierung. Die Christians Methode benutzt die RTT (Round

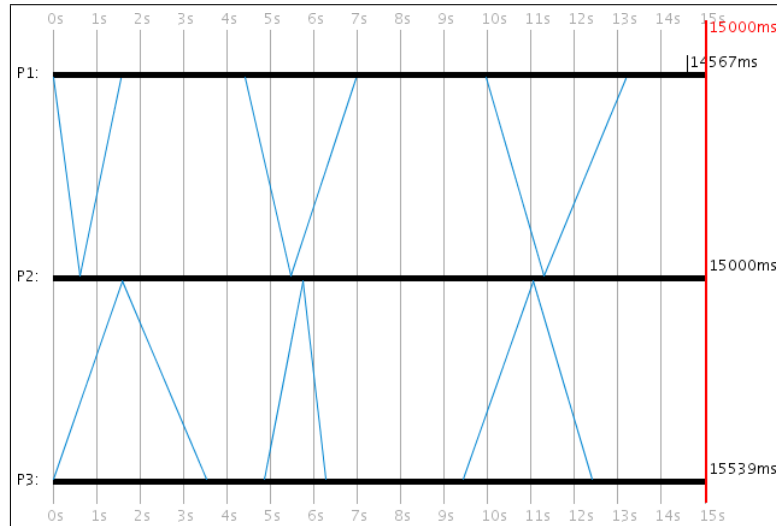


Abbildung 2.19.: Interne Synchronisierung und Christians Methode im Vergleich

Trip Zeit) t_{rtt} , um die Übertragungszeiten von einzelnen Nachrichten zu approximieren.

Wenn der Client seine lokale Zeit t_c bei einem Server synchronisieren möchte, so verschickt er eine Anfrage, und misst dabei die RTT t_{rtt} bis die Serverantwort eintrifft. Die Serverantwort beinhaltet die lokale Prozesszeit vom Server t_s 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 eine Christians-Simulation so aus wie in Abbildung 2.18, daher wird hier auf eine einfache Abbildung vom Christians-Protokoll verzichtet. Viel Interessanter ist der direkte Vergleich zwischen dem Protokoll zur internen Synchronisierung und der Christians Methode der externen Synchronisierung (Abbildung 2.19). Hier stellt P1 den Client zur internen Synchronisierung und P3 den Client zur externen Synchronisierung dar. P2 fungiert für beide Protokolle gleichzeitig als Server. P1 und P3 starten jeweils zu den lokalen Prozesszeiten 0ms, 5000ms und 10000ms eine Clientanfrage (Tabelle 2.7). P1 und P3 haben als Uhrabweichung 0.1 eingestellt und die Simulationsdauer beträgt insgesamt 15000ms.

Es ist zu ablesbar, dass 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 erneuerten Ausführung alle Nachrichten wieder eine neue zufällige Übertragungszeit haben werden, die auf das eine oder andere Protokoll schlechte oder gute Auswirkungen haben können.

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 2.7.: Programmierte Ereignisse, Vergleich interne und externe Synchronisierung

2.5.6. Der Berkeley Algorithmus zur internen Synchronisierung

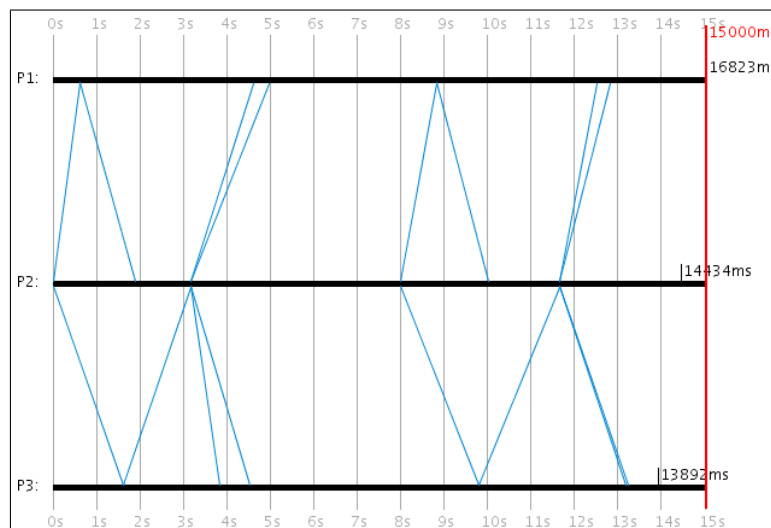


Abbildung 2.20.: Der Berkeley Algorithmus zur internen Synchronisierung

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

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 2.8.: Programmierte Ereignisse zum Berkeley Algorithmus

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 seiner eigener Zeit sowie aller Prozesszeiten. 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 stellt t'_i die derzeit aktuelle Prozesszeit des jeweiligen Clients dar. Denn bis zum Eintreffen des Korrekturwertes ist inzwischen wieder Zeit verstrichen.

