# Analyse und Korrektur des Datenkommunikationsmodells einer zeitkontinuierlichen verteilten Anwendung mit Schwerpunkt auf Zustandskonsistenz und Fehlertoleranz

**Bachelorarbeit**

Stefano Albrecht
Betreuer: Dipl.-Inform. Christof Leng
Gutachter: Prof. Alejandro P. Buchmann, Ph. D.

TECHNISCHE UNIVERSITÄT DARMSTADT

Technische Universität Darmstadt
Fachbereich Informatik

Fachgebiet
Datenbanken und Verteilte Systeme

Analyse und Korrektur des Datenkommunikationsmodells einer zeitkontinuierlichen verteilten Anwendung mit Schwerpunkt auf Zustandskonsistenz und Fehlertoleranz

Bachelorarbeit

Eingereicht von Stefano Albrecht
Tag der Einreichung: 30.03.2010

Betreuer: Dipl.-Inform. Christof Leng
Gutachter: Prof. Alejandro P. Buchmann, Ph. D.

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Datenbanken und Verteilte Systeme

## Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 30.03.2010 Stefano Albrecht

## Danksagung

An dieser Stelle möchte ich mich bei Herrn Prof. Alejandro Buchmann dafür bedanken, dass er offen für meine Ideen war und es mir ermöglicht hat, diese in Form einer Bachelorarbeit umzusetzen. Meinem Betreuer Christof Leng danke ich für eine Reihe anregender Diskussionen und außerdem dafür, dass er mich diese Arbeit frei und nach eigenem Ermessen hat erstellen lassen. Abschließend bedanke ich mich bei Herrn Andreas John, der mir den Testserver für die Evaluation zur Verfügung gestellt hat.

## Inhaltsverzeichnis

## Abbildungsverzeichnis

## Tabellenverzeichnis

# 1 Einleitung

Netzwerktechnologien, insbesondere die verwendeten Übertragungsmedien, verbessern sich zusehends von Jahr zu Jahr. Im Zuge der immer besser werdenden Verfügbarkeit und Zuverlässigkeit von Netzwerken erhöht sich auch die Anzahl sogenannter verteilter Anwendungen. Eine *verteilte Anwendung* ist ein Programm, das mehrfach instantiiert auf mehreren Systemen läuft, wobei die Systeme über ein Netzwerk miteinander verbunden sind. Jede Instanz verwaltet ihren eigenen *Zustand*. Eine zentrale Aufgabe von verteilten Anwendung besteht darin, diese Zustände zu synchronisieren, d.h. sie untereinander *konsistent* zu halten. Zwei Zustände sind genau dann konsistent, wenn eine bestimmte Menge von Informationen dieser Zustände identisch oder äquivalent im Sinne einer gegebenen Äquivalenzrelation ist. Konsistenz ist demnach anwendungsspezifisch. Betrachten wir ein verteiltes Schachspiel mit zwei Instanzen, so sind die Zustände beider Instanzen genau dann konsistent, wenn die Menge der Spielfiguren und der zugehörigen Positionen identisch ist. Die Zustände zweier Instanzen einer Flugsimulation dagegen könnten bereits konsistent sein, wenn die Menge der Flugzeuge und der jeweiligen Positionen, Geschwindigkeiten, Beschleunigungen und Flugrichtungen eine maximale Abweichung nicht überschreiten. Eine verteilte Anwendung wird schließlich als *zustandskonsistent* bezeichnet, wenn die Menge der Zustände aller ihrer Instanzen immer konsistent ist.

Verteilte Anwendungen werden typischerweise in *zeitdiskrete* und *zeitkontinuierliche* Anwendungen unterteilt. Instanzen einer zeitdiskreten Anwendung ändern ihren Zustand ausschließlich als Antwort auf Benutzerinteraktionen. Instanzen einer zeitkontinuierlichen Anwendung ändern ihren Zustand zusätzlich im Laufe der Zeit, d.h. auch ohne Benutzerinteraktion. Das verteilte Schachspiel ist eine zeitdiskrete Anwendung, weil sich der Zustand der Instanzen nur als Antwort auf einen Spielzug ändert. Die verteilte Flugsimulation hingegen ist eine zeitkontinuierliche Anwendung, weil sich z.B. die Positionen der Flugzeuge mit der Zeit ändern. Offensichtlich ist Zustandskonsistenz in zeitkontinuierlichen Anwendungen inhärent schwieriger als in zeitdiskreten Anwendungen, weil Mechanismen zur Wahrung der Konsistenz nicht nur auf Benutzerinteraktionen, sondern auch auf Änderungen des Zustands aufgrund der Zeit reagieren müssen. Verfahren zum Erhalt der Konsistenz in zeitkontinuierlichen Anwendungen unterscheiden sich daher von denjenigen in zeitdiskreten Anwendungen.

Ein weitere Aufgabe von verteilten Anwendungen ist das Behandeln von Fehlern, wobei Fehler im hiesigen Kontext Ausfälle von Systemen oder verspätete bzw. verlorene Nachrichten bezeichnen. Eine Anwendung, die solche Fehler derart behandelt, dass die betroffenen Instanzen der Anwendung weiter funktionieren können (idealerweise so, als wäre der Fehler niemals passiert), bezeichnen wir im Rahmen dieser Arbeit als *fehlertolerant*. Da alle Verfahren zur Wahrung der Zustandskonsistenz versagen, wenn hierzu relevante Nachrichten verloren gehen, ist Fehlertoleranz eine wichtige und wünschenswerte Eigenschaft in verteilten Anwendungen.

Inwiefern sind Zustandskonsistenz und Fehlertoleranz relevante Fragestellungen? Ersteres lässt sich intuitiv sehr einfach erklären. Zustandskonsistenz muss gewährleistet werden, da die einzelnen Instanzen der Anwendung anderenfalls ein Eigenleben entwickeln, d.h. jede Instanz hat einen anderen Zustand. Ohne Zustandskonsistenz werden die Zustandsveränderungen einer Instanz bei den anderen Instanzen nicht oder nicht korrekt sichtbar. Wenn sich ein Benutzer einer verteilten Anwendung aufgrund des ihm vorliegenden Zustands für eine bestimmte Aktion entscheidet, so erwartet er, dass die Aktion in allen anderen Instanzen der Anwendung genau so wie vom ihm intendiert ausgeführt wird. Wenn die Aktion des Benutzers nun direkt an die anderen Nutzer gesendet wird, könnte es bei einem inkonsistenten Zustand dazu kommen, dass die Aktion bei den anderen Nutzern nicht ausgeführt werden kann, weil

der (inkonsistente) Zustand des ausführenden Nutzers erforderlich ist. Falls andererseits nur die Auswirkungen der Aktion des Benutzers propagiert werden, so erscheint den anderen Nutzern womöglich ein unlogisches Artefakt, weil die propagierten Auswirkungen nicht kompatibel zu den jeweiligen Zuständen sind. Fehlertoleranz, insbesondere das Behandeln von Nachrichtenausfällen, ist wichtig, weil verlorene Nachrichten in den meisten Fällen zu Inkonsistenzen führen. Insbesondere diejenigen Nachrichten, die zur Wahrung der Konsistenz dienen, müssen verlässlich transportiert werden.

Zwei konkrete Beispiele sollen die Wichtigkeit der beiden Eigenschaften weiter verdeutlichen:

**Computerspieleindustrie** Mit über 1,56 Milliarden Euro Umsatz allein in Deutschland im Jahr 2009 [1] ist die Computerspieleindustrie eine der umsatzstärksten Branchen der Softwareindustrie. Ein beträchtlicher Anteil der heutzutage gebotenen Computerspiele bietet neben dem Einzelspielermodus einen netzwerkbasierten Mehrspielermodus. Einige Spiele sind sogar vollständig als Netzwerkspiel ausgelegt. Allen gemein ist, dass das Spiel nur dann richtig funktionieren kann, wenn die Zustände der Spielinstanzen (d.h. der Teilnehmer am Spiel) konsistent sind. Sind sie es nicht, so wird der Spielspaß geschmälert, da kein korrekter Spielfluss zustande kommt. Eine unmittelbare Folge wäre ein finanzieller Verlust, den es natürlich zu verhindern gilt. Es müssen also Maßnahmen getroffen werden, um Zustandskonsistenz und Fehlertoleranz zu gewährleisten.

**Verteilte Simulationen** Komplexe Simulationen sind oftmals nur dann durchführbar, wenn die Berechnungen auf mehrere Systeme verteilt werden. Eine solche Simulationsanwendung bezeichnet man als verteilte Simulation. Beispiele hierfür sind komplexe Flugsimulationen, Straßenverkehrssimulationen und Schlachtfeldsimulationen. Die Zustände der Simulationsinstanzen müssen konsistent gehalten werden, da das Simulationsergebnis anderenfalls verfälscht werden könnte. Spätestens dann, wenn Simulationsergebnisse Grundlage für Entscheidungen mit Konsequenzen sind, muss die Korrektheit der Ergebnisse gewährleistet sein. Entscheidungen, die aufgrund falscher Simulationsergebnisse getätigt werden, können den Verlust von großen Geldmengen zur Folge haben, und im schlimmsten Fall sogar das Leben von Menschen gefährden.

Im Fokus dieser Arbeit steht das Datenkommunikationsmodell einer zeitkontinuierlichen verteilten Anwendung. Es handelt sich hierbei um das Spiel *Saga Online*. Das Modell wird zunächst auf Zustandskonsistenz überprüft, wobei die genaue Bedeutung der Konsistenz formal definiert wird. Die Analyse wird zeigen, dass das Modell in seiner ursprünglichen Form das Kriterium der Zustandskonsistenz nicht erfüllt. Das Modell wird daraufhin um Verfahren ergänzt, die Zustandskonsistenz gewährleisten sollen. Im Anschluss daran wird das Modell hinsichtlich Fehlertoleranz überprüft, wobei auch hier Schwächen offenbart werden. Es wird ein Verfahren zur Behandlung von verspäteten und ausgefallenen Nachrichten vorgestellt. Die neuen Verfahren werden schließlich evaluiert.

Der Rest dieser Arbeit gliedert sich wie folgt:

*Kapitel 2* beschreibt Saga Online und gibt grundlegende Details zur Architektur und Implementierung des Spiels. *Kapitel 3* führt Zustandskonsistenz als formalen Begriff ein und überprüft das Datenkommunikationsmodell von Saga Online darauf hin. Das Modell wird anschließend um Verfahren ergänzt, die Konsistenz sicherstellen sollen. *Kapitel 4* führt den Begriff der Fehlertoleranz ein und überprüft das Modell von Saga Online diesbezüglich. Es wird ein Verfahren vorgestellt, das verspätete und ausgefallene Nachrichten behandelt. *Kapitel 5* enthält die Evaluation der Verfahren. *Kapitel 6* fasst die Arbeit schließlich kurz zusammen, während *Kapitel 7* einen Ausblick auf weitere Arbeiten gibt.

# 2 Saga Online

## 2.1 Allgemeines

*Saga Online*[1] ist ein netzbasiertes Mehrspielerspiel. Die Entwicklungen an dem Spiel haben Ende 2006 begonnen und schreiten seitdem in unregelmäßigen Abständen voran. Entwurf und Umsetzung des Spiels geschehen allein durch den Autor dieser Arbeit. Das Spiel wurde zum Zeitpunkt der Erstellung dieser Arbeit noch nicht veröffentlicht.

Das Spiel basiert auf einer Client-Server-Architektur. Clients (d.h. die Spieler) sind mit einem Server verbunden, der die Zustände der Clients synchronisiert (d.h. konsistent hält). Der Server selbst ist mit einem Hauptserver verbunden, der eine Liste mit allen registrierten Servern unterhält. Clients können diese Liste vom Hauptserver beziehen und sich von dort aus mit einem Server verbinden. Weitere Informationen zur Architektur des Spiels finden sich in Abschnitt 2.4.

Saga Online ist eine Java-Anwendung. Zum Ausführen des Spiels wird eine Java Virtual Machine in der Version 1.6 oder höher benötigt. Diese kann auf den Webseiten der Firma Sun herunter geladen werden.[2] Ältere Versionen sind möglicherweise kompatibel, werden jedoch nicht getestet. Informationen zur Implementierung des Spiels finden sich in Abschnitt 2.5.

## 2.2 Spielphasen

Der Spielablauf von Saga Online lässt sich in vier Phasen unterteilen:

**Login** Die Login-Phase ist die initiale Phase des Spiels. Darin wird der Benutzer aufgefordert, sich mit seinem Nutzernamen und Passwort anzumelden. Bei erfolgreicher Anmeldung wird die Login-Phase verlassen, sie kann fortan nicht mehr erreicht werden.

**Serverliste** Nachdem sich der Benutzer angemeldet hat, bekommt er eine Liste von verfügbaren Spielservern angezeigt. Der Benutzer sucht sich hieraus einen Server aus und verbindet sich mit diesem, womit er die Serverliste wieder verlässt.

**Serverlobby** Sobald sich der Benutzer mit einem Server verbunden hat, gelangt er in dessen Serverlobby. Darin können Einstellungen an der Spielfigur vorgenommen werden. Andere Spieler in derselben Lobby werden angezeigt und können über einen Chat kommunizieren. Der Benutzer kann das Spiel nun starten oder zur Serverliste zurückkehren.

**Spiel** Wurde das Spiel gestartet, so tritt die Spiel-Phase ein. Hier wird das Spielgeschehen angezeigt und ausgetragen. Der Benutzer kann die Spiel-Phase zu jeder Zeit verlassen und zurück zur Serverlobby oder direkt zur Serverliste gelangen.

Abbildung 2.1 zeigt die Beziehungen zwischen den vier Phasen. Phasenübergänge werden durch Pfeile dargestellt, wobei die Login-Phase die initiale Phase ist. Die gestrichelte Linie soll verdeutlichen, dass die Login-Phase nicht mehr erreicht werde kann, sobald sie verlassen wurde.

[1] Saga ist ein Akronym für *Stefano Albrecht's Game*. Der Titel ist auf Englisch zu lesen, der Apostroph ist also beabsichtigt.
[2] Siehe *http://java.sun.com* und *http://java.sun.com/javase/downloads* für weitere Informationen und Downloads zu Java.

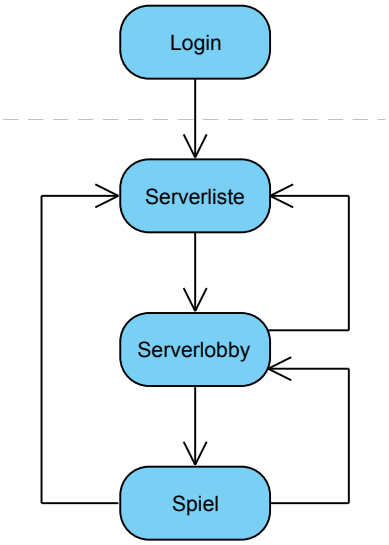


**Abbildung 2.1:** Spielphasen in Saga Online

## 2.3 Bildschirmfotos

Um sich eine konkrete Vorstellung von den vier Spielphasen und damit vom gesamten Spiel verschaffen zu können, werden nachfolgend Bildschirmfotos zu allen Phasen dargestellt. Jede Phase wird anhand der Bildschirmfotos genauer erläutert.


**Abbildung 2.2:** Bildschirmfoto von Phase 1 (Login)

In der Login-Phase (Abbildung 2.2) wird der Benutzer aufgefordert, sich mit seinem Nutzernamen und Passwort anzumelden. Die Anmeldung erfolgt hierbei beim Rootserver (siehe Abschnitt 2.4). Die Zugangsdaten können optional in eine lokale Datei gespeichert werden, so dass der Anmeldevorgang künftig verkürzt wird. Das Programm informiert außerdem über Falscheingaben, bei drei oder mehr aufeinanderfolgenden Falscheingaben erfolgt eine zeitlich befristete Anmeldesperre. Mehrfachanmeldungen eines Benutzers werden vom Rootserver unterbunden.


**Abbildung 2.3:** Bildschirmfoto von Phase 2 (Serverliste)

Die Serverliste (Abbildung 2.3) zeigt alle beim Rootserver registrierten Gameserver (siehe Abschnitt 2.4). Die Liste kann nach vorgegebenen Kriterien sortiert werden, z.B. Servername oder Spieleranzahl. Durch Klick auf *Aktualisieren* wird die Liste erneut vom Rootserver bezogen. Durch Klick auf *Schnell aktualisieren* werden die in der Liste enthaltenen Server direkt überprüft (d.h. neue Server werden nicht angezeigt). Möchte man sich mit einem Server verbinden, so muss der entsprechende Listeneintrag markiert und auf *Verbinden* geklickt werden.


**Abbildung 2.4:** Bildschirmfoto von Phase 3 (Serverlobby)

Sobald man mit einem Server verbunden ist, gelangt man in dessen Lobby (Abbildung 2.4). Zu sehen ist ein Bereich mit allgemeinen Informationen zum Spiel und eine verkleinerte Karte der Spielwelt. Andere Spieler in derselben Lobby werden in einer Liste angezeigt und können über den Chat miteinander kommunizieren. Über der Spielerliste können Einstellungen zur Spielfigur vorgenommen werden. Bei Klick auf *Anmelden* registriert sich der Benutzer beim Server. Die Schaltfläche ändert sich dann zu *Start*, über welche man dem Spiel beitreten kann. Ein Klick auf *Aktualisieren* erneuert die in der Serverlobby angezeigten Informationen. Möchte der Benutzer zurück zur Serverliste, so klickt er auf *Zurück*.


**Abbildung 2.5:** Bildschirmfoto von Phase 4 (Spiel)

Die Spiel-Phase (Abbildung 2.5) ist die Hauptphase des Spiels. Darin wird der Spielverlauf aus Sicht der Benutzers angezeigt. Oben rechts sind die Statuswerte des Spielers und die zur Verfügung stehenden Fähigkeiten zu sehen. Über die kleinen Symbole am rechten Bildschirmrand lassen sich weitere Menüs öffnen, beispielsweise eine Minikarte zur Spielwelt oder ein Fähigkeitenbaum, in dem der Spieler neue Fähigkeiten erlernen kann. Der Spieler navigiert seine Spielfigur mit Maus und Tastatur durch die Spielwelt, die Tastenbelegung lässt sich frei konfigurieren.

## 2.4 Architektur

Die Architektur von Saga Online basiert auf dem Client-Server-Modell. Die Zustände der Clients werden über einen zentralen Server synchronisiert, indem dieser die Nachrichten der Clients untereinander verteilt (vgl. *Broadcasting*). Eine besondere Eigenschaft des Servers ist, dass die von den Clients empfangenen Nachrichten nicht interpretiert, sondern lediglich weitergeleitet werden. Der Server verwaltet somit keinen Zustand des Spiels, wodurch er weniger Ressourcen in Anspruch nimmt.

Die Architektur von Saga Online besteht aus drei Komponenten:

**Gameclient** Der Gameclient stellt den Client im Client-Server-Modell dar. Ein Benutzer meldet sich über den Gameclient zunächst beim Rootserver an. Von diesem erhält er anschließend eine Liste mit registrierten Gameservern. Der Benutzer kann sich nun mit einem der Gameserver verbinden und am Spiel des Servers teilnehmen. Alle Nachrichten des Clients gehen fortan nur noch an diesen Server, der sie wiederum an die anderen Clients weiterleitet.

**Gameserver** Der Gameserver stellt den Server im Client-Server-Modell dar. Ein Server korrespondiert hierbei zu einer Spielwelt. Clients melden sich beim Server an, um am Spiel teilnehmen zu können. Empfängt der Server eine Nachricht von einem Client, so verteilt er diese an alle anderen Clients. Um Zustandskonsistenz und Fehlertoleranz zu gewährleisten, muss der Server einige Maßnahmen vorkehren. Dies ist Hauptbestandteil dieser Arbeit (siehe Kapitel 3 und 4).

**Rootserver** Der Rootserver ist hierarchisch ganz oben angesiedelt. Eine seiner beiden Hauptaufgaben ist das Verwalten der Benutzerprofile. Die andere Hauptaufgabe besteht im Verwalten einer Serverliste. Die Serverliste enthält alle aktuell registrierten Gameserver, mit denen sich Benutzer verbinden können. Um in diese Liste aufgenommen zu werden, müssen sich die Gameserver beim Rootserver registrieren. Benutzerprofile und Serverliste werden somit zentral organisiert.

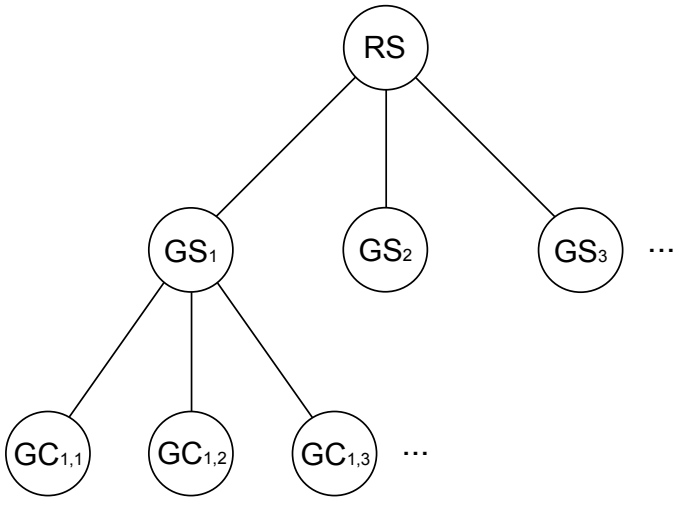


**Abbildung 2.6:** Komponenten-Hierarchie in Saga Online

Abbildung 2.6 zeigt die Hierarchie der zuvor beschriebenen Komponenten. Der Rootserver (RS) ist ganz oben in der Hierarchie angesiedelt. Die zweite Ebene der Hierarchie setzt sich aus den Gameservern (GS) zusammen, die jeweils mit dem Rootserver verbunden sind. Ebene drei der Hierarchie beinhaltet alle Gameclients (GC), die mit einem Gameserver verbunden sind (zu erkennen an den Indizes).

## 2.5 Implementierung

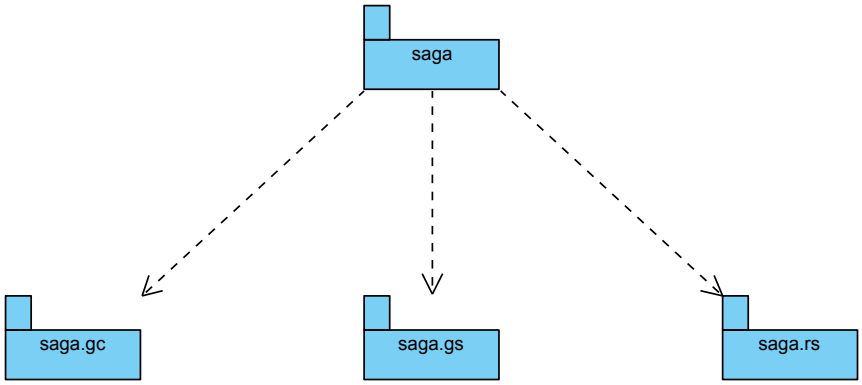


**Abbildung 2.7:** Paket-Hierarchie der Implementierung von Saga Online

Saga Online basiert auf Java. Zum Ausführen des Spiels wird eine Java Virtual Machine in der Version 1.6 oder höher benötigt. Ältere Versionen sind möglicherweise kompatibel, werden jedoch nicht getestet. Abbildung 2.7 zeigt einen Ausschnitt der Paket-Hierarchie des Spiels. Sie ist angelehnt an die drei Komponenten aus Abschnitt 2.4. Das Paket `saga` ist Grundlage des gesamten Projekts und teilt sich in weitere Unterpakete auf, unter anderem die Pakete `saga.gc`, `saga.gs` und `saga.rs`:

**`saga.gc`** Dieses Paket beinhaltet alle diejenigen Klassen, die in ihrer Gesamtheit den *Gameclient* bilden.
⇒ Hauptklasse ist die Klasse `saga.gc.GameClient`.

