

# A Network-Agnostic and Cheat-Resistant Framework for Multiplayer Online Games



Vom Fachbereich Informatik  
der Technischen Universität Darmstadt  
genehmigte

## DISSERTATION

zur Erlangung des akademischen Grades  
eines Doktors der Ingenieurwissenschaften (Dr.-Ing.)

von

**Dipl.-Inform. Patric Kabus**

geboren in Dieburg

Referenten:

Prof. Alejandro P. Buchmann, PhD, TU-Darmstadt  
Prof. Dr. Bettina Kemme, McGill University

Tag der Einreichung: 22.04.2009  
Tag der mündlichen Prüfung: 15.05.2009  
Darmstadt 2009

D17



## **Erklärung**

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades "Dr.-Ing." mit dem Titel "A Network-Agnostic and Cheat-Resistant Framework for Multiplayer Online Games" selbständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, den 22.04.2009

Patric Kabus



## **Akademische Laufbahn**

**1984 - 1988** Wendelinusschule Klein-Umstadt

**1988 - 1997** Max-Planck-Gymnasium Groß-Umstadt

**1998 - 2003** Studium der Informatik an der TU Darmstadt

**2003 - 2008** Wissenschaftlicher Mitarbeiter an der TU Darmstadt



*To my wife and my family.*



## Zusammenfassung

”Kinderkram” mag der erste Gedanke vieler sein, wenn es auf das Thema Computer- und Videospiele kommt. Doch das Wachstum der Branche und ihre Verkaufszahlen sprechen da eine ganz andere Sprache: Die weltweiten Umsätze haben längst zweistellige Milliardenbeträge erreicht und von den Wachstumsraten können andere Branchen nur träumen. Alleine in den USA wurden im Jahr 2007 18,8 Milliarden US-Dollar mit Soft- und Hardware für Computer- und Videospiele umgesetzt. Dies bedeutet eine Steigerung um vierzig Prozent gegenüber dem Vorjahr. Analysten sagen voraus, dass dieser Industriezweig in absehbarer Zeit sogar die Umsatzzahlen der Musikindustrie übertreffen wird.

Mit der zunehmenden Bedeutung der weltweiten Vernetzung über das Internet steigt auch der Anteil sogenannter ”Online-Spiele”. Bei dieser Art von Spielen können sich Teilnehmer, die über die ganze Welt verteilt sind, zum gemeinsamen Spielen über das Internet miteinander verbinden. Auch die Zukunftsaussichten solcher Online-Spiele sind glänzend: bis 2011 soll der weltweite Umsatz auf über 13 Milliarden US-Dollar ansteigen. Die kommerziell wohl erfolgreichste Art von Online-Spielen sind die sogenannten ”Massively Multiplayer Online Games (MMOGs)”. Dieses Genre bietet riesige virtuelle Spielwelten, in denen tausende von Spielern gleichzeitig interagieren können. Dazu erschaffen sie individuelle virtuelle Avatare, die in Anlehnung an reale Personen Eigenschaften und Fähigkeiten entwickeln sowie Besitztümer anhäufen können. Die Spielwelten sind rund um die Uhr verfügbar, ein Spieler kann sie jederzeit mit einem Avatar betreten. Anders als bei anderen Spielgenres gibt es kein vorgegebenes Ziel nach dessen Erreichen das Spiel zuende ist. Stattdessen existieren die virtuellen Welten oft über viele Jahre hinweg und binden somit die Spieler langfristig. Der erfolgreichste Vertreter der MMOGs ist zur Zeit ”World of Warcraft”, der Anfang 2008 über zehn Millionen Teilnehmer weltweit vorweisen konnte und damit einen Marktanteil von über 62 Prozent innehatte. Die Teilnahme an diesen Spielen wird in der Regel über Abonnements realisiert, für die monatliche Beträge von bis zu 15 US-Dollar erhoben werden.

Die Entwicklung von heutigen Computer- und Videospiele ist eine komplexe und kostenintensive Herausforderung. Im Jahr 2008 hat das erste Videospiel die Grenze von 100 Millionen US-Dollar an Entwicklungskosten überschritten. Zusätzlich müssen Anbieter von Online-Spielen die notwendige Infrastruktur bereitstellen und betreiben, damit ein Spiel über das Internet gespielt werden kann. Traditionell werden diese Spiele als Client/Server-Architektur realisiert. Der Client dient dabei nur als eine Art Terminal, das die Spielwelt audiovisuell darstellt und Kommandos des Spielers entgegennimmt um sie an den Server zu schicken. Alle notwendigen Berechnungen um diese Kommandos zu verarbeiten und den Zustand der Spielwelt zu verwalten werden auf dem Server durchgeführt. Um hunderte oder gar tausende von Spielern in einer Spielwelt unterzubringen sind leistungsfähige Rechner und breitbandige Internetverbindungen notwendig. Dazu kommt ein erheblicher Personalaufwand für das Betreiben der Server, das Erstellen von Softwareupdates sowie Kundenservice und Abonnementverwaltung. Beispielsweise liefen für World of Warcraft seit dem Start im November 2004 rund 200 Millionen US-Dollar an Kosten an.

Zusätzlich zu dem Aufwand, den Betrieb eines Online-Spiels aufrecht zu erhalten, kommt eine weitere Herausforderung hinzu: das Spiel frei von Betrugern, sogenannten "Cheatern" zu halten. Als Cheater bezeichnet man Spieler, die sich unfaire Vorteile gegenüber anderen Spielern verschaffen. Dies hat erheblichen Einfluß auf das Spielerlebnis ehrlicher Spieler und damit letztendlich auch auf den kommerziellen Erfolg eines Spiels. Ehrliche Spieler werden durch Cheater benachteiligt, was dazu führt, dass sie häufig ihre Abonnements kündigen. Die Betreiber von MMOGs gehen in der Regel hart gegen Cheater vor und zögern nicht diese sofort vom Spiel auszuschließen. Beispielsweise hat Blizzard Entertainment, der Betreiber von World of Warcraft, im Jahr 2006 innerhalb eines einzigen Monats 59.000 Spieler wegen Cheatings des Spiels verwiesen.

In dieser Arbeit stellen wir eine Netzwerkarchitektur für Online-Spiele vor, die darauf abzielt, die Kosten für das Bereitstellen der notwendigen Dienste erheblich zu senken. Dies geschieht, indem die benötigte Rechenzeit und Bandbreite nicht mehr vom Server, sondern von den Clients, d.h. den Rechnern der Spieler, bereitgestellt wird. In der Regel verfügen Spieler über sehr leistungsfähige Hardware, die bei Online-Spielen bislang nicht voll ausgelastet wird, da die Spielwelt vollständig auf dem Server verwaltet wird. Weiterhin sind Spieler häufig über breitbandige Verbindungen an das Internet angeschlossen. Unsere Architektur nutzt diese Ressourcen indem sie die Verwaltung der Spielwelt auf die Clients verlagert. Zu diesem Zweck wird die Spielwelt in kleinere Regionen, deren Verwaltung von einem einzelnen Spielerrechner bewältigt werden kann, unterteilt. Ein Spieler, dessen Avatar sich in einer bestimmten Region befindet, verbindet sich mit dem Rechner, der für die Verwaltung dieser Region zuständig ist. Der Spielbetreiber muss nun nur noch Dienste bereitstellen, die verhältnismäßig wenig Ressourcen in Anspruch nehmen. Zum einen wird ein Dienst benötigt, der die Spielregionen den Clients zur Verwaltung zuweist. Dieser Dienst kann gleichzeitig als Zutrittspunkt zum System fungieren, der jeden Spieler zu dem Rechner weiterleitet, der gerade für seine Region zuständig ist. Weiterhin sollte die Abonnementverwaltung nur von einem vertrauenswürdigen Server durchgeführt werden, da hier sensible Daten gespeichert sind.

Die gerade beschriebene Netzwerkarchitektur wird in ein Framework integriert, das netzwerkspezifischen Programmcode vor dem Spielentwickler verbirgt. Das vermindert die Komplexität des Entwicklungsprozess' erheblich und damit auch die verbundenen Kosten. Gleichzeitig wird die Wiederverwendbarkeit deutlich gesteigert. Die Abstraktion vom Netzwerk wird über das Publish/Subscribe-Paradigma erreicht. Das Framework sorgt dafür, dass Änderungen des Spielstandes über eine Publikation automatisch zu den Rechnern verteilt werden, die an dieser Änderung interessiert sind. Auf diese Weise wird der Zustand des Spiels auf allen Knoten konsistent gehalten, ohne dass der Spielentwickler dazu manuell eingreifen muss. Das Framework abstrahiert aber nicht nur von der oben genannten Netzwerkarchitektur. Prinzipiell kann jede Architektur verwendet werden, solange die Kommunikation auf die entsprechenden Subskriptionen und Publikationen abgebildet werden kann. Momentan unterstützen wir zusätzlich die traditionelle Client/Server-Architektur und einen reinen Peer-to-Peer-Modus. Zusätzlich zur Netzwerkabstraktion beschleunigt das Framework den Entwicklungsprozess durch einen datenzentrierten Ansatz. Jeder Aspekt eines Spielobjekts — Zustand, Typ und Operationen — können aus einer externen Datei geladen und zur Laufzeit verändert werden. Dadurch wird zeitaufwändiges Neukompilieren bei Änderungen am

Spieldesign vermieden.

Weiterhin stellen wir eine Lösung vor, die regelwidrige Veränderungen am Spielstand verhindert. Dieses Problem entsteht, wenn der Zustand des Spiels nicht auf vertrauenswürdigen Servern, sondern auf den Clients der Spieler verwaltet wird. Weil diese Clients prinzipiell nicht vertrauenswürdig sind, können wir uns nicht ohne weiteres auf deren Berechnungen verlassen. Anstatt einen einzelnen Client über den Zustand einer Region entscheiden zu lassen, wird der Zustand auf mehreren Clients repliziert. Jede Replik votiert nun für einen bestimmten Zustand des Spiels und die Mehrheit entscheidet. Solange die Mehrzahl der Repliken sich regelkonform verhält, können Manipulationen dadurch verhindert werden. Der Abstimmungsprozess erfordert keine direkte Synchronisation zwischen Repliken. Dadurch wird der Kommunikationsaufwand minimiert und einzelne Repliken können den Entscheidungsprozess nicht blockieren.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation and Problem Statement . . . . .	1
1.2	Proposed Approach . . . . .	2
1.3	Contributions of this Thesis . . . . .	3
1.4	Thesis Organization . . . . .	4
<b>2</b>	<b>Related Work</b>	<b>7</b>
2.1	Distributed Gaming . . . . .	7
2.2	Network Abstraction . . . . .	10
2.3	Cheating Prevention in Online Games . . . . .	11
<b>3</b>	<b>Framework Architecture</b>	<b>15</b>
3.1	Introduction . . . . .	15
3.2	Overview . . . . .	17
3.2.1	Game Layer . . . . .	17
3.2.2	Object Interface . . . . .	17
3.2.3	Object Layer . . . . .	18
3.2.4	Network Interface . . . . .	19
3.2.5	Network Layer . . . . .	19
3.2.6	Concluding Overview . . . . .	20
3.3	Game Layer . . . . .	21
3.3.1	Scheduler . . . . .	21
3.3.2	Input Manager . . . . .	22
3.3.3	Presentation Manager . . . . .	23
3.4	Object Layer . . . . .	24
3.4.1	Game Object Model . . . . .	24
3.4.2	Object Storage and Retrieval . . . . .	27
3.4.3	Updates and Ownership Management . . . . .	28
3.4.4	Login and Logout . . . . .	28
3.4.5	Class Diagram . . . . .	28
3.5	Network Layer . . . . .	29
3.5.1	Publish/Subscribe . . . . .	29
3.5.2	Optimizations . . . . .	33
3.5.3	Class Diagram . . . . .	34
3.6	Example Game Implementation . . . . .	35
3.7	Case Study: Integrating BubbleStorm . . . . .	37
3.8	Conclusion . . . . .	41
3.8.1	Performance Impact . . . . .	41

3.8.2	Cheating . . . . .	42
<b>4</b>	<b>Resilience against Cheating</b>	<b>43</b>
4.1	Introduction . . . . .	43
4.1.1	System Classification . . . . .	43
4.1.2	Definition and Taxonomy of Cheating . . . . .	44
4.1.3	Cheating Attacks Specific to P2P Online Games . . . . .	45
4.1.4	Impact of Successful Attacks . . . . .	45
4.2	Main Concepts . . . . .	46
4.2.1	Addressing Misplaced Trust . . . . .	46
4.2.2	Addressing Lack of Secrecy . . . . .	48
4.2.3	Preventing vs. Detecting Cheating . . . . .	49
4.3	Region Replication . . . . .	49
4.3.1	Partitioning of the Game World . . . . .	49
4.3.2	Distribution of Game State and Logic . . . . .	51
4.3.3	Replica Selection . . . . .	52
4.3.4	Consistency . . . . .	53
4.3.5	Update Propagation . . . . .	55
4.4	Normal Operation . . . . .	55
4.4.1	Bootstrapping . . . . .	55
4.4.2	Game Client Login . . . . .	56
4.4.3	Game Client Logout . . . . .	56
4.4.4	Region Controller Login . . . . .	58
4.4.5	Region Controller Logout . . . . .	59
4.4.6	Player Changing Regions . . . . .	59
4.5	Cheating Attack Scenarios . . . . .	59
4.5.1	Attacks performed by Game Clients . . . . .	61
4.5.2	Attacks performed by Region Controllers . . . . .	64
4.5.3	Collusion Attacks . . . . .	66
4.5.4	Message Omission . . . . .	66
4.5.5	Region Controller Replacement . . . . .	68
4.6	Scalability . . . . .	68
4.7	General Applicability . . . . .	70
4.8	Conclusion . . . . .	71
<b>5</b>	<b>Evaluation</b>	<b>73</b>
5.1	Introduction . . . . .	73
5.2	The Request-Update Cycle . . . . .	73
5.3	Correctness . . . . .	74
5.4	Frame Execution Time . . . . .	76
5.4.1	Modeling Message Transmission Time . . . . .	76
5.4.2	Transmission Delay . . . . .	77
5.4.3	Propagation Delay . . . . .	78
5.4.4	Total Message Transmission Time . . . . .	78
5.4.5	Probability of an Inconsistency . . . . .	78
5.5	Simulation Setup . . . . .	79
5.5.1	Scenario 1 - 2 MBit ADSL Node, 25 Game Clients . . . . .	81
5.6	Adding Realism to Scenario 1 . . . . .	84
5.6.1	Clock Skew . . . . .	84
5.6.2	Node Churn . . . . .	86

5.6.3	Node crashes . . . . .	88
5.6.4	Combination of all extensions . . . . .	90
5.7	Further Scenarios . . . . .	90
5.7.1	Scenario 2 - 16 MBit ADSL Node, 25 Game Clients . . . .	91
5.7.2	Scenario 3 - 16 MBit ADSL Node, 100 Game Clients . . . .	94
5.7.3	Scenario 4 - 50 MBit VDSL Node, 100 Game Clients . . . .	97
5.7.4	Scenario 5 - 50 MBit VDSL Node, 250 Game Clients . . . .	100
5.8	Results . . . . .	102
5.9	Conclusion . . . . .	104
<b>6</b>	<b>Summary and Future Work</b>	<b>105</b>
6.1	Future Work . . . . .	106