In den Beispiel in Abbildung 2.20 gibt es 2 Clientprozesse P1 und P3 sowie den Serverprozess P2. Der Server startet nach jeweils 0ms und 7500ms eine Synchronisationsanfrage (Tabelle 2.8). In der Abbildung ist zu erkennen, dass der Server stets 2 Korrekturwerte verschickt, die jeweils P1 und P2 erreichen. Es werden hier also pro Synchronisierungsvorgang 4 Korrekturwerte ausgeliefert. Eine Korrekturnachricht enthält neben dem Korrekturwert k_i auch die PID des Prozesses, für den die Nachricht bestimmt ist. Ein Client verarbeitet so nur die für ihn bestimmten Korrekturwerte, indem das Protokoll die PID vorher überprüft.

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 synchronisieren soll. Wenn hier eine PID angegeben wird, die gar nicht existiert oder nicht das Berkeley Protokoll clientseitig unterstützt, funktioniert das Protokoll nicht. Dann wird ewig auf eine fehlende Clientantwort gewartet.

2.5.7. Das Ein-Phasen Commit Protokoll

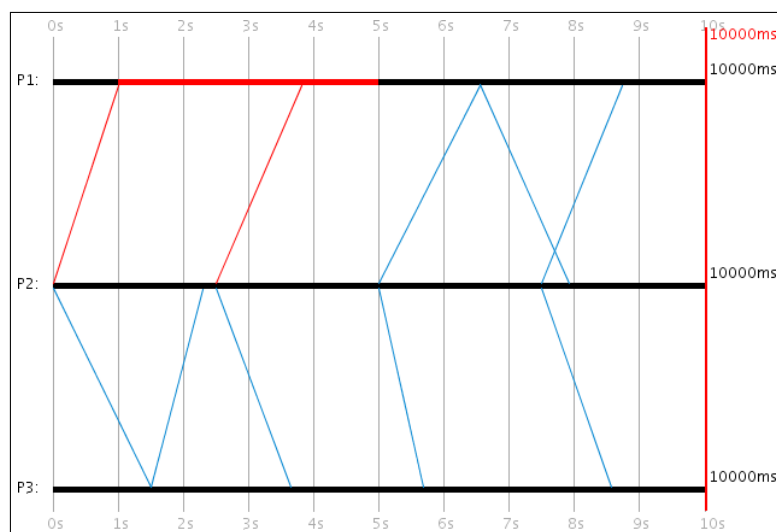


Abbildung 2.21.: Das Ein-Phasen Commit Protokoll

Das Ein-Phasen Commit Protokoll ist dafür da, 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 erneuertes Versenden des Festschreibewunsches eingestellt bekommen.

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	Prozessweiderbelebung

Tabelle 2.9.: Programmierte Ein-Phasen Commit Ereignisse

Die programmierten Ereignisse des Beispiels in Abbildung 2.21 sind in Tabelle 2.9 aufgelistet. P1 und P3 simulieren jeweils einen Client und P2 den Server. Damit die Simulation mehrere Festschreibewünsche verschickt, stürzt in der Simulation P1 nach 1000ms ab und nach 5000ms steht er wieder zur Verfügung. Die ersten beide Festschreibewünsche erreichen dadurch P1 nicht und erst der dritte Versuch verläuft erfolgreich. Bevor die Bestätigung von P1 bei P2 eintrifft, läuft jedoch der Wecker erneuert ab, so dass ein weiterer Festschreibewunsch versendet wird. Da P1 und P3 jeweils schon eine Bestätigung verschickt haben, wird dieser Wunsch ignoriert.