**`saga.gs`** Dieses Paket beinhaltet alle diejenigen Klassen, die in ihrer Gesamtheit den *Gameserver* bilden.
⇒ Hauptklasse ist die Klasse `saga.gs.GameServer`.

**`saga.rs`** Dieses Paket beinhaltet alle diejenigen Klassen, die in ihrer Gesamtheit den *Rootserver* bilden.
⇒ Hauptklasse ist die Klasse `saga.rs.RootServer`.

Jede Hauptklasse verfügt über eine eigene `main`-Methode und ist somit ausführbar. Die genannten Pakete sind isoliert voneinander, d.h. es existieren keine Querverweise zwischen den Paketen. Solche Klassen, die wiederverwendet werden sollen, sind im Paket `saga.global` ausgelagert. Saga Online greift weiterhin auf die Klassen zweier anderer Projekte zu, die parallel zum Spiel entstanden sind:

**Universal Utilities** Dieses Projekt entstand als eine Ansammlung verschiedenster Klassen, die allgemein und unabhängig von Saga Online zu gebrauchen sind. Beispiele hierfür sind Datenstrukturen für Bytefelder (hier verwendet für Netzkommunikation), Klassen zur Datenbankanbindung (hier verwendet für MySQL[3]) und Klassen zum Datentransport in Netzwerken (UDP und TCP, ebenfalls im Spiel verwendet). Das Projekt bietet viele weitere nützliche Klassen.

**Dynamic Desktop Engine** Die *Dynamic Desktop Engine* ($D^2E$) ist eine Erweiterung für AWT[4] und Swing[5]. Mit ihr werden GUI-Komponenten (z.B. Frames oder Panels) um die Fähigkeit ergänzt, grafische Objekte (z.B. Bilder) und eingebettete GUI-Komponenten (z.B. Buttons und Textfelder) dynamisch zu manipulieren. Beispiele sind Filteroperationen für Ein- und Ausblendung von Bildern oder Rotation von Panels. Mithilfe dieses Projekts werden die Fenster in Saga Online realisiert.

[3] http://www.mysql.com
[4] http://java.sun.com/javase/6/docs/technotes/guides/awt
[5] http://java.sun.com/javase/6/docs/technotes/guides/swing

# 3 Zustandskonsistenz

## 3.1 Einleitung

In diesem Kapitel wird das Datenkommunikationsmodell von Saga Online hinsichtlich Zustandskonsistenz überprüft. Zu diesem Zweck wird die Bedeutung der Konsistenz formal definiert. Die Analyse des Modells wird zeigen, dass es in seiner ursprünglichen Form keine Zustandskonsistenz gewährleistet. Das Modell wird daraufhin um Verfahren ergänzt, die Zustandskonsistenz erzielen sollen. Eine abschließende Analyse wird zeigen, dass das so veränderte Modell dem Kriterium der Konsistenz gerecht wird.

Die Wichtigkeit der Zustandskonsistenz wurde bereits in Kapitel 1 diskutiert. Verfahren zum Erhalt der Zustandskonsistenz sind notwendig, um die Zustände der Instanzen einer verteilten Anwendung identisch oder äquivalent zu halten. Inkonsistente Zustände verursachen in der Regel Berechnungsfehler und können sogar zu Ausfällen führen. Eines der genannten Beispiele war die verteilte Simulation, bei der inkonsistente Zustände zu falschen Simulationsergebnissen führen können.

Zustandskonsistenz in einem zeitkontinuierlichen verteilten Spiel wie Saga Online ist ein besonders wichtiges Qualitätsmerkmal. Inkonsistente Zustände werden vom Spieler meist dadurch entdeckt, dass Mitspieler offensichtlich unlogische Aktionen durchführen, wie etwa ein Angriff ins Leere oder ausbleibende Reaktionen. Besonders tragisch wird es, wenn ein Angriff eines Spielers aus dessen Sicht erfolgreich ist, aus Sicht des Getroffenen jedoch nicht, da die Zustände voneinander abweichen. Hat der vermeintlich erfolgreiche Angriff zum virtuellen Tod der getroffenen Figur geführt, so wird sich der Angreifer besonders darüber freuen, während der Getroffene in seiner Instanz des Spiels unbeschadet weiter spielt. Dies alles schmälert den Spielspaß, im schlimmsten Fall macht es das Spiel sogar ungenießbar. Zustandskonsistenz ist somit eine wichtige Eigenschaft für das Spiel.

Das Kapitel ist wie folgt aufgebaut:

Abschnitt 3.2 enthält eine formale Definition von Konsistenz, die fortan Grundlage für die Analyse sein wird. Abschnitt 3.3 analysiert das Datenkommunikationsmodell von Saga Online hinsichtlich des Konsistenzkriteriums. In Abschnitt 3.4 werden Verfahren zur Zustandskonsistenz eingeführt. Abschnitt 3.5 schließt das Kapitel mit einer Postanalyse des Modells ab.

## 3.2 Definition von Konsistenz

Die Bedeutung der Konsistenz wurde bisher nur informell gegeben. Für eine formale Analyse des Modells muss der Begriff jedoch formal definiert werden. Es werden zwei Varianten der Konsistenz definiert, das *schwache Konsistenzkriterium* und das *starke Konsistenzkriterium*. Beide Kriterien beziehen sich auf zeitkontinuierliche verteilte Anwendungen. Die Bedeutung der Zustandskonsistenz ist hierbei gegeben durch das starke Konsistenzkriterium. Die Definitionen in diesem Abschnitt sind angelehnt an [9].

Sei $A$ eine zeitkontinuierliche verteilte Anwendung. Mit $I(A)$ bezeichnen wir die Menge aller Instanzen von $A$. Den Zustand einer Instanz $i \in I(A)$ zum Zeitpunkt $t \in T$ bezeichnen wir mit $s_{i,t}$, wobei $T \subseteq \mathbb{R}_0^+$ die Menge aller betrachteten Zeitpunkte ist. Sei weiterhin $O(A)$ die Menge aller Operationen, die von Instanzen von $A$ ausgeführt werden. Ein Element $o_{i,t^o,t^*} \in O(A)$ bezeichnet eine Operation, die von Instanz

$i$ zum Zeitpunkt $t^o \in T$ initiiert und zum Zeitpunkt $t^* \in T$ $(t^* \geq t^o)$ ausgeführt wird.

Wir definieren nun die Funktion $R_A : I(A) \times T \times O(A) \rightarrow \{true, false\}$ als

$R_A(i, t, o_{j,t^o,t^*}) = true$ falls $o_{j,t^o,t^*}$ von Instanz $i$ zum Zeitpunkt $\hat{t} \leq t$ empfangen wurde, sonst $false$.

$R_A$ gibt also Auskunft darüber, ob eine Operation von einer bestimmten Instanz zu einem gegebenen Zeitpunkt bereits empfangen wurde. Die Funktion wird für die nachfolgenden Definitionen benötigt.

### 3.2.1 Schwache Konsistenz

Eine zeitkontinuierliche verteilte Anwendung $A$ erfüllt das schwache Konsistenzkriterium (auch *schwache Konsistenz*) genau dann, wenn folgender Ausdruck wahr ist:

$$\forall t \in T \ \forall i, j \in I(A) : \left[\forall o_{k,t^o,t^*} \in O(A) : t^* \leq t \Rightarrow R_A(i, t, o_{k,t^o,t^*}) \wedge R_A(j, t, o_{k,t^o,t^*})\right] \Rightarrow s_{i,t} = s_{j,t}$$

Der Ausdruck besagt, dass zu jedem Zeitpunkt $t$ die Zustände zweier Instanzen $i$ und $j$ gleich sein müssen (d.h. $s_{i,t} = s_{j,t}$), wenn alle Operationen $o_{k,t^o,t^*}$, die noch vor $t$ ausgeführt werden sollen (d.h. $t^* \leq t$), rechtzeitig von diesen Instanzen empfangen wurden (d.h. $R_A(i, t, o_{k,t^o,t^*})$ und $R_A(j, t, o_{k,t^o,t^*})$ sind wahr).

Zu beachten ist, dass das Kriterium nicht durch verspätete oder verlorene Nachrichten verletzt wird. Es sagt nichts über die Zustände zweier Instanzen zu einem bestimmten Zeitpunkt aus, wenn die zuvor auszuführenden Operationen nicht oder nicht rechtzeitig von beiden Instanzen empfangen wurden. Dies ist insofern wichtig, als dass keine verteilte Anwendung jemals Konsistenz gewährleisten könnte, wenn man die zeitlichen Verzögerungen in einem Netzwerk miteinbezieht (die prinzipiell unbeschränkt sind).

### 3.2.2 Starke Konsistenz

Das schwache Konsistenzkriterium ist insofern schwach, als dass es lediglich das Potential für korrekte Zustände hergibt, nicht jedoch die Korrektheit selbst. Ein Zustand $s_{i,t}$ ist hierbei *korrekt* genau dann, wenn alle Operationen $o_{j,t^o,t^*}$ mit $t^* \leq t$ rechtzeitig von $i$ empfangen (d.h. $R_A(i, t^*, o_{j,t^o,t^*})$ ist wahr) und zu ihren intendierten Ausführungszeitpunkten $t^*$ ausgeführt wurden. Eine verteilte Anwendung, die das schwache Konsistenzkriterium erfüllt, muss nicht notwendigerweise auch die Korrektheit aller Zustände gewährleisten. Wir führen hierzu das starke Konsistenzkriterium (auch *starke Konsistenz*) ein. Zu ihrer formalen Definition benötigen wir den Begriff der virtuellen perfekten Instanz.

Die *virtuelle perfekte Instanz* $P$ einer zeitkontinuierlichen verteilten Anwendung $A$ zeichnet sich durch folgende Eigenschaften aus:

- $\forall o_{i,t^o,t^*} \in O(A) : R_A(P, t^*, o_{i,t^o,t^*})$ (d.h. $P$ empfängt alle Operationen rechtzeitig)
- $P$ führt alle Operationen $o_{i,t^o,t^*}$ zum Zeitpunkt $t^*$ aus (d.h. wie ursprünglich intendiert)

$P$ hat also immer denjenigen Zustand, den eine zu $A$ identische nichtverteilte Anwendung mit denselben Operationen hätte. Dies entspricht unserer Vorstellung von Korrektheit, die wir nun wie folgt mithilfe der starken Konsistenz ausdrücken.

Eine zeitkontinuierliche verteilte Anwendung $A$ erfüllt das starke Konsistenzkriterium genau dann, wenn folgender Ausdruck wahr ist:

$$\forall t \in T\ \forall i \in I(A) : \left[\forall o_{k,t^o,t^*} \in O(A) : t^* \leq t \Rightarrow R_A(i,t,o_{k,t^o,t^*})\right] \Rightarrow s_{i,t} = s_{P,t}$$

Der Ausdruck besagt, dass zu jedem Zeitpunkt $t$ der Zustand einer Instanz $i$ gleich demjenigen der virtuellen perfekten Instanz $P$ sein muss (d.h. $s_{i,t} = s_{P,t}$), wenn alle Operationen $o_{k,t^o,t^*}$, die noch vor $t$ ausgeführt werden sollen (d.h. $t^* \leq t$), rechtzeitig von der Instanz $i$ empfangen wurden (d.h. $R_A(i,t,o_{k,t^o,t^*})$ ist wahr). Man kann sich leicht davon überzeugen, dass das starke Konsistenzkriterium die schwache Konsistenz impliziert. Letzteres wird lediglich um den Korrektheitsbegriff erweitert.

Eine verteilte Anwendung, die das starke Konsistenzkriterium erfüllt, bezeichnen wir im Rahmen dieser Arbeit als *zustandskonsistent*. Wenn wir das Datenkommunikationsmodell von Saga Online formal analysieren, so beziehen wir uns zu diesem Zweck auf das starke Konsistenzkriterium.

### 3.2.3 Beispiel

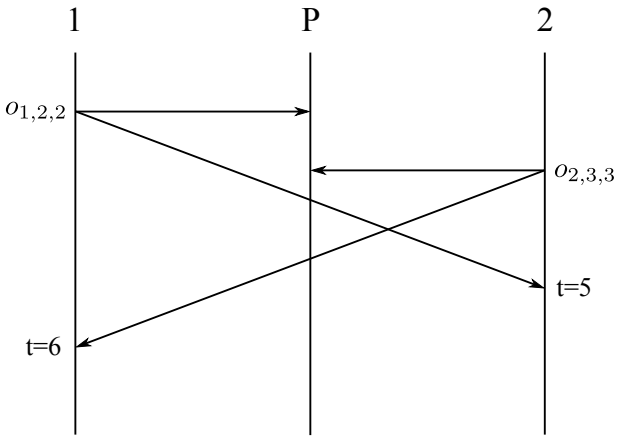


**Abbildung 3.1:** Schwache und starke Konsistenz

Abbildung 3.1 veranschaulicht die Bedeutung der schwachen und starken Konsistenz. Zu sehen sind zwei Instanzen 1 und 2 sowie die virtuelle perfekte Instanz $P$ einer zeitkontinuierlichen verteilten Anwendung. Die senkrechten Linien unter den Instanzen stellen perfekt synchronisierte Zeitachsen dar.[1]

Instanz 1 initiiert eine Operation zum Zeitpunkt 2 und führt sie noch im selben Zeitpunkt aus. Die Operation wird erst zum Zeitpunkt 5 von Instanz 2 empfangen. Die Verzögerung entsteht, weil der Transport der Nachricht einige Zeit benötigt. Instanz 2 initiiert eine Operation zum Zeitpunkt 3 und führt sie ebenfalls sofort aus. Die Operation wird von Instanz 1 zum Zeitpunkt 6 empfangen. Beide Operationen werden ohne Verzögerungen an $P$ übertragen und korrekt im Sinne der starken Konsistenz ausgeführt.

Die betrachtete verteilte Anwendung erfüllt das schwache Konsistenzkriterium, wenn die Zustände der Instanzen 1 und 2 für die Zeitpunkte $t < 2$ und $t \geq 6$ gleich sind. Sie erfüllt darüber hinaus das starke Konsistenzkriterium, wenn der Zustand von Instanz 1 für die Zeitpunkte $t < 3$ und $t \geq 6$, und derjenige

[1] Eine perfekte Synchronisation der Uhren ist nicht möglich und im Sinne der Konsistenzkriterien auch nicht notwendig. Die Uhren der Instanzen werden als perfekt synchronisiert angenommen, um die Zeiten der Operationen direkt miteinander vergleichen zu können.

von Instanz 2 für die Zeitpunkte $t < 2$ und $t \geq 5$ gleich dem Zustand von $P$ ist. Das Beispiel verdeutlicht, dass das starke Konsistenzkriterium die schwache Konsistenz impliziert.

## 3.3 Analyse des Modells

In diesem Abschnitt soll das Datenkommunikationsmodell der zeitkontinuierlichen verteilten Anwendung Saga Online (siehe Kapitel 2) hinsichtlich Zustandskonsistenz untersucht werden. Zustandskonsistenz entspricht hierbei der starken Konsistenz, d.h. das Modell ist zustandskonsistent genau dann, wenn es das starke Konsistenzkriterium erfüllt (siehe hierzu Abschnitt 3.2.2). Für allgemeine Informationen zur Architektur des Spiels, siehe Kapitel 2, Abschnitt 2.4.

Die Analyse des Modells geschieht auf formale Weise, wir verwenden daher die Notation aus den vorhergehenden Abschnitten. Die zugrundeliegende verteilte Anwendung ist das Spiel Saga Online, das fortan mit $SO$ bezeichnet werden soll. Zunächst einmal soll untersucht werden, wie der Konsistenzmechanismus in der ursprünglichen Form des Modells funktioniert. Jede Instanz $i \in I(SO)$ ist mit einem zentralen Server $S$ verbunden. Instanzen korrespondieren hierbei zu Spielern und sollen daher synonym gebraucht werden. Sobald eine Instanz $i$ eine Operation $o_{i,t^o,t^*} \in O(SO)$ initiiert, wird diese sofort ausgeführt (d.h. es gilt $t^o = t^*$) und anschließend an den Server gesendet, so dass dieser die Operation an die restlichen Instanzen der Anwendung verteilen kann (auch als *Broadcasting* bezeichnet). Empfängt eine Instanz $i$ eine Operation $o_{j,t^o,t^*}$, so führt sie diese sofort aus, d.h. noch im selben Zeitpunkt. Modelle dieser Art nennt man *eventbasiert*, wobei ein Event das Eintreffen einer Operation bezeichnet.

Die Navigation der Spielfiguren ist die häufigste Operation im gesamten Spiel. Würde man jede Bewegung als einzelne Operation übermitteln, so wäre das Netzwerk binnen kürzester Zeit überlastet. Um dies zu umgehen, die Zustände jedoch trotzdem konsistent zu halten, verwendet das Modell ein Verfahren namens *Dead-Reckoning*. Informationen zum Verfahren finden sich in Abschnitt 3.3.2.

### 3.3.1 Beweis

Es soll nun gezeigt werden, dass $SO$ nicht zustandskonsistent ist, d.h. es erfüllt das starke Konsistenzkriterium nicht. Der Beweis soll durch einen Widerspruch geführt werden, indem gezeigt wird, dass die Negation des Kriteriums erfüllbar ist. Die Negation der starken Zustandskonsistenz lautet wie folgt:

$$\exists t \in T\ \exists i \in I(A) : \left[\forall o_{k,t^o,t^*} \in O(A) : t^* \leq t \Rightarrow R_A(i,t,o_{k,t^o,t^*})\right] \wedge s_{i,t} \neq s_{P,t}$$

Es ist $A = SO$ und $P$ die virtuelle perfekte Instanz von $SO$. Wir betrachten nun ein Szenario in $SO$ mit den Instanzen 1 und 2. Instanz 1 initiiert eine Operation $o_{1,t^*,t^*}$, d.h. Initialisierung und Ausführung der Operation finden zum selben Zeitpunkt statt. Die Operation wird anschließend an den Server $S$ und von dort zur Instanz 2 gesendet, welche die Operation zu einem Zeitpunkt $\hat{t} > t^*$ empfängt. Der Abstand von $t^*$ zu $\hat{t}$ ist hierbei beliebig groß (und positiv).

Das linke Konjunktionsglied der obigen Formel ist erfüllt für $t = \hat{t}$ und $i = 2$. Die Frage ist nun, ob auch das rechte Glied erfüllbar ist, ob also der Zustand von Instanz 2 zum Zeitpunkt $\hat{t}$ ungleich demjenigen der perfekten Instanz $P$ zum Zeitpunkt $\hat{t}$ sein kann (d.h. ob $s_{2,\hat{t}} \neq s_{P,\hat{t}}$ wahr sein kann). Bei genauerer Betrachtung fällt auf, dass $P$ in diesem Szenario identisch zur Instanz 1 ist. Es gilt $R_{SO}(1,t^*,o_{1,t^*,t^*}) = true$ und die Operation wird von Instanz 1 zum Zeitpunkt $t^*$ ausgeführt. Dies entspricht den Eigenschaften der perfekten Instanz. Die Frage lässt sich also darauf reduzieren, ob die Zustände der Instanzen 1 und 2 zum Zeitpunkt $\hat{t}$ ungleich sein können (d.h. ob $s_{1,\hat{t}} \neq s_{2,\hat{t}}$ wahr sein kann). Hierfür ist wichtig, dass $SO$ eine zeitkontinuierliche Anwendung ist, d.h. Zustände können sich zu jeder Zeit verändern. Dies kann zur

Folge haben, dass die Operation $o_{1,t^*,t^*}$ zum Zeitpunkt $\hat{t}$ nicht mehr ordnungsgemäß ausgeführt werden kann oder zum falschen Ergebnis führt. Beispielsweise könnte die Operation eine Fähigkeit des Spielers 1 sein, die den Zustand $s_{1,t^*}$ voraussetzt. Es ist möglich, dass die Zustände $s_{1,t^*}$ und $s_{2,\hat{t}}$ derart voneinander abweichen, dass die Operation zum Zeitpunkt $\hat{t}$ nicht in Instanz 2 ausgeführt werden kann, wodurch $s_{1,\hat{t}} \neq s_{2,\hat{t}}$ folgen könnte. Insbesondere ist $s_{1,\hat{t}} = s_{2,\hat{t}}$ nicht allgemeingültig, womit $s_{1,\hat{t}} \neq s_{2,\hat{t}}$ und somit die ganze Formel erfüllbar ist. Das starke Konsistenzkriterium wird folglich *nicht* von $SO$ erfüllt.

□

### 3.3.2 Dead-Reckoning

Dead-Reckoning ist ein weit verbreitetes Verfahren für Konsistenz in zeitkontinuierlichen verteilten Anwendungen. Typische Anwendungen, die das Verfahren einsetzen, sind Flug-, Verkehrs-, und Schlachtfeldsimulationen, aber auch Netzwerkspiele, wie z.B. Saga Online und andere. Eine der wohl häufigsten Operationen in den genannten Anwendungen ist die Navigation bzw. Bewegung von Entitäten, wobei letzteres die Flugzeuge in der Flugsimulation oder die Spielfiguren im Netzwerkspiel bezeichnet. Würde man jede Positionsänderung einzeln als Operation übermitteln, so wäre das Netzwerk schnell überfordert. An dieser Stelle setzt das Verfahren an. Anstatt jede Positionsänderung einzeln zu übertragen, werden in periodischen oder wohldefinierten Abständen Positionsnachrichten übermittelt. Die Nachrichten enthalten außerdem Informationen, die zur Fortbewegung der Entitäten benötigt werden, wie beispielsweise Bewegungsrichtung, Geschwindigkeit und Beschleunigung. Mithilfe dieser Informationen kann die Entität selbstständig fortbewegt werden, so dass nicht jede Positionsänderung separat übermittelt werden muss. Die periodischen Positionsnachrichten verhindern, dass die Positionen derselben Entität in den verschiedenen Instanzen der Anwendung zu weit voneinander abweichen.

Saga Online nutzt Dead-Reckoning für die Fortbewegung der Spielfiguren. Sobald ein Spieler eine Bewegungstaste drückt, wird dies als Operation mitsamt der aktuellen Position der Spielfigur an die anderen Instanzen weitergeleitet. Die Spielfigur wird fortan solange in die entsprechende Richtung fortbewegt, bis der Spieler die Taste wieder loslässt. Letzteres wird ebenfalls als Operation mit der aktuellen Position der Spielfigur übermittelt. Jeder Spieler sendet zusätzlich periodische Positionsnachrichten, um zu verhindern, dass die Positionen derselben Figur in den Instanzen zu weit voneinander abdriften.

Dead-Reckoning ist kein neues Verfahren. Eine der ältesten Anwendungen, die das Verfahren implementieren, ist das Spiel *Amaze* [2]. Mit dem US-amerikanischem Simulationsnetzwerk *SIMNET* folgte schnell die erste große kommerzielle Implementierung [10]. SIMNET war die seinerzeit größte verteilte Schlachtfeldsimulation und erlaubte bis zu 15 Teilnehmer in einer Simulation. Mit *DIS* (Distributed Interactive Simulation) folgte schließlich der erste IEEE-Standard [7]. Der Standard wurde für größere Simulationen entworfen und erlaubt mehrere hundert Teilnehmer in einer Sitzung.

## 3.4 Korrektur des Modells

In Abschnitt 3.3 wurde gezeigt, dass das starke Konsistenzkriterium von der ursprünglichen Version des Datenkommunikationsmodells in Saga Online *nicht* erfüllt wird. In diesem Abschnitt sollen daher Verfahren eingeführt werden, die speziell zum Erhalt der Zustandskonsistenz entworfen wurden. Genauer sind dies ein Verfahren zur zeitlichen Ordnung von Operationen *nach* Empfang beim Server, und eines zur zeitlichen Ordnung von Operationen *vor* Empfang beim Server. Da beide Verfahren global synchronisierte Uhren voraussetzen, wird zunächst ein Verfahren zur Uhren-Synchronisation eingeführt. In Abschnitt 3.5 wird gezeigt, dass das derart modifizierte Modell das Kriterium der starken Konsistenz erfüllt.