# List of Figures

3.1	High-level overview . . . . .	17
3.2	Detailed overview . . . . .	20
3.3	Scheduler Class Diagram . . . . .	22
3.4	Input Manager Class Diagram . . . . .	23
3.5	Object Layer Class Diagram . . . . .	29
3.6	Request/update in CCS mode . . . . .	30
3.7	Request/update in RS mode . . . . .	31
3.8	Request/update in RS mode (updating peer is the object's owner)	31
3.9	Request/update in AC mode . . . . .	32
3.10	Client login in CCS mode . . . . .	32
3.11	Client login in AC mode . . . . .	33
3.12	Peer login in RS mode . . . . .	33
3.13	Network Layer Class Diagram . . . . .	35
3.14	Example game . . . . .	36
3.15	Example of a game object type definition . . . . .	38
3.16	Intersecting query and data bubbles in a BubbleStorm network .	39
3.17	Publish/Subscribe on top of BubbleStorm . . . . .	40
4.1	Game Client login procedure . . . . .	57
4.2	Game Client logout procedure . . . . .	58
4.3	Region Controller login procedure . . . . .	60
4.4	Region Controller logout procedure . . . . .	61
4.5	Player region change procedure . . . . .	62
4.6	Game Client sends forged request to all Region Controllers . . . .	63
4.7	Game Client sends forged request to a minority of Region Con- trollers . . . . .	63
4.8	Game Client sends forged request to a majority of Region Con- trollers . . . . .	64
4.9	Region Controller sends forged update to Game Clients . . . . .	64
4.10	Multiple Region Controllers send forged update to Game Clients	65
4.11	Game Client colludes with Region Controllers . . . . .	67
4.12	Region Controller replacement procedure . . . . .	69
5.1	Sequence of actions within a frame . . . . .	74
5.2	Density and distribution function for the network jitter . . . . .	83
5.3	Distribution of NTP time offset samples . . . . .	86
5.4	Distribution of player session times . . . . .	88



# List of Tables

5.1	Simulation results for scenario 1 . . . . .	84
5.2	Simulation results for scenario 1 including clock skew . . . . .	87
5.3	Simulation results for scenario 1 including node churn . . . . .	89
5.4	Simulation results for scenario 1 including node churn and node crashes . . . . .	90
5.5	Simulation results for scenario 1 including clock skew, node churn and node crashes . . . . .	91
5.6	Simulation results for scenario 2 with ideal-world setting and 20ms buffer . . . . .	93
5.7	Simulation results for scenario 2 with ideal-world setting and 25ms buffer . . . . .	94
5.8	Simulation results for scenario 2 with real-world setting and 20ms buffer . . . . .	95
5.9	Simulation results for scenario 2 with real-world setting and 25ms buffer . . . . .	95
5.10	Simulation results for scenario 3 with ideal-world setting . . . . .	97
5.11	Simulation results for scenario 3 with real-world setting . . . . .	98
5.12	Simulation results for scenario 4 with ideal-world setting . . . . .	99
5.13	Simulation results for scenario 4 with real-world setting . . . . .	100
5.14	Simulation results for scenario 5 with ideal-world setting . . . . .	102
5.15	Simulation results for scenario 5 with real-world setting . . . . .	103
5.16	Results of the different extensions of scenario 1 . . . . .	103
5.17	Results for the real-life scenarios . . . . .	104



# Chapter 1

## Introduction

### 1.1 Motivation and Problem Statement

“Video games are kids’ stuff” may be still in the minds of many people. But the video games industry is far beyond its infancy and has already grown into a multi-billion dollar business. The NPD Group reports [79] that in 2007 the revenues generated in the U.S. with video game soft- and hardware for consoles and personal computers reached a total of 18.8 billion dollars, a 40 percent increase over 2006. According to PricewaterhouseCoopers [85], the global sales will even surpass those of the music industry within the next years.

With the success of the Internet, online games are a constantly increasing part of these sales. According to DFC Intelligence [32], the worldwide online game market will grow to over 13 billion dollars in 2011. The probably most successful online game genre today is that of the so-called *Massively Multiplayer Online Games (MMOGs)*. This kind of games provides vast virtual worlds, where thousands of players can meet and interact simultaneously. Most of these worlds are persistent, i.e. they may be online for years. They are hosted on Internet servers which are online 24/7 and players can join and leave the game whenever they like to. The persistence of the game world allows for long-term development of virtual avatars with individual characteristics and possessions. The leader of the MMOG market today is Blizzard Entertainment with the title *World of Warcraft* [14]. In the beginning of 2008, World of Warcraft had 10 million subscribers (each paying up to 15 dollars per month) and a market share of 62 percent [109].

Developing today's video games is a complex and cost-intensive task and multiplayer online functionality has a significant share in this. In 2008, the first video game hit the 100 million dollar mark [100] for development costs. In addition to that, publishers of online games need to provide the necessary services to allow their customers to play the game over the Internet. Traditionally, most online games and nearly all MMOGs are built relying on the Client/Server architecture. The client software runs on the player's computers and shows only an audio-visual representation of the game world. It accepts commands issued by the player and transmits them to the server. Processing the commands and managing the state of the game is completely done on the server-side. Thus, to be able to handle hundreds or thousands of players simultaneously, large

amounts of computing power and network bandwidth are required. Additionally, the service requires a large staff for server maintenance, software updates, billing and customer services. In [62] it was revealed that the provision of the World of Warcraft service did cost about 200 million dollars since its launch in November 2004.

In addition to the effort of maintaining a multiplayer online game service after its launch there arises another challenge: keeping the game free of cheaters. A cheater may be defined as a user that performs an “action that gives an advantage over his opponents that is considered unfair by the game developer” [103]. One must be aware that cheating is a major concern in multiplayer games as it seriously affects the game experience of honest players [76]. Especially for subscription-based online games this is fatal, since customers will cancel their subscriptions if the experience doesn’t meet their expectations. Game publishers usually do not hesitate to close the accounts of players that they believe to have cheated. For example, in 2006 Blizzard Entertainment announced in their forums [80] that they have banned 59,000 players from World of Warcraft within a single month.

## 1.2 Proposed Approach

This thesis proposes a network architecture for multiplayer online games that aims at reducing the costs for providing online game services by shifting most of the computational effort and the bandwidth requirements on to the customers’ computers. In traditional Client/Server games the client acts merely as a dumb terminal which shows an audiovisual representation of the game and accepts input from the player. However, players of computer games tend to be equipped with powerful hardware and usually access the Internet via broadband connections. Thus, many of the client-side resources remain unused. Our architecture utilizes these resources by letting the player nodes carry out the management of the game state. For this purpose, the game world is partitioned into smaller sized regions which can be handled by a single player computer. Those players, whose virtual avatars are located in a certain region, connect to the corresponding node that manages the region. The game publisher has just to provide servers for tasks that have comparably low resource requirements. A central server is necessary to assign the region management tasks to player nodes. It also serves as an entry point into the system so newly joined players know which node is responsible for their region. Finally, the subscription management should only be performed by a trusted server since it handles sensitive player data (e.g. credit card data).

The architecture mentioned above is integrated into a framework that tries to hide the networking related code from a regular game developer. This reduces the complexity of the development process and thus the corresponding costs while at the same time enhances reusability. Network abstraction is achieved by applying the Publish/Subscribe paradigm [37]. Our framework automatically generates an update publication whenever the state of the game changes which is routed to the nodes that need to be informed. This way, the game state is kept consistent on all nodes of the network without the need for manual intervention by the game developer. However, the pub/sub mechanism does not only abstract from the above network architecture. In principle, any kind

of network architecture may be used as long as the message passing can be mapped onto the appropriate publications and subscriptions. We currently also support the traditional Client/Server architecture as well as a pure Peer-to-Peer one. In addition to the network abstraction, our framework tries to speed up the development process by following a data-driven approach. This means that all objects of the game's state are completely dynamic. Every aspect of a game object — state, type and operations — can be loaded from a configuration file and changed during runtime. This way, time-consuming recompilations are avoided whenever changes in the game object's design occur.

Finally, we present a solution for preventing malicious manipulations of the game state. This is a problem that arises from our proposed network architecture: the game services are now provided by untrusted player nodes instead of trusted servers run under the authority of the game publisher. Since it is not feasible to establish full trust into player nodes, we must cope with the fact that a certain fraction of the nodes may be malicious. Instead of letting a single node being responsible for managing the state of a game region, we replicate a region's state on multiple nodes. Each replica votes for its state and the majority determines the correct one. As long as the majority of nodes is honest, unfair manipulations can be prevented. Our voting procedure avoids a direct synchronization between region replicas. This way, the messaging overhead is reduced and single malicious nodes are not able to disturb the voting procedure.

### 1.3 Contributions of this Thesis

In this work we develop an easy-to-use framework for creating multiplayer online games. The main benefits of this framework include the following aspects:

- Although the focus of the framework lies on multiplayer online games, the framework is flexible enough to allow the creation of single player games as well as local area multiplayer games.
- The framework allows to use different networking modes without changing the code of the game. Currently, Client/Server, Peer-to-Peer and a hybrid anti-cheating mode are supported. Custom modes can easily be integrated with little effort.
- Game developers can focus on the actual design of the game without worrying about networking or consistency issues. They can create and manipulate game objects as if they were stored locally. Updates of game objects are automatically disseminated to the interested nodes.
- The framework is modular so any custom or off-the-shelf components (e.g. graphics, sound or physics engines) can be integrated.
- For realizing the Publish/Subscribe service that provides the network abstraction, custom implementations that are optimized for special requirements may be used.
- The framework easily integrates into a game developer's workflow. All game objects may be created and manipulated with specialized external tools. The completely dynamic and data-driven object model allows to import game objects on the fly without the need for recompiling code.

- We present an example game that demonstrates that our framework can actually be used for implementing real games.

We develop a distributed and cheat-resistant network architecture for online multiplayer games. This architecture integrates seamlessly into our framework and can be used as one of the many possible networking modes. The main contributions are:

- We identify the cheating attacks relevant for distributed online games, analyze their impact and point out the main concepts to counteract them.
- Based on these concepts, we develop a distributed gaming architecture that addresses the most important issues like consistency, replica placement and update propagation.
- We thoroughly analyze relevant attack scenarios and show how our system deals with them.

Finally, to prove that our approach is feasible under realistic network conditions, the proposed architecture underwent an evaluation. The evaluation included

- a mathematical model to estimate the probability that a voting failure will occur because of network latencies and jitter.
- a comparison of the model to the results of a simulation. The simulation covered all interactions of the participating network nodes.
- multiple realistic scenarios which all were based on real-life parameters and included clock skew, node churn and node crashes.

## 1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 gives an overview of related scientific work. We start with the general topic of distributed gaming. Next we discuss papers in the more specific areas of network abstraction for online games and addressing cheating. The overview shows that no previous research has covered all issues addressed in this thesis.

In chapter 3 we discuss the design and implementation of our framework. We start with giving a high-level overview and continue describing each layer in detail. We show how network abstraction is achieved and explain how the framework can be optimized for specific requirements. We conclude this section with a description of an example game realized using our framework.

Chapter 4 presents the distributed network architecture that provides appropriate countermeasures against cheating. We first give an introduction into the taxonomy of cheating and explain which attacks are relevant for us. We then discuss general principles to counterattack these attacks before going into the details of our approach. The flow of information within our architecture is described in detail and visualized by sequence diagrams. We conclude this chapter with analyzing concrete attack scenario and discussions on how our system deals with them.

The architecture above is evaluated in chapter 5. For this we develop an analytical model to predict the behavior of our system. This model is compared to an implementation of our network architecture using a simulation framework. For the comparison we examine five scenarios which are based on real-life parameters and realistic network conditions.

Finally, in chapter 6, we give a summary of our work and discuss possible directions of future work.



## Chapter 2

# Related Work

In this chapter we present an overview of the research that has been done in the area. We start with publications that are related to the general topic of distributed gaming. We then present work that has been done in the areas of network abstraction and cheating prevention in online games.

### 2.1 Distributed Gaming

In this section we give an overview of research projects that distribute the computational effort of managing a game among the nodes of the players. The projects mentioned in this section do not address the problem of cheating at all or declare it as future work.

*MiMaze* [66, 46, 45] labels itself to be a descendant of *Amaze* [8] and claims to be the first 3D multiplayer game designed with a distributed architecture. It follows some of the rules of the IEEE Standard for Distributed Interactive Simulation [53, 54].

In *MiMaze* each client maintains its own local view of the global game state using information received from other clients. A server is only needed when a new client joins a session. The underlying transport protocol is RTP [90] over UDP/IP multicast. Clients are synchronized via a mechanism called *bucket synchronization*. Simulation time is divided into fixed length sampling periods and a bucket is associated with each sampling period. Updates received by a player that were issued during a certain period are gathered in the corresponding bucket.

*MiMaze* has undergone a performance evaluation. It was performed with 25 clients on the Mbone [36], a virtual network on top of the Internet, that allows for multicasting. The evaluation showed that although there was a significant loss of updates (usually only about 70% of the update messages were incorporated into the calculation of current game state) and therefore inconsistencies, these losses had no visible impact on gameplay. This may be due to the high update frequency (25 updates per second) which makes a few lost updates not noticeable. Scalability seems to be a major problem in *MiMaze*. The authors didn't perform tests with more than 25 clients, but they argue that every additional clients adds about 10 kilobit/second, so the Mbone

will be saturated at about 50 clients.

*Mercury* was first introduced as a distributed content-based publish/subscribe infrastructure for Internet multiplayer games [12]. It mainly addresses the scalability issues of broadcast-based architectures like MiMaze. Broadcasting updates to all clients leads to network flooding and therefore limits the number of players. Publish/subscribe systems deliver *publications* (in this context: updates) only to clients which are interested in them, i.e. which have registered an appropriate *subscription*. For example, a player may want to receive position updates of other players only if they are within his line of vision. He would consequently register for a position update subscription that is based on his current location in the game. In *Mercury* a publication is composed of simple pairs of typed attributes and values (e.g. the x- and y-coordinates of a player's position). A subscription is a conjunction of predicates over these values. If a publication's values evaluate a subscription's predicate conjunction to "true", the publication will be routed to the appropriate subscribers. For example, if a player lingers in the region determined by the coordinates  $100 < x < 200$  and  $400 < y < 500$ , he may receive all position updates within these boundaries.

The *mercury* infrastructure is divided into hubs, each consists of multiple nodes and is responsible for a certain attribute. Any subscription will be sent to a single hub, that is responsible for one of its attributes. The choice of the hub has a significant impact on flooding because a publication will be sent to all hubs that are responsible for one of its attributes. Inside a hub the nodes are arranged logically as a circle, each node responsible for a range of the attribute's values. Every node is connected to its predecessor and successor. A publication is passed along the circle until it reaches the node whose range meets the attribute value. Subscriptions on the other hand may be routed to multiple nodes because they may match a range of attribute values. Eventually, a publication will reach the nodes where the matching subscription is stored, the "rendezvous" point. These nodes will forward the publication to the subscribers. Because hubs are organized as circles, a message that is sent to a hub with  $n$  nodes will pass through  $n/2$  nodes on average, causing a very high latency.

In [10] the focus of *Mercury* has changed to a system that supports multi-attribute range queries. But a multiplayer game (called "Caduceus") is still used as an example application. The routing has undergone further optimizations, but latency is still very high.

*SimMud* [61] is a simple P2P massively multiplayer game. It is built on top of the P2P overlay Pastry [88]. Like other overlays, Pastry provides the functionality of a distributed hash table (DHT), by mapping a given object key to a unique node in the network. Game state is disseminated using Scribe [20], a multicast infrastructure built on top of Pastry. In *SimMud* the game world is partitioned into fixed size regions. Players in the same region form an interest group so that object updates that can be seen by all players in a region are disseminated only within the group. Interactions between players are handled by direct connections. Every object has a coordinator that has authority over the object's state and therefore enforces single-copy consistency. If any player wants to manipulate the state of an object, he has to send an update to it's

coordinator. Although there is a single coordinator for every object's state, there can be any number of replicas. All updates to the object are sent to the coordinator as well as to the replicas. Whenever the coordinator fails, a replica can take over its place and become the new coordinator. Experiments with 1000 and 4000 players respectively show that most messages take less than six hops. Given a random delay between 3 and 100ms between nodes, most messages are delivered in less than 200ms. The average bandwidth requirement is 7.2KB/s, and peaks at 22.34KB/s. These figures show that the architecture is suitable for multiplayer online games over consumer broadband connections. SimMud declares cheating issues as future work. In its current version, the fact that every client is the coordinator for its own player object makes arbitrary manipulations possible.