Protokollvariablen

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

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

2.5.8. Das Zwei-Phasen Commit Protokoll

Das Zwei-Phasen Commit Protokoll ist eine Erweiterung des Ein-Phasen Commit Protokolls. Der Server statet 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 ein Ergebnis aller Clients vorliegt. Nach Erhalt aller Abstimmungen überprüft der

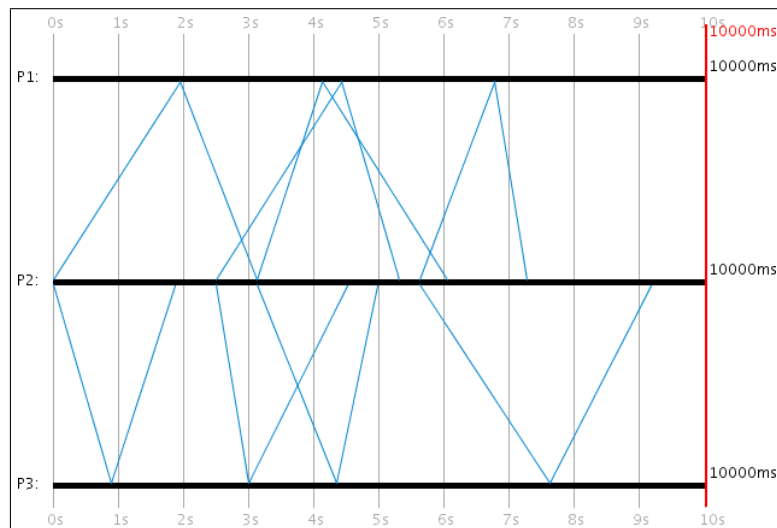


Abbildung 2.22.: Das Zwei-Phasen Commit Protokoll

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 2.10.: Programmierte Zwei-Phasen Commit Ereignisse

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 alle jedoch mit true abstimmten, soll festgeschrieben werden. Dabei wird das globale Abstimmungsergebnis true verschickt. Das globale Abstimmungsergebnis wird periodisch so oft erneuert verschickt, bis von jedem Client eine Bestätigung des Erhalts vorliegt.

In dem Beispiel (Abbildung 2.22) sind P1 und P3 Clients und P2 der Server. Der Server verschickt nach 0ms seine initiale Anfrage (Tabelle 2.10). Da diese Simulation recht unübersichtlich ist, liegen In den Tabellen 2.11 und 2.12 Auszüge dem Loggfenster vor. Hier stimmen P1 und P3 jeweils mit true, d.h. es soll festgeschrieben werden, ab. Auf die Lamport- und Vektorzeitstempel sowie die lokalen Prozesszeiten wurde in der Darstellung wegen Irrelevanz verzichtet. Da keine Uhrabweichungen konfiguriert wurden, sind die lokalen Prozesszeiten stets gleich der globalen Zeit. 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.

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 2.11.: 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 2.12.: Auszug aus der Loggausgabe des Zwei-Phasen Commit Beispiels (2)

Protokollvariablen

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

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

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

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

2.5.9. Der ungenügende (Basic) Multicast

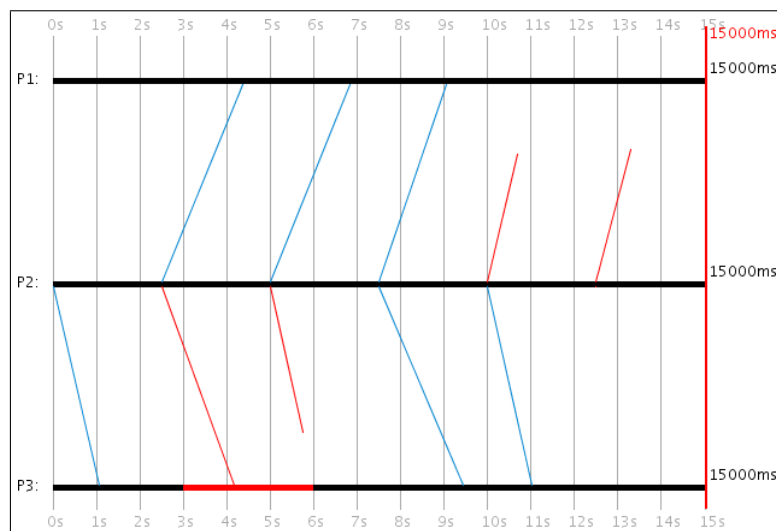


Abbildung 2.23.: Das Basic-Multicast Protokoll

Das Basic-Multicast Protokoll ist sehr einfach aufgebaut. In dem Beispiel auf [Abbildung 2.23](#) sind P1 und P3 Server und P2 der Client. Bei diesem Protokoll startet der Client immer

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 2.13.: Programmierte Basic-Multicast Ereignisse

die Anfrage, was bei diesem Protokoll eine einfache Multicast-Nachricht ist, die jeder Server empfangen kann. Wie in Tabelle 2.13 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 auch keine Nachrichten empfangen kann. 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 gesetzt, worauf alle in dieser Simulation verschickten Nachrichten mit einer Wahrscheinlichkeit von 30 Prozent ausfallen. Hier ging die 3. Multicast-Nachricht auf den Weg zu P3- und die 5. sowie 6. Nachricht auf den Weg zu P1 verloren.