### 3.4.1 Uhren-Synchronisation

Das Datenkommunikationsmodell von Saga Online sieht in seiner ursprünglichen Form keine Synchronisaton der Uhren vor. Die in den Folgeabschnitten beschriebenen Verfahren zur Wahrung der Zustandskonsistenz gehen jedoch von synchronisierten Uhren aus. In diesem Abschnitt soll daher zunächst ein Verfahren zur Synchronisation der Uhren aller Instanzen eingeführt werden. Es handelt sich hierbei um den *Algorithmus von Cristian* [4].

Gegeben sind ein Client $C$ (die Instanz) und ein Server $S$. Mit $T_C$ bezeichnen wir die Uhrzeit von $C$, mit $T_S$ diejenige von $S$. Wir gehen davon aus, dass die Uhren eine ausreichend große Auflösung besitzen, d.h. eine Auflösung im Millisekundenbereich, falls Sekunden gemessen werden, und eine Auflösung im Nanosekundenbereich, falls Millisekunden gemessen werden. Ziel des Verfahrens ist es nun, die Uhren von $C$ und $S$ zu synchronisieren, d.h. es gilt $T_C = T_S + \epsilon$ mit möglichst kleinem Fehler $\epsilon$.[2]

Das Verfahren funktioniert wie folgt:

$C$ sendet zum Zeitpunkt $t_0$ eine Zeitanfrage an $S$. $S$ empfängt die Anfrage zum Zeitpunkt $t_1$, die Zeit für den Hinweg der Anfrage beträgt somit $t_1 - t_0$. $S$ bestimmt nun seine Zeit $T_S$ und sendet sie zum Zeitpunkt $t_2$ an $C$. $C$ empfängt die Antwort zum Zeitpunkt $t_3$, die Zeit für den Rückweg der Antwort beträgt somit $t_3 - t_2$. Die Zeit von $t_0$ nach $t_3$ bezeichnet man als *Round Trip Time* (RTT). Da wir die Zeit von $t_1$ nach $t_2$ nicht kennen, nehmen wir sie als 0 an, womit sich $RTT = (t_1 - t_0) + (t_3 - t_2)$ ergibt. $C$ setzt seine Zeit abschließend auf $T_C = T_S + RTT/2$.

Jedes Netzwerk hat eine minimale Zeit für den Transport von Nachrichten. Diese Zeit sei mit $t_{min}$ bezeichnet und es gilt $RTT \geq 2 * t_{min}$. Der früheste Zeitpunkt, an dem $S$ seine Zeit $T_S$ bestimmten kann, ist $t_0 + t_{min}$. Da wir von $t_1 = t_2$ ausgehen, muss die Zeit $T_S$ von $S$ im Bereich $T_S + t_{min}$ bis $T_S + RTT - t_{min}$ liegen, wenn $C$ die Antwort von $S$ erhält. Diese Zeitspanne hat eine Breite von $RTT - 2 * t_{min}$, womit wir den durchschnittlichen Fehler $\epsilon = RTT/2 - t_{min}$ erhalten. Üblicherweise ist die Wahrscheinlichkeit für kleinere Verzögerungen im Netzwerk höher als für größere Verzögerungen. Das Verfahren macht sich diesen Umstand zunutze, indem es mehrfache Zeitanfragen an $S$ stellt und diejenige Zeit $T_S$ verwendet, für die RTT am kleinsten ist (das Verfahren wird daher als *probabilistisch* bezeichnet). Idealerweise ist $RTT = 2 * t_{min}$, womit $\epsilon = 0$ gilt. Das Verfahren wird anschaulich in Abbildung 3.2 dargestellt.

Saga Online implementiert das Verfahren als Dienst aufseiten des Gameservers (siehe Kapitel 2, Abschnitt 2.4). Gameclients können ihre Uhren über diesen Dienst untereinander synchronisieren, indem sie alle die Zeit vom Gameserver übernehmen. Da die Synchronisation als Dienst angeboten wird, kann sie zu jeder Zeit und insbesondere unabhängig vom Spiel ausgeführt werden.

Unglücklicherweise ist eine einmalige Synchronisation nicht ausreichend, um alle Uhren über das gesamte Spiel hinweg synchron zu halten. Dies ist dadurch bedingt, dass die Uhren von der darunterliegenden Hardware betrieben werden. Die Hardware selbst läuft mit einer bestimmten Frequenz, die zwischen den jeweiligen Instanzen zum Teil stark voneinander abweichen kann. Dies hat zur Folge, dass die Uhren unterschiedlich schnell ticken, weswegen anfänglich synchrone Uhren nach einer Weile nicht mehr synchron sind. Um dieses Problem zu umgehen, werden die Uhren im Spiel periodisch synchronisiert. In der Praxis hat sich hierfür eine Periode von 20 Sekunden etabliert. Dies soll gleiche Messungen im Millisekundenbereich gewährleisten.

[2] Eine perfekte Synchronisation, d.h. $\epsilon = 0$, ist wegen Verzögerungen im Netzwerk nicht möglich.

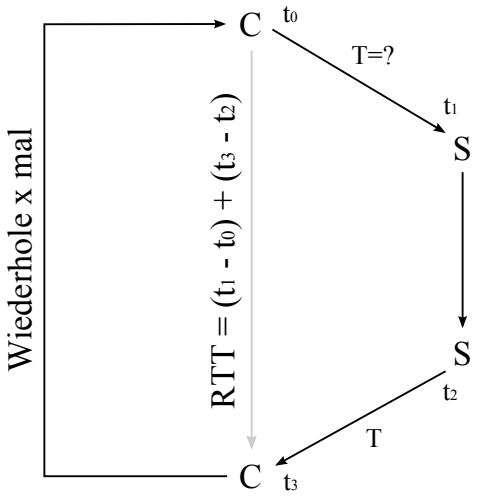


**Abbildung 3.2:** Probabilistische Uhren-Synchronisation

### 3.4.2 Zeitliche Ordnung von Operationen nach Empfang bei Server

In diesem Abschnitt wird ein Verfahren eingeführt, dass die zeitliche Ordnung von Operationen *nach* Empfang beim Server gewährleisten soll. Genauer bedeutet dies, dass alle Operationen $o_{i,t^o,t^*} \in O(SO)$, die von einer Instanz $i \in I(SO)$ zu einem Zeitpunkt $t^o \in T$ initiiert wurden, zu einem für alle Instanzen einheitlichen Zeitpunkt $t^* \in T$ ausgeführt werden, wobei $t^*$ relativ zur Empfangszeit $\hat{t}$ der Operation beim Server gewählt wird. In Abschnitt 3.4.3 wird hingegen ein Verfahren eingeführt, dass die zeitliche Ordnung von Operationen *vor* Empfang beim Server garantieren soll.

Wir definieren zunächst zwei Begriffe:

Sei $A$ eine zeitkontinuierliche verteilte Anwendung. Wir sagen, dass die Operation $o_{j,t^o,t^*} \in O(A)$ eine *Kurzzeitinkonsistenz* in $A$ verursacht, wenn es eine Instanz $i \in I(A)$ gibt, so dass $R_A(i, t^*, o_{j,t^o,t^*}) = false$ gilt. Der Begriff Kurzzeitinkonsistenz rechtfertigt sich dadurch, dass ein Zeitpunkt $\tilde{t} > t^*$ angenommen wird, für den $R_A(i, \tilde{t}, o_{j,t^o,t^*}) = true$ gilt[3]. Die Zeitspanne von $t^*$ bis $\tilde{t}$ wird hierbei als kurz angesehen. Wir bezeichnen weiterhin die Zeit $t^* - t^o$ als *Reaktionszeit* der Operation.

Kurzzeitinkonsistenzen sind idealerweise nicht existent. Falls sie dennoch auftreten, so sollen sie möglichst kurz sein, d.h. der Abstand zwischen $t^*$ und $\tilde{t}$ ist möglichst klein. Reaktionszeiten sollen ebenfalls möglichst kurz sein, so dass Benutzer das Resultat ihrer Aktionen schnell erkennen und dementsprechend weiterhandeln können. Dies gilt insbesondere für Netzwerkspiele, in denen sich zu lange Reaktionszeiten als starkes Ruckeln äußern. Idealerweise gilt also $t^* - t^o = 0$ für eine Operation $o_{i,t^o,t^*}$. Die beiden Idealzustände sind widersprüchlich in dem Sinne, als dass niedrigere Reaktionszeiten potentiell mehr bzw. längere Kurzzeitinkonistenzen verursachen und das Verhindern von Kurzzeitinkonsistenzen längere Reaktionszeiten erfordert. Es muss also ein Kompromiss zwischen ihnen gefunden werden.

Abbildung 3.3 zeigt eine Situation in der ursprünglichen Form des Modells von Saga Online. Instanz 1 initiiert eine Operation $o_{1,t^*,t^*}$ und führt sie noch im selben Zeitpunkt $t^*$ aus (d.h. Reaktionszeit ist 0). Die Operation wird anschließend an den Server $S$ und von dort zur Instanz 2 gesendet. Letztere empfängt die Operation zu einem Zeitpunkt $t > t^*$, weswegen $R_{SO}(2, t^*, o_{1,t^*,t^*}) = false$ ist. Die Operation verursacht somit eine Kurzzeitinkonsistenz in $SO$ (genauer in Instanz 2).

[3] Der Fall, in dem kein solches $\tilde{t}$ existiert (d.h. die Operation kommt niemals an), wird in Kapitel 4 behandelt.

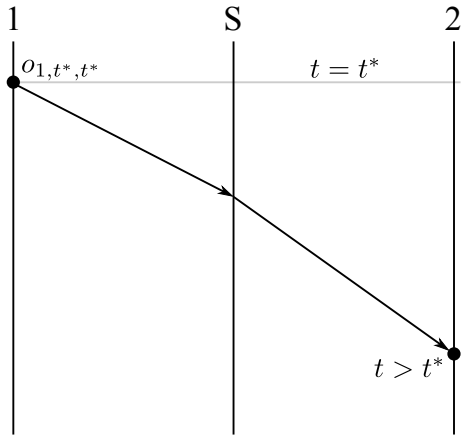


**Abbildung 3.3:** Kurzzeitinkonsistenz wegen sofortiger Reaktion

Wir führen nun ein Verfahren ein, das den genannten Kompromiss finden soll. Es nutzt das Missverhältnis zwischen Kurzzeitinkonsistenz und Reaktionszeit, indem es die Reaktionszeit soweit erhöht, bis Kurzeitinkonsistenzen unwahrscheinlich geworden und ansonsten möglichst kurz sind, falls sie dennoch auftreten. Die Reaktionszeit bleibt im Rahmen des Zumutbaren, sofern die Verzögerungen im Netzwerk nicht zu hoch sind (siehe hierzu Kapitel 5, Abschnitt 5.3).

Das Verfahren funktioniert wie folgt:

Eine Operation $o_{i,t^o,t^*}$, die von einer Instanz $i$ zum Zeitpunkt $t^o$ initiiert wurde, wird fortan nicht mehr sofort ausgeführt (d.h. es gilt $t^o \neq t^*$). Die Operation wird ohne Umwege an den Server $S$ gesendet, der sie zum Zeitpunkt $\hat{t}$ empfängt. Man beachte, dass die Uhren der Instanzen und des Servers synchronisiert sind (siehe Abschnitt 3.4.1). $S$ setzt den Zeitpunkt $t^*$ nun derart, dass die Wahrscheinlichkeit für den rechtzeitigen Erhalt der Operation (d.h. noch vor $t^*$) bei allen Instanzen möglichst hoch ist, die Reaktionszeit jedoch möglichst klein bleibt. Der Zeitpunkt $t^*$ wird demnach so gewählt, dass der Ausdruck $\forall j \in I(A) : R_A(j, t^*, o_{i,t^o,t^*})$ mit hoher Wahrscheinlichkeit wahr ist, zusammen mit einer möglichst kurzen Reaktionszeit. Um dies zu erreichen, wird $t^* = \hat{t} + t_{lat}$ gesetzt, wobei $t_{lat}$ gleich der durchschnittlichen maximalen Latenzzeit[4] aller Instanzen ist. $S$ sendet die Operation anschließend an alle Instanzen. Sobald die Operation von einer Instanz empfangen wird, wartet diese solange, bis der Ausführungszeitpunkt $t^*$ erreicht ist, und führt die Operation anschließend aus.

Der Vorgang wird in Abbildung 3.4 dargestellt. Instanz 1 initiiert eine Operation zum Zeitpunkt $t^o$ (markiert durch leeren Kreis), und sendet die Operation anschließend zum Server $S$, der sie zum Zeitpunkt $\hat{t}$ empfängt. $S$ setzt den Ausführungszeitpunkt $t^*$ der Operation auf $\hat{t} + t_{lat}$ und sendet sie an die Instanzen 1 und 2. Diese empfangen die Operation zu unterschiedlichen Zeitpunkten, jedoch vor dem Ausführungszeitpunkt. Beide Instanzen warten, bis dieser Zeitpunkt erreicht ist (gekennzeichnet durch graue Linie), und führen die Operation dann gleichzeitig aus (markiert durch ausgefüllte Kreise).

Eine alternative Wahl für den Wert von $t_{lat}$ ist die durchschnittliche minimale Latenzzeit aller Instanzen oder der Gesamtdurchschnitt aller Latenzzeiten. Wir setzen $t_{lat}$ auf die durchschnittliche maximale Latenzzeit aller Instanzen, da dies die Wahrscheinlichkeit für den rechtzeitigen Erhalt aller Operationen im Vergleich zu den anderen Wahlen maximiert. Zwar erhöht dies ebenso die Reaktionszeit aller Operationen, in der Praxis befinden sich die tatsächlichen Reaktionszeiten jedoch in einem Bereich, der vom Menschen nicht oder kaum bemerkbar ist (siehe hierzu die Evaluation in Kapitel 5). Die alternativen

[4] Als *Latenzzeit* bezeichnen wir die Zeit, die eine Nachricht von einem Rechner zu einem anderen benötigt.

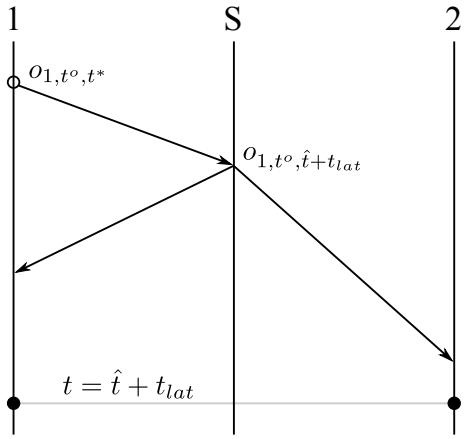


**Abbildung 3.4:** Übertragung einer Operation mit Verzögerung durch Server

Wahlen sind insofern vorteilhaft, als dass sie die Reaktionszeiten minimieren. Unsere Wahl von $t_{lat}$ stellt ein Optimum dar, da sich die Reaktionszeiten in einem Bereich befinden, der nicht oder nur kaum bemerkbar ist, die Wahrscheinlichkeit für den rechtzeitigen Erhalt aller Operationen jedoch maximiert ist.

Das Verfahren kann Kurzzeitinkonsistenzen *nicht verhindern*. Es wird nach wie vor Latenzzeiten geben, die über dem durchschnittlichen Maximalwert liegen. In einem solchen Fall kann es sein, dass der Zustand der betroffenen Instanz repariert werden muss. Ein Verfahren hierfür wird in Kapitel 4 beschrieben.

Die Latenzzeiten für eine Internetanwendung rangieren typischerweise zwischen 20 (national) und 150 (international) Millisekunden (ms). Bedingt durch die geringe Auflösung der menschlichen Wahrnehmung sind Reaktionszeiten von bis zu 100 ms nicht oder nur selten bemerkbar (siehe u.a. [14], [12] und [3]). Die Evaluation in Kapitel 5 zeigt, dass auch für Saga Online Reaktionszeiten von bis zu 100 ms nicht oder kaum bemerkbar sind. Reaktionszeiten von bis zu 150 ms liegen im Rahmen des Erträglichen.

Es sei erwähnt, dass das Verfahren keinesfalls neu ist. Eine ausführliche Beschreibung des Verfahrens ist in [9] zu finden. Es wird dort als *Local-Lag* bezeichnet, wobei sich der Name auf die künstlich herbeigeführte Verzögerung (en. *Lag*) durch $t_{lat}$ bezieht. Ein anderes Beispiel für den Einsatz des Verfahrens ist das Internetspiel *MiMaze* von Diot und Gautier [5]. Darin wird ein konstanter Wert für $t_{lat}$ verwendet, wohingegen unser Ansatz einen Kompromiss zwischen Kurzzeitinkonsistenz und Reaktionszeit findet.

### 3.4.3 Zeitliche Ordnung von Operationen vor Empfang bei Server

Während das Verfahren in Abschnitt 3.4.2 die zeitliche Ordnung von Operationen *nach* Empfang beim Server regelt, soll das in diesem Abschnitt eingeführte Verfahren die zeitliche Ordnung von Operationen *vor* Empfang beim Server regeln. Genauer soll das Verfahren gewährleisten, dass alle Operationen in ihrer initiierten Reihenfolge ausgeführt werden. Wir werden sehen, dass dieses Ziel mit Ausnahme eines Sonderfalls erreicht wird, der jedoch unkritisch im Sinne der Zustandskonsistenz ist.

Für die Definitionen in diesem Abschnitt benötigen wir zunächst eine Zuordnung von Operationen zu Empfangszeiten. Sei $A$ eine zeitkontinuierliche verteilte Anwendung und $S$ der Server, der alle Instanzen in $I(A)$ synchronisiert. Die Funktion $\pi_S : O(A) \rightarrow T$ ordnet allen Operationen $o_{i,t^o,t^*} \in O(A)$ denjenigen Zeitpunkt $\hat{t} \in T$ zu, an dem die Operation von $S$ empfangen wurde. Es gilt $\pi_S(o_{i,t^o,t^*}) \rightarrow \infty$, falls die Operation $o_{i,t^o,t^*}$ niemals von $S$ empfangen wird. Wegen der Verzögerungen im Netzwerk gilt zudem immer $\forall o_{i,t^o,t^*} \in O(A) : \pi_S(o_{i,t^o,t^*}) > t^o$ (alle Uhren sind synchronisiert).

Wir führen nun den Begriff des Aggregats ein:

Ein *ungeordnetes Aggregat* ist ein Tripel $A_k = (t_k, l_k, O_k)$, wobei $t_k \in T$ die Startzeit des Aggregats und $l_k > 0$ die Länge oder Dauer des Aggregats ist. $O_k$ bezeichnet die Menge aller Operationen des Aggregats und ist definiert als $O_k := \left\{ o_{i,t^o,t^*} \in O(A) \mid t_k \leq \pi_S(o_{i,t^o,t^*}) < t_k + l_k \right\}$. Wir gehen hier von *uniformen* Aggregaten aus, d.h. es existiert eine gemeinsame Länge $l$ so dass $\forall A_k : l_k = l$ gilt. Außerdem dürfen sich die Aggregate nicht überlappen, d.h. es gilt $\forall A_k : t_k + l_k = t_{k+1}$. Die Folge $(\alpha_k)_{k\in\mathbb{N}}, \alpha_k \mapsto A_k$ ist die Folge der ungeordneten Aggregate.

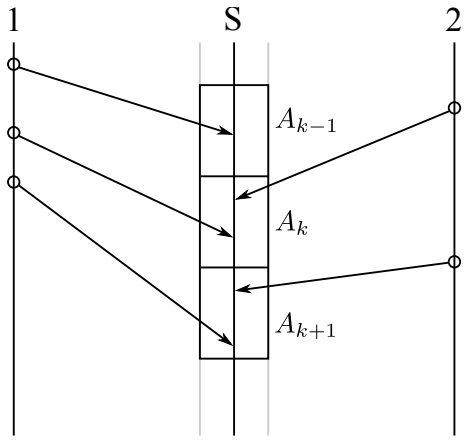


**Abbildung 3.5:** Operationen in ungeordneten Aggregaten

Abbildung 3.5 zeigt die Situation einer verteilten Anwendung mit Server $S$ und den beiden Instanzen 1 und 2. $S$ implementiert hierbei die Folge $(\alpha_k)_{k\in\mathbb{N}}$ der ungeordneten Aggregate. Das Aggregat $A_{k-1}$ beinhaltet eine Operation von Instanz 1. Die Aggregate $A_k$ und $A_{k+1}$ beinhalten jeweils eine Operation von beiden Instanzen, wobei die Reihenfolge der Operationen nicht berücksichtigt wird.

Der Begriff wird nun um eine zeitliche Ordnung ergänzt:

Ein *geordnetes Aggregat* ist ein Paar $A_k^\Delta = (A_k, \Delta_k)$, wobei $A_k$ ein ungeordnetes Aggregat ist. $\Delta_k$ ist eine Relation auf den Operationen in $O_k$ und wird definiert als $\Delta_k := \left\{ (o_{i,t_i^o,t_i^*}, o_{j,t_j^o,t_j^*}) \in O_k^2 \mid t_i^o < t_j^o \right\}$, mit der Semantik einer zeitlichen Ordnung. Wir gehen davon aus, dass es keine zwei Operationen $o_{i,t_i^o,t_i^*}$ und $o_{j,t_j^o,t_j^*}$ mit $t_i^o = t_j^o$ gibt.[5] Die Folge $(\alpha_k^\Delta)_{k\in\mathbb{N}}, \alpha_k^\Delta \mapsto A_k^\Delta$ ist die Folge der geordneten Aggregate.

Abbildung 3.6 zeigt die Situation einer verteilten Anwendung mit Server $S$ und den beiden Instanzen 1 und 2. $S$ implementiert hierbei die Folge $(\alpha_k^\Delta)_{k\in\mathbb{N}}$ der geordneten Aggregate. Instanz 1 initiiert eine Operation zum Zeitpunkt $t_1$. Die Operation wird von $S$ zu einem Zeitpunkt $\hat{t}_1$ empfangen. Instanz 2 initiiert ebenfalls eine Operation zu einem Zeitpunkt $t_2 > t_1$. Die Operation wird von $S$ zu einem Zeitpunkt $\hat{t}_2 < \hat{t}_1$ empfangen, womit die zeitliche Ordnung im Aggregat $A_k^\Delta$ verletzt wird (markiert durch roten Kreis). $S$ stellt die Operation von Instanz 2 daher hinter diejenige von Instanz 1.

[5] Die Annahme ist dadurch gerechtfertigt, dass es immer eine ausreichend hohe Auflösung der Uhren gibt, so dass die beiden Zeiten ungleich sind. Für unsere Zwecke genügt hierfür bereits eine Auflösung im Millisekundenbereich. Auflösungen im Nanosekundenbereich sind aufwendiger zu berechnen, bieten jedoch mehr Sicherheit.

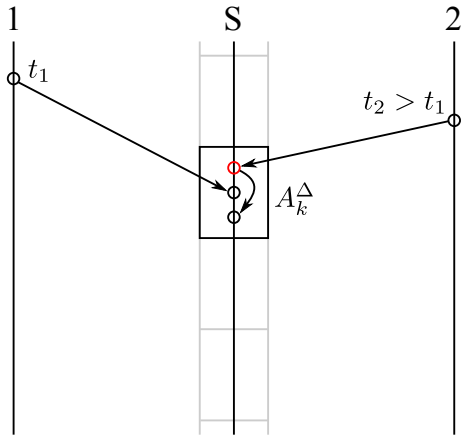


**Abbildung 3.6:** Ordnung zweier Operationen in einem geordneten Aggregat

Das Ziel lautet nun, die Folge $(\alpha_k^\Delta)_{k\in\mathbb{N}}$ als Grundlage für die Verteilung der Operationen in $S$ zu verwenden, $S$ soll also alle Operationen $o_{i,t^o,t^*}$ einem geordneten Aggregat $A_k^\Delta$ der Folge zuweisen. Außerdem soll das Verfahren aus Abschnitt 3.4.2 beibehalten werden. Statt jede Operation einzeln zu verteilen, werden fortan die Operationen ganzer Aggregate verteilt. Die Operationen sind dabei zeitlich für die Länge des Aggregats geordnet.