*Colyseus* [11] is another P2P game architecture based on distributed hash tables. The objects of the game world are distributed among the nodes of the players. Each object has a single owner which serializes all operations on a primary copy while other nodes may only keep cached replicas for local access which are periodically updated. To speed up the updating process, the DHT is only used for locating the primary object. After the owning node of a game object is known, updates are propagated to the replicas using a direct connection. One of Colyseus' main features is a subsystem that allows prefetching of game objects to reduce latencies. This prefetching is controlled by the interest management system. Usually, a player node is only interested in game objects that are in the interaction range of the player's avatar. Colyseus tries to discover the primary copies of objects before they get into this range. This way, the delay until a local replica is available is hidden from the player. The authors adapted the commercial first-person-shooter Quake 2 [52] and showed that their architecture can handle even fast-paced games very well. The authors declare cheating as future work. Currently, nodes can tamper with primary copies they own, withhold updates of game objects or receive updates of objects that should not be available to them.

*Mediator* [38] adopts a hybrid communication architecture for multiplayer online games. In the peer node bootstrapping process, a structured P2P overlay is used. The game world is split into zones and their structure is maintained using an application layer multicast. Finally, time critical events are transmitted over direct connections between peer nodes. The major contribution of this work is that multiple super-peer roles (called mediators) are used to perform the different management tasks of a multiplayer online game. The boot mediator is the peer node that is closest to a zone in the P2P overlay and handles the bootstrapping of new nodes. Distributed resource discovery and interest management are performed by their own mediators. Zone mediators are responsible for balancing out the workload among super-peers that manage the game zones. The authors argue that the framework is extensible and new mediator roles can easily be introduced according to additional requirements.

There also exist variations of common consistency models and implemen-

tations more specific for distributed games. *Rendezvous* [25] is a decentralized consistency management mechanism that is targeted at multiplayer games in high latency environments. A key feature is that it always maintains a certain degree of inconsistency in order to improve response time. Unfortunately, as will be discussed in chapter 4, inconsistencies affect the correctness of our proposed anti-cheating system and thus *Rendezvous* cannot be applied here. Mauve et al. [70, 102, 69] propose a scheme that tries to hide short-term inconsistencies which are caused by network delays. Updates performed on local game state copies are delayed to compensate for the propagation delay to other replicas. This way, the local player will perceive local changes with approximately the same delay as remote players. However, global consistency among nodes is not addressed.

## 2.2 Network Abstraction

In this section we present projects related to network abstraction in online games. To our knowledge, no scientific or commercial work exists that deals with the complete abstraction from different network architectures within a gaming context.

Kaneda et al. [60] propose *PeerBooster*, a middleware that allows the reuse of Client/Server-based games in a Peer-to-Peer mode. The authors argue that this may be useful if the publisher of a game discontinues to provide the necessary servers. A reason for this may be that the hosting becomes uneconomical because players have lost interest in the game. This might be the case with older games or games that were not very successful from the beginning. Each player has to install an application on his node which connects to the other player nodes in a P2P fashion. The application acts as a fake server to the local game application by capturing and answering the game related traffic. The global state is synchronized between all nodes, making it appear as if all players were connected to the same server. A major drawback of this approach is that the game's network protocol must either be openly specified or reverse-engineered. Every implementation of this middleware is specific to a certain game and hardly reusable for other games.

*Kosmos* [4] is a simple game built upon a distributed server architecture which is hidden behind a publish/subscribe abstraction. The game world is split into segments and each segment is managed by a server. A focus of this paper is to make the segments of the game world appear as a single seamless world to the players. For this purpose, subscriptions to updates of player avatars and game objects are automatically adjusted if these objects get close to the borders of a segment or cross them. Consistency is enforced by a locking mechanism which serializes access to all game objects. Since game regions are always hosted on servers provided by the game publisher and not on untrusted player clients, cheating is not an issue in this paper.

Another multiplayer online game architecture based on the pub/sub

paradigm is proposed by Fiedler et al. [42]. Like the paper presented above, they split the game world into distinct segments and subscriptions are chosen according to the players position. Additionally, the game communication is split into two different channels. The first channel is used for position updates, while the second for interactions between players. The authors argue that the second channel can be handled by the player nodes directly without the engagement of a server. This way, the bandwidth demands on the server side are reduced. The server only needs to receive position updates from the first channel and these updates may be aggregated to save even more bandwidth. Since player nodes handle the interaction between player avatars themselves, cheating is possible. The authors declare dealing with this as future work.

The Real-Time Framework (RTF) [47] also aims at providing an abstraction from the underlying network, but from a different perspective. It does not address pure P2P or hybrid architectures. Instead, it abstracts from the way a multiplayer game is distributed in a multi-server architecture. RTF supports three distribution concepts, namely *zoning*, *instancing* and *replication*. Similar to our framework, RTF provides a way for game developers to deal with game objects without concerning about synchronization issues. The paper does not go into detail about the underlying network architecture. Thus, it is currently difficult to say in which parts our works complement each other.

Modern commercial game engines usually provide some level of network abstraction, but are mostly tied to a certain network architecture. The technology overview of the latest *Unreal 3 Engine* [43] states that it is possible to run games either in a Client/Server or P2P mode. Unfortunately, the architecture is not openly documented and details thus unavailable. It is uncertain whether the engine supports a transition from P2P to C/S or vice versa without altering code. Moreover, it is very unlikely that the engine easily supports hybrid or custom network architectures.

## 2.3 Cheating Prevention in Online Games

In this section we present projects related to cheat prevention in multiplayer online games. Much of the work done in this area only addresses very specific attacks for certain game genres which are not discussed in detail here. Instead, we focus Projects that address the general problem of arbitrary game state manipulations and discuss the differences to our approach.

*FreeMMG* [23, 22] is a hybrid between Peer-to-Peer and Client/Server architecture and similar to our anti-cheating approach. While a server part is responsible for managing subscriptions, authentication and storing backups of the virtual world, the game itself is running in a distributed fashion on the clients. The game world is split into segments and segments are replicated on the nodes of the players. Unlike the system presented in this work, FreeMMG stores a replica of a segment's state on the node of the players within that segment. This opens up the possibility of disclosing secret information directly

to the players. The replicas use a lock-stepping synchronization mechanism to keep the replicas consistent. This allows a single malicious node to block the synchronization process indefinitely. Unfortunately, many aspects of the system remain unclear. First, there is no systematic classification of attacks with an explanation of how the system counteracts them. Only very few cheating scenarios are considered briefly. It is also not clear how the correct game state is determined in the presence of cheaters. Finally, the authors haven't found an appropriate consistency protocol yet. Although central parts are missing, a prototype of the system has been implemented. How this implementation is supposed to function in the presence of these gaps is not explained.

Another hybrid system that claims to provide cheat resistance is published in [26]. As in the system discussed above, the game world is split up into smaller regions which are managed by player nodes. The assignment of regions to nodes is realized through the Pastry [88] P2P overlay. Each region has a master copy and several secondary replicas, following the primary backup approach. The authors argue that because there exist multiple replicas of a region, a manipulation of a region's state cannot go unnoticed. However, they do not explain how the correct state of a region can be determined among probably conflicting replica states. As will be explained later in our work, either a agreement or a voting procedure has to be performed to determine the correct state. Generally, the paper stays on a very abstract level without providing any details about consistency among replicas, attack scenarios and appropriate countermeasures or latency issues incurred by the P2P overlay.

*Trusted Computing (TC)* is an initiative of the Trusted Computing Group [97]. It offers two features that are of interest to online game publishers. First, the possibility that only software that is signed by the publisher may run on a TC enabled node. Second, the possibility that a TC enabled node can prove its trustworthiness to other nodes of the system. The former guarantees that the client software (and its state) cannot be manipulated, the latter enables game publishers to identify trusted nodes over the Internet. As long as the game state is only distributed among trusted nodes, no manipulations are possible.

As the public discussion shows, Trusted Computing comes along with many dangers to the autonomy and privacy of the user. However, from an online game publisher's point of view, it seems to be an ideal solution, provided that the security mechanisms are functional and cannot be circumvented. Players could be encouraged to equip their Personal Computers with TC features by lowering their subscription fees or offering them access to exclusive game content.

All modern video game console are already equipped with TC-like security mechanisms. However, most of these mechanisms have been circumvented shortly after the release of the consoles. Since then, the console manufacturers have tried to fix security loopholes with updated firmwares until new ones are found. This example clearly shows a major drawback of TC systems: as soon as the security mechanisms is circumvented, all TC nodes become untrusted since exploits are spread over the Internet very fast. Securing the nodes again becomes a cat-and-mouse game between manufacturers and hackers.

An interesting anti-cheating approach is presented in [75] which breaks up with the assumption that a client is inherently not trustworthy. To ensure the integrity of a client, a protection mechanism is embedded into the software. In order to prevent an attacker from bypassing the protection, the protection code will be constantly changed within short intervals. The client has to download always the latest version of the code in order to be allowed to play. The authors claim that breaking the protection within the small period when it is active is not feasible. Since this approach is orthogonal to the system presented in this work, they could be combined to provide a higher level of protection.

There are also many publications on other kind of attacks that are specific to certain game genres or scenarios. We will only give a very brief overview. Baughman et. al. [6, 7] propose a scheme that uses a lock-stepped commitment protocol to prevent cheats on the protocol level. The *NEO* protocol [44] was developed as an improvement to the one presented above. It addresses a broader range of cheats while at the same time reduces latency but still addresses only cheats on the protocol level. Another approach on a similar level is *AC/DC* [41], which addresses cheats based on game event timing. Buro [18] presents a server-based architecture which addresses a cheat popular in Real-Time Strategy Games (RTS) that discloses the positions of enemy players. Chambers et al. [24] show that this kind of attack can also be addressed in a Peer-to-Peer architecture. Mogaki et al. [74] try to address the problem of delaying or denying the sending of game commands with a time-stamp service. Finally, *RACS* [106, 104] is an anti-cheating scheme for hybrid architectures which only reduces the outgoing bandwidth requirements of the server but not the incoming bandwidth and processing requirements.



## Chapter 3

# Framework Architecture

### 3.1 Introduction

“Ten or twenty years ago it was all fun and games. Now it’s blood, sweat, and code.” [15] In the early days, computer games could be developed by a only few people or even a single person. Most of the work was about writing optimized game code for hardware with very limited resources. Due to these resource limitations, other aspects of a game, like design, graphics or sound, had to remain very simple. Today’s games are multi-million dollar projects including dozens of highly specialized professionals, like 3D artists, level designers, musicians or storytellers.

Despite the fact that creative work makes up the largest fraction of a gaming project today, it still remains a challenging software engineering effort. As in all software engineering projects, reusability is one of the key issues which can significantly lower complexity, production costs and time-to-market. Any game uses at its core a central component, called the *game engine*, that handles all the computational tasks necessary for a game. First of all, it manages all the objects that show up in the game, like players, enemies and the game world itself. It performs the necessary logic to make these game objects come alive, like performing artificial intelligence for objects that represent living things or physics for inanimate objects. The game engine receives commands that are issued by human players (e.g. via mouse, keyboard or gamepads) that sit in front of the computer and turns them into actions that are performed by the game objects representing the players. Last but not least, the engine provides an audio-visual real-time representation of the game. Many of these tasks can be encapsulated in a separate component. This way, a component can easily be replaced by a more specialized one or reused in different projects. Moreover, the components can provide an abstraction from the underlying hardware, enabling games to run on different platforms. Nowadays, many game engines are customized and reused by multiple game projects and selling engine licenses is even part of the business model of some producers.

Besides providing essential technical components, the game engine serves as an interface to incorporate all the digital content (called *assets*) created by various artists into the game. Examples for assets are character and level designs, 3D models and textures, sound effects and music or text and dialogs.

Assets are created with specialized tools and later converted into a format that can be imported by the game engine. Usually, most asset creators have a very limited knowledge about writing code. Thus, the interface to the game engine must require a minimum of programming skills. But at least when creating assets that exhibit behavior (like an enemy whose behavior is determined by an artificial intelligence) one usually cannot avoid getting in touch with coding. For this purpose, easy-to-learn scripting languages are incorporated into the game engine. Together with predefined methods, which handle common in-game functionality (e.g. a *move(x,y)* method, which moves a game object to a certain position and automatically performs path finding and collision detection) and can be called from within a script, the programming task is kept as simple as possible.

Hiding complexity gets even more difficult when network gaming comes into play. Network functionality is probably the most important gaming feature today, with networks ranging from a few nodes in a LAN environment to a few thousand nodes in Massively Multiplayer Online Games. Providing a sufficiently consistent view of the game on all nodes of the network is non-trivial. Consequently, asset creators should not be burdened with the task of handling inconsistencies or performing manual synchronization of game objects. However, even programmers that work on different engine components benefit from being shielded from complex consistency issues. Thus, it is generally a good idea to keep consistency-related code within a single module, allowing developers of other modules to focus on their specific tasks. Again, a clean separation of concerns is a good basis for reusability.

In this chapter we present a framework for a game engine that, in addition to providing support for the necessary components, completely shields game developers from network and consistency related issues. Unlike existing game engines, our system does not only abstract from a specific network architecture. Games built using our framework can be deployed in many different environments by simply changing a configuration file. Besides running the game in single player mode locally, we currently support three network modes: classic Client/Server, a pure Peer-to-Peer mode usually known as *Replicated Simulation* [9] and a P2P mode with special anti-cheating guarantees that is presented in detail in the following chapter. In the following we will refer to these network modes as *CCS*, *RS* and *AC* respectively. All three modes provide some protection against cheating, an essential property for today's games. The underlying abstraction allows developers to extend the framework with their own custom network modes, if necessary. Without the need to commit to a specific network mode, it is much easier to reuse a game engine in different projects. Furthermore, game developers may allow players of a certain game to change the network mode by simply altering a configuration file. If a group of players doesn't trust a single node to host a server for a Client/Server session, they could switch to Peer-to-Peer mode where each node maintains its own local copy of the game state. Finally, home-brewn or independent games as well as academic projects may benefit from the possibility of playing around with different network modes without having to change their game code.

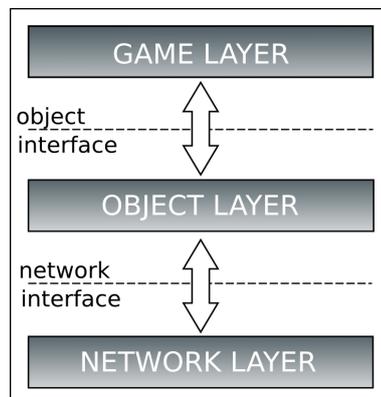


Figure 3.1: High-level overview

## 3.2 Overview

Our proposed framework architecture can be divided into three layers and two intermediate interfaces, as shown in Figure 3.1. The discussion in this section remains on a rather abstract level; important details are addressed in the following sections. We start on the highest layer, the *Game Layer*, and work our way down to the lowest one, the *Network Layer*.

### 3.2.1 Game Layer

The Game Layer is the place where most of the "action" takes place. It contains nearly all the important components of a game engine, like the input manager, the presentation manager and the scheduler. The input manager is responsible for accepting commands issued by the player via keyboard, mouse, a gamepad or any other kind of input device. The presentation manager provides the player with an audiovisual real-time representation of the game and probably even some haptic feedback. At the core of any game engine there is a scheduler which controls at which intervals the game world is updated and triggers certain components of the engine.