2.5.10. Der zuverlässige (Reliable) Multicast

Bei dem zuverlässigen (Reliable) Multicast verschickt der Client so oft periodisch seine Multicast-Nachricht, bis er von allen beteiligten Servern eine Bestätigung erhalten hat. Nach jedem erneuten Versuch vergisst der Client, von welchen Servern er bereits eine Bestätigung erhalten hat, womit jeder erneuter Versuch von allen Teilnehmern aufs Neue bestätigt werden muss. In dem Beispiel (Abbildung 2.24, Tabelle 2.14, sowie den Loggs in den Tabellen 2.15 und 2.16) ist P2 der Multicast-verschickende Client, während P1 und P3 die Server sind. Bei 0ms initiiert der Client seine Multicast-Nachricht. Die Nachrichtenverlustwahrscheinlichkeiten sind überall auf 30 Prozent eingestellt.

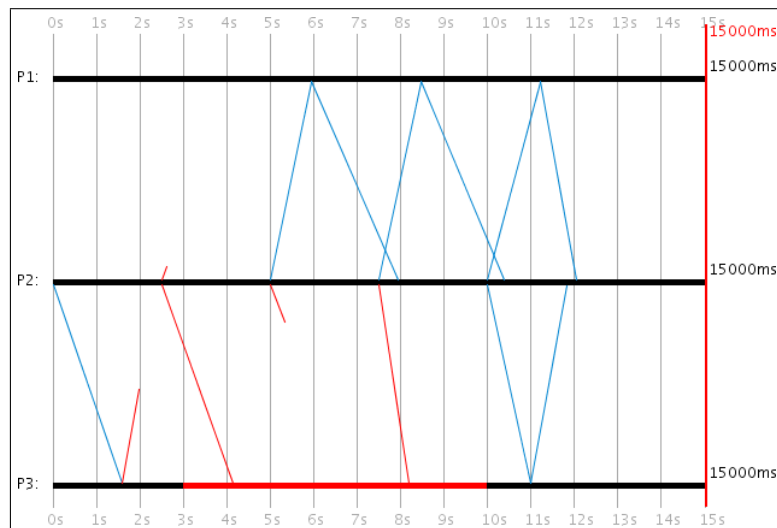


Abbildung 2.24.: Das Reliable-Multicast Protokoll

In diesem Beispiel benötigt der Client bis zur korrekten Auslieferung eines zuverlässigen Multicasts genau 5 Versuche:

1. Versuch:

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

2. Versuch:

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

3. Versuch:

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

4. Versuch:

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

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 2.14.: Programmierte Reliable-Multicast Ereignisse

- P3: Die Multicast-Nachricht erreicht P3, aber P3 ist abgestürzt und kann somit keine Nachricht verarbeiten.

5. Versuch:

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

Protokollvariablen

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

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

2.6. Weitere Beispiele

2.6.1. Vektor- und Lamportzeitstempel

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 erneuert 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 erneuert 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 2.15.: Auszug aus der Loggausgabe des Reliable-Multicast Beispiels

Zeit (ms)	PID	Loggnachricht
011213	1	ACK erneuert 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 2.16.: Auszug aus der Loggausgabe des Reliable-Multicast Beispiels (2)

Kapitel 3.

Die Implementierung

3.1. Gliederung der Pakete

3.2. Editoren

3.3. Ereignisse

3.3.1. Interne Ereignisse

3.4. Protokolle

3.4.1. Protokoll-API

3.5. Serialisierung von Simulationen

3.5.1. Rückwärtskompatibel

3.6. Programmierrichtlinien

3.7. Entwicklungsumgebung

Kapitel 4.

Ausblick

Anhang A.

Akronyms

API Application Programming Interface

GUI Graphical User Interface

NID Nachrichten-Identifikationsnummer

PID Prozess-Identifikationsnummer

RTT Round Trip Time

VS Verteiltes System

Anhang B.

Literaturverzeichnis

- [Oßm07] Martin Oßmann. Vorlesung “verteilte systeme” an der fh aachen. Vorlesung, 2007.
- [Tan03] Andrew Tanenbaum. Verteilte systeme - grundlagen und paradigmten. Buch, 2003.
2. Autor Marten van Steen; ISBN: 3-8273-7057-4.