Betrachten wir ein geordnetes Aggregat $A_k^\Delta = ((t_k, l_k, O_k), \Delta_k)$ und das Tupel $\sigma_k := (o_{i,t_i^o,t_i^*}, o_{j,t_j^o,t_j^*}, ...)$ mit allen Operationen aus $O_k$, wobei die Einträge des Tupels gemäß $\Delta_k$ sortiert sind. Sei $o_{i,t_i^o,t_i^*}$ das erste Element des Tupels, d.h. es gilt $\forall o_{j,t_j^o,t_j^*} \in \sigma_k : j \neq i \Rightarrow t_i^o < t_j^o$. Bevor das Tupel an alle Instanzen verteilt werden kann, müssen die Ausführungszeitpunkte der darin enthaltenen Operationen angepasst werden. Der Ausführungszeitpunkt einer Operation $o_{j,t_j^o,t_j^*} \in \sigma_k$ wird hierfür auf $t_j^* = t_k + l_k + (t_j^o - t_i^o) + t_{lat}$ gesetzt, wobei $t_{lat}$ die künstliche Verzögerung aus dem Verfahren in Abschnitt 3.4.2 ist. Die positive Differenz $t_j^o - t_i^o$ sichert die relativen Zeitabstände der Operationen und somit eine korrekte zeitliche Ordnung. Sobald das Tupel $\sigma_k$ von einer Instanz empfangen wird, führt diese die im Tupel enthaltenen Operationen in gegebener Reihenfolge zu den jeweiligen Zeitpunkten aus.

Das Verfahren garantiert die zeitliche Ordnung der Operationen für die *meisten* Fälle. Einen speziellen Fall deckt das Verfahren jedoch nicht ab, siehe hierzu Abbildung 3.7. Darin zu sehen sind zwei Instanzen 1 und 2 und der Server $S$. Instanz 1 initiiert eine Operation zum Zeitpunkt $t_1$, die von $S$ dem Aggregat $A_k^\Delta$ zugeordnet wird. Instanz 2 initiiert ebenfalls eine Operation zu einem Zeitpunkt $t_2 > t_1$, die jedoch derart früher bei $S$ ankommt, dass sie dem Aggregat $A_{k-1}^\Delta$ zugeordnet wird. In diesem Fall kann die zeitliche Ordnung der beiden Operationen nicht wiederhergestellt werden, da sie sich in unterschiedlichen Aggregaten befinden. Die Operationen werden folglich in falscher Reihenfolge ausgeführt. Da sie jedoch von *allen* Instanzen in derselben (falschen) Reihenfolge ausgeführt werden, kann kein inkonsistenter Zustand erfolgen. Die Wahrscheinlichkeit für das Auftreten eines solchen Falls sinkt, wenn die Längen $l_k$ der Aggregate vergrößert werden. Andererseits verursachen längere Aggregate höhere Reaktionszeiten, die jedoch möglichst gering sein sollen, wie wir in Abschnitt 3.4.2 festgehalten haben. Es ergibt sich ein Missverhältnis ähnlich demjenigen zwischen Kurzzeitinkonsistenz und Reaktionszeit. Abgesehen von diesem Sonderfall funktioniert das Verfahren jedoch sehr gut, wie die Evaluation in Kapitel 5 zeigt.

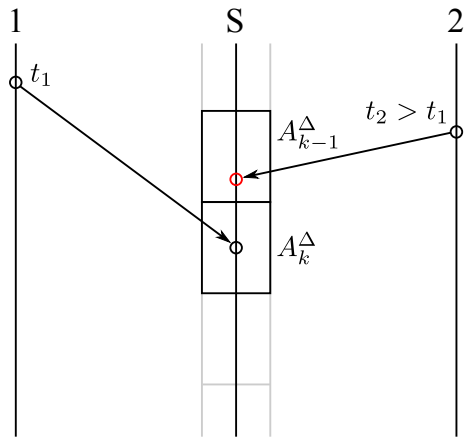


**Abbildung 3.7:** Ausnahmefall der zeitlichen Ordnung vor Empfang beim Server

## 3.5 Postanalyse

In diesem Abschnitt soll formal bewiesen werden, dass die hier eingeführten Modifikationen am Datenkommunikationsmodell von Saga Online ausreichen, um das starke Konsistenzkriterium zu erfüllen (siehe Abschnitt 3.2.2). Es werden zwei Varianten für einen Beweis vorgestellt. Der erste Beweis wird durch einen Widerspruch herbeigeführt, der zweite mithilfe einer Reduktion. Es sei darauf hingewiesen, dass insbesondere die zeitliche Ordnung von Operationen *nach* Empfang beim Server (siehe Abschnitt 3.4.2) bereits ausreicht, um das Kriterium zu erfüllen.

### 3.5.1 Beweis durch Widerspruch

Der hier vorgestellte Beweis basiert auf einem Widerspruch. Es soll gezeigt werden, dass das Komplement des starken Konsistenzkriteriums unerfüllbar ist. Den Widerspruch führen wir herbei, indem wir annehmen, dass das Komplement des Kriteriums erfüllbar ist, die Annahme jedoch als falsch heraus stellen. Das Komplement der starken Konsistenzkriteriums lautet wie folgt:

$$\exists t \in T\ \exists i \in I(A) : \left[\forall o_{k,t^o,t^*} \in O(A) : t^* \leq t \Rightarrow R_A(i,t,o_{k,t^o,t^*})\right] \wedge s_{i,t} \neq s_{P,t}$$

Es ist $A = SO$ und $P$ die virtuelle perfekte Instanz von $SO$. Wir nehmen an, dass obiger Ausdruck in der modifizierten Variante von $SO$ erfüllbar ist, d.h. es existiert ein Zeitpunkt $t$ und eine Instanz $i$, so dass alle Operationen $o_{k,t^o,t^*}$ mit $t^* \leq t$ rechtzeitig von $i$ empfangen werden, der Zustand von $i$ zum Zeitpunkt $t$ jedoch ungleich demjenigen von $P$ zum Zeitpunkt $t$ ist (d.h. $s_{i,t} \neq s_{P,t}$). Wegen $s_{i,t} \neq s_{P,t}$ existiert eine Operation $o_{k,t^o,t^*}$ mit $t^* \leq t$, die entweder nicht rechtzeitig empfangen (d.h. $R_A(i,t,o_{k,t^o,t^*}) = false$), oder nicht exakt zum Zeitpunkt $t^*$ ausgeführt wurde. Letzteres ist per Definition ausgeschlossen, denn die Modifikationen an $SO$ sehen die Ausführung zum Zeitpunkt $t^*$ vor. Ersteres wird durch die zeitliche Ordnung *nach* Empfang beim Server verhindert, denn die künstliche Verzögerung $t_{lat}$ kann immer so gewählt werden, dass $R_A(i,t^*,o_{k,t^o,t^*})$ und damit insbesondere $R_A(i,t,o_{k,t^o,t^*})$ wahr ist. Demnach kann eine solche Operation nicht existieren, womit wir einen Widerspruch erhalten.

□

### 3.5.2 Beweis durch Reduktion

Eine andere Möglichkeit für einen Beweis ergibt sich, indem wir das starke Konsistenzkriterium auf einen in der modifizierten Variante von $SO$ allgemeingültigen Ausdruck reduzieren. Wir erreichen dies, indem wir zeigen, dass alle Instanzen $i \in I(SO)$ äquivalent zur virtuellen perfekten Instanz $P$ sind. Wegen $\forall i \in I(SO) : i \equiv P$ gilt dann auch $\forall t \in T : s_{i,t} = s_{P,t}$, womit sich das Konsistenzkriterium auf folgenden allgemeingültigen Ausdruck reduzieren lässt:

$$\forall t \in T \ \forall i \in I(A) : \left[\forall o_{k,t^o,t^*} \in O(A) : t^* \leq t \Rightarrow R_A(i, t, o_{k,t^o,t^*})\right] \Rightarrow true \ \text{ mit } \ A = SO$$

Wir zeigen nun, dass alle Instanzen $i \in I(SO)$ äquivalent zu $P$ sind. $P$ ist definiert als eine Instanz mit der Eigenschaft, alle Operationen $o_{k,t^o,t^*}$ bis spätestens $t^*$ zu empfangen und sie zu diesem Zeitpunkt auszuführen. Letztere Eigenschaft gilt gemäß Modifikation auch für alle regulären Instanzen. Erstere Eigenschaft ist für ein genügend großes $t_{lat}$ erfüllt. Angenommen es existiert eine maximale Verzögerungszeit $t_{max}$ im Netzwerk, so dass keine Verzögerung größer als $t_{max}$ ist. Wir können dann $t_{lat} = t_{max}$ setzen, um $R_{SO}(i, t^*, o_{k,t^o,t^*}) = true$ für alle Instanzen $i$ und Operationen $o_{k,t^o,t^*}$ zu erhalten. Somit sind alle Instanzen $i \in I(SO)$ äquivalent zur virtuellen perfekten Instanz $P$.

□

In der Praxis existiert ein solches $t_{max}$ üblicherweise nicht, insbesondere nicht im Internet.[6] Der Beweis ist damit rein theoretischer Natur. Allerdings kann $t_{lat}$ ohne Weiteres derart hoch gesetzt werden, dass $R_{SO}(i, t^*, o_{k,t^o,t^*})$ für alle Instanzen $i$ und Operationen $o_{k,t^o,t^*}$ mit extrem hoher Wahrscheinlichkeit wahr ist. Dass die Reaktionszeit hiervon negativ beeinflusst wird, ist für den Beweis der Konsistenz irrelevant.

Die zeitliche Ordnung von Operationen *vor* Empfang beim Server benötigt eine Rechtfertigung, da sie für den Erhalt der Zustandskonsistenz (zumindest so, wie sie im Rahmen dieser Arbeit definiert ist) nicht notwendig ist. Tatsächlich werden beide Beweise nur mithilfe der zeitlichen Ordnung von Operationen *nach* Empfang beim Server geführt. Trotzdem stellt die Ordnung vor Empfang beim Server eine gravierende Verbesserung dar, da sie die zeitliche Reihenfolge der Operationen wie ursprünglich *intendiert* garantiert. Dies ist insbesondere daher wichtig, da die Uhren aller Instanzen synchronisiert sind, und sie somit eine gemeinsame zeitliche Sichtweise auf das Spiel haben. Für die Zustandskonsistenz ist dies nur insofern irrelevant, als dass sie unabhängig von der zeitlichen Ordnung von Operationen vor Empfang beim Server ist, solange die Operationen von allen Instanzen *zur selben Zeit* ausgeführt werden.

Die zeitliche Ordnung von Operationen vor Empfang beim Server wird außerdem für das Erfüllen der *Fehlertoleranz* benötigt (siehe Kapitel 4). Einer der dort beschriebenen Fehlerfälle, deren Behandlung eine fehlertolerante Anwendung auszeichnen, ist der verspätete Empfang einer Nachricht (d.h. Operation) von Clients (d.h. Instanzen) zum Server. Eine Operation $o_{i,t_i^o,t_i^*}$ ist hierbei verspätet, wenn sie von $S$ zum Zeitpunkt $t_1$ empfangen wird, und eine andere Operation $o_{j,t_j^o,t_j^*}$ mit $t_j^o > t_i^o$ existiert, die von $S$ zu einem Zeitpunkt $t_2 < t_1$ empfangen wird. Dieser Fall wird durch durch die zeitliche Ordnung von Operationen vor Empfang beim Server korrigiert, so dass $t_j^o > t_i^o \Rightarrow t_j^* > t_i^*$ gilt.

[6] Tatsächlich definiert das Internet Protocol [11] ein 8-Bit langes Feld namens TTL (*Time to Live*), das mit einem Wert größer 0 initialisiert wird. Alle Stellen, die ein IP-Paket weiterleiten, müssen diesen Wert um die Anzahl an Sekunden zur Bearbeitung des Pakets, zumindest jedoch um 1, senken. Ist der Wert 0 erreicht, so wird das Paket verworfen. Damit ergibt sich theoretisch eine maximale Lebenszeit von 255 Sekunden für ein IP-Paket (ungeachtet der Latenzzeiten). In der Praxis ist jedoch keinesfalls gewährleistet, dass der Wert immer gemäß Spezifikation behandelt wird. Außerdem berücksichtigt der Wert nicht den Ausfall eines Pakets, womit auch hier keine obere Schranke für die Dauer des Transports gegeben ist.

# 4 Fehlertoleranz

## 4.1 Einleitung

In Kapitel 3 wurde das Datenkommunikationsmodell von Saga Online hinsichtlich Zustandskonsistenz untersucht und um Verfahren erweitert, die die Eigenschaft erfüllen sollen. Aufbauend auf dieser Modifikation soll das Modell nun hinsichtlich Fehlertoleranz untersucht werden. Der Begriff wird hierzu auf eine nicht ganz formale Weise definiert, indem sechs Fehlerfälle spezifiziert werden, die von einem fehlertoleranten System geeignet zu behandeln sind. Die anschließende Analyse wird zeigen, dass zwei dieser Fehlerfälle nicht behandelt werden, womit das Modell in seiner derzeitigen Form nicht fehlertolerant ist. Es wird ein Verfahren eingeführt, dass die beiden Fälle geeignet behandelt. In einer Postanalyse wird schließlich gezeigt, dass das derart modifizierte Modell fehlertolerant ist.

Eine verteilte Anwendung ist allgemein dann fehlertolerant, wenn sie trotz Aufkommen von Störungen, Fehlern oder Ausfällen[1] bis zu einem gewissen Grad weiter arbeiten kann, und idealerweise so, als wäre nichts von all dem passiert. Da Fehler allgegenwärtig sind, insbesondere in einem komplexen Netz wie dem Internet, ist Fehlertoleranz eine wichtige Eigenschaft. Im Rahmen dieser Arbeit betrachten wir die Fehlertoleranz zudem als eine unterstützende Maßnahme für die Zustandskonsistenz, da ein Fehler (beispielsweise der Ausfall einer Nachricht), oftmals zu einem inkonsistenten Zustand führt. Ein fehlertolerantes System weist daher tendenziell weniger Inkonsistenzen auf.

Fehlertoleranz ist für Saga Online insbesondere deshalb wichtig, weil die Anwendung keinerlei Anforderungen an die Anbindung oder Systeme der teilnehmenden Spieler stellt. Dies kann zur Folge haben, dass ein Spiel einen oder mehrere Teilnehmer mit schlechter Konnektivität beinhaltet, womit verspätete oder ausgefallene Nachrichten sehr viel wahrscheinlicher sind. Ein System, das nicht über ausreichende Kapazitäten zur Ausführung des Spiels verfügt, ist außerdem anfälliger für einen Systemabsturz, der das laufende Spiel ernsthaft beeinträchtigen kann, sofern er nicht korrekt behandelt wird.

Eine allgemeine und weitaus umfassendere Behandlung des Themas findet sich in [8]. Ein System wird darin als fehlertolerant bezeichnet, wenn das Verhalten des Systems trotz Ausfall eines oder mehrerer Teilkomponenten konsistent mit seiner Spezifikation ist. Ein Fehler, sofern aufgetreten, soll maskiert oder verdeckt werden, d.h. er soll nicht nach außen hin sichtbar sein. Besonders hervorzuheben sind weiterhin die Ausarbeitungen in [6]. Abschnitt 2.3.1 des Buchs enthält eine formale Definition von Fehlertoleranz. Tanenbaum [13] beschreibt Fehlertoleranz als die Eigenschaft eines Systems, dessen Dienste auch in Anwesenheit bestimmter Störungen bereitstellen zu können, und setzt sie in enge Verbindung mit den Eigenschaften *Verfügbarkeit*, *Zuverlässigkeit*, *Funktionssicherheit* und *Wartbarkeit*.

Das Kapitel ist wie folgt aufgebaut:

Abschnitt 4.2 beinhaltet unsere Definition von Fehlertoleranz. In Abschnitt 4.3 wird das Datenkommunikationsmodell von Saga Online hinsichtlich Fehlertoleranz untersucht (basierend auf unserer Definition). Abschnitt 4.4 führt ein Verfahren ein, dass zwei Fehlerfälle aus der Definition in geeigneter Weise behandelt. Abschnitt 4.5 beendet das Kapitel mit einer Postanalyse, in der gezeigt wird, dass das mit dem Verfahren erweiterte Datenkommunikationsmodell die Definition der Fehlertoleranz erfüllt.

[1] Der Begriff *Fehler* soll die Begriffe *Störung* und *Ausfall* künftig miteinschließen.

## 4.2 Definition von Fehlertoleranz

Wir definieren nun den Begriff der *Fehlertoleranz*. Dabei verfolgen wir einen weniger formalen Ansatz, indem wir sechs *Fehlerfälle* und ihre *Nachbedingungen* spezifizieren. Eine verteilte Anwendung ist genau dann fehlertolerant, wenn sie die Fehlerfälle so behandelt, dass ihre Nachbedingungen gelten. Insbesondere darf keiner der Fehlerfälle die Zustandskonsistenz verletzen, d.h. sollte ein Fehlerfall eintreten, so muss die Zustandskonsistenz spätestens *nach* Behandlung des Fehlers wieder erfüllt sein.

Unsere Definition von Fehlertoleranz richtet sich stark nach dem Modell von Saga Online, d.h. wir betrachten eine zeitkontinuierliche verteilte Anwendung $A$, deren Instanzen $i \in I(A)$ von einem zentralen Server $S$ synchronisiert werden. Wir gehen außerdem davon aus, dass $A$ die in Kapitel 3 eingeführten Verfahren verwendet. Trotz dieser Beschränkung spiegelt unsere Definition den Kern der Fehlertoleranz hinreichend gut wieder: Eine Anwendung ist fehlertolerant, wenn sie Fehler *tolerieren* kann.

Es folgen die Spezifikationen der sechs Fehlerfälle und ihre Nachbedingungen:

### Client fällt aus

*Beschreibung* Als *Client* ($C$) bezeichnen wir ein System, auf dem ein oder mehrere Instanzen $i \in I(A)$ der Anwendung ausgeführt werden. Wir gehen hierbei ohne Beschränkung der Allgemeinheit davon aus, dass ein Client nur eine Instanz ausführt, und benutzen die Begriffe daher synonym. Mit $C_i$ bezeichnen wir den Client, der die Instanz $i$ ausführt. Ein Client fällt aus, wenn er seine Arbeit stoppt oder nicht ordnungsgemäß fortführt. Es kann viele Gründe für einen Ausfall geben, als Beispiel sei hier der Systemabsturz genannt. Charakteristisch für den Ausfall ist, dass er nicht oder nur durch Eingreifen des Benutzers behoben werden kann.

*Nachbedingung* Der Ausfall eines Clients darf die Arbeit der verteilten Anwendung langfristig nicht beeinträchtigen, d.h. sie kann unmittelbar nach dem Ausfall bis zu einem gewissen Grad weitergeführt werden, und nach Behandlung des Ausfalls, d.h. nach endlicher Zeit, wieder normal fortgesetzt werden. Dies setzt in den meisten Fällen voraus, dass der Ausfall von der Anwendung erkannt wird. Insbesondere darf der Ausfall eines Clients keine weiteren Ausfälle verursachen.

### Server fällt aus

*Beschreibung* Als *Server* ($S$) bezeichnen wir diejenige Komponente der verteilten Anwendung, die die Zustände der Instanzen synchronisiert, gleichzeitig aber auch das System, auf dem die Komponente ausgeführt wird. Der Server fällt aus, wenn er seine Arbeit stoppt oder nicht ordnungsgemäß fortführt (analog zum Ausfall eines Clients). Im Gegensatz zu den Clients ist der Server, von dem es typischerweise nur einen gibt, ein zentraler Bestandteil der verteilten Anwendung. Ein Ausfall des Servers kann daher fatal sein in dem Sinne, als dass die Arbeit der Anwendung nicht oder nur mit erheblichem Aufwand in ihrer bisherigen Weise fortgesetzt werden kann.

*Nachbedingung* Der Ausfall des Servers darf nicht den Ausfall eines oder mehrerer Clients zur Folge haben. Dies setzt normalerweise voraus, dass der Ausfall von den Clients erkannt wird. Nachdem der Ausfall des Servers von einem Client erkannt wurde, soll dieser den Ausfall derart behandeln, dass ein Wechsel zu einem anderen Server möglich ist, sofern ein solcher existiert.

### Nachricht von Client zu Server kommt zu spät

*Beschreibung* Eine Nachricht von einem Client $C_i$ zum Server $S$ bezeichnet in diesem Kontext eine Operation $o_{i,t_i^o,t_i^*} \in O(A)$, die von Instanz $i$ zu $S$ gesendet wird. Die Operation kommt zu spät bei $S$ an, wenn sie von $S$ zum Zeitpunkt $t_1$ empfangen wird, und eine andere Operation $o_{j,t_j^o,t_j^*} \in O(A)$ mit $t_j^o > t_i^o$ existiert, die von $S$ zu einem Zeitpunkt $t_2 < t_1$ empfangen wird. Dies bedeutet, dass die Operationen in falscher Reihenfolge bei $S$ ankommen.

*Nachbedingung* Es muss gewährleistet sein, dass die Operationen in korrekter Reihenfolge ausgeführt werden. Formal bedeutet dies, dass $t_j^o > t_i^o \Rightarrow t_j^* > t_i^*$ gilt. Idealerweise ist der zeitliche Abstand zwischen den Operationen zur Initiierung und Ausführung identisch, d.h. es gilt $t_j^o - t_i^o = t_j^* - t_i^*$. Natürlich gilt weiterhin, dass die Zustandskonsistenz nicht verletzt werden darf. Außerdem darf eine verspätete Nachricht keinen Ausfall verursachen.

### Nachricht von Client zu Server fällt aus

*Beschreibung* Dies ist ein Spezialfall des vorhergehenden Fehlerfalls, bei dem die Nachricht mit unendlicher Verspätung beim Server ankommt (d.h. niemals). Formal fällt eine Operation $o_{i,t^o,t^*} \in O(A)$ genau dann aus, wenn $\pi_S(o_{i,t^o,t^*}) \to \infty$ gilt (siehe Kapitel 3, Abschnitt 3.4.3).

*Nachbedingung* Die Nachbedingungen sind identisch zu denjenigen des allgemeinen Falls, bei dem sich die Nachricht nur um endliche Zeit verspätet. Hier gilt insbesondere, dass durch den Ausfall der Nachricht kein Ausfall aufseiten des Servers oder der Clients erfolgen darf.

### Nachricht von Server zu Client kommt zu spät

*Beschreibung* Eine Nachricht vom Server $S$ zu einem Client $C_i$ bezeichnet in diesem Kontext ein geordnetes Aggregat $A_k^\Delta$ (siehe Kapitel 3, Abschnitt 3.4.3), das von $S$ zur Instanz $i$ gesendet wird. Das Aggregat kommt zu spät bei $i$ an, wenn es von $i$ zum Zeitpunkt $t_1$ empfangen wird, und ein anderes geordnetes Aggregat $A_h^\Delta$ mit $h > k$ existiert, das von $i$ zu einem Zeitpunkt $t_2 < t_1$ empfangen wird. Dies bedeutet, dass die Aggregate in falscher Reihenfolge bei $i$ ankommen.

*Nachbedingung* Auch hier muss gewährleistet sein, dass die in den Aggregaten enthaltenen Operationen in korrekter Reihenfolge ausgeführt werden, idealerweise zu den zugewiesenen Ausführungszeitpunkten. Insbesondere darf die Zustandskonsistenz durch verspätete Nachrichten nicht dauerhaft verletzt werden, d.h. spätestens nach Behandlung des Fehlerfalls muss die Zustandskonsistenz wieder erfüllt sein. Wichtig ist zudem, dass verspätete Nachrichten keine Ausfälle verursachen können.