Although virtually every game is made of components like those mentioned above, actual implementations may show a great variety. Professional games today will most likely consist of much more components, while simple games may combine everything into a single one. Note that these components do not necessarily have to be implemented by the game developers themselves. There are many implementations that can be bought off the shelf or are available for free.

Please refer to Section 3.5 for a detailed discussion of the Game Layer.

### 3.2.2 Object Interface

The central element of a game is a collection of objects that constitute the state of the virtual world. The *game objects* may represent nearly every aspect of the game: the players' avatars, computer-controlled enemies or allies, interactive objects (like vehicles and machines) or completely static objects (like trees and

walls). Even purely logical entities that have no perceptible representation (at least none that is perceived by a human player), like containers that aggregate game objects into a logical unit or triggers that activate in-game actions, may be modeled as game objects. The Object Interface allows the creation and deletion of game objects as well as reading and changing their state.

In a multiplayer game, multiple participants share the same game world and thus need to have a consistent view of its state. If the players are located on different nodes of a network, local copies of the game objects, which as a whole represent the state, need to be synchronized. The Object Interface hides this synchronization effort completely, allowing a game developer to access and manipulate game objects as if they were local. All components that run on the Game Layer may work as usual. E.g., the input manager translates input events into appropriate changes of the player's avatar object. The presentation manager may read the state of the game objects and generate audio-visual and haptic feedback. And last not least, the scheduler triggers updates of game objects whenever the rules and the logic of the game require it.

Furthermore, the Object Interface provides methods that perform the necessary bootstrapping when setting up or joining a network session as well as methods to leave a network or shut down a session. Although these methods are not directly related to game objects, they are included in the Object Interface to provide a seamless abstraction to the game developer.

Note that the Object Interface is the lowest interface that a regular game developer should get in touch with. Deciding in which network mode a game runs is done via a configuration file, not by writing code. Only if the game uses a custom network mode, code has to be written for the layers below.

### 3.2.3 Object Layer

The Object Layer is responsible for holding up the illusion that all game objects seem to be local and can be manipulated through the Object Interface without concerning about synchronization. Furthermore, it has to handle the necessary bootstrapping when a new node joins the network or cleanup when a node leaves.

In our framework, every game object has an owner which keeps a master copy of it. Whenever a node wants to change a local copy of an existing game object it must send a *request* to the owner. If the request is granted, the owner changes the object state accordingly and sends an *update* to every node that keeps a local copy (including the one which has sent the request). Whenever a node receives an update sent by the owner of an object, it will perform the contained change on its local copy. This way we achieve a single-copy consistency since the owner of an object serializes all operations on it. Note that in the AC example a group of nodes acts as the common owner of a game object. Each node in the group receives a request, processes it independently and sends an update. Whichever node has a local copy will receive the updates and elect the one which holds the majority. Please refer to chapter 4 for a detailed discussion.

Note that all operations needed for the management of an object can be mapped onto two types of messages, namely a request message and an update message. We still need a third kind of message to inform nodes about organizational events like the joining and leaving of nodes. Whenever a node joins the network it sends an *announcement* to the existing nodes. Every node that owns a game object which is relevant for the newly joined node may now send

an update containing the current state of this object. This way, a new node can be provided with the current state of the game. When the node leaves again, it may inform the other nodes that it won't process request or updates anymore. If the objects it owns are still needed, it may request the creation of replacements on remaining nodes.

Please refer to section 3.4 for a detailed discussion of the Object Layer.

### 3.2.4 Network Interface

The discussion above showed that the messages needed for game object synchronization and node housekeeping may be divided into three categories: *requests*, *updates* and *announcements*. What we have to make sure is that messages are sent to the appropriate recipients. For instance, a client in the CCS example is never interested in receiving request messages, since it doesn't own any objects. On the contrary, the server doesn't care about updates since — due to the fact that it owns all the objects — it is the only one to send them. To complicate matters, nodes join and leave and thus the list of senders and recipients changes dynamically.

However, this problem is not new and a solution for it is well-established: the *Publish/Subscribe (pub/sub)* paradigm [37]. One of the main advantages of pub/sub systems is the decoupling of message senders from message receivers. Participants of such a system only need to know what *kind* of messages they want to send. They do not need to know who are actually the recipients of these messages. The other way round, receivers only need to know what kind of messages they are interested in, not who may actually be sending them. The sending of messages of a certain kind is called a *publication*, while registering interest for a certain kind is called a *subscription*. The pub/sub system matches every publication to its respective subscriptions and thus takes care that a message will reach its intended recipients. Both, publishers and subscribers, may join and leave dynamically without requiring other participants to take notice of this.

Applying this concept to our framework avoids that owners of game objects and keepers of local copies have to be aware of each other. Any node which wants to manipulate an object simply publishes an appropriate request message. Owners of game objects are subscribed to this kind of message and thus will automatically receive change requests. After processing the request, they publish an update and nodes which keep a local copy will receive the change since they are subscribed to update messages. To sum it up, the networking interface has to provide methods to issue publications and register subscriptions.

### 3.2.5 Network Layer

The lowest layer of our framework's architecture is responsible for implementing the pub/sub methods that are offered by the network interface. Publications have to be routed over the network to the appropriate subscribers. This layer also has to take care of managing publishers and subscribers which dynamically join and leave the network.

Please refer to section 3.5 for a detailed discussion of the Network Layer.

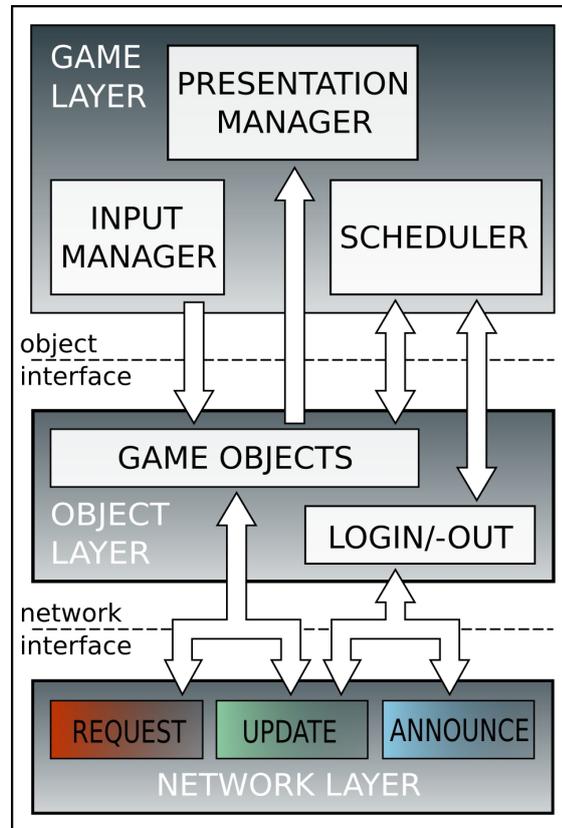


Figure 3.2: Detailed overview

### 3.2.6 Concluding Overview

Figure 3.2 gives a more detailed overview of our three-layer framework including its two interfaces. On top is the Game Layer which may access the lower layers of our framework via the Object Interface. Within the Game Layer, one may simply manipulate game objects as if they were local without paying attention to the layers below. The only thing that may be noticeable is a delay until a manipulation actually takes effect. (This delay may be hidden from the player by using commonly known techniques like *Dead Reckoning* [82].) Below the Object Interface is the Object Layer where the configuration of the desired network mode takes place. A node has to define to which topics it publishes and subscribes and which factory it uses for creating objects with the correct ownership. Supporting custom network modes means providing the appropriate definitions and factories. This layer is also responsible for handling the login and logout of nodes. Finally, the Network Interface serves as an abstraction to the message handling. By using a generic interface one may use different implementations in order to fulfill certain performance or scalability requirements or simply to experiment.

## 3.3 Game Layer

As mentioned in the section above, the Game Layer is the place where most of the actual development takes place. Ideally, game developers will only get in touch with the lower layers by using the Object Interface. Our framework provides default implementations for the most important components, namely the Scheduler, the Input Manger and the Presentation Manager, which will be presented in the following.

### 3.3.1 Scheduler

The scheduler is the central component of any game engine. A game is basically a real-time simulation and any of its objects and components must perform the actions and operations at the correct point in time. In its most basic form, a game engine scheduler is a simple loop that repeatedly reads input, updates game objects and renders them to the screen. However, in modern games there are many more tasks that need to be managed in a timely fashion, like disseminating and receiving network updates or performing physical computations for game objects. Moreover, the scheduler must make sure that a game runs at the correct speed on machines with different processing capacities. In early days, computer games repeatedly executed their main loop as fast as they could. As long as the game was only played on a certain hardware, it always ran at the same speed. However, with the success of the IBM PC and its successors, it was more and more common that computers, which had basically the same architecture and operating system, ran at different speeds. Thus, schedulers now had to take into account the amount of real-time that passed between two executions of the main loop. Another problem is that different passes of the main loop may have different execution times. For example, when many objects are currently visible on the screen, updating and rendering these objects may take significantly longer than in a situation with only a few objects. Variable execution times may result in a jerky gaming experience.

In order to support arbitrary tasks that can be put under the control of the scheduler, an appropriate interface has been defined. Every component that wants to be triggered by the scheduler implements the `Task` interface and registers itself at the scheduler. The scheduler iterates over all registered tasks and calls an update method which activates the task. The order in which the tasks are activated during one pass of the main loop may be defined at task registration time. In addition to the queue of tasks that are activated while the game is running, there is a queue of tasks when the game is paused. This is necessary, for example, to present a configuration menu while the game is paused.

Whenever a task is activated, it receives the amount of time that has elapsed since the last loop pass. This way, variable execution times of different passes can be compensated. For instance, if the position of a moving game object needs to be updated, the new position can be calculated by multiplying the speed of the object with the time that has elapsed. As a result, the object will move at a constant speed no matter how long a pass of a loop actually takes. For determining the time that has elapsed since the last pass, a timer is used. Our framework provides a default implementation that uses the standard JDK timer, but it may easily be replaced with a custom high-precision timer.

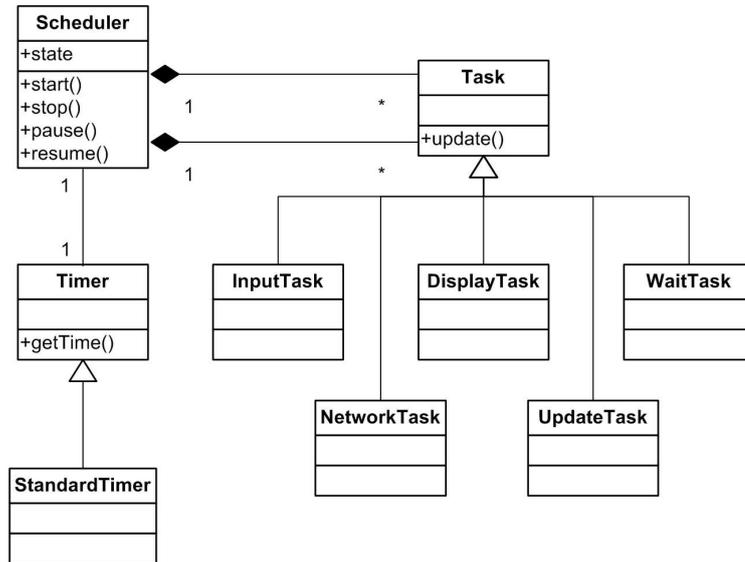


Figure 3.3: Scheduler Class Diagram

The framework provides some default tasks. The `InputTask` triggers the polling of player input which may be translated into corresponding state changes of the player’s avatar object (see section below). Informing active game objects about the time that has elapsed, so they can update their state accordingly (e.g. continue a movement), is performed by the `UpdateTask`. The `NetworkTask` applies game object updates that are received from other nodes over the network and sends updates of local objects to other nodes. Reading the state of all game objects and rendering them onto the player’s display is done by the `DisplayTask`.

Finally, the `WaitTask` simply suspends the game thread for a certain time which is useful if the game runs on very fast machines. Usually, the main loop is executed consecutively without pausing. On very fast machines this leads to a very high update rate of the game objects and the display. Up to a certain degree, this results in a smoother presentation of the game. However, beyond a certain point this is simply a waste of resources. Suspending the game thread regularly leaves more processing resources for other processes running on the machine or at least to less energy consumption while being in an idle mode.

### 3.3.2 Input Manager

The Input Manager is the component that is responsible for accepting commands issued by the player and turning them into appropriate actions. Today there exists a multitude of input devices: the keyboard, mice, gamepads, joysticks or steering wheels. All these devices may provide input data in different formats. Moreover, the same command may be issued by a player in various ways, so some kind of abstraction is needed. For example, pressing the left arrow on the keyboard, moving the mouse left or turning the wheel to the left will probably all result in the same command: move the player to the left. In our framework, commands are represented by `InputEvent` objects. For instance, an input event

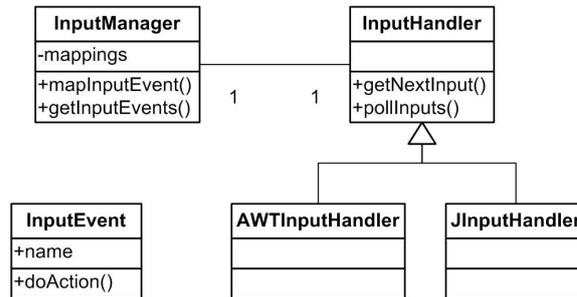


Figure 3.4: Input Manager Class Diagram

called “move player left” represents the above mentioned command, no matter through which device it was issued. Moreover, a method can be defined for this event that automatically changes the state of the corresponding player object. The Input Manager allows arbitrary mappings between input data and input events. The same command may be issued through different devices or multiple commands may be issued by a single button press.

The actual hardware devices can be accessed through the **InputHandler** interface. Our framework provides two implementations for this interface, namely **AWTInputHandler** and **JInputHandler**. The former is an abstraction for the *Abstract Window Toolkit (AWT)* which is the standard API for graphical user interfaces in Java. The AWT allows reading player input — just keyboard and mouse are supported — only through an event queue. Every time the player generates input a corresponding event is generated and put into the queue. However, for responsive gameplay, direct polling of the input devices is usually preferred. *JInput* is an external library that not only allows asynchronous polling of input devices but also supports many more devices than the AWT. Actually, all devices that are supported by the underlying operating system can be used with *JInput*. However, *JInput* is partly written in native code and thus runs only on supported platforms while AWT is available on all Java platforms. The input handler interface provides an abstraction for polling input data even if the underlying implementation only provides an input event queue. The **AWTInputHandler**, for example, returns the most recent events of the queue when `pollInputs()` is called.

### 3.3.3 Presentation Manager

The Presentation Manager is responsible for providing a real-time audiovisual representation of the game. There exist plenty of commercial and open-source engines for two- and three-dimensional graphics as well as sound engines. In this work we only provide a very basic two-dimensional display manager without sound. This is sufficient for the example game that we implemented using our architecture and which is presented in section 3.6.

Our implementation provides methods for opening a display in windowed or full-screen mode. It allows creating graphics that are managed by the graphics hardware for optimal rendering speed. In addition to that, it uses double buffering for drawing display frames. This means that a new frame is always rendered in an invisible back buffer. As soon as the rendering is finished, the currently

visible frame (called the front buffer) is replaced by the back buffer which itself becomes the new back buffer onto which the next frame is rendered. This way, the rendering of a frame is not visible to the player in front of the display. If only one buffer is used for rendering, a disturbing flickering may be visible.