### Nachricht von Server zu Client fällt aus

*Beschreibung* Dies ist ein Spezialfall des vorhergehenden Fehlerfalls, bei dem die Nachricht mit unendlicher Verspätung beim Client ankommt (d.h. niemals). Formal fällt ein geordnetes Aggregat $A_k^\Delta$ dann aus, wenn alle Glieder $\alpha_h^\Delta$ der Folge $(\alpha_n^\Delta)_{n\in\mathbb{N}}$ (siehe Kapitel 3, Abschnitt 3.4.3) mit $h > k$ noch vor dem Glied $\alpha_k^\Delta$ bei der Instanz $i$ ankommen (die Folge hat unendlich viele Glieder).

*Nachbedingung* Die Nachbedingungen sind wieder identisch zu denjenigen des allgemeinen Falls, bei dem sich die Nachricht nur um endliche Zeit verspätet. Auch hier gilt insbesondere, dass durch den Ausfall der Nachricht kein Ausfall aufseiten des Servers oder der Clients erfolgen darf.

## 4.3 Analyse des Modells

In diesem Abschnitt wird das Datenkommunikationsmodell von Saga Online hinsichtlich Fehlertoleranz untersucht. Als Grundlage hierfür dient unsere Definition von Fehlertoleranz aus dem vorhergehenden Abschnitt, d.h. es wird geprüft, ob alle darin spezifizierten Fehlerfälle derart behandelt werden, dass ihre Nachbedingungen erfüllt sind. Die Analyse erfolgt im Weiteren unter der Annahme, dass das Modell um die in Kapitel 3 eingeführten Verfahren ergänzt wurde.

Zunächst soll geprüft werden, ob das Modell den Ausfall eines Clients oder des Servers geeignet behandelt. In der ursprünglichen Form des Modells senden Clients in regelmäßigen Abständen eine leere Nachricht an den Server und umgekehrt, jedoch nur dann, wenn in jenem zeitlichen Abstand nicht bereits eine andere Nachricht gesendet wurde. Mithilfe dieser periodischen Nachricht teilen die jeweiligen Sender ihre Anwesenheit mit. Fällt die Nachricht aus (d.h. es kommen auch außer dieser Nachricht keine anderen mehr an) und steht die Verbindung noch (im Falle von TCP[2]) oder wird UDP[3] für die Kommunikation genutzt (das Spiel unterstützt beide Protokolle), so wird die vermeintliche Abwesenheit mittels einer expliziten Anfrage auf Anwesenheit überprüft. Falls auch dann keine Nachricht eintrifft, so wird die Abwesenheit des Clients oder Servers als bestätigt angenommen, womit der Fehlerfall erkannt wurde. Wenn ein Client den Ausfall des Servers feststellt, so trennt er die Verbindung (im Falle von TCP), und begibt sich zurück zur Serverliste (siehe Kapitel 2, Abschnitt 2.2). Von dort aus kann er sich mit einem anderen Server verbinden. Wenn dagegen der Server den Ausfall eines Clients feststellt, so teilt er dies den anderen Clients mit (diese entfernen den ausgefallen Client dann vom Spiel) und entfernt den Client von seiner Verteilerliste. Insbesondere jedoch verursacht der Ausfall des Servers oder eines Clients keine weiteren Ausfälle. Die beiden Fehlerfälle werden somit geeignet im Sinne der Spezifikation behandelt.

Wir überprüfen nun den Fehlerfall, bei dem eine Nachricht von einem Client (d.h. eine Operation) zu spät beim Server ankommt. Die Operation sei bezeichnet mit $o_{i,t_i^o,t_i^*}$ und vom Server $S$ zum Zeitpunkt $t_1$ empfangen. Gemäß Spezifikation des Fehlerfalls kommt die Operation genau dann zu spät bei $S$ an, wenn eine Operation $o_{j,t_j^o,t_j^*}$ mit $t_j^o > t_i^o$ existiert, die von $S$ zu einem Zeitpunkt $t_2 < t_1$ empfangen wird. In der ursprünglichen Form des Modells hätte dies $t_j^* < t_i^*$ unmittelbar zur Folge. Wir zeigen, dass eine solche Operation mit hinreichend großer Aggregatlänge nicht existieren kann, bedingt durch die zeitliche Ordnung von Operationen *vor* Empfang beim Server (siehe Kapitel 3, Abschnitt 3.4.3). Sei $A_k^\Delta = (t_k, l_k, O_k, \Delta_k)$ ein geordnetes Aggregat, dann gilt per Definition $\forall o_{i,t_i^o,t_i^*}, o_{j,t_j^o,t_j^*} \in O_k : t_i^o < t_j^o \Rightarrow t_i^* < t_j^*$. Wenn wir also garantieren, dass die Operationen $o_{i,t_i^o,t_i^*}$ und $o_{j,t_j^o,t_j^*}$ im selben Aggregat $A_k^\Delta$ enthalten sind, dann gilt auch $t_j^* > t_i^*$. Hierfür muss die uniforme Aggregatlänge $l$ derart gewählt werden, dass sie den zeitlichen Abstand der Empfangszeiten $t_1$ und $t_2$ kompensiert. In der Theorie genügt es, $l \to \infty$ zu setzen. In der Praxis muss $l$ so gewählt werden, dass beide Operationen *praktisch immer* (d.h. mit höchster Wahrscheinlichkeit) zum selben Aggregat gehören. Natürlich würde dies die Reaktionszeit ebenso erhöhen (was nicht wünschenswert ist), für den theoretischen Beweis ist das jedoch irrelevant. Es wurde somit gezeigt, dass es keine solche Operation $o_{j,t_j^o,t_j^*}$ geben kann (für hinreichend großes $l$). Offensichtlich ist das Modell zustandskonsistent, da sich in dieser Hinsicht nichts verändert. Ebenso gilt, dass verspätete Nachrichten keine Ausfälle verursachen. Der Fehlerfall wird somit geeignet im Sinne der Spezifikation behandelt.

Abbildung 4.1 veranschaulicht das Argument der Aggregatlänge. Beide Seiten der Abbildung zeigen den Server $S$ und die Instanzen 1 und 2. Im Unterschied zur linken Seite hat die rechte Seite jedoch eine verdoppelte Aggregatlänge. Instanz 1 sendet eine Operation zum Zeitpunkt $t_1$. Instanz 2 sendet ebenfalls eine Operation zu einem Zeitpunkt $t_2 > t_1$. Auf der linken Seite der Abbildung wird die Operation von

[2] *Transmission Control Protocol* (TCP), ein verbindungsorientiertes Protokoll zur Datenübertragung im Internet.
[3] *User Datagram Protocol* (UDP), ein verbindungsloses Protokoll zur Datenübertragung im Internet.

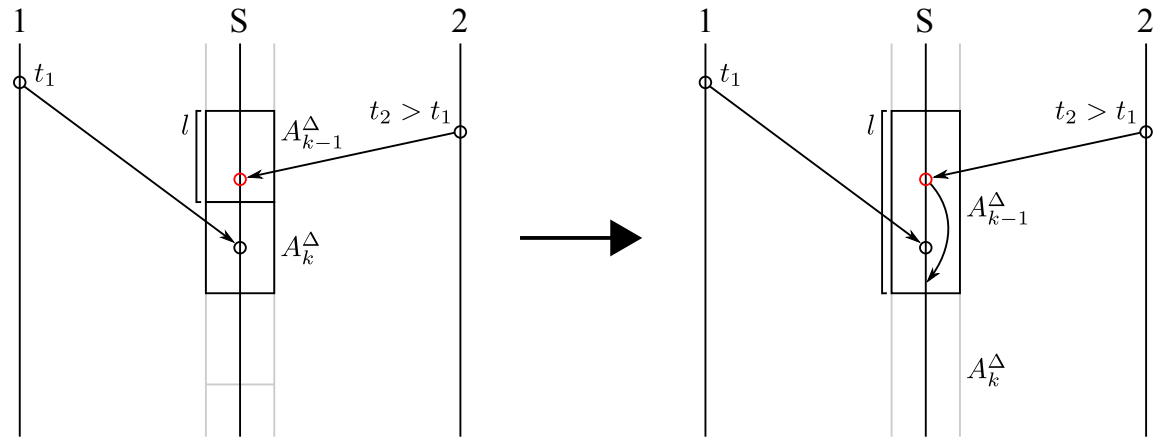


**Abbildung 4.1:** Zeitliche Ordnung von Operationen durch verdoppelte Aggregatlänge

Instanz 1 dem Aggregat $A^{\Delta}_{k}$ zugeordnet, während diejenige von Instanz 2 dem Aggregat $A^{\Delta}_{k-1}$ zugeordnet wird. Die Operation von Instanz 1 kommt somit zu spät bei $S$ an, und da sich die beiden Operationen in unterschiedlichen Aggregaten befinden, wird der Fehlerfall nicht behandelt. Die rechte Seite der Abbildung zeigt dieselbe Situation, durch die verdoppelte Aggregatlänge werden beide Operationen jedoch demselben Aggregat zugewiesen. Der Fehlerfall kann folglich behandelt werden, indem die Operationen in die richtige zeitliche Reihenfolge gestellt werden.

Es soll nun der Fehlerfall betrachtet werden, bei dem eine Nachricht von einem Client zum Server ausfällt. Die Operation sei wieder mit $o_{i,t^o_i,t^*_i}$ bezeichnet. Es gilt $\pi_S(o_{i,t^o_i,t^*_i}) \to \infty$, d.h. die Operation kommt mit unendlicher Verspätung bei $S$ an. Da es sich hierbei um einen Spezialfall des vorigen Fehlerfalls handelt, könnte man annehmen, dass auch hier keine Operation $o_{j,t^o_j,t^*_j}$ mit $t^o_j > t^o_i$ existiert, die von $S$ zu einem Zeitpunkt $t_2 < t_1 = \pi_S(o_{i,t^o_i,t^*_i})$ empfangen wird, so dass auch $t^*_j < t^*_i$ gilt. Dies ist jedoch nicht der Fall, da es keine uniforme Aggregatlänge $l$ gibt, die den Abstand von $t_2$ zu $t_1$ (der unendlich ist) kompensiert, auch dann nicht, wenn wir $l \to \infty$ setzen. Wir argumentieren in diesem Fall auf eine andere Weise, indem wir feststellen, dass es irrelevant im Sinne der Zustandskonsistenz ist, ob die Operation bei $S$ ankommt oder nicht. Kommt die Operation an, d.h. wir haben den Normalfall, so ist die Zustandskonsistenz erfüllt. Kommt sie nicht an, so entspricht dies dem Fall, als wäre die Operation niemals initiiert worden. Niemand, weder der Server noch die Clients (auch nicht der Initiator der Operation), wird jemals erfahren, dass die Operation initiiert wurde.[4] Aus diesem Grunde kann auch kein Ausfall entstehen. Der Fehlerfall wird also korrekt im Sinne der Spezifikation toleriert.

Bis hierhin konnten wir zeigen, dass der Ausfall des Servers oder eines Clients und die Verspätung bzw. der Ausfall einer Nachricht von einem Clients zum Server geeignet im Sinne der Definition behandelt werden. Was nun noch bleibt, sind die Fehlerfälle, bei dem eine Nachricht vom Server zu einem Client verspätet ankommt oder gänzlich ausfällt. Wir zeigen für beide Fälle, dass sie in der momentanen Form des Modells nicht geeignet behandelt werden (genauer: sie werden gar nicht behandelt).

Eine Nachricht vom Server $S$ zu einem Client $C_i$ bezeichnet ein geordnetes Aggregat $A^{\Delta}_{k}$. Die Nachricht ist dabei gerichtet an die Instanz $i$, die von $C_i$ ausgeführt wird. Per Definition kommt das Aggregat zu spät bei $i$ an, wenn es von $i$ zum Zeitpunkt $t_1$ empfangen wird, und ein Aggregat $A^{\Delta}_{h}$ mit $h > k$ existiert, das von $i$ zu einem Zeitpunkt $t_2 < t_1$ empfangen wird. Wir zeigen, dass die Zustandskonsistenz in einem

[4] Der Ausfall einer Operation ist natürlich unoptimal aus Sicht des Benutzers, da dieser die Operation initiiert, sie jedoch nicht ausgeführt wird. In der Regel wird er die Operation einfach wiederholen, sobald er den Ausfall bemerkt hat, und spätestens dann sollte sie auch ausgeführt werden. In diesem Sinne ist der Ausfall also unkritisch.

solchen Fall nicht länger gewährleistet ist, indem wir die Erfüllbarkeit der Negation des starken Konsistenzkriteriums nachweisen. Hierfür betrachten wir den einfachen aber nicht untypischen Fall, dass beide Aggregate genau eine Operation beinhalten, d.h. $A_k^\Delta$ enthält die Operation $o_{a,t_a^o,t_a^*}$ und $A_h^\Delta$ die Operation $o_{b,t_b^o,t_b^*}$. Jeder andere Fall lässt sich mit hinreichend kleinem $l$ (uniforme Aggregatlänge) auf diesen Fall reduzieren. Es gilt $t_a^* = t_k + l_k + t_{lat}$ und $t_b^* = t_h + l_h + t_{lat}$ (siehe Kapitel 3, Abschnitt 3.4.3). Wegen $h > k$ gilt $t_h > t_k$, zusammen mit $l_k = l = l_h$ gilt dann auch $t_b^* > t_a^*$.

Die Negation des starken Konsistenzkriteriums lautet wie folgt:

$$\text{(F1)}: \exists t \in T\ \exists j \in I(SO) : \left[\forall o_{q,t^o,t^*} \in O(SO) : t^* \leq t \Rightarrow R_A(j,t,o_{q,t^o,t^*})\right] \wedge s_{j,t} \neq s_{P,t}$$

Wir betrachten den folgenden erfüllbarkeitsäquivalenten Ausdruck:

$$\text{(F2)}: \exists t \in T : t \geq t_b^* \wedge R_{SO}(i,t,o_{a,t_a^o,t_a^*}) \wedge R_{SO}(i,t,o_{b,t_b^o,t_b^*}) \wedge s_{i,t} \neq s_{P,t}$$

Wir können $t$ so wählen, dass die ersten drei Konjunktionsglieder in F2 erfüllt sind. Es bleibt zu zeigen, dass dann auch $s_{i,t} \neq s_{P,t}$ erfüllbar ist. Per Definition sind $R_{SO}(P,t_a^*,o_{a,t_a^o,t_a^*})$ und $R_{SO}(P,t_b^*,o_{b,t_b^o,t_b^*})$ erfüllt, $P$ führt die Operationen unmittelbar zu ihren Ausführungszeiten aus. Da das Aggregat mit der Operation $o_{a,t_a^o,t_a^*}$ beliebig verspätet bei $i$ ankommt, ist jedoch nicht gewährleistet, dass auch $R_{SO}(i,t_a^*,o_{a,t_a^o,t_a^*})$ wahr ist. Falls letzteres nicht erfüllt ist, so kann $i$ die Operation nicht wie intendiert zum Zeitpunkt $t_a^*$ ausführen, womit $s_{i,t_a^*} \neq s_{P,t_a^*}$ bereits erfüllbar ist. Wegen $t_a^* < t_b^* \leq t$ ist dann auch $s_{i,t} \neq s_{P,t}$ und somit F2 erfüllbar. Weil F2 erfüllbarkeitsäquivalent zu F1 ist, ist auch F1 erfüllbar. Es folgt, dass das starke Konsistenzkriterium nicht erfüllt ist.

□

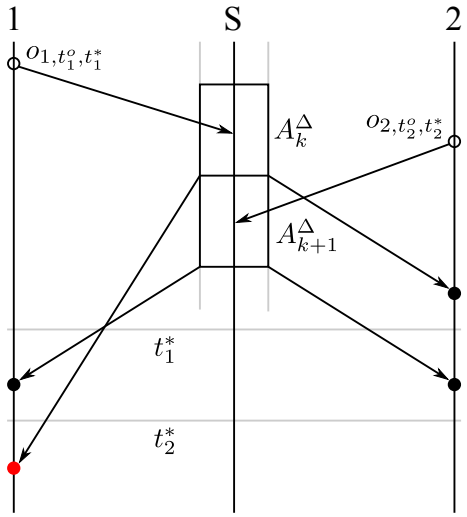


**Abbildung 4.2:** Potentielle Inkonsistenz durch verspätetes Aggregat

Abbildung 4.2 veranschaulicht die im Beweis betrachtete Variante des Fehlerfalls. Zu sehen sind der Server $S$ und die Instanzen 1 und 2. Instanz 1 sendet eine Operation $o_{1,t_1^o,t_1^*}$ an $S$, die dem Aggregat $A_k^\Delta$ zugeordnet wird. Das Aggregat wird anschließend an beide Instanzen verteilt. Instanz 2 sendet ebenfalls eine Operation $o_{2,t_2^o,t_2^*}$ an $S$, die wiederum dem Aggregat $A_{k+1}^\Delta$ zugeordnet wird. Auch dieses Aggregat

wird anschließend an beide Instanzen verteilt. Die grauen Linien markieren die intendierten Ausführungszeitpunkte der Operationen, d.h. die Zeitpunkte $t_1^*$ und $t_2^*$. Instanz 2 empfängt beide Aggregate rechtzeitig und kann die Operationen daher wie intendiert ausführen. Instanz 1 empfängt zunächst das Aggregat $A_{k+1}^{\Delta}$ und kann die darin enthaltene Operation $o_{2,t_2^o,t_2^*}$ rechtzeitig ausführen. Das Aggregat $A_k^{\Delta}$ kommt jedoch verspätet bei Instanz 1 an, und zwar derart verspätet, dass die darin enthaltene Operation $o_{1,t_1^o,t_1^*}$ nicht mehr rechtzeitig ausgeführt werden kann (markiert durch roten Kreis).

Der Beweis berücksichtigt ebenso den Ausfall eines Aggregat. Hier lässt sich jedoch auch intuitiv argumentieren, dass der Zustand einer Instanz $i$ ganz offensichtlich nicht konsistent mit den Zuständen der anderen Instanzen sein muss, wenn diese eine Operation ausführen, die $i$ niemals ausführen wird. Dies entspräche dem Fall, dass $i$ kurzfristig von der Spielwelt ausgeschlossen würde, womit ein konsistenter Zustand gar nicht erst zu erwarten wäre. Folglich wird auch der Ausfall einer Nachricht vom Server zu einem Client nicht geeignet bzw. gar nicht behandelt.

## 4.4 Korrektur des Modells

Die Analyse des Modells hat gezeigt, dass die Verspätung und der Ausfall eines Aggregats nicht geeignet im Sinne unserer Definition von Fehlertoleranz behandelt werden. In diesem Abschnitt führen wir ein Verfahren ein, dass die beiden Fehlerfälle korrekt behandelt. Hierzu definieren wir zunächst die Begriffe Sequenz und Konsistenzrelevanz. In Abschnitt 4.4.1 werden Details zur Implementierung des Verfahrens gegeben. Abschnitt 4.4.2 beschreibt ein alternatives Verfahren, das als *Timewarp* bezeichnet wird.

Wir führen zunächst den Begriff der Sequenz ein:

Eine *Sequenz* ist ein Paar $S_k^{\Delta} = (A_k^{\Delta}, s_k)$, wobei $A_k^{\Delta}$ ein geordnetes Aggregat ist (siehe Kapitel 3, Abschnitt 3.4.3). Mit $s_k \in \mathbb{Z}$ bezeichnen wir die *Sequenznummer* von $S_k^{\Delta}$. Sequenznummern müssen *eindeutig* sein, d.h. sie müssen $\forall S_k^{\Delta}, S_h^{\Delta} : k \neq h \Rightarrow s_k \neq s_h$ erfüllen. Außerdem müssen sie *fortlaufend* sein, es muss also $\forall S_k^{\Delta}, S_h^{\Delta} : h = k+1 \Rightarrow s_h = s_k + 1$ gelten (dies impliziert Eindeutigkeit). Wir gehen fortan von $s_k = k$ für alle Sequenzen $S_k^{\Delta}$ aus. Die Folge $(\varphi_k)_{k\in\mathbb{N}}, \varphi_k \mapsto S_k^{\Delta}$ ist die Folge der Sequenzen.

Unser Verfahren setzt voraus, dass der Server $S$ die Folge $(\varphi_k)_{k\in\mathbb{N}}$ der Sequenzen implementiert, nicht wie bisher die Folge $(\alpha_k^{\Delta})_{k\in\mathbb{N}}$ der geordneten Aggregate. Der Unterschied besteht darin, dass jedem Aggregat eine eindeutige und fortlaufende Nummer zur Identifikation zugeordnet wird. Statt wie bisher die Aggregate $A_k^{\Delta}$ an die Instanzen zu verteilen, werden fortan die Sequenzen $S_k^{\Delta}$ verteilt. Dies ermöglicht den Instanzen die Erkennung der betrachteten Fehlerfälle. Das Verfahren basiert weiterhin auf dem Konzept der Konsistenzrelevanz, das nachfolgend definiert wird:

Sei $A$ eine zeitkontinuierliche verteilte Anwendung und $i \in I(A)$ eine Instanz von $A$. Mit $s_{i,t}$ bezeichnen wir den Zustand der Instanz zum Zeitpunkt $t \in T$. Sei weiterhin $o_{i,t^o,t} \in O(A)$ eine Operation, die von $i$ zum Zeitpunkt $t^o \leq t$ initiiert und zum Zeitpunkt $t$ (d.h. im Zustand $s_{i,t}$) ausgeführt wird. Es ist $s_{i,\hat{t}}$ der Zustand, in den $i$ nach Ausführung der Operation $o_{i,t^o,t}$ im Zustand $s_{i,t}$ gelangt. Die Operation $o_{i,t^o,t}$ ist *konsistenzrelevant* genau dann, wenn $s_{i,t} \neq s_{i,\hat{t}}$ gilt. Ein Aggregat $A_k = (t_k, l_k, O_k)$ ist konsistenzrelevant, wenn mindestens eine Operation $o_{i,t^o,t^*} \in O_k$ existiert, die konsistenzrelevant ist (analog für geordnete Aggregate). Eine Sequenz $S_k^{\Delta} = (A_k^{\Delta}, s_k)$ ist konsistenzrelevant genau dann, wenn $A_k^{\Delta}$ dies ist. Ein Beispiel für eine konsistenzrelevante Operation in Saga Online ist die Bewegung von einer Stelle zu einer anderen, eine Textnachricht hingegen ist nicht konsistenzrelevant.

Wir definieren die Hilfsfunktion $\delta : O(A) \rightarrow \{true, false\}$, die genau dann wahr für eine Operation $o_{i,t^o,t^*} \in O(A)$ ist, wenn sie konsistenzrelevant ist. Wir erlauben außerdem die Schreibweise $\delta(S_k^{\Delta})$ für

eine Sequenz $S_k^\Delta$ mit $\delta(S_k^\Delta) = true \Leftrightarrow \exists o_{i,t^o,t^*} \in O_k : \delta(o_{i,t^o,t^*}) = true$.

Eine weitere Voraussetzung des Verfahrens besteht nun darin, dass $S$ eine Menge $\Phi$ aus Sequenzen verwaltet. Die Menge besteht aus genau denjenigen Sequenzen $S_k^\Delta$, die von $S$ bereits an die Instanzen verteilt wurden und *nicht* konsistenzrelevant sind. $\Phi$ muss also folgende Eigenschaft erfüllen:

$$\forall t \in T\ \forall S_k^\Delta : \left[ t \geq t_k + l_k \wedge \neg\delta(S_k^\Delta) \right] \Leftrightarrow S_k^\Delta \in \Phi$$

In der Praxis wird die Eigenschaft abgeschwächt, indem das $t$ aus obiger Formel nicht größer als die Summe $t_k + l_k + t_\Phi$ für eine positive Konstante $t_\Phi$ sein darf. Anderenfalls würde $\Phi$ unaufhaltsam wachsen, was nur in der Theorie tragbar ist. Eine andere Möglichkeit der Reduktion besteht darin, die Anzahl der Sequenzen in $\Phi$ nach oben zu beschränken, und nur die neuesten Sequenzen beizubehalten.