Since our simple implementation of the Presentation Manager fits into a single class, we omit showing a class diagram.

## 3.4 Object Layer

The Object Layer offers a transparent access to all game objects disregarding whether they are stored locally or on a remote node. It provides the necessary implementation of the storage and retrieval methods offered by the Object Interface. Additionally, it implements methods for joining and leaving a network.

In this section we discuss the data model of the game objects, object management and how updates and ownership are handled. Next, we show how and logging in and out of the system actually works. Finally, we give an overview of the Object Layer's architecture.

### 3.4.1 Game Object Model

The game object model has to meet two basic requirements: it must be flexible enough to allow the modeling of arbitrary game objects and it must support the creation of objects by means of specialized tools.

Many different ways exist to model objects within a virtual gaming environment [13, 27, 33, 35]. We have chosen an approach that provides high flexibility as well as ease of use. It is completely dynamic, i.e. every aspect of a game object can be changed at runtime without the need for a recompilation.

The creation process of a game object is a very important issue. Game engine programmers only provide the data model for the game objects; the task of turning dull data structures into interesting objects that make up a fascinating game world is performed by asset creators which are mainly artists. To come alive, game objects need among other things detailed audiovisual representations and realistic behavior. This has to be done without writing complicated program code or time-consuming recompiling of object structures. Thus, game artists need easy-to-use creation tools and a way to get a direct feedback of their work. For this reason, the so-called *data-driven* development approach has been established. Nearly all aspects of a game object are provided in a separate data structure instead of hard-coding them into the software. These data structures can be generated by specialized creation tools and imported into the game engine without the need to recompile program code. This way, asset creators can see the changes they made to the game objects nearly instantly reflected within the game. This speeds up the development process and should make it easy to integrate this framework into their workflow.

The game object model consists of four parts:

1. a game object type system,
2. the definition of game object attributes,
3. operations that can be performed on game objects and

4. the relationship between game objects.

In the following, we describe each of these parts.

### Type System

Every game object has an object type associated to it. In contrast to programming language type systems, the focus of the game object type system is not on providing features like type safety or polymorphism. Today's computer games may have many thousand different kinds of game objects. A type system helps to categorize these objects and bring them into a hierarchical order. This way, complex object types can be derived from simpler ones. For example, at the root of the hierarchy may be simple types that define whether an object is visible, whether it can move or receive input from a player. From these basic type more complex ones can be derived, e.g. a visible and moving avatar that can receive input from a player. Game object types also provide an easy way to create many objects of the same kind, e.g. an large army of uniform foot soldiers.

Of course, this functionality is also provided by type systems of common object-oriented programming languages. However, experience has shown that during the game development process the design of game objects may change very frequently [33]. A static type hierarchy as offered by common programming languages would force game developers to perform time-consuming recompilation even for small changes. Moreover, asset creators would need to get in touch with programming code since they had to change class hierarchies. For this reason, the game object type system provided by our framework is completely dynamic. A game object type keeps references to its base types, its attributes and methods. These references can be changed dynamically during run-time without the need for a recompilation. This way, an asset creator can directly import any changes into a running test environment and see these changes reflected instantly. Moreover, changing game object types can be done through the use of specialized tools. These tools may provide a graphical interface to compose game objects and generate data structures that can be imported into the game engine, thus following the data-driven approach.

### Attributes

Every game object has a certain state attached to it that distinguishes it from other instances of the same type. An object's state is composed of its attributes, which are themselves represented by state objects. Thus, a game object keeps only references to its state objects which can be changed dynamically at any time. Which attributes an object has, is determined by its type. On creation, any game object is equipped with all the states that its type and all base types define. The instantiation of game state objects can be done in a lazy manner. As long as attributes of an object only contain the default value, a default state object provided by the game object type can be used. Not until an attribute gets a value assigned that differs from the default, it may create its own state object instance. This way the memory footprint may be kept small and unnecessary instantiation overhead is avoided.

A state object itself has an identifier which must be unique among the states of a certain object type. By this identifier an attribute may be accessed, e.g. by operations defined on the object. For example, a *move(x,y,z)* method may

change the values of an object's position attributes. The position attributes may also be read by the presentation manager to render the object at the proper position on the screen. Every state object has a value that contains the actual state and corresponding getter and setter methods.

An important aspect is that a state object may inform attached listeners about changes of its value. By default, a state object informs the game object it belongs to whenever its value changes. The game object accumulates all changes which can, for example, be sent as a whole over the network or written to persistent storage. This way, resource-intensive polling for changes can be avoided.

### Operations

Every aspect of a game object can be represented by its attributes. Whenever an object exhibits behavior, i.e. it "does something", this is reflected by a change of its attributes. E.g. an object that moves changes its position attributes constantly. Moreover, it may change certain flag attributes that provide hints to other engine components. For example, a moving object may set a certain flag to inform the presentation manager that it should render an appropriate animation.

Performing operations on game objects could be done by simply changing its attributes. However, it is tiresome to change the position of a moving object constantly by hand. Instead, it would be convenient to simply set the desired endpoint of a movement and let the finding of intermediate waypoints (including path-finding to avoid obstacles) be handled automatically. For this reason, operations can be defined on objects that abstract complex attribute change patterns into single commands.

Of course, like types and attributes, operations should be dynamically linked to a game object to enable the flexibility of a data-driven approach. For this reason, operations are encapsulated into objects that are referenced by the corresponding game object. The operation objects themselves represent small pieces of code written in a scripting language. Since most scripting languages are kept rather simple, they are ideal for asset creators that cannot avoid getting in touch with coding. Simple scripts may be programmed by themselves while more complex one can be provided by programmers. The scripting interface also allows the use of different scripting languages, which may be tailored for certain tasks like artificial intelligence or simulating physics. Since our framework is implemented in Java, all scripting languages that implement the *Java Scripting API* [71] may be used.

Game operations are defined by the game object type and a type inherits all methods from its base types. Methods with the same name are overridden.

### Relations

All objects of the game world form a hierarchical tree. Every game object thus has a single parent and zero to many children. In most cases the hierarchy will be ordered spatially. At the root is an object that represents the game world as a whole, followed by subdivisions like regions or buildings and so on. However, any ordering that fits the needs of a certain game design best may be chosen. In

addition to the hierarchical ordering, it is possible to form game object groups with regard to any criteria. This is described in the following section.

### 3.4.2 Object Storage and Retrieval

Since the game world may consist of thousands of game objects, a game engine needs to provide efficient storage and retrieval methods. Some engine components need to access game objects to change their state. For example, the Input Manager translates commands issued by a player into changes of the corresponding avatar object. In a game that is played over the network, state changes that are received from other nodes also need to be applied to the local objects. Finally, many game objects will change their state over time, e.g. a vehicle that follows a specific path changes its position constantly. Other components need to read the state of game objects regularly. For example, the Presentation Manager needs to read the state of game objects to create an audiovisual representation of the game. The networking component may need to inform other nodes of the system about the changes of local game objects.

#### Object Manager

The Object Manager serves as a central repository for game objects. It allows the insertion and retrieval of all objects of the game world. Every object has a unique identifier under which it can be retrieved. Additionally, the manager offers the possibility to search for objects or groups of objects using regular expressions. For example, since objects are organized hierarchically, it is possible to retrieve all objects that share the same prefix.

The Object Manager also provides access to the game object type system. As explained above, game object types are represented as objects themselves and can be retrieved by using their canonical names.

Finally, the manager offers convenience methods which are only used by components located on the Object Layer. For example, game object updates retrieved from the network can automatically be applied to local objects. Furthermore, arbitrary observers can be attached to game objects to monitor changes.

#### Object Views

Although all objects of the game world are ordered hierarchically, in some cases it is useful to form groups of objects according to other criteria. For example, a physics component may want to know all objects that need accurate simulation of physical properties. In the hierarchical order of the game world these objects are not necessarily grouped together but may be spread all over the hierarchy. For this reason it is possible to create game object views according to certain criteria. A view can register itself as a game object observer at the object manager. Whenever a new object is created or an existing one changes its state, a listening view may add the object if it matches the view definition. The same way, objects can be removed if they are deleted or do not match the view criteria anymore. By using a view, any engine component can obtain a current set of all game objects that are of interest to it.

### 3.4.3 Updates and Ownership Management

As described above, game objects gather all changes that have been made to them. These changes (also called *deltas*) are encapsulated in a separate object. Not only changes to its attributes, but also the creation and deletion of whole objects, is stored in a delta. The delta object provides serialization and deserialization methods which make it easy to transmit changes to other nodes over the network.

At this point, the ownership management comes into play. In a networked game, only the owner of an object may allow changes to it. Whenever a change is made to a game object, the framework automatically checks whether the performing node of the change is also the owner of the object. If not, the game object is not directly changed. Instead, a delta object is created that reflects the desired changes and contains a special flag that marks it as a change request. This request is transmitted to the actual owner node which may decide whether it wants to perform the change or not. If it performs the change, an update delta is automatically created in the usual way. This is a regular delta which is not flagged as a request. It is transmitted to all nodes that need to be informed about the change (including the node that sent the request). The receiving nodes then update the object's state accordingly.

Until now we have only talked about existing game objects which contain the owner information in their metadata. What remains is the question of how ownership is determined when creating a game object. Burdening a game developer with this task when creating an object would break our abstraction. To avoid this, the object layer has to provide a factory method for each supported network mode which encapsulates the knowledge about determining ownership. A game developer simply creates an object (through the Object Interface) and, depending on the network configuration, an appropriate factory is chosen. In our CCS example, the server is the owner of all game objects and whenever a client needs to create one, the respective object factory determines the server as the owner of this object. In contrast, in the RS example a peer node always takes ownership of objects it creates. Finally, in the AC mode, the owner id addresses the whole group of owners. As we can see, a node does not only create objects for itself but it may also request the creation on another node. Thus, the creation of a new game object is treated the same way as the manipulation or deletion of an existing one: it is sent as a request to the future owner. Upon receiving and processing a creation request, the owner sends an update to all nodes the creation may concern.

### 3.4.4 Login and Logout

Logging in and out of the system is straightforward: the other nodes of the system must be informed about the joining or leaving of the local node. For this purpose, an appropriate announcement has to be published on the network. How this is done is explained in detail in section 3.5.1.

### 3.4.5 Class Diagram

Figure 3.5 shows an UML class diagram containing the most important classes. At the center is the `GameObject` class. Every game object has a parent object

and an arbitrary number of children, thus forming a hierarchical order. Every game object is composed of one to many `GameObjectState` objects which constitute the attributes of a game object. The game object is registered as a `GameObjectStateListener` which gets informed every time an attribute is changed. All changes are accumulated in a `GameObjectDelta` object which can be serialized and deserialized for easy transmission over the network. Every game object has a `GameObjectType` associated to it that contains all attributes of the type and its default values. Moreover, the type references the script objects that make up the operations that can be performed on game objects of this type. Finally, each type may have an arbitrary number of base types which it is composed of. Each game object may have a `GameForm` associated to it which is responsible for rendering it to the screen (if the object is visible). The `ObjectManager` is the actual implementation of the Object Interface and acts as a facade to the classes of the Object Layer.

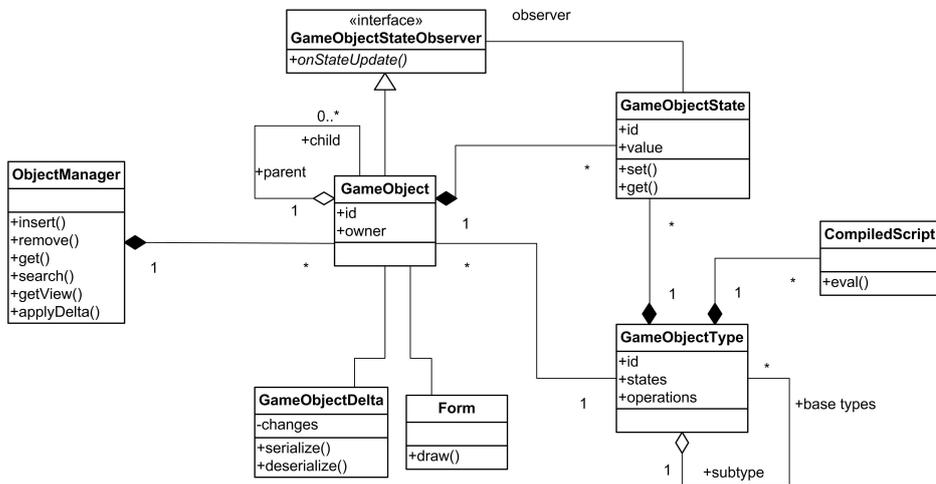


Figure 3.5: Object Layer Class Diagram

## 3.5 Network Layer

The lowest layer is responsible for distributing updates of local objects or sending change requests for objects that are owned by other nodes. Additionally, it sends announcements of nodes that join or leave the system.

We start with an explanation of how the underlying publish/subscribe system handles the dissemination of messages mentioned above. Next, we shortly discuss possible optimizations by using different flavors of pub/sub. Finally, we give a short overview of the involved classes.

### 3.5.1 Publish/Subscribe

To demonstrate how a pub/sub messaging service can be integrated into our framework we have chosen a simple form of pub/sub, a *topic-based* approach.

Later on we will discuss how more powerful approaches may be used to lower bandwidth consumption or improve scalability.

As the name implies, in a topic-based pub/sub system participants publish and subscribe to *topics* and each topic represents a certain kind of message. The obvious way to model our communication is to assign each type of message — requests, updates and announcements— its own topic. We first demonstrate how requesting a change and sending an update works within the three example network modes we have implemented. Next, we will show how the announce topic may be used for handling nodes joining and leaving the network.

The following is a short overview of how these network modes distribute the ownership of game objects.

**Classic Client/Server (CCS)** The central server is the owner of all objects and thus keeps all master copies. Clients only store local copies which are updated by the server.

**Replicated Simulation (RS)** Each peer may own certain objects for which it keeps the master copies. It stores local copies of the objects owned by other peers.

**Anti-Cheating (AC)** In order to avoid arbitrary manipulations by malicious nodes, each object is owned by multiple owners called *Region Controllers (RCs)* (see next chapter for a detailed discussion on this mode). Thus, each RC keeps its own master copy of an object and any change request has to be sent to each RC. After changing the state of a master copy, each RC sends an update to the local copies on the clients. The client compares the update messages and elects the one that holds the majority.

Figure 3.6 shows the request/update process in the CCS context. Client 1 wants to change an object and publishes a message to the request topic. The server which owns all objects has subscribed to this topic and thus receives all requests. After performing the requested changes the server publishes a message containing the changes to the update topic. All clients, including the one that has sent the request, are subscribed to this topic and receive the update.

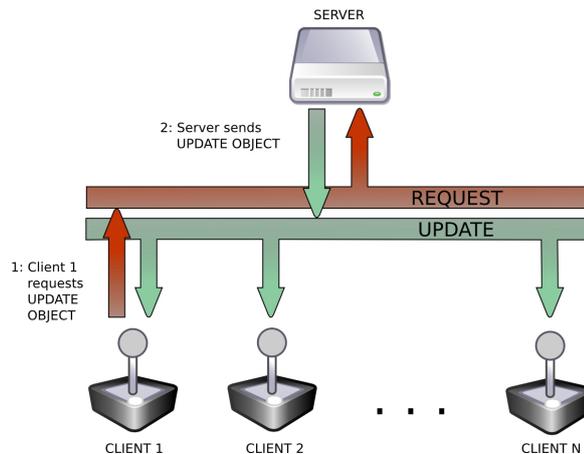


Figure 3.6: Request/update in CCS mode

In the RS context (Figure 3.7), a peer that wants to change an object publishes a request. All peers within the system are subscribed to the request topic, but only the owner of that object needs to process the request. The state update is then published and received by all peers, since each of them is subscribed to the update topic. A special case is when a peer wants to change an object that

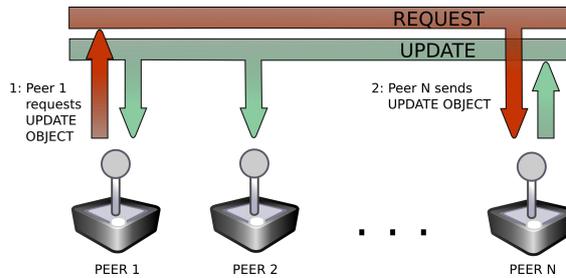


Figure 3.7: Request/update in RS mode

it owns. In this case the peer may directly send an update to the other peers (figure 3.8).