Das Verfahren funktioniert nun wie folgt:

Jede Instanz $i \in I(A)$ besitzt eine Variable $c_i$, in der die Sequenznummer $s_k$ der zuletzt empfangenen Sequenz $S_k^\Delta$ enthalten ist. Sobald eine neue Sequenz $S_h^\Delta$ bei $i$ eintrifft, wird geprüft, ob $c_i + 1 = s_h$ gilt (dies entspricht dem zu erwartenden Fall, da Sequenznummern fortlaufend sind). Falls dem so ist, so wird die Sequenz normal verarbeitet. Anderenfalls wird geprüft, ob $c_i + 1 < s_h$ gilt (in diesem Fall sind die Sequenzen $S_{c_i+1}^\Delta$ bis $S_{h-1}^\Delta$ verspätet oder ganz ausgefallen). Falls dem so ist, so wurde der Fehlerfall erkannt. Falls aber $c_i + 1 > s_h$ gilt (d.h. die Sequenz $S_h^\Delta$ kommt zu spät), so wird das Aggregat ignoriert. Der Grund hierfür ist, dass es einen früheren Zeitpunkt gegeben haben muss, zu dem $c_i + 1 \leq s_h < s_x$ für eine Sequenz $S_x^\Delta$ galt, womit die Verspätung von $S_h^\Delta$ bereits behandelt wurde. Eine Ausnahme hiervon besteht in der verspäteten Teilnahme der Instanz $i$. Da hier jedoch zwangsläufig $c_i + 1 < s_h$ gilt, wird dieser Fall auf den Ausfall einer Nachricht reduziert und somit vom Verfahren abgedeckt.

Falls der Fehlerfall entdeckt wurde (d.h. es ist $c_i + 1 < s_h$), so ist noch unklar, ob es sich um eine Verspätung oder den Ausfall der Sequenzen $S_{c_i+1}^\Delta$ bis $S_{h-1}^\Delta$ handelt. Das Verfahren behandelt beide Fälle auf dieselbe Weise, daher wird diese Information nicht benötigt. Instanz $i$ sendet eine Anfrage $(c_i, s_h)$ an $S$. $S$ überprüft nun, ob alle Sequenzen $S_{c_i+1}^\Delta$ bis $S_{h-1}^\Delta$ in $\Phi$ enthalten sind. Falls dem so ist, so antwortet $S$ mit den Operationen all dieser Sequenzen. Dies ist ohne Weiteres möglich, da es sich um nicht konsistenzrelevante Operationen handelt, die von $i$ zu jeder Zeit ausgeführt werden können. Falls dem nicht so ist, so leitet $S$ ein 4-phasiges Synchronisierungsverfahren ein, das wie folgt abläuft:

**Phase 1** $S$ sendet eine Anweisung an alle Instanzen in $I(A)$, in der sie für eine unbestimmte Zeit pausiert werden. Pausierte Instanzen können keine Operationen initiieren oder ausführen und ändern ihren Zustand nicht. Die Anweisung wird von allen Instanzen bestätigt. Sobald dies geschehen ist oder eine bestimmte Zeitspanne abgelaufen ist, geht $S$ in Phase 2 über.

**Phase 2** $S$ erfragt den momentanen Zustand $s_{j,t}$ aller Instanzen $j \neq i$ (d.h. von allen außer der betroffenen Instanz $i$). Diese beantworten die Anfrage umgehend. Sobald $S$ den Zustand $s_{j,t}$ einer Instanz $j$ erhalten hat, merkt er sich diesen und verwirft die Antworten aller anderen Instanzen, ohne jedoch auf diese zu warten. $S$ geht direkt über in Phase 3.

**Phase 3** $S$ sendet den Zustand $s_{j,t}$ an alle Instanzen $k \neq j$. Diese übernehmen den Zustand $s_{j,t}$ (d.h. sie setzen $s_{k,t} = s_{j,t}$) und bestätigen dies bei $S$. Die Pause bleibt hierbei nach wie vor bestehen. Sobald alle Instanzen die Übernahme des Zustands bestätigt haben oder eine maximale Zeitspanne abgelaufen ist, geht $S$ in Phase 4 über.

**Phase 4** $S$ sendet eine Anweisung an alle Instanzen in $I(A)$, in der die Pause zu einem gemeinsamen Zeitpunkt aufgehoben wird (alle Uhren sind synchronisiert) und die Sequenznummern aller Instanzen

gleichgesetzt werden. Jede Instanz bestätigt den Erhalt der Anweisung. Das Synchronisierungsverfahren ist damit abgeschlossen und die Anwendung kann normal fortgesetzt werden.

Die Gleichsetzung der gespeicherten Sequenznummern aller Instanzen in Phase 4 muss natürlich derart geschehen, dass das erste Aggregat nach Abschluss des Synchronisierungsverfahrens die unmittelbar folgende Sequenznummer besitzt. Anderenfalls würde das Synchronisierungsverfahren sofort wieder aufgerufen, sobald eine Operation initiiert wird.

In Phase 3 wird der Zustand $s_{j,t}$ von allen Instanzen $k \neq j$ übernommen, nicht nur von $i$. Der Grund hierfür ist, dass der Pausebefehl aus Phase 1 zu unterschiedlichen Zeitpunkten bei den Instanzen eintreffen kann, womit sich die Zustände $s_{k,t}$ prinzipiell von $s_{j,t}$ unterscheiden können. Die Maßnahme in Phase 3 garantiert, dass alle Instanzen denselben Zustand haben.

Mit Ausnahme von Phase 2 verlangen alle Phasen eine Bestätigung des Phasenübergangs. Was aber passiert, wenn die Bestätigung von einer oder mehreren Instanzen ausfällt? In den einzelnen Phasen wird hierzu eine maximale Zeitspanne abgewartet, nach deren Ablauf die nächste Phase einsetzt. Eine andere Möglichkeit besteht darin, die ausbleibenden Bestätigungen erneut anzufordern. Insgesamt stellt der Ausfall einer Bestätigung kein ernsthaftes Problem dar. Auch die andere Richtung, d.h. der Ausfall eines Phasenübergangs vom Server zu den Instanzen, ist kein ernsthaftes Problem. Falls der Übergang zur Phase $p$ mit $1 \leq p < 4$ ausfällt, so ist es unwahrscheinlich, dass auch der Übergang zur Phase $p+1$ und $p+2$ ausfällt. Sobald eine Instanz eine übersprungene Phase bemerkt, kann diese das Synchronisierungsverfahren erneut einleiten. Kritisch wird es nur dann, wenn der Übergang zur Phase 4 ausfällt, womit die Pause nicht aufgehoben wird. Dieser Fall wird spätestens dann erkannt, wenn die betroffene Instanz eine Sequenz $S_k^{\Delta}$ von $S$ empfängt, was ein Indiz dafür ist, dass die Pause bereits aufgehoben wurde. In diesem Fall kann das Synchronisierungsverfahren einfach erneut aufgerufen werden.

Ein besonderer Vorteil unseres Verfahrens besteht darin, dass auch die *verspätete Teilnahme* einer Instanz (en. *Late Join*) behandelt wird. Wenn wir davon ausgehen, dass alle Sequenznummern positiv sind, können wir die Instanzen $i \in I(A)$ mit $c_i = 0$ initialisieren, womit $c_i + 1 < s_h$ zwangsläufig gilt, wenn die Instanz $i$ verspätet (d.h. nicht zum Start der Anwendung) teilnimmt. Wir können alternativ annehmen, dass $s_{min}$ die kleinste mögliche Sequenznummer ist, und initialisieren $c_i$ dann mit $s_{min} - 1$. Die verspätete Teilnahme wird somit auf den Ausfall einer Nachricht vom Server reduziert, womit folglich das Synchronisierungsverfahren eingeleitet wird. Der Unterschied besteht nur darin, dass $S$ die Teilnahme von $i$ explizit bekannt machen muss, anderenfalls bleibt $i$ unbekannt für die restlichen Instanzen.

### 4.4.1 Implementierung der Zustände

Im Zuge der Implementierung des Verfahrens müssen grundlegende Fragen zur Umsetzung beantwortet werden. Eine wichtige Frage betrifft die Modellierung der Zustände: Aus welchen Informationen setzt sich ein Zustand zusammen und wie wird er repräsentiert? Die Frage ist hochgradig anwendungsspezifisch und hängt wesentlich von den Anforderungen an die Anwendung ab. In diesem Abschnitt beschreiben wir die Modellierung der Zustände in Saga Online und geben einige Details zur Umsetzung. Grundlegende Informationen zur Implementierung des Spiels finden sich in Kapitel 2, Abschnitt 2.5.

Zustände werden in Saga Online nach dem Prinzip *"so viel wie nötig, so wenig wie möglich"* modelliert. Dies bedeutet, dass ein Zustand einerseits all diejenigen Informationen enthalten muss, die zum erwarteten Ablauf der Anwendung benötigt werden, andererseits so kompakt wie möglich sein soll. Die erste Anforderung betrifft vor allem den Austausch von Zuständen, so wie er in unserem Verfahren beschrieben wird. Wenn eine Instanz $i$ mit Zustand $s_{i,t}$ einen Zustand $s_{j,t}$ einer anderen Instanz $j$ übernimmt, so muss gewährleistet sein, dass die Informationen in $s_{j,t}$ ausreichen, um einen identischen Spielverlauf

in beiden Instanzen zu erreichen (unter der Annahme, dass zukünftige Operationen zu exakt denselben Zeiten ausgeführt werden). In diesem Sinne bezeichnet ein Zustand $s_{i,t}$ also nicht den *vollständigen* Zustand der Instanz $i$, sondern einen *partiellen* Zustand, der nur diejenigen Informationen enthält, die für den Spielverlauf relevant sind. Wir bezeichnen $s_{i,t}$ daher künftig als *Spielzustand*. Die zweite Anforderung ergibt sich schließlich aus dem einfachen Grund, dass Zustände möglichst wenig Netzlast erzeugen sollen. Zustände, die weniger Informationen enthalten, sind in der Regel einfacher zu berechnen. Auch dies ist eine wünschenswerte Eigenschaft.

Ein Zustand in Saga Online ist ein Paar $s_{i,t} = (P_{i,t}, O_{i,t})$, wobei $i \in I(A)$ die Instanz des Zustands ist und $t \in T$ der Zeitpunkt, zu dem sich $i$ im Zustand $s_{i,t}$ befindet. $P_{i,t}$ ist die Menge der Zustände aller Spielfiguren im Spiel. Genauer bezeichnet $p_x \in P_{i,t}$ den Zustand einer Spielfigur $x$ aus Sicht von $i$ zum Zeitpunkt $t$. $O_{i,t}$ ist die Menge der Zustände aller Operationen im Spiel, dabei enthält $O_{i,t}$ nur solche Operationen, die zum Zeitpunkt $t$ noch aktiv sind. Genauer bezeichnet $o_y \in O_{i,t}$ den Zustand einer aktiven Operation $y$ aus Sicht von $i$ zum Zeitpunkt $t$. Die Zustände $p_x$ und $o_y$ enthalten wiederum nur diejenigen Informationen, die für den Spielverlauf relevant sind. Ein Zustand $p_x$ wird im Spiel durch die Klasse *saga.gc.Character* implementiert und ein Zustand $o_y$ durch die Klasse *saga.gc.ability.Ability* und alle seine abgeleiteten Klassen. Beide Klassen implementieren das Interface *saga.gc.recovery.Recoverable*, das zur Berechnung und Übernahme der Zustände $p_x$ und $o_y$ benötigt wird.

Das Paket *saga.gc.recovery* beinhaltet Interfaces (dt. *Schnittstellen*) und Klassen, die für die Berechnung und Übernahme von Zuständen (d.h. $p_x$, $o_y$ und $s_{i,t}$) entworfen wurden. Insbesondere wichtig ist die Klasse *saga.gc.recovery.RecoveryManager*, die für die Berechnung und Übernahme von Spielzuständen $s_{i,t}$ zuständig ist. Die Interfaces und Klassen werden nachfolgend kurz erläutert, wobei wir Spielfiguren und Operationen zusammenfassend als *Entitäten* bezeichnen:

**saga.gc.recovery.Pausable** Dieses Interface deklariert die Methoden *public boolean isPaused()* und *public void setPaused(boolean paused)*. Mit ersterer Methode lässt sich abfragen, ob eine Entität pausiert ist, mit letzterer lässt sich eine Entität pausieren oder deren Pause aufheben. Eine pausierte Entität ändert ihren Zustand nicht. Das Interface wird von *saga.gc.recovery.Recoverable* geerbt.

**saga.gc.recovery.Backable** Dieses Interface deklariert die Methoden *public ByteSegment getBackup()* und *public void useBackup(ByteSegment backup) throws RecoveryException*. Mit ersterer Methode lässt sich der Zustand einer Entität berechnen, mit letzterer wird der Zustand einer kompatiblen Entität übernommen. *ByteSegment* ist eine Klasse aus dem Projekt UNIVERSAL UTILITIES (siehe Kapitel 2, Abschnitt 2.5) und stellt ein Bytefeld mit Operationen zur Manipulation des Feldes bereit. Das Interface wird von *saga.gc.recovery.Recoverable* geerbt.

**saga.gc.recovery.Recoverable** Dieses Interface deklariert die Methoden *public int getEntityID()* und *public void setEntityID(int id)*. Mit ersterer Methode lässt sich die Identifikationsnummer einer Entität abfragen, mit letzterer lässt sich die Nummer einer Entität setzen. Das Interface erbt die Interfaces *saga.gc.recovery.Pausable* und *saga.gc.recovery.Backable*.

**saga.gc.recovery.RecoveryManager** Diese Klasse stellt das Herzstück des Pakets dar. Sie implementiert das Interface *saga.gc.recovery.Recoverable* und stellt Methoden bereit, um alle Entitäten (d.h. das gesamte Spiel) zu pausieren und den aktuellen Spielzustand $s_{i,t}$ zu berechnen oder einen anderen Spielzustand zu übernehmen. Die Klasse ist ein zentraler Bestandteil des in diesem Abschnitt vorgestellten Verfahrens zur Synchronisation aller Spielzustände.

**saga.gc.recovery.RecoveryException** Diese Klasse repräsentiert eine Ausnahme (d.h. typischerweise einen Fehler) die während der Übernahme eines Zustands auftreten kann. Sie tritt typischerweise dann auf, wenn versucht wird, den Zustand einer inkompatiblen Entität zu übernehmen. Die Klasse erbt von *java.lang.Exception* und kann somit in die throws-Klausel eingebunden werden.

### 4.4.2 Alternatives Verfahren: Timewarp

Eine Alternative zum hier vorgestellten Verfahren wird in [9] beschrieben. Der Algorithmus wird dort als *Timewarp* bezeichnet und behandelt das Problem der verspäteten Nachricht vom Server zum Client. Wir stellen den Algorithmus an dieser Stelle kurz vor und vergleichen ihn anschließend mit unserem Ansatz.

Sei $A$ eine zeitkontinuierliche verteilte Anwendung. Wir nehmen an, dass alle Instanzen $i \in I(A)$ zu einem Zeitpunkt $t_i^I \in T$ mit korrektem Zustand $s_{i,t_i^I} = s_{P,t_i^I}$ initialisiert werden, wobei $P$ die virtuelle perfekte Instanz von $A$ ist (siehe Kapitel 3, Abschnitt 3.2.2). Der Algorithmus setzt voraus, dass jede Instanz $i$ eine Liste $L_i^o$ von Operationen und eine Liste $L_i^s$ von Zuständen verwaltet, die mit $L_i^o = \emptyset$ und $L_i^s = (s_{i,t_i^I})$ initialisiert werden. Die Listen sind stets aufsteigend nach Ausführungszeit $t^*$ (für $L_i^o$) bzw. Zustandszeit $t$ (für $L_i^s$) sortiert. Timewarp funktioniert dann wie folgt:

**Schritt 1** Warte für eine konstante Zeit $t^f$. Empfange in dieser Zeit Operationen und setze sie in die Menge $O_i$. Mit $t^*_{min}$ bezeichnen wir die früheste Ausführungszeit der Operationen in $O_i$, d.h. $t^*_{min} = min\left\{t^* \mid o_{j,t^o,t^*} \in O_i\right\}$. Übertrage alle Operationen von $O_i$ in die Liste $L_i^o$ und setze $O_i = \emptyset$. Die Intervallzeit $t^f$ bestimmt die Frequenz, mit der Zustände berechnet und ausgegeben werden.

**Schritt 2** Bestimme denjenigen Zustand in $L_i^s$, der zeitlich unmittelbar vor $t^*_{min}$ gelegen ist. Mit $t^s_{max}$ bezeichnen wir den Zeitpunkt dieses Zustands, d.h. $t^s_{max} = max\left\{t \mid s_{i,t} \in L_i^s \wedge t < t^*_{min}\right\}$.

**Schritt 3** Sei $t^c$ die aktuelle Zeit. Tue das Folgende für alle Zeitpunkte $t \in T$ mit $t^s_{max} < t < t^c$, für die mindestens eine Operation $o_{j,t^o,t} \in L_i^o$ existiert (in aufsteigender Reihenfolge): Nehme den Zustand $s_{i,t^s_{max}}$, falls dies die erste Iteration ist, sonst den resultierenden Zustand der vorigen Iteration, und bezeichne ihn mit $s_{i,t^b}$. Ausgehend von $s_{i,t^b}$, berechne erneut alle $s_{i,n} \in L_i^s$ mit $t^b \leq n < t$ und ersetze sie in $L_i^s$. Berechne schließlich den Zustand $s_{i,t}$, führe hierzu alle Operationen $o_{j,t^o,t}$ aus.

**Schritt 4** Berechne den aktuellen Zustand $s_{i,t^c}$ basierend auf dem zuletzt berechneten Zustand aus Schritt 3 und speichere ihn in $L_i^s$, gebe ihn abschließend aus. Gehe zurück zu Schritt 1.

Der Name Timewarp (dt. *Zeitsprung*) leitet sich davon ab, dass der aktuelle Zustand $s_{i,t^c}$ ausgehend vom Zustand $s_{i,t^s_{max}}$ im Schnelldurchlauf berechnet wird. Verspätete Operationen werden in Schritt 2 des Algorithmus berücksichtigt, wo gegebenenfalls ein entsprechend früherer Zustand $s_{i,t^s_{max}}$ bestimmt wird, sodass die Operation in die Berechnungen miteinbezogen wird.

Der Algorithmus weist im Vergleich zu unserem Verfahren einige gravierende Nachteile auf. Zunächst einmal behandelt der Algorithmus nur verspätete Nachrichten vom Server zum Client, nicht aber ausgefallene. Abgesehen davon verbraucht der Algorithmus wesentlich mehr Speicher, da viele Operationen und Zustände über längere Zeit gespeichert werden müssen. Unklar ist auch die Frage, wie lange diese im Speicher gehalten werden sollen, bis sie mit hoher Wahrscheinlichkeit nicht mehr benötigt werden. Falls diese Zeit zu kurz gewählt wird, so versagt der Algorithmus für all diejenigen Fälle, in denen eine Operation derart verspätet ankommt, dass kein Zustand $s_{i,t^s_{max}}$ gefunden werden kann. Wird sie zu groß gewählt, so wird unnötig viel Speicherplatz verwendet.

Timewarp bestimmt zudem verhältnismäßig viele Modellierungsdetails der Anwendung, z.B. wird die Ausgabefrequenz der Anwendung festgelegt. Im Gegensatz hierzu ist unser Verfahren weitestgehend allgemein gehalten und kann in nahezu allen Anwendungen eingesetzt werden. Nachträgliche Änderungen an der Anwendung können somit unabhängig von unserem Verfahren durchgeführt werden. Ein weiterer großer Vorteil unseres Verfahrens ist, dass es auch die verspätete Teilnahme einer Instanz abdeckt, indem dieser Fall auf den Ausfall einer Nachricht reduziert wird. Für Timewarp müssen hierzu weitere

Mechanismen erdacht werden (siehe z.B. [15]).

Der Vorteil an Timewarp ist, dass verspätete Operationen ohne Verzögerung in die Zustandsberechnungen miteinbezogen werden, solange das Intervall $t^f$ ausreichend groß ist.[5] In unserem Verfahren wird dagegen eine Pause für alle Instanzen vorgeschrieben, die erst aufgehoben wird, wenn die Zustandskonsistenz wiederhergestellt ist. Eine Möglichkeit, die sich bietet, ist die Kombination beider Verfahren. Dabei werden Zustände gemäß Timewarp berechnet, wobei nicht einzelne Operationen, sondern ganze Sequenzen verwendet werden. Verspätete oder ausgefallene Sequenzen werden gemäß unseres Verfahrens identifiziert und bei $S$ erfragt. Erhält $S$ eine solche Anfrage, so werden die verspäteten bzw. ausgefallenen Sequenzen bei den anderen Instanzen erfragt und an die anfragende Instanz weitergereicht, womit diese den aktuellen Zustand berechnen kann. Alternativ kann $S$ eine eigene Liste von Sequenzen führen und die Anfrage direkt beantworten. Während des ganzen Prozesses ist keine Pausierung notwendig. Das Synchronisierungsverfahren kommt nur dann zum Einsatz, wenn eine Instanz verspätet teilnimmt. Dabei muss nur die verspätete Instanz pausiert werden, alle anderen Instanzen können das Spiel normal fortsetzen und $S$ verteilt deren Operationen nach wie vor, auch an die verspätete Instanz. Sobald letztere ihren initialen Zustand von $S$ erhalten hat, kann sie den aktuellen Zustand mithilfe der Operationen in $L_i^o$ berechnen. Die hier beschriebene Variante einer Kombination beider Verfahren stellt natürlich nur eine Möglichkeit dar, andere Varianten sind durchaus möglich.

## 4.5 Postanalyse

Nachdem wir in Abschnitt 4.4 ein Verfahren zur Behandlung verspäteter oder ausgefallener Nachrichten vom Server zum Client eingeführt haben, soll in diesem Abschnitt überprüft werden, ob das so modifizierte Datenkommunikationsmodell fehlertolerant im Sinne unserer Definition aus Abschnitt 4.2 ist.

Unsere Definition von Fehlertoleranz besagt, dass eine zeitkontinuierliche verteilte Anwendung $A$ mit Server $S$ genau dann fehlertolerant ist, wenn sie sechs Fehlerfälle derart behandelt, dass ihre jeweiligen Nachbedingungen erfüllt sind. Die Analyse in Abschnitt 4.3 hat gezeigt, dass vier der sechs Fehlerfälle bereits korrekt im Sinne unserer Definition behandelt werden. Wir überprüfen nun, ob dies immer noch gilt, nachdem das Modell um das in diesem Kapitel eingeführte Verfahren erweitert wurde.

Die ersten beiden Fehlerfälle besagen, dass $A$ den Ausfall des Servers oder eines Clients tolerieren muss. In der ursprünglichen Form des Modells werden hierzu periodische Nachrichten verschickt, mit deren Hilfe der Sender seine Anwesenheit ausdrückt. Stellt ein Client die Abwesenheit des Servers fest, so begibt er sich schlichtweg zurück in die Serverliste (siehe Kapitel 2, Abschnitt 2.2). Stellt dagegen ein Server die Abwesenheit eines Clients fest, so entfernt er diesen aus seinem Verteiler und informiert die restlichen Clients. Da sich in dieser Hinsicht nichts verändert hat, stellen wir fest, dass das Modell nach wie vor den Ausfall des Servers oder eines Clients toleriert.

Die zwei folgenden Fehlerfälle besagen, dass $A$ verspätete oder ausgefallene Nachrichten vom Client zum Server tolerieren muss. Die Analyse hat festgestellt, dass die in Kapitel 3 eingeführte zeitliche Ordnung von Operationen vor Empfang beim Server bereits ausreicht, um verspätete Nachrichten zu tolerieren (genauer: zu korrigieren). Den Ausfall einer Nachricht haben wir als unwichtig angesehen in dem Sinne, als dass er die Zustandskonsistenz nicht betrifft. Auch in dieser Hinsicht hat sich im Modell nichts verändert. Die zeitliche Ordnung der Operationen findet nach wie vor statt, der Unterschied besteht nur darin, dass Aggregate eindeutige Sequenznummer mit sich führen und daher Sequenzen genannt werden. Wenn wir davon ausgehen, dass Nachrichten vom Server zum Client niemals verspätet

[5] Verzögerungen könnten dagegen sichtbar werden, falls eine Operation stark verspätet ankommt oder das Intervall $t^f$ zu klein gewählt ist, womit die Dauer der Zustandsberechnung zu lang (d.h. größer als $t^f$) sein könnte.

ankommen (dieser Fehlerfall also wegfällt), so stellen wir fest, dass die Modifikationen in diesem Kapitel keine Auswirkung auf jene im vorigen Kapitel haben. Verspätete und ausgefallene Nachrichten vom Client zum Server werden also wie zuvor toleriert.

Es bleibt zu zeigen, dass das modifizierte Modell auch die letzten beiden Fehlerfälle toleriert. Diese besagen, dass $A$ verspätete oder ausgefallene Nachrichten vom Server zum Client tolerieren muss. Wir haben hierzu ein Verfahren eingeführt, dass die Fehlerfälle anhand von Sequenznummern identifiziert und alle Instanzen synchronisiert, wenn die verspäteten oder ausgefallenen Nachrichten konsistenzrelevant sind (sind sie es nicht, so betreffen sich die Zustandskonsistenz nicht und müssen nicht gesondert behandelt werden). Ausgehend von der Notation in Abschnitt 4.4 bezeichnen wir die vermeintlich verspäteten oder ausgefallenen Sequenzen mit $S^{\Delta}_{c_i+1}$ bis $S^{\Delta}_{h-1}$, wobei $c_i$ und $s_h$ die zuletzt bzw. aktuell empfangenen Sequenznummern sind. Der Fehlerfall wurde erkannt, wenn die Ungleichung $c_i + 1 < s_h$ erfüllt ist. Falls mindestens eine der Sequenzen $S^{\Delta}_{c_i+1}$ bis $S^{\Delta}_{h-1}$ konsistenzrelevant ist, so leitet $S$ ein Synchronisierungsverfahren ein, nach dessen korrekter Durchführung $\forall i, j \in I(A) : s_{i,t} = s_{j,t}$ für ein $t \in T$ gilt, womit die Zustandskonsistenz eindeutig wiederhergestellt ist. Falls dagegen $c_i + 1 > s_h$ erfüllt ist, so wird die Sequenz $S^{\Delta}_h$ ignoriert. Der Fall wird darauf zurückgeführt, dass es einen früheren Zeitpunkt geben muss, an dem $c_i + 1 \leq s_h < s_x$ für eine Sequenz $S^{\Delta}_x$ gilt, womit die Verspätung der Sequenz $S^{\Delta}_h$ bereits behandelt wurde. Eine Ausnahme hiervon bildet die verspätete Teilnahme der Instanz. Hier gilt jedoch zwangsläufig $c_i + 1 < s_h$ für alle Sequenzen $S^{\Delta}_h$, womit auch dieser Fall abgedeckt wird. Der Fall $c_i + 1 = s_h$ ist der zu erwartende Fall und führt zur ordnungsgemäßen Verarbeitung der Sequenz. Insgesamt halten wir damit fest, dass verspätete und ausgefallenen Nachrichten vom Server zum Client korrekt im Sinne unserer Definition von Fehlertoleranz behandelt werden, womit das Modell vollständig fehlertolerant ist.

□

# 5 Evaluation

In diesem Kapitel werden die im Rahmen dieser Arbeit eingeführten Modifikationen am Datenkommunikationsmodell von Saga Online evaluiert. Wir verfolgen hierbei einen allgemeinen Ansatz, in dem alle Modifikationen miteinbezogen werden, statt sie isoliert voneinander zu testen. Hierdurch soll gewährleistet werden, dass die Ergebnisse der Evaluation möglichst praxisrelevant sind. In Abschnitt 5.1 führen wir zunächst einen neuen Gleichheitsbegriff ein, den wir benötigen, um Zustände miteinander vergleichen zu können. Das Verfahren der Evaluation wird anschließend in Abschnitt 5.2 erläutert. Abschnitt 5.3 beschreibt die genaue Durchführung der Evaluation und fasst die Ergebnisse zusammen.

## 5.1 Konsistenzrelation

Unser Verfahren basiert im Wesentlichen darauf, die Spielzustände aller Instanzen miteinander zu vergleichen, und somit festzustellen, ob sie untereinander konsistent sind. In der Theorie haben wir hierzu die strikte Gleichheit verwendet, d.h. wir haben gefragt, ob zwei Spielzustände $s_{i,t}$ und $s_{j,t}$ *identisch* sind. In der Praxis erweist sich jedoch ein schwächerer Gleichheitsbegriff als ausreichend, um zu entscheiden, ob zwei Spielzustände *konsistent* sind. Ziel dieses Abschnitts ist es, eine Relation zu definieren, die genau dann wahr für zwei Spielzustände ist, wenn sie konsistent sind.

In Abschnitt 4.4.1 aus Kapitel 4 legten wir die Struktur eines Spielzustands in Saga Online fest, indem wir sagten, dass ein Spielzustand einer Instanz $i$ zum Zeitpunkt $t$ ein Paar $s_{i,t} = (P_{i,t}, O_{i,t})$ ist, wobei $P_{i,t}$ die Menge der Zustände aller Spielfiguren und $O_{i,t}$ die Menge der Zustände aller aktiven Operationen im Spiel ist. Ein $p_x \in P_{i,t}$ haben wir als Zustand der Spielfigur $x$ aus Sicht von $i$ zum Zeitpunkt $t$ definiert. Analog dazu haben wir ein $o_y \in O_{i,t}$ als Zustand der Operation $y$ aus Sicht von $i$ zum Zeitpunkt $t$ definiert, wobei wir nur aktive Operationen betrachten. Wir gehen nun einen Schritt weiter, und spezifizieren die Zustände $p_x$ und $o_y$ auf informelle Weise.

Ein Zustand $p_x \in P_{i,t}$ einer Spielfigur $x$ besteht aus *Statuswerten* und *Bewegungswerten*. Statuswerte sind u.a. Gesundheit und maximale Gesundheit, Stärke und maximale Stärke, Intelligenz und maximale Intelligenz, sowie Erfahrungspunkte und Stufe der Spielfigur. Bewegungswerte sind u.a. Position, Bewegungsrichtung und Geschwindigkeit der Spielfigur. Auch ein Zustand $o_y \in O_{i,t}$ einer Operation $y$ besteht aus Statuswerten und Bewegungswerten, die ganz ähnlich denjenigen von $p_x$ sind, sich jedoch von Operation zu Operation unterscheiden können. Alle Wertzuweisungen sind bezogen auf den Zustand $s_{i,t}$ und aus Sicht von $i$ zum Zeitpunkt $t$ zu verstehen.

Wir definieren nun die Relation $\sim_p$, die genau dann wahr für zwei Zustände $p_{x_1}$ und $p_{x_2}$ ist, wenn sie zur selben Spielfigur gehören (d.h. es ist $x_1 = x_2$) und die Statuswerte der beiden Zustände identisch sowie die Bewegungswerte äquivalent im Sinne einer maximalen Abweichung sind. Zwei Positionen sind beispielsweise äquivalent, wenn sie eine maximale Distanz voneinander nicht überschreiten (dies lassen wir im Rahmen dieser Arbeit unterspezifiziert). Wir erlauben außerdem die Schreibweise $P_{i,t_1} \sim_p P_{j,t_2}$ mit $P_{i,t_1} \sim_p P_{j,t_2} \iff \left[\forall p_x \in P_{i,t_1}\ \exists q_x \in P_{j,t_2} : p_x \sim_p q_x\right] \wedge \left[\forall q_x \in P_{j,t_2}\ \exists p_x \in P_{i,t_1} : q_x \sim_p p_x\right]$. Analog hierzu ist die Relation $\sim_o$ für zwei Zustände $o_{y_1}$ und $o_{y_2}$ bzw. $O_{i,t_1}$ und $O_{j,t_2}$ definiert.

Mithilfe der Relationen $\sim_p$ und $\sim_q$ können wir nun eine Relation definieren, die genau dann wahr für zwei Spielzustände ist, wenn sie konsistent sind. Die *Konsistenzrelation* $\sim$ vergleicht zwei Spielzustände $s_{i,t_1} = (P_{i,t_1}, O_{i,t_1})$ und $s_{j,t_2} = (P_{j,t_2}, O_{j,t_2})$ und ist genau dann wahr, wenn die Zustände aller Spielfiguren

und Operationen beider Spielzustände gleich sind, d.h. es gilt $s_{i,t_1} \sim s_{j,t_2} \Leftrightarrow P_{i,t_1} \sim_p P_{j,t_2} \wedge O_{i,t_1} \sim_o O_{j,t_2}$. Die Konsistenzrelation ist *reflexiv* (es gilt immer $s_{i,t} \sim s_{i,t}$) und *symmetrisch* (es gilt immer $s_{i,t_1} \sim s_{j,t_2} \Leftrightarrow s_{j,t_2} \sim s_{i,t_1}$), sie ist jedoch nicht *transitiv* (es gilt nicht $s_{i,t_1} \sim s_{j,t_2} \wedge s_{j,t_2} \sim s_{k,t_3} \Rightarrow s_{i,t_1} \sim s_{k,t_3}$), da $\sim_p$ und $\sim_o$ dies nicht sind, und ist somit keine Äquivalenzrelation. Um also die Konsistenz von mehr als zwei Spielzuständen feststellen zu können, müssen alle betrachteten Spielzustände paarweise miteinander verglichen werden. Wenn wir in den nachfolgenden Abschnitten von Gleichheit zweier Spielzustände sprechen, so meinen wir damit gleich im Sinne der Konsistenzrelation.

Abschließend soll noch begründet werden, wieso wir die Konsistenzrelation als alternativen Gleichheitsbegriff verwenden. Der Unterschied besteht darin, dass die Konsistenzrelation Abweichungen in den Bewegungswerten toleriert und damit potentiell mehr Zustände konsistent sind. In der Praxis ist dies insofern relevant, als dass minimale Abweichungen in den Bewegungswerten, insbesondere in den Positionen, nur sehr unwahrscheinlich Abweichungen in den Spielverläufen produzieren. Betrachten wir eine Situation in Saga Online mit den Spielfiguren $x_1$ und $x_2$, die zu den Instanzen $i_1$ und $i_2$ korrespondieren. Die Position von $x_1$ ist aus Sicht von $i_1$ in $(x, y)$, aus Sicht von $i_2$ jedoch in $(x+1, y+1)$, d.h. sie weicht minimal ab. Angenommen $i_1$ führt über $x_1$ einen Angriff auf $x_2$ bzw. $i_2$ aus, dann ist es extrem unwahrscheinlich, dass der Angriff bedingt durch die Abweichung unterschiedlich in beiden Instanzen verläuft. Genau diese Fälle soll die Konsistenzrelation abdecken, indem sie die betrachtete und ähnliche Situation trotz minimaler Abweichung als konsistent einstuft.

## 5.2 Verfahren

Wir beschreiben nun das Verfahren der Evaluation. Dabei halten wir uns zunächst allgemein, indem wir das Verfahren parametrisieren. Für die konkrete Durchführung der Evaluation, siehe Abschnitt 5.3. Die Idee des Verfahrens lässt sich in einem Satz wiedergeben und lautet wie folgt:

*Wie viele Sekunden dauert es, bis die Zustandskonsistenz verletzt ist?*

Wir untersuchen also die Fragestellung, wie lange das Spielgeschehen konsistent in allen Instanzen bleibt. Hierzu verwenden wir zunächst das Modell von Saga Online in seiner ursprünglichen Form und anschließend die modifizierte Variante. Ersteres bezeichnen wir fortan als *altes* Modell und letzteres als *neues* Modell. Ein Modell ist im Sinne der Fragestellung genau dann besser als ein anderes, wenn es die Zustandskonsistenz länger aufrecht hält. Idealerweise erfüllt es die Zustandskonsistenz immer.

Wir stellen realitätsnahe Spielsituationen nach, um zu gewährleisten, dass die Evaluation möglichst praxisrelevante Ergebnisse liefert. Genau genommen machen wir nichts anders, als viele kurze Runden zu spielen und die Spielzustände der einzelnen Instanzen periodisch aufzuzeichnen, um sie anschließend miteinander zu vergleichen. Zu Beginn stellen wir eine Menge von $\eta_S$ Spielern zusammen. Die Spieler sollen dabei aus möglichst unterschiedlichen und weit entfernten Regionen stammen, sodass das zu überprüfende Modell mit unterschiedlichen Latenzzeiten konfrontiert wird. Außerdem sollen die Spieler das Spiel bereits kennen und es problemlos bedienen können. Damit soll sichergestellt sein, dass sie während des Spiels keine Zeit zum Erlernen der Funktionen benötigen, was typischerweise weniger initiierte Operationen zur Folge hätte. Als nächstes legen wir die Zahl $\eta_R$ der Runden fest, die gespielt werden sollen. Jede Runde dauert dabei genau $\eta_T$ Sekunden an, sobald sie gestartet wurde.

Der genaue Ablauf des Verfahrens ergibt sich nun wie folgt:

Die Spieler werden von 1 bis $\eta_S$ durchnummeriert und damit eindeutig gekennzeichnet. Alle Spieler erhalten ihre persönlichen Zugangsdaten, mit denen sie sich beim Rootserver[1] anmelden. Sobald getan, gelangen sie in die Serverliste, von wo aus sie die Serverlobby unseres Testservers erreichen. Dort nehmen sie wie gewohnt Einstellungen an ihren Spielfiguren vor und begeben sich anschließend ins Spiel. Sobald alle Spieler im Spiel angekommen sind, verteilt der Server ein Signal, um den Start der Runde zu bestätigen. Bis dahin sind die Zustände aller Spieler identisch und erst jetzt können sie sich ändern. Die Runde dauert nun genau $\eta_T$ Sekunden an. Alle Spieler erstellen zu jeder Sekunde eine Momentaufnahme ihres Spiels, d.h. sie berechnen ihre Spielzustände $s_{i,t}$ mit $t \in \{1, ..., \eta_T\}$. Nach Ablauf der Runde gelangen sie zurück in die Serverlobby, von wo aus die nächste Runde gestartet wird, sodass insgesamt $\eta_R$ Runden gespielt werden. Letztlich erzeugt jeder Spieler $i \in \{1, ..., \eta_S\}$ eine Zustandsmenge $S_{i,r} := \{s_{i,t} \mid t \in \{1, ..., \eta_T\}\}$ für jede Runde $r \in \{1, ..., \eta_R\}$. All diese Mengen werden an eine zentrale Stelle gesandt, von wo aus für jede Runde überprüft wird, nach wie vielen Sekunden mindestens zwei Zustände inkonsistent sind. Hierfür wird die Konsistenzrelation aus Abschnitt 5.1 verwendet. Das komplette Verfahren wird zweimal durchlaufen, einmal für das alte Modell und einmal für das neue Modell. Beim Durchlauf mit dem neuen Modell werden außerdem die Anzahl der Synchronisierungen in den jeweiligen Runden aufgezeichnet (siehe hierzu Kapitel 4, Abschnitt 4.4).

## 5.3 Durchführung

Für die Evaluation wurden insgesamt zehn Personen aus vier Ländern versammelt (d.h. $\eta_S = 10$). Alle Personen wurden eingehend in das Spiel eingearbeitet und sind bestens mit den Spielfunktionen vertraut. Die Herkunftsländer der Personen liegen zum Teil über 6000 km voneinander entfernt, womit hohe Latenzzeiten garantiert sind. Wenn wir im Folgenden von Latenzzeiten sprechen, so meinen wir die Zeit, die eine Nachricht von einem Rechner zu einem anderen im Netzwerk benötigt. Mit RTT (*Round Trip Time*) dagegen bezeichnen wir die Zeit, die eine Nachricht von einem Rechner zu einem anderen und wieder zurück benötigt. Alle Zeiten werden in Millisekunden angegeben. Abbildung 5.1 zeigt einen Ausschnitt einer Weltkarte, in dem Europa und Teile von Nordamerika zu sehen sind. Jeder Teilnehmer ist durch einen Punkt markiert. Die roten Pfeile stellen die Verbindungen zwischen den Teilnehmern und unserem Testserver dar, der sich in Frankfurt am Main in Hessen befindet.

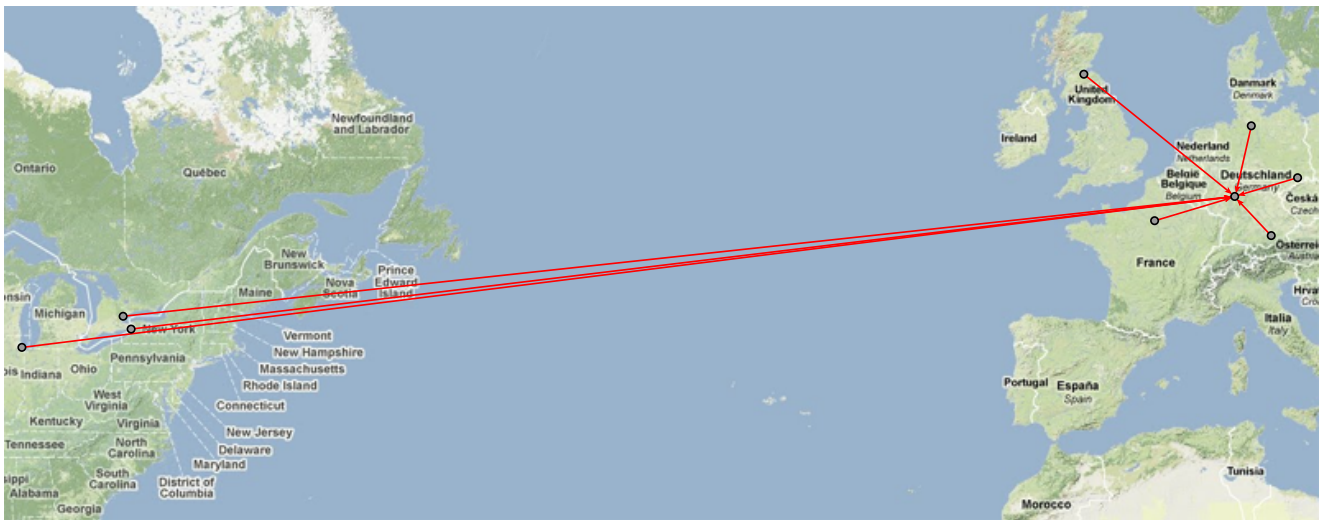


**Abbildung 5.1:** Weltkarte mit den Positionen aller Teilnehmer der Evaluation (Quelle: Google Maps)

[1] Für *Rootserver*, *Serverliste* und *Serverlobby*, siehe Kapitel 2, Abschnitt 2.4 und 2.2.

Tabelle 5.1 enthält die Aufenthaltsorte der Teilnehmer und führt die durchschnittlichen RTT-Zeiten vom jeweiligen Teilnehmer zu unserem Testserver auf. Die Teilnehmer wurden derat durchnummeriert, dass eine höhere Nummer gleichbedeutend mit einer höheren Latenzzeit zum Server ist. Der Tabelle kann bereits entnommen werden, dass sich die Länge der künstlichen Verzögerung im neuen Modell (d.h. der Wert von $t_{lat}$; siehe Kapitel 3, Abschnitt 3.4.2) auf etwa 60 ms belaufen wird. Mit einer Aggregatlänge von mindestens 50 ms bewegen wir uns hier bereits in einem Reaktionsbereich, der vom Menschen theoretisch wahrnehmbar ist (siehe z.B. [3]).

| # | **Stadt** | **RTT** |
|---|---|---|
| **1** | Frankfurt (GER) | $\approx < 1$ ms |
| **2** | Darmstadt (GER) | $\approx$ 2 - 4 ms |
| **3** | München (GER) | $\approx$ 6 - 8 ms |
| **4** | Paris (FRA) | $\approx$ 13 - 15 ms |
| **5** | Dresden (GER) | $\approx$ 18 - 20 ms |
| **6** | Hamburg (GER) | $\approx$ 20 - 22 ms |
| **7** | Edinburgh (GBR) | $\approx$ 29 - 31 ms |
| **8** | Toronto (CAN) | $\approx$ 100 - 105 ms |
| **9** | Buffalo (USA) | $\approx$ 109 - 112 ms |
| **10** | Chicago (USA) | $\approx$ 115 - 118 ms |

**Tabelle 5.1:** Herkunft und RTT-Zeiten aller Teilnehmer der Evaluation

Wir setzen $\eta_R = 30$ und $\eta_T = 60$, es werden also 30 Runden gespielt, wovon jede insgesamt 60 Sekunden andauert. Ersteren Wert halten wir im Rahmen dieser Arbeit für ausreichend, um Unterschiede zwischen den Modellen heraus zu stellen. Letzterer Wert ergibt sich aus früheren Tests basierend auf dem alten Modell, bei denen typischerweise schon nach 40 bis 50 Sekunden Inkonsistenzen festgestellt wurden (durch Sichtung der Spielzustände). Die Runden werden auf einer Spielwelt gespielt, die gerade nur so groß wie der Spielausschnitt selbst ist und zudem keine Hindernisse enthält (vergleichbar mit Abbildung 2.5). Hierdurch kommt es schon nach kurzer Zeit zum Kontakt zwischen den Spielern, womit tendenziell mehr Aktionen durchgeführt werden. Alle Teilnehmer spielen gegeneinander.

| # | t | i | j | # | t | i | j |
|---|---|---|---|---|---|---|---|
| **1** | 7 | 10 | 8 | **16** | 8 | 7 | 5 |
| **2** | 11 | 4 | 10 | **17** | 12 | 10 | 8 |
| **3** | 8 | 6 | 9 | **18** | 14 | 6 | 9 |
| **4** | 16 | 7 | 8 | **19** | 10 | 8 | 9 |
| **5** | 13 | 10 | 9 | **20** | 9 | 10 | 7 |
| **6** | 7 | 10 | 3 | **21** | 17 | 4 | 10 |
| **7** | 11 | 7 | 9 | **22** | 9 | 9 | 5 |
| **8** | 10 | 5 | 10 | **23** | 15 | 3 | 10 |
| **9** | 17 | 8 | 9 | **24** | 17 | 8 | 9 |
| **10** | 10 | 2 | 6 | **25** | 11 | 5 | 7 |
| **11** | 9 | 9 | 10 | **26** | 13 | 10 | 8 |
| **12** | 15 | 6 | 8 | **27** | 10 | 5 | 10 |
| **13** | 19 | 9 | 4 | **28** | 7 | 2 | 9 |
| **14** | 17 | 8 | 10 | **29** | 13 | 10 | 5 |
| **15** | 11 | 10 | 9 | **30** | 18 | 8 | 6 |

**Tabelle 5.2:** Ergebnisse der Evaluation mit altem Modell

Tabelle 5.2 zeigt die Ergebnisse der Evaluation mit dem alten Modell. Die erste Spalte (#) enthält die Rundennummer. Die zweite Spalte (t) enthält die Anzahl Sekunden, nach denen mindestens zwei Spielzustände inkonsistent waren. Die zwei folgenden Spalten (i und j) enthalten die Nummern der beiden Spieler, deren Spielzustände zum Zeitpunkt $t$ inkonsistent waren. Wenn es mehr als zwei inkonsistente Spielzustände gab, so sind diejenigen Spieler aufgeführt, deren Zustände am weitesten voneinander abgewichen sind. Formal bezeichnet jede Zeile eine Zeit $t$ und diejenigen Zustände $s_{i,t}$ und $s_{j,t}$, für die zuerst $s_{i,t} \not\sim s_{j,t}$ galt (in der jeweiligen Runde).