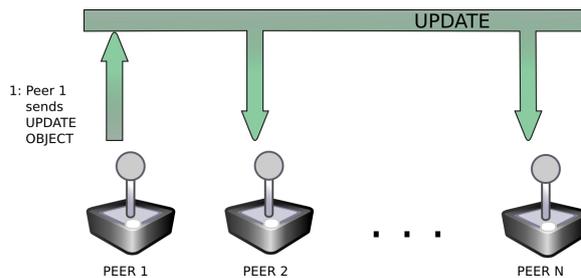


Figure 3.8: Request/update in RS mode (updating peer is the object's owner)

Our last example, the AC context (Figure 3.9), is very similar to the CCS mode. Instead of having a single server, all RCs are subscribed to the request topic. After performing the requested change, each RC publishes an update. The clients, which are subscribed to the update topic, receive all updates from the RCs. Before an update will be performed, the correct one is elected out of the received updates.

To handle events like nodes logging in and out of the system, a third topic, called *announce*, is used. Whenever a new player joins the game, an object has to be created that represents that player. The nodes already in the system need to be informed about the state of this new player object. Figure 3.10 illustrates this process in the CCS context. The server, which is subscribed to the announce topic, receives a login announcement published by the new client. It creates a new avatar object representing that player and publishes an appropriate update. This update is received by all clients, since they are subscribed to the update topic. The AC mode (figure 3.11) is very similar, the only difference is that all Region Controllers have to vote for the creation of the player object and send an appropriate update.

After logging in, the new client needs to be supplied with the current state

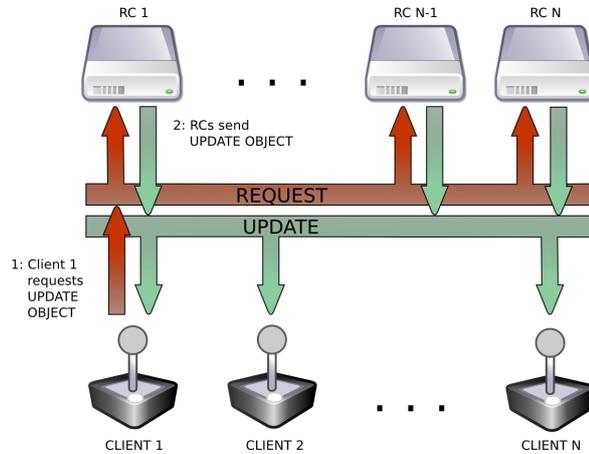


Figure 3.9: Request/update in AC mode

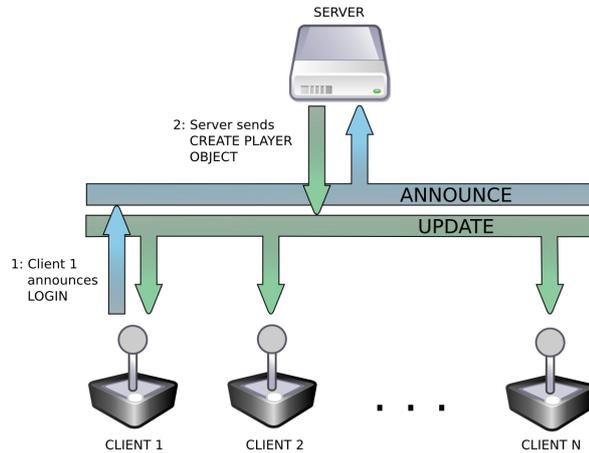


Figure 3.10: Client login in CCS mode

of the game. For this purpose, every node that owns game objects must be subscribed to the announce topic. Upon receiving the login message, the owners may publish an update containing the complete state of their master copies. Figure 3.12 shows this for the RS scenario. All peers that own local objects have to inform the new peer about the state of the already existing objects. Unfortunately, publishing the whole state of all master copies every time a node joins the game would be a waste of bandwidth. Every node subscribed to the update topic would receive the current state, even if its local copy is up-to-date. Optimizations that avoid this are discussed in the following subsection.

If a node wants to leave the network it simply publishes a log-out announcement. In CCS mode, after receiving this message, the server publishes an update that removes the avatar object of the corresponding player from the game. The same update is published by the Region Controllers in the AC mode. In the RS mode things are slightly more complex, since a leaving peer node may be itself the owner of certain game objects which are still needed. Before leaving the

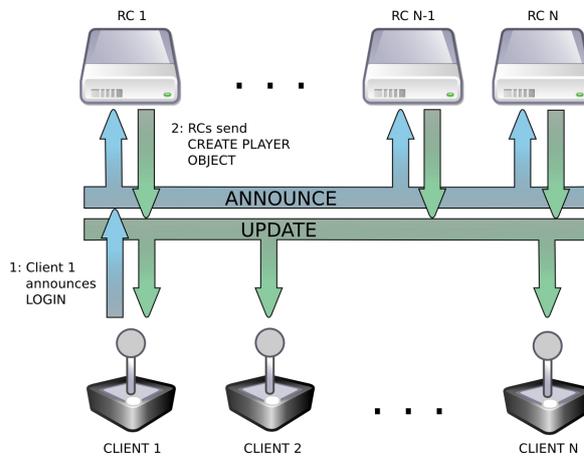


Figure 3.11: Client login in AC mode

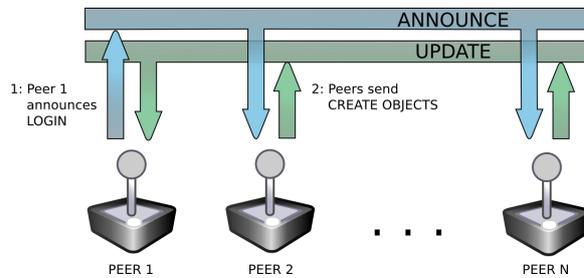


Figure 3.12: Peer login in RS mode

network, the node has to make sure that these objects are transferred to other peers. In order to do so, it can request the creation of an object on another peer by specifying this peer's id as the owner id.

### 3.5.2 Optimizations

An important way to reduce network bandwidth requirements in online games is to restrict the amount of updates a certain node receives. Obviously, a node does not need to be informed about changes of game objects that the local player can neither perceive nor interact with in any way. Limiting the update message to ones relevant for the player is commonly known as *Interest Management*. Instead of subscribing to all messages that are published to the update topic, a filtering based on the in-game position of objects may be performed.

For example, the Java Message Service [95] combines a topic-based pub/sub approach with filtering based on key/value pairs. Every update published may be enriched with additional properties that contain the position of the updated object. Only when the player's avatar is in the interaction range of that object the update will be sent to that player's node.

Instead of using a flat topic, a hierarchical one may be employed to restrict messages to certain game regions. This approach is usually referred to as *subject-based* filtering [81]. E.g. in a game that uses a real-world setting, sub-

jects like `Earth`, `Earth.Europe` and `Earth.Europe.Germany` could exist. Whenever an avatar enters a region (e.g. `Germany`) the node subscribes to the corresponding subjects. On the one hand, this makes sure that the node won't be bothered with unrelated messages of events that happen in a different country or even on a different continent. On the other hand, the node will receive messages of events that are relevant for the whole continent or even globally. Naturally, changes made by the node will be published to the appropriate subjects in the same manner, depending on their relevance.

Not only the addressing model but also the implementation of a specific model has an impact on performance and scalability. One very important performance criteria of network games is the latency when propagating updates of game objects. Usually nodes of gaming networks talk directly to each other, be it a client talking to a server or peers talking to each other. The delay of changing an object (i.e. issuing a request and getting a reply) equals the roundtrip time between nodes. In an implementation that wants to avoid higher latencies, a node that requests the change of an object must send the request directly to the owner node. Afterwards, the owner has to send its updates directly to all nodes which keep a local copy of the updated object. This way, extra delay caused by additional hops on the network path is avoided. In such an implementation a local software component running on each node can provide the pub/sub interface to the object layer. Internally, this component stores a list of all subscriber nodes for all topics it publishes messages to. Whenever a node publishes a message it can send it directly to the appropriate nodes. The subscription management service may be located on a separate node. Every time a node subscribes for a topic, the management service can inform the publishers about it. By sending a so called *advertisement*, a node can inform the management service about its intention to act as a publisher for a certain topic.

A further optimization is that whenever a node wants to change a game object that it owns, it may directly publish an update without the need to send a request first. But one should be aware that this may affect fairness. While the change is propagated to other nodes with the delay of a single hop it is perceived nearly instantly on the local node. This may enable the local player to react much faster than players on remote nodes. To avoid this, an artificial delay may be introduced (e.g. *Local Lag* [70]).

While the implementation above minimizes latency caused by network delays, it severely limits scalability. Think of a node in a Replicated Simulation which has to send updates to a very large amount of other nodes in the game. This way a node will soon reach the limits of its network connection, especially when using an asynchronous DSL connection with a very limited upload bandwidth. This is where pub/sub systems that rely on intermediate brokers play out their strength. While introducing additional delays for message delivery, the intelligent routing and filtering mechanisms can minimize bandwidth and connectivity requirements on the game nodes.

### 3.5.3 Class Diagram

Figure 3.13 shows an UML class diagram containing the most important classes. Since we only provide a basic implementation which is rather straightforward, we don't go into much detail. The `Comm` class provides the implementation of the Network Interface and acts as a facade to the network subsystem. It has

a unique identifier which can be used as the ownership information for game objects. The class provides appropriate methods for publishing messages and to subscribe to message topics.

In order to get informed about incoming messages for a certain topic, a `SubscriptionHandler` object has to be registered. Whenever an appropriate message arrives, it is passed to the `handle()` method of the corresponding handler. Our implementation provides the needed handlers for the three topics *request*, *update* and *announce*. Depending on the network mode for which the system is configured, the appropriate handlers are chosen. Currently, our implementation supports the three network modes discussed in this chapter. The actual sending over the network is performed by a `NIOServer`. This class is a network server implementation based on the *Java New I/O (NIO)* system. Java NIO provides non-blocking network connections based on connection multiplexing with selectors. Instead of spawning a thread for each network connection, a single thread uses a selector to iterate over all connections and check whether they are ready for reading or writing. Because NIO is able to use native I/O operations of the underlying operating system directly and can handle multiple connections within a single thread, it is very efficient and scalable.

For each connection, the NIO server instantiates a `MessageProcessor` object. This object reads incoming data when its available and reassembles it into messages. Whenever a message has been completely received, it is put into a queue which can be accessed by the server. Outgoing messages are also put into a queue and whenever the connection is ready for writing, the messages in the queue are transferred.

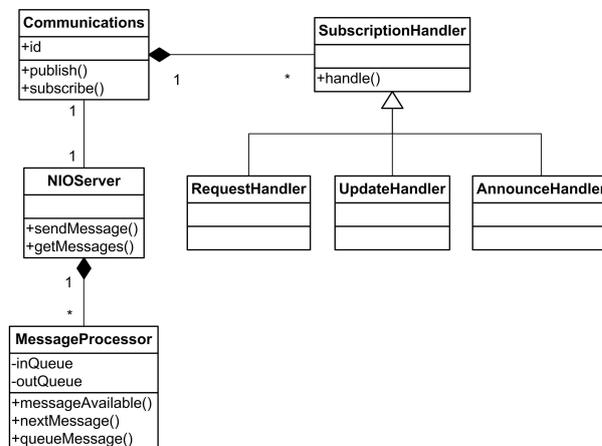


Figure 3.13: Network Layer Class Diagram

## 3.6 Example Game Implementation

For demonstrating the feasibility of our approach, we implemented a game that includes many important aspects found in today's games. These aspects include a graphical representation, changes in object state through player input or progress of time and interaction between game objects. While in our example

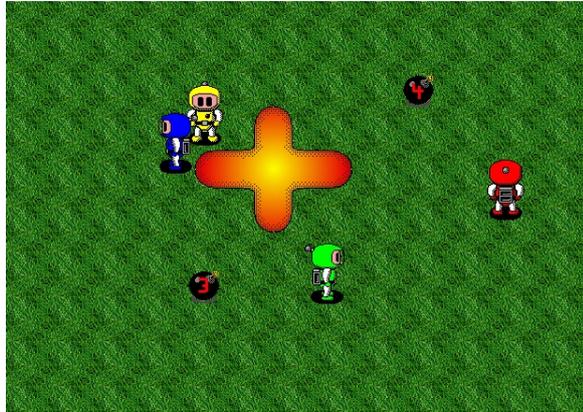


Figure 3.14: Example game

they remain very basic, our framework imposes no limits onto their implementation. For example, rich three-dimensional graphics and sound are possible as well as control of game objects through complex artificial intelligence.

In this section we will give a short introduction on how to develop games using the framework by means of this example game. Our game is a simplified version of a famous multiplayer game concept that has been implemented by the open-source game XBlast [16]. Every player controls an avatar which may move freely around the game field. By pressing a button, he can place a bomb at his current location. Placing the bomb starts a timed detonator and when the countdown reaches zero the bomb explodes. All avatars that are in the vicinity of the detonation are removed from the field and, as in the original XBlast game, the last remaining player wins. Figure 3.14 shows a screenshot of the game.

The starting point for implementing a game is a framework class that we haven't introduced yet, the class `GameNode`. This is merely a convenience class that keeps references to the `ObjectManager`, which is the implementation of the Object Interface, and the `Comm` object. In addition to that, it performs the necessary initialization, i.e. read the network mode from a configuration file and sets up the appropriate object factories as well as the necessary subscriptions.

The generic `GameNode` class is extended by the game-specific class `BomberNode`. Here we add the Presentation Manager, the Input Manager and the Scheduler. First this class initializes a display for the game, either a regular or a full-screen window. Next, it configures the input system either with an AWT or a `JInput` handler and defines the appropriate mappings, e.g. the up arrow on the keyboard is mapped to a "MovePlayerUp" input event. Then the necessary tasks are registered to the scheduler, for instance tasks that read input from the player, update the display and update game objects. Finally, it requests the creation of an avatar object for the local player and starts the scheduler.

In the game there exist two types of objects: player avatars and bombs. The avatars may move around freely and drop a bomb at their current location by pressing a button. As soon as a bomb is placed, it starts a countdown. When the countdown reaches zero, the bomb explodes and all players in its vicinity are removed from the game. All game object types are defined in an external file. The type definition is currently written in XML, but by providing

an appropriate import plug-in, any data format may be used.

Figure 3.15 shows how a such XML type definition may look like. Lines 3 to 8 show the type definition of a player object. A player has a two-dimensional position represented by the `x` and `y` states. The state `facing` shows in which direction the player avatar is facing. Since the player type only stores the current position, the flag `moving` shows whether the player is currently performing a movement or stands still (alternatively, we could use a speed vector). Both, the `facing` and the `moving` states, are used by the rendering system to determine which animation frames to draw. Whenever a player generates an input event, the states of the corresponding object gets updated accordingly. The lines 10 to 49 show the type definition of a bomb object. Like a player object, it has a two-dimensional position. Additionally, `countdown` contains the detonation timer and `accumulatedMillis` is used for accumulating milliseconds, as will be explained below. Lines 16 to 30 contain the `update` method which is regularly called by the update task registered at the scheduler. This method, which is only called by the node owning the object, allows to trigger time-dependent behavior like decrementing the internal counter of the bomb. Note that the scripting engine allows to pass references to any script. In our example, `TIMEPASSED` contains the amount of milliseconds that have passed since the last call of the update method. On each call of the update method, the elapsed milliseconds are accumulated. If the value is larger or equal to a thousand, one second or more has passed. In this case the detonation counter is decreased by one and a second is deducted from the accumulated time. When the detonation counter reaches zero, the `explode` method is called. The `explode` script first creates an iterator over all existing player objects. For this purpose, it uses a reference to the Object Manager to search for all objects that have an identifier starting with the string “player”. It then iterates over all player objects and, if the position of the player is within a certain range of the bomb, removes the player object from the game. Finally, the bomb removes itself.

As mentioned above, the network mode is specified in a configuration file. In addition to that, we need to provide an information where a node can connect to the network. If we use a Client/Server configuration, every client needs the address of the server. In case of the Anti-Cheating mode, every client needs all addresses of the Region Controllers. In the Replicated Simulation configuration, we need the address of a node that serves as a log-in point to the system.

### 3.7 Case Study: Integrating BubbleStorm

In section 3.5 we describe how the pub/sub paradigm is used to abstract from a specific network system. We claim that any system that provides this abstraction can actually be used with our framework. To substantiate this claim, we used the Peer-to-Peer network *BubbleStorm* [98, 99], extended it with a pub/sub interface and integrated it into our system.

BubbleStorm is an unstructured decentralized Peer-to-Peer system, that has some interesting characteristics. It provides an exhaustive search mechanism with probabilistic guarantees. A query is evaluated at the peer that received it, so any kind of query evaluator may be used. BubbleStorm is fast and scalable, in a network with a million nodes a search takes usually less than a second. It exploits the heterogeneity of the nodes’ bandwidth to improve its performance,

```

1 <types>
2
3 <type id="player">
4 <state name="x" default="0"/>
5 <state name="y" default="0"/>
6 <state name="facing" default="south"/>
7 <state name="moving" default="false"/>
8 </type>
9
10 <type id="bomb">
11 <state name="x" default="0"/>
12 <state name="y" default="0"/>
13 <state name="accumulatedMillis" default="0"/>
14 <state name="countdown" default="5"/>
15
16 <script name="update" lang="js">
17 <![CDATA[
18     accumulatedMillis += TIMEPASSED;
19     if (accumulatedMillis >= 1000)
20     {
21         countdown -= 1;
22         accumulatedMillis -= 1000;
23     }
24     if (countdown == 0)
25     {
26         this.execute("explode");
27     }
28 ]]>
29 </script>
30
31 <script name="explode" lang="js">
32 <![CDATA[
33     playerIterator = MANAGER.search('/player.*').iterator();
34
35     while (playerIterator.hasNext())
36     {
37         player = playerIterator.next();
38         playerX = player.getState('x');
39         playerY = player.getState('y');
40         if (((playerX > (x - 50)) && (playerX < (x + 50))) &&
41             ((playerY > (y - 50)) && (playerY < (y + 50))))
42         {
43             MANAGER.remove(player);
44         }
45     }
46     MANAGER.remove(this);
47 ]]>
48 </script>
49 </type>
50
51 </types>

```

Figure 3.15: Example of a game object type definition

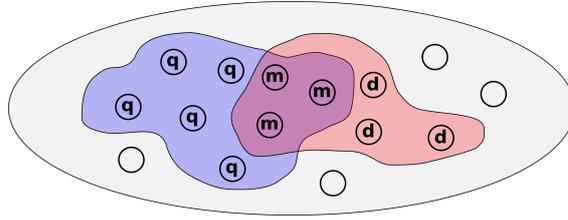


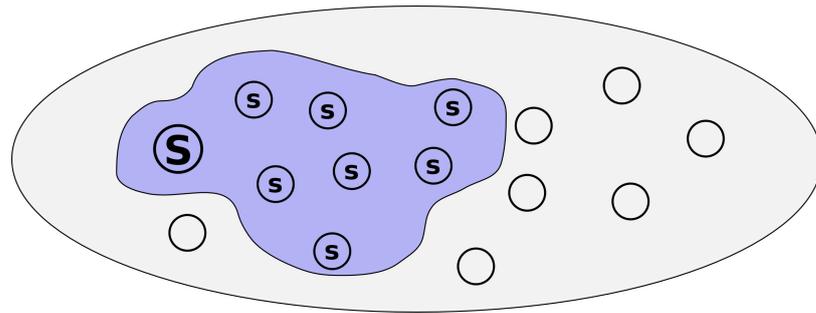
Figure 3.16: Intersecting query and data bubbles in a BubbleStorm network

provides load-balancing that avoids hotspots and is very robust against churn and crashes. BubbleStorm has not been developed with games in mind and thus may not provide optimal performance for this purpose. However, the authors are currently investigating online multiplayer games as a possible use case.

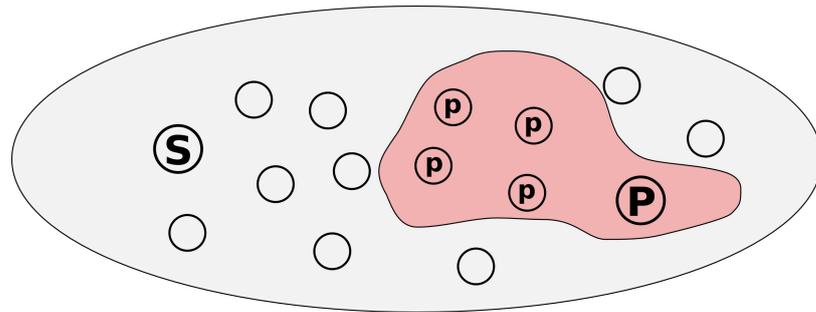
On an abstract level, BubbleStorm is very simple. It replicates both, data and queries, on a certain amount of nodes. The set of all nodes that store a replica of a certain data item or query is called a *bubble*. The intersection of a query bubble and a corresponding data item bubble is the set of nodes that are able to answer the query successfully. In a network with  $n$  nodes where a query is replicated  $q$  times and a data item is replicated  $d$  times, the chance of successfully matching a query is greater than  $1 - e^{-qd/n}$ . For example, if  $qd = 4n$ , the chance of matching a query is greater than 98 percent. Figure 3.16 shows a simplified view of a BubbleStorm network. Nodes marked with a "q" replicate a query, while those marked with a "d" replicate the corresponding data item. The nodes which replicate both are able to successfully answer the query and are marked with an "m". Note that the figure does not reflect realistic ratios between the number of nodes in each set.

Realizing a publish/subscribe abstraction on top of BubbleStorm is rather straightforward. A node that wants to register a subscription stores it into the network. The system creates a corresponding bubble and replicates the subscription on all nodes of this bubble. Figure 3.17(a) shows this process. The node marked with a capital "S" registers a subscription which contains the subscription definition (e.g. a channel name for channel-based pub/sub) and the address of the subscriber. This subscription is replicated on all nodes of the corresponding bubble (marked with a lowercase "s"). A publication is replicated the same way as a subscription (see figure 3.17(b)). Each node of the publication bubble checks whether it stores a subscription matching the received publication. Nodes that are in the intersection of both bubbles forward the publication directly to the subscriber using the address stored within the subscription (figure 3.17(c)). Note that a subscriber may receive a publication multiple times. Nodes that receive a publication matching a locally stored subscription cannot know which subscribers already received this publication.

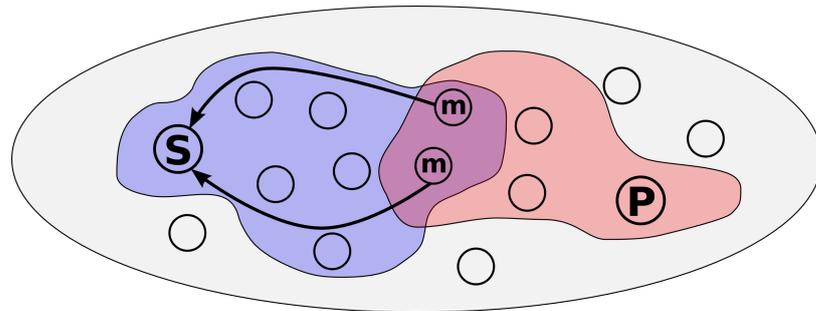
After implementing a pub/sub interface, BubbleStorm can be used by our framework as described in section 3.5.1. In principle, it is possible to run a game in any networking mode. However, using the Classic Client/Server networking mode together with the BubbleStorm network would be pointless since this way we do not utilize any of the system's advantages. On the contrary, BubbleStorm seems to be well suited for large scale multiplayer games running in the Replicated Simulation and Anti-Cheating modes. The scalability as well



(a) Creating a subscription bubble



(b) Creating a publication bubble



(c) Rendezvous nodes send publication to subscriber

Figure 3.17: Publish/Subscribe on top of BubbleStorm

as bandwidth consumption and message latency for networks in the order of millions of nodes outperforms other systems. Since any query evaluator may be used, more powerful pub/sub approaches than the simple channel-based one may be applied. As mentioned above, further research on optimizing BubbleStorm for the needs of multiplayer online games is planned for the near future.

## 3.8 Conclusion

In this chapter we have presented a framework that provides a game developer with a complete abstraction from network related issues. The framework can be divided into three layers: on the highest level the game layer, underneath the object layer and at the bottom the network layer.

On the game layer, standard components, like the game engine and components managing audiovisual feedback and player input, are located. This is also where a game developer has to implement the rules and the logic of a specific game. All components on this layer communicate through an interface with the layer below, the object layer. Game developers can create, manipulate and delete all game objects as if they were local; network consistency as well as ownership management are handled automatically. The networking interface below hides network related issues behind a publish/subscribe abstraction. If it is necessary to optimize the network layer for different quality requirements, like higher scalability or lower latency, custom implementations can be used.

With network implementation details hidden, game developers can focus more on game design rather than writing specialized code. Implementation details like data-driven game objects further emphasize this approach.

### 3.8.1 Performance Impact

In some cases, additional layers of indirection may cause a significant degradation of performance. Since our framework adds two layers of abstraction, we will have a closer look on this issue. The upper layer, which is visible to the game developer, handles the ownership management of objects. As we have seen in section 3.4.3, this is not very complex and in our implementation it boils down to a few lines of code. This layer will hardly have any noticeable impact on performance.

The second layer below handles the pub/sub message dissemination and involves sending data over the network. In order to minimize messaging overhead, a custom implementation for the target architecture should be chosen in favor of a generic pub/sub system. Generic systems may provide a great variety of functionality which isn't always necessary or even useful in a gaming context. Naturally, these systems cause a far greater overhead compared to a custom implementation. Our implementation of the three network modes currently supported does not cause additional messaging overhead in comparison to traditional systems. And in any case the maximum network delay is the roundtrip time between the node sending a request and the node that answers with an update. In the case where the owner manipulates an object directly, only the delay for sending the update is incurred. What is left is the overhead to determine the nodes to which requests and updates have to be sent. By using a naming

scheme that allows mapping of owner names to nodes, only a single operation on a lookup table has to be performed.

### 3.8.2 Cheating

No multiplayer online game today can come along without some protection against cheating, since the possibility to cheat poses a major threat to the fairness of the game.[31, 58, 86] Fairness is a critical factor for enjoying a game and consequently cheating may drive away paying customers. While the Anti Cheating mode is discussed in detail in the following section, we want to touch on this topic in the context of the other two network modes. In the CCS mode, all trust is imposed on the server and our framework doesn't change this. A P2P node within the Replicated Simulation is responsible for the object it owns. However, all peers receive updates about changes of that object and they may check themselves if those changes conform to the rules of the game. Otherwise they may reject an update.

The only thing the framework has to guarantee is that no one is able to forge messages. E.g., if a node receives an update, it must be sure that the sender is really the owner of that object. Nodes may simply be identified by IP addresses or, if a higher level of security is necessary or object ownership must outlast network sessions, cryptographic signatures may be used. For this purpose a public key infrastructure is necessary which can be run by the game publisher.

# Chapter 4

## Resilience against Cheating

### 4.1 Introduction

In this chapter we will evolve a cheat-resistant Peer-to-Peer game system design. First, we give a characterization of our system and an introduction to the topic of cheating. Next, we start the discussion of our approach on a rather abstract level to point out the main concepts. Later on we will discuss its implications in more detail and give a in-depth description of the actual system and how it handles different cheating scenarios. Finally, we discuss scalability issues and the general applicability of our approach.

#### 4.1.1 System Classification

Before we can classify our system, we give a very brief overview of its structure.

The objective of our system is to shift the computational load and network bandwidth consumption from the server to the players' computers. The basic idea is that a player's node may act at the same time as a client and a server for different parts of the game world (which are referred to as a *regions* in the following). As will be described later, a region is replicated among multiple nodes in order to prevent cheating. Players whose avatars are located in a certain region connect as clients to the player nodes that act as a server for a replica of that region. This way, the game publisher is relieved from the resource-intensive task of providing servers for the game world.

According to the definitions given in [93], our system classifies as a Peer-to-Peer system for the following reasons:

- Resources like bandwidth, storage and processing power are located on the peer nodes. Each peer utilizes resources provided by other peers.
- In order to utilize these resources, peers directly interact with each other over a network.
- Each peer can act both as a client and a sever for a game region. All peers are equal partners with symmetric functionality.

To be more specific, our system falls into the category of an *unstructured centralized P2P system*. The system is *unstructured* in the sense that the content

(i.e. a game region replica) stored on a certain peer and the peer's IP address are unrelated. Assignment of region replicas to peers is described in section 4.3.3. Our system is *centralized* since it needs a central server that acts as an entry point to the network and manages the assignment of regions to peers. However, the central server is only needed when nodes join or leave the system. In contrast to centralized P2P systems like Napster [77], where the server has to be contacted for every query operation and thus poses a potential bottleneck, the resource demand on the server side is minimized.

### 4.1.2 Definition and Taxonomy of Cheating

Though cheating is rampant in today's online games, there is often no clear understanding of this topic and a lack of terms and definitions. Yan and Randell [110] were the first to give a rather comprehensive overview of cheating in online games and to define a cheating taxonomy. They define *cheating* as

...any behavior that a player uses to gain an advantage over his peer players or achieve a target in an online game [...] if, according to the game rules [...], the advantage or the target is one that he is not supposed to have achieved.

Their cheating taxonomy consists of three dimensions:

**By vulnerability.** A cheat can be performed either by exploiting a flaw in the game system or vulnerabilities of the people involved in the game. The former includes implementation errors and game design flaws, the latter social engineering attacks and abuse by insiders (e.g. game operators).

**By consequence.** Players can try to violate the integrity of the game, e.g. making their avatars more powerful by raising their strength or their hit-point values. They can also achieve unfair advantages by gaining access to confidential information, e.g. finding out about the position of hidden enemy players. Another possibility to put other players at a disadvantage is the denial of services they want to use. The whole purpose of cheating is the violation of fairness. The paper lists this as a separate consequence. However, we believe that fairness violation is rather a subsumption of the consequences mentioned before.

**By cheating principal.** Cheats can be performed by players, game operators or a cooperating group that may include both.

Yan and Randell do not mention a different kind of cheaters, the so-called "griefers" [56]. As the name implies, the sole intention of these people is to hurt other players' game experience. While griefing may actually be performed without breaking any game rules (e.g. insulting other players through the player chat), griefers may also exploit possible cheats to hurt other players, e.g. killing their avatars or stealing their items. The difference between cheaters and griefers is that griefers do not expect any game-related benefit from their actions. From a technical point of view, the possible attacks for griefers are the same as for regular cheaters. However, one should keep in mind that griefers may tend to use those attacks that are not particularly attractive to others.