Keine Runde war länger als 19 Sekunden lang zustandskonsistent, im Durchschnitt sind bereits nach 12 Sekunden Inkonsistenzen aufgetreten. Die Zustände wichen teilweise so stark voneinander ab, dass die Inkonsistenzen selbst durch Sichtung problemlos erkannt werden konnten. Ein richtiger Spielfluss konnte hier niemals entstehen, da sich die Spielzustände zu schnell und zu stark verselbstständigten. Auffällig ist auch die Tatsache, dass Spieler mit höheren Nummern häufiger in der Tabelle enthalten sind, als solche mit niedrigeren Nummern. Dies ist auf die sofortige Ausführung der Operationen im alten Modell zurück zu führen, die in Kombination mit hohen Latenzzeiten leicht zu inkonsistenten Zuständen führen kann. Wenn beispielsweise Spieler 10 in Chicago eine Operation ausführt, so kommt diese erst nach ungefähr 115 ms bei Spieler 9 in Buffalo an. Wenn Spieler 2 in Darmstadt unmittelbar nach Spieler 10 eine Operation ausführt, so kommt diese schon nach ungefähr 60 ms bei Spieler 9 an, d.h. noch vor derjenigen von Spieler 10. Tatsächlich wurde die Operation aber erst nach derjenigen von Spieler 10 ausgeführt. Folglich besteht die erhöhte Wahrscheinlichkeit, dass die Zustände der Spieler 9 und 10 (und zudem aller anderen) inkonsistent werden.

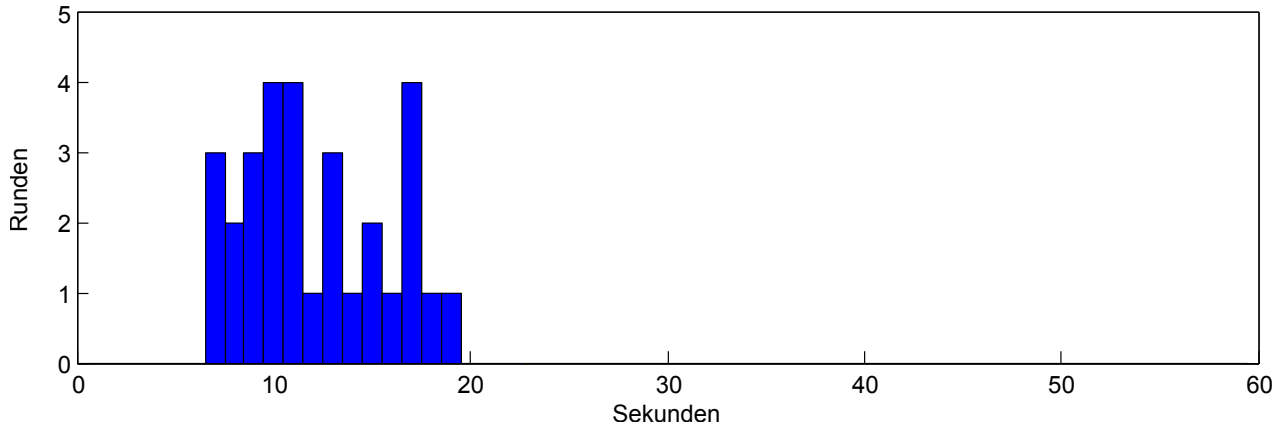


**Abbildung 5.2:** Verteilung der Spieldauer nach Evaluation mit altem Modell

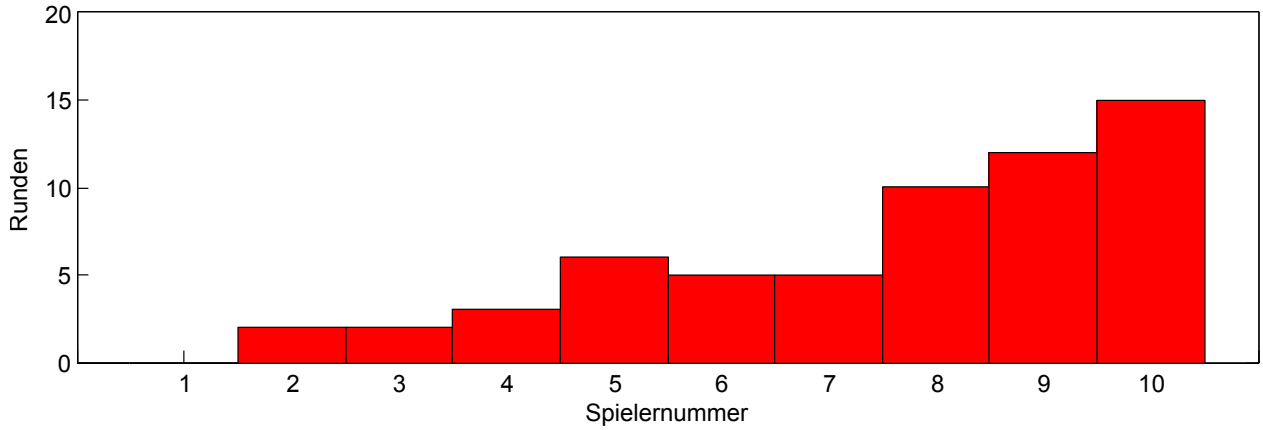


**Abbildung 5.3:** Verteilung der Inkonsistenzen nach Evaluation mit altem Modell

Abbildung 5.2 zeigt die Verteilung der Spieldauer basierend auf den Ergebnissen der Evaluation mit dem alten Modell. Die Spieldauern rangieren zwischen minimal 7 und maximal 19 Sekunden. Am häufigsten kamen Spieldauern von 10, 11 und 17 Sekunden zustande, wobei das Diagramm eine Verdichtung zwischen 9 und 11 Sekunden andeutet. Abbildung 5.3 zeigt die Anzahl von Vorkommnissen, in denen Spieler die ersten Inkonsistenzen aufwiesen. Das Diagramm verdeutlicht die Tatsache, dass Spieler mit höheren Latenzzeiten prinzipiell anfälliger für Inkonsistenzen sind. Spieler 10 stellt mit 15 Vorkommnissen das Maximum dar, während Spieler 1 keinen einzigen inkonsistenten Spielzustand hatte.

| # | t | i | j | sync | # | t | i | j | sync |
|---|---|---|---|---|---|---|---|---|---|
| **1** | - | - | - | - | **16** | - | - | - | - |
| **2** | - | - | - | - | **17** | 54 | 3 | 10 | - |
| **3** | - | - | - | 1 | **18** | - | - | - | - |
| **4** | - | - | - | - | **19** | - | - | - | - |
| **5** | 47 | 5 | 2 | - | **20** | - | - | - | - |
| **6** | - | - | - | - | **21** | - | - | - | - |
| **7** | - | - | - | - | **22** | - | - | - | - |
| **8** | - | - | - | - | **23** | - | - | - | 1 |
| **9** | - | - | - | - | **24** | - | - | - | - |
| **10** | - | - | - | - | **25** | - | - | - | - |
| **11** | - | - | - | - | **26** | - | - | - | - |
| **12** | - | - | - | 1 | **27** | - | - | - | - |
| **13** | - | - | - | - | **28** | - | - | - | - |
| **14** | - | - | - | - | **29** | 33 | 7 | 9 | 2 |
| **15** | - | - | - | - | **30** | - | - | - | - |

**Tabelle 5.3:** Ergebnisse der Evaluation mit neuem Modell

Tabelle 5.3 enthält die Ergebnisse der Evaluation mit dem neuen Modell. Die zusätzliche Spalte (sync) enthält die Anzahl der durchgeführten Synchronisationen in einer Runde. Alle Sequenzen des Modells hatten dieselbe Länge von 50 ms. Die durchschnittliche maximale Latenzzeit lag immer zwischen 60 und 70 ms. Insgesamt hatten Operationen damit eine maximale Reaktionszeit[2] von bis zu 120 ms für Spieler 1 und bis zu 190 ms für Spieler 10. Die erhöhte Reaktionszeit wurde zwar von einigen Spielern hin und wieder bemerkt, hatte ansonsten jedoch keinen weiteren Einfluss auf das Spiel.

Beinahe alle Runden waren für die betrachtetet Zeit von 60 Sekunden zustandskonsistent. Nur in wenigen Ausnahmen wurden Inkonsistenzen entdeckt, den Spielern selbst sind diese jedoch nie aufgefallen, da die betroffenen Spielzustände nur geringfügig voneinander abwichen. Die Spielzustände mussten nur in wenigen Fällen synchronisiert werden, und dann zumeist nur ein einziges Mal. Den Spielern ist dies nach eigenen Angaben nicht störend aufgefallen, da der Vorgang zur Synchronisierung meist in wenigen Sekunden abgeschlossen war. Zwischenzeitlich wurde die Länge der Sequenzen von 50 ms auf 100 ms angehoben, was den Spielern jedoch nur teilweise negativ aufgefallen ist. Das Spiel war also auch mit einer maximalen Reaktionszeit von 170 ms für Spieler 1 bis 240 ms für Spieler 10 noch spielbar.

[2] Die maximale Reaktionszeit für eine Operation eines Spielers ergibt sich aus der durchschnittlich maximalen Latenzzeit des Spielers für den Hinweg der Operation zum Server, der Länge der zugeordneten Sequenz, und schließlich der durchschnittlich maximalen Latenzzeit aller Spieler ($t_{lat}$) für den Rückweg zum Spieler.

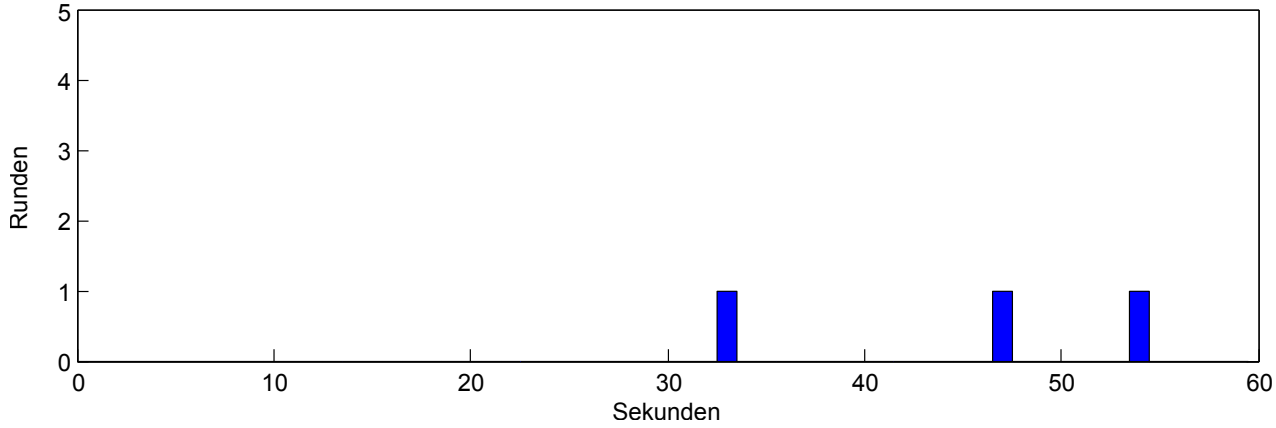


**Abbildung 5.4:** Verteilung der Spieldauer nach Evaluation mit neuem Modell

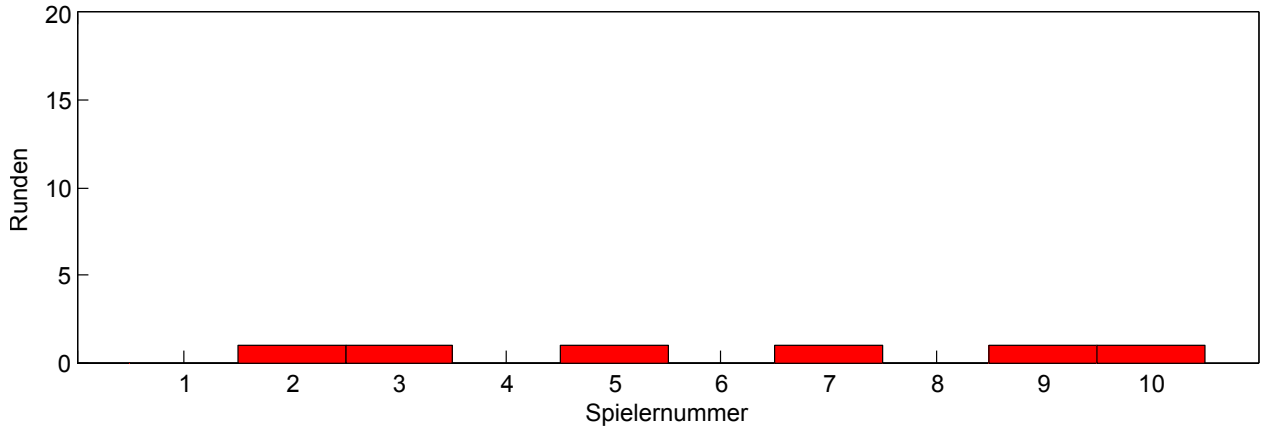


**Abbildung 5.5:** Verteilung der Inkonsistenzen nach Evaluation mit neuem Modell

Abbildung 5.4 zeigt die Verteilung der Spieldauer basierend auf den Ergebnissen der Evaluation mit dem neuen Modell. Die wenigen inkonsistenten Spielzustände kamen erst nach 33, 47 und 54 Sekunden auf, was im Vergleich zum alten Modell deutlich später ist. Beinahe alle Runden blieben für die betrachtete Zeit von 60 Sekunden zustandskonsistent. Abbildung 5.5 zeigt wieder die Anzahl von Vorkommnissen, in denen Spieler die ersten Inkonsistenzen aufwiesen. Im Vergleich zum alten Modell fällt zunächst auf, dass wesentlich weniger Spieler an inkonsistenten Spielzuständen beteiligt waren. Weiterhin ist zu erkennen, dass höhere Latenzzeiten nicht zwangsläufig zu mehr Inkonsistenzen führen. Dies resultiert aus den gemeinsamen Ausführungszeiten der Operationen, die derart gewählt sind, dass auch Instanzen mit höheren Latenzzeiten die Möglichkeit haben, die Operation rechtzeitig auszuführen.

Die Evaluation zeigt deutlich, dass das neue Modell im Sinne unser anfänglichen Fragestellung besser als das alte Modell ist, da wesentlich weniger Inkonsistenzen auftreten. Die Zustände mussten zudem nur selten synchronisiert werden, womit dies nicht weiter negativ ins Gewicht fällt. Insgesamt können wir die im Rahmen dieser Arbeit unternommenen Korrekturen am Modell als erfolgreich bezeichnen.

## 6 Zusammenfassung

Ziel dieser Arbeit war es, das Datenkommunikationsmodell einer zeitkontinuierlichen verteilten Anwendung hinsichtlich Zustandskonsistenz und Fehlertoleranz zu untersuchen und die vorhandenen Defizite mittels geeigneter Verfahren auszumerzen. Als *verteilte Anwendung* bezeichneten wir hierbei solche Anwendungen, die mehrfach instantiiert auf unterschiedlichen Systemen laufen, wobei die Systeme über ein Netzwerk miteinander verbunden sind. Jede *Instanz* verwaltet ihren eigenen Zustand. Wir bezeichneten eine verteilte Anwendung als *zeitdiskret*, wenn sich die Zustände ihrer Instanzen nur als Antwort auf Benutzerinteraktionen änderten. Falls sich die Zustände zusätzlich im Laufe der Zeit änderten, so bezeichneten wir sie als *zeitkontinuierlich*. Die Zustände zweier Instanzen waren *konsistent*, falls eine bestimmte Teilmenge beider Zustände identisch oder äquivalent im Sinne einer gegebenen Relation war. Wir bezeichneten eine verteilte Anwendung als *zustandskonsistent*, wenn die Menge der Zustände aller ihrer Instanzen immer konsistent war. Wir bezeichneten die Anwendung weiterhin als *fehlertolerant*, wenn sie bestimmte Fehler wie etwa den Ausfall eines Systems tolerieren konnte.

Die in dieser Arbeit untersuchte Anwendung ist das Java-basierte Netzwerkspiel *Saga Online*. Die Architektur des Spiels basiert auf einem zentralen Server, der die Operationen der Instanzen untereinander verteilt. Das Modell des Spiels wurde zunächst hinsichtlich Zustandskonsistenz untersucht, die wir hierfür mithilfe des *starken Konsistenzkriteriums* definiert hatten. Die Analyse hatte gezeigt, dass das Modell in seiner ursprünglichen Form nicht zustandskonsistent ist. Das Modell wurde daraufhin um ein Verfahren für die zeitliche Ordnung von Operationen *nach* Empfang beim Server erweitert. Das Verfahren sah eine *künstlich verspätete Ausführungszeit* für Operationen vor, sodass sie zeitgleich von allen Instanzen der Anwendung ausgeführt werden konnten. Das Modell wurde anschließend noch um ein Verfahren für die zeitliche Ordnung von Operationen *vor* Empfang beim Server erweitert. Hierbei wurden Operationen in *geordneten Aggregaten* gesammelt, die eine korrekte zeitliche Reihenfolge ihrer Operationen garantierten. Da beide Verfahren synchronisierte Uhren voraussetzten, wurde das Modell zu aller erst um ein Verfahren für die *Uhren-Synchronisation* ergänzt. In einer abschließenden Postanalyse konnte letztlich gezeigt werden, dass das derart modifizierte Modell zustandskonsistent ist.

Das Modell wurde anschließend hinsichtlich Fehlertoleranz untersucht. Die Eigenschaft wurde anhand sechs *Fehlerfälle* und ihren *Nachbedingungen* definiert, darunter der Ausfall des Servers und verspätete oder ausgefallene Nachrichten vom oder zum Server. Eine Anwendung wurde als fehlertolerant bezeichnet, wenn sie die Fehlerfälle derart behandelte, dass ihre Nachbedingungen erfüllt waren. In einer anfänglichen Analyse wurde gezeigt, dass das Modell mit den bisherigen Modifikationen bereits vier der sechs Fehlerfälle korrekt im Sinne unserer Definition von Fehlertoleranz behandelte. Nicht behandelt wurden verspätete oder ausgefallene Nachrichten, die vom Server verschickt wurden. Folglich wurde ein Verfahren eingeführt, dass die beiden Fehlerfälle mithilfe von *Sequenznummern* erkennen konnte und bei Bedarf ein *Synchronisierungsverfahren* einleitete, um die Zustandskonsistenz wiederherzustellen. Zudem wurde das Konzept der *Konsistenzrelevanz* eingeführt, mit welcher die Anzahl der benötigten Synchronisationen prinzipiell verringert wurde. Auch hier konnte in einer abschließenden Postanalyse gezeigt werden, dass das derart modifizierte Modell vollständig fehlertolerant ist.

Das modifizierte Modell wurde schließlich auf Praxistauglichkeit überprüft. Das Verfahren der Evaluation sah vor, dass eine Gruppe von Teilnehmern mehrere Runden spielte, wobei die *Spielzustände* der Teilnehmer sekundenweise aufgezeichnet wurden. Die Zustände wurden dann mithilfe einer *Konsistenzrelation* verglichen um herauszufinden, nach wie vielen Sekunden die ersten Inkonsistenzen auftraten. Das Ergebnis der Evaluation zeigte deutlich, dass die neuen Verfahren gut funktionieren.

# 7 Ausblick

Abschließend soll ein Ausblick auf offene Probleme und interessante Fragestellungen gegeben werden, die aus zeitlichen oder organisatorischen Gründen nicht im Rahmen dieser Arbeit untersucht wurden.

**Beziehung zw. künst. Verzögerung und Aggregatlänge** Ungeklärt ist die Frage, ob und inwiefern die künstliche Verzögerung aus der zeitlichen Ordnung von Operationen *nach* Empfang beim Server mit der Aggregatlänge aus der zeitlichen Ordnung von Operationen *vor* Empfang beim Server in Verbindung steht. Eine Antwort auf diese Frage könnte die Wahl der beiden Werte regeln.

**Optimale und automatische Aggregatlänge** Interessant ist auch die Frage, welche Aggregatlängen in welchen Situationen optimal sind. Eine Antwort auf diese Frage hätte zur Folge, dass die Anwendung zur Laufzeit stets einen optimalen Wert für die Aggregatlänge bestimmten könnte. Dies wiederum würde die Anzahl der Inkonsistenzen nochmals verringern.

**Formale Definition von Fehlertoleranz** Die Definition von Fehlertoleranz in dieser Arbeit stützt sich auf die Beschreibung von sechs Fehlerfällen und ihren Nachbedingungen. Neben dieser nicht ganz formalen Definition könnte man die Eigenschaft auch vollständig formal definieren, womit sie sich formal beweisen ließe. Ein möglicher Ansatz hierfür findet sich in [6].

**Korrumpierte Nachrichten tolerieren** Nachteilig ist auch die Beschränktheit der Definition von Fehlertoleranz dieser Arbeit. Nicht betrachtet werden beispielsweise korrumpierte Nachrichten, d.h. solche Nachrichten, die auf dem Weg vom Sender zum Empfänger versehentlich oder absichtlich beschädigt wurden. Es handelt sich also auch um einen sicherheitsrelevanten Aspekt.

**Kombination mit Timewarp** Das in Kapitel 4 eingeführte Verfahren leitet einen Synchronisierungsvorgang ein, um verspätete oder ausgefallene Nachrichten vom Server zu korrigieren. Ein Nachteil hieran ist, dass alle Instanzen pausiert werden. In Kombination mit dem Timewarp-Algorithmus (siehe Kapitel 4, Abschnitt 4.4.2) wären diese Pausen nicht länger notwendig.

## Literaturverzeichnis


[1] Marktzahlen Computer- und Videospiele, Gesamtjahr 2009. Bundesverband Interaktive Unterhaltungssoftware e.V., 2010. www.biu-online.de.

[2] E. J. Berglund and D. R. Cheriton. Amaze: A Multiplayer Computer Game. *IEEE Software*, 2(1):30–39, May 1985.

[3] S. Card, T. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

[4] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[5] C. Diot and L. Gautier. A Distributed Architecture for Multiplayer Interactive Applications on the Internet. *IEEE Network*, 13(4), July 1999.

[6] K. Echtle. *Fehlertoleranzverfahren*. Springer-Verlag, 1990.

[7] Institute for Electrical and Electronics Engineers. IEEE Standard for Distributed Interactive Simulation - Application Protocols. *IEEE Std 1278.1-1995*, September 1995.

[8] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[9] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-Lag and Timewarp: Providing Consistency for Replicated Continuous Applications. *IEEE Transactions on Multimedia*, 6(1), February 2004.

[10] A. Pope. The SIMNET Network and Protocols. Technical Report 7102, BBN Systems and Technologies, Cambridge, Massachusetts, July 1989.

[11] J. Postel. RFC 791: Internet Protocol. Internet Engineering Task Force, September 1981.

[12] B. Schneiderman. Response Time and Display Rate in Human Performance with Computers. *ACM Computing Surveys (CSUR)*, 16(3):265–285, September 1984.

[13] A. S. Tanenbaum and M. Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2nd edition, 2006.

[14] S. Teal and A. Rudnicky. A Performance Model of System Delay and User Strategy Selection. *Proc. Human Factors in Computing Systems*, pages 295–305, May 1992.

[15] J. Vogel, M. Mauve, V. Hilt, and W. Effelsberg. Late Join Algorithms for Distributed Interactive Applications. *Multimedia Systems*, 9(4):327–336, 2003.