### 4.1.3 Cheating Attacks Specific to P2P Online Games

Many of the cheating attacks mentioned by Yan and Randell also apply to traditional Client/Server architectures and countermeasures have already been developed in this context. In this work we do not address attacks like exploiting game design bugs and implementation errors, hacking into servers or compromising passwords through social engineering. Instead, we will focus on those attacks that we identified to be inherent (but not necessarily exclusive) to a Peer-to-Peer online gaming system. These are

**Exploiting Misplaced Trust** In a Peer-to-Peer online gaming system, the software as well as game state data may be stored locally on players' machines (which we assume to be untrusted, we will come back to this issue later). This makes them susceptible to any kind of malicious manipulation.

**Exploiting Lack of Secrecy** As stated above, all game state is stored on untrusted nodes. Without further protection a node is not only able to access all data that is stored locally but it may also disclose it to other nodes.

**Collusion** Any untrusted nodes within the system may collude in performing cheats.

It is unclear why the paper lists *Collusion* as a separate cheating attack and at the same time mentions cooperating attackers in the cheating principal dimension. We consider *Collusion* as being orthogonal to the other two attacks since both, *Exploiting Misplaced Trust* and *Exploiting Lack of Secrecy*, can be performed either by a single player or by multiple colluding players. Consequently we will discuss both in the context of a single attacker and multiple colluding attackers.

Within the first dimension of the cheating taxonomy, the vulnerability, both attacks above fall into the category of a game design flaw and thus have to be addressed by the design of the system. This is discussed in section 4.2. Within the second dimension, the cheating consequence, they either violate integrity or disclose confidential information. How these attacks affect gameplay will be discussed in the following section. Within the last dimension, the cheating principal, the attacks can either be performed by a single player or by multiple cooperating players. The differences that arise from this are discussed in the respective sections on the cheating attack scenarios.

### 4.1.4 Impact of Successful Attacks

We consider *Exploiting Misplaced Trust* by manipulating game logic to be the most dangerous of all possible attacks, since it is relevant to any kind of game and can have an arbitrarily high impact. In games, where the player directly controls a virtual character, a cheater may make himself invincible by altering his avatar's attributes. In a racing game he can raise the speed of his vehicle, while in a strategy game he can provide himself with unlimited resources or money. As soon as a cheater has the possibility to modify the game state or logic, his options are virtually unlimited. The system we propose focuses on this kind of attack and thus provides appropriate countermeasures.

In contrast to that, the relevance of *Exploiting Lack of Secrecy* is heavily dependent on the kind of game. In a strategy game, for example, knowing the position of enemy troops can give a crucial advantage over other players and eventually decide over winning or losing. But in games where reflexes and other skills are more essential than strategy, information about other players' avatars are much less relevant. Often these game do provide this information anyway, e.g. players can directly see the health status of others within the game. Note that this kind of attack can only partially be addressed, since one cannot prevent a player that has legal access to a certain piece of information from disclosing it to other players. For example, for team-mates in online games it is quite common to use an external voice channel. Nobody can prevent them from exchanging information through the channel that cannot be exchanged directly within the game. Our system tries to hide data from prying eyes as much as possible.

## 4.2 Main Concepts

Before we can start a detailed description of our system, it is necessary to introduce the basic principles by which the attacks identified above can be counteracted.

### 4.2.1 Addressing Misplaced Trust

In traditional Client/Server online games, misplaced trust is not a major issue. Assuming that the system is properly designed, all data is stored and processed out of the reach of a client on a trusted server. Clients usually act as a graphical terminal which takes input from the player, sends it as a request to the server and receives an update of the game state which it displays on the screen. A client may cache data and perform its own calculations on it but only for local purposes. Only the data stored on the server is authoritative and clients never exchange any data directly.

A regular node in a Peer-to-Peer system is usually not trusted because the player has unrestricted access to the software and the data that is stored on his computer. There exists an attempt to improve the trustworthiness of computers that are not under direct control of a trusted authority. However, this approach (called *Trusted Computing*) has certain shortcomings which are discussed in section 2.3. For our system, we assume that a client is inherently not trustworthy and thus any data that it stores and any result that it computes may be falsified.

On the one hand, we want to utilize the computing capacity of the players' machines. On the other, we cannot trust the data that the node stores and computes. Obviously, we need a way to check the validity of the information from the node. Having a trusted server running in parallel to reproduce and check the results of the nodes would be pointless. If it has to perform all the calculations again, we have won nothing but incur an additional message passing overhead. Having a trusted server performing only random samples occasionally does not work, too. In order to reproduce a node's calculation the server would need to know the exact state of the game before and after the calculation. But since the server does not keep track of the whole game state, it would need again

to trust the node to send the correct states. The only option that is left is to let other peers in the system check the results that a node produces.

Since there is no way of directly judging the trustworthiness of a node, we cannot rely on a single one to perform this checking. However, it is safe to assume that most of the players are honest and would report a malicious node if they notice one, since it is disrupting their game experience. If a significant part of the players were trying to cheat, playing a game wouldn't be a lot of fun even if measures are taken to prevent cheating. If we replicate the processing of game state data on randomly chosen nodes, the majority of them will very likely agree on the correct state.

### The Byzantine Generals Problem

The agreement problem that arises here reminds us of one well-known to computer science: the *Byzantine Generals Problem* [65]. In order to illustrate the problem, Lamport et al. describe a scenario where a group of generals of the Byzantine army are camped with their troops around an enemy city. A commanding general issues an order to the camped lieutenant generals (e.g. whether to attack the city or not). All generals may communicate only via messenger and one or more of them (including the commanding general) may be traitors which try to confuse the others. The generals need an algorithm to guarantee that

- a) all loyal lieutenant generals follow the same order.
- b) if the commanding general is loyal, every loyal lieutenant general follows his order.

The paper shows that, assuming signed messages are used, an agreement (called *Byzantine Agreement*) can be achieved if there are at least two loyal generals. Signed messages mean

- a) a loyal general's signature cannot be forged and any alteration of his message can be detected.
- b) anyone can verify the authenticity of a general's signature.

An important contribution of Lamport et al. was the introduction of the *Byzantine error model*. This model assumes that nodes may not only crash and simply stop functioning, but they may also malfunction without stopping. A malicious node can be seen as a node that fails in a Byzantine manner, since, from a technical point of view, it makes no difference whether a node exhibits a wrong behavior intentionally or not.

Now, how is the Byzantine Generals Problem similar to our situation? The commanding general can be seen as a player that sends a request to change the state of the game. If the request is legal with regard to the game rules (i.e. the commander is loyal), all loyal peers will fulfill the request. If not, all loyal peers will drop the request and agree not to make a change to the game state. Cheating nodes could perform illegal changes to the game state that benefit them and try to make the other nodes to agree on this change.

Unfortunately, the original Byzantine Agreement solution is very expensive in both the amount of time and the number of messages required. In case there

are  $m$  malicious nodes, the algorithm requires message paths of length up to  $m + 1$ . This would cause a messages latency that is intolerable for todays online games. Moreover, the additional traffic generated by the amount of necessary messages would slow the system down even more (please refer to the paper for a calculation of the message overhead). However, we can avoid this overhead by relaxing our requirements.

We need the loyal nodes to agree on a certain state change in the presence of Byzantine nodes. However, it is not necessary that the loyal nodes *know* that they agree. Above we made the assumption that most of the nodes belong to honest players. If they receive the same legal request, they perform the correct change without the need to communicate with each other. Afterwards they send an update that reflects the change to the interested nodes. Since the majority of updates that any receiver gets are equal, the receiver knows which update to accept.

Naturally, our simplified approach has some drawbacks. While the Byzantine agreement (using signed messages) can tolerate that  $m$  out of  $m + 2$  nodes are malicious, we can tolerate only  $m$  malicious nodes out of  $2m + 1$  nodes. This meets the assumption we have made above. A special case we have not mentioned yet is when a node sends different legal requests to the other nodes. In this case each node will perform a different but legal change and send an appropriate update. The nodes receiving the different updates cannot determine the correct one anymore. As we will see later, our system can only *detect* those attacks (by using signed messages), but cannot *prevent* them in the first place.

## 4.2.2 Addressing Lack of Secrecy

If we look at the traditional Client/Server scenario, protecting confidential information is straightforward. All data is stored on a trusted server and each client receives only the information that it is allowed to have. For example, the position information of enemy troops which are outside the vision range of a certain player is not send to that player's node. Restricting the flow of information to a client is usually referred to as *Interest Management* [50]. Interest management is not only used for keeping information confidential; an important application is to save bandwidth consumption by avoiding to send unnecessary information.

As mentioned above, once a player has gained access to a certain piece of information he is free to forward it to any other player. It is common practice today that players of online games communicate through voice channels in order to coordinate their actions. Nobody can prevent a player who has seen an enemy hiding behind a corner from informing another player who approaches the hiding spot unsuspectingly. Most of the online games actually benefit from such information exchange as it adds an additional strategic component to the game. However, there are games which become pointless if such an exchange is performed. Think of a digital version of a card game like *Bridge* [108], where two players form a partnership but they do not know each other's cards. Disclosing information about the cards on your hand to your partner would destroy the whole appeal of the game. Since there is no way for a game provider to prevent this, certain games are not suitable for playing online if the players don't trust each other.

Keeping data confidential in a Peer-to-Peer system can be done in two ways.

First, a piece of data is only stored on nodes that are allowed to access it. Second, if confidential data must be stored on a node that is not allowed to access it (e.g. for caching or relaying purposes), it must be encrypted. Obviously, the same problem as in a Client/Server system arises. A node that may legally access a piece of data may disclose it to other nodes through channels external to the game. The system we propose distributes game state data in a way that minimizes the chance that one of two colluding nodes has access to confidential data that may be of interest for the other.

### 4.2.3 Preventing vs. Detecting Cheating

In some cases it is not possible to prevent cheating in the first place. Above we mentioned the case that a malicious player sends different but legal action requests to different game state replicas. If the replicas do not communicate with each other, none of them can detect this situation and will process these request, resulting in different states at the replicas. Eventually, if enough replicas arrived at a different state, it becomes impossible that any update achieves a majority. No later than this the attack will be detected and the question arises who is to be blamed. If we require every request to be signed, it is easy to detect the origin of the attack. Every replica can prove that it received a request that differs from the others since it was signed by the sender.

Detecting attacks only after they have been successfully performed may sound unattractive at first. However, remember that all players are paying subscribers of the online game service. If their cheating attempt is detected, they will most likely be banned from the game service. They lose all their achievements in the game and at least the money they have paid for their current subscription period. Additionally, the terms of conditions could require a subscriber to pay a fine for disrupting the service or the game publisher could even take legal actions. Since a cheater can be clearly identified by his signature, it is very unlikely that a player will take this risk.

## 4.3 Region Replication

We have discussed so far that the game state and logic is replicated on peer nodes. Whenever a player's node requests a change of the game world, it will send a corresponding message to the replicas. The replicas perform the change and send updates to all nodes that have to be informed about the change. Finally, the receiving nodes compare the updates and accept the one that holds the majority. This explanation is rather abstract, so in the following we will go into more detail.

### 4.3.1 Partitioning of the Game World

Depending on the game, the size of its state and the computational resources to manage it varies heavily. Session-based games with few players usually have a very small state and low resource consumption so that in a Client/Server system the server may host multiple sessions at once. On the contrary, the state of a single Massively Multiplayer Online Game world sometimes is huge so that it has to be distributed over multiple servers. Though players usually own rather

powerful computers, their computing power cannot be compared to that of a dedicated server. While smaller session-based games can easily be managed by a player machine, it is obvious that it cannot handle the complete state of large online games.

Since splitting large game worlds into parts that can be handled by single servers is a common approach, we will adopt it for the players' computers as well. Game worlds usually represent two- or three-dimensional space, so a segmentation into spatial regions is the most natural one. Managing game world regions on different nodes is straightforward, as long as there is no interaction between them that would require a synchronization of shared objects. Every node that hosts a region manages only the game objects contained in it and performs all the logic necessary, i.e. it receiving change requests, performing the changes and sending updates to interested players. Of course regions may not be completely separated from each other. In order to form a contiguous world, players must at least be able to travel between regions. Speaking technically, moving from one region to each other means a transfer of a player object between the nodes that manage the regions. Since the object simply disappears from one region and reappears in another this still doesn't introduce any shared state and thus there is no need for synchronization between regions.

Having separated regions which only allow players to move from one region to another but not any further inter-region interactions is still prevalent in today's online games. Instead of aiming at seamless game worlds that hide region borders from players with complex synchronization techniques, modern game providers go in the opposite direction. Nearly every successful MMOG today makes use of so-called *instances*. An instance is a separate region of the game world that is only shared by a small group of players (usually between five and forty). Different groups of players can occupy the same region, but each group gets their own copy of the region from scratch. The benefit for the player group is that they can explore the region, kill monsters there and loot the treasures without being disturbed by other players. The most successful MMOG today, *World of Warcraft* [14], uses instances for special parts of the world, mostly dungeons. Another very successful online game, *Guild Wars* [2], uses instances for nearly all areas. Only gathering places like cities that serve as connection points between instance regions are managed in the usual way. For the game provider, the most important advantage of instances is that they avoid costly synchronization between regions that would eventually limit the scalability of the game in terms of the number of simultaneous players. Having small and separate regions allows the provider to create as many of them as necessary with negligible overhead. Moreover, instances can be spawned on any server that is currently not working at its full capacity. The small size of instances makes their resource consumption highly predictable, thus allowing for an optimal utilization of resources. The huge success of the instance concept shows that partitioning game worlds into rather small regions is not a serious limitation for the game experience as long as it is properly integrated.

Note that when we talk about inter-region interactions, this does not include communication systems which allow sending text or voice messages to other players. These systems can be seen as external services that have no direct effect on the game state and do not require any synchronization of in-game objects.

### 4.3.2 Distribution of Game State and Logic

We have discussed how to split the game state into smaller chunks that are small enough to be handled by players' computers. Next, we show how these chunks are distributed among the peer nodes. For every region of the game world we need a set of nodes that manage the state of that region. For the rest of this work we refer to these nodes as *Region Controllers (RCs)*. Every Region Controller manages the complete state of its corresponding region, i.e. the state of a region is *replicated* on all its RCs. The counterpart of a Region Controller is the *Game Client* which represents a player in the game. Every Game Client sends requests according to the player's desired actions to the RCs that manage the region in which the player is located. The Region Controllers process the request and send updates to all Game Clients of the region that need to be informed about the change. Finally, the Game Clients display the changes onto the players' screens. Note that "Region Controller" and "Game Client" are just roles played by a node in the system. Every node can play both roles, even at the same time. The interaction between the two roles is nearly identical to that between a client and a server in a traditional Client/Server online game.

Now how do nodes become Region Controllers or Game Clients? The assignment of roles to nodes is handled by a *Management Service* that is offered by the game provider and serves as the central entry point to the system. This service may also take care of subscription management, accounting and billing. E.g. it would reject a player that hasn't paid his subscription fees or was banned from the game.

In the bootstrapping phase of the game, where no or only a few nodes are online, the service offers some initial Region Controller instances. Because of the low number of players at that time this not a very resource-intensive task. Note that because these initial RCs are run by the game provider, they are trustworthy and thus a single one is enough to manage a region (there is no mutual checking of RCs required). As more nodes join the system, the initial Region Controllers can be replaced with Region Controller groups that consist of regular peer nodes. As soon as there are enough nodes in the system, the initial Region Controllers are not needed anymore.

Whenever a player wants to enter the game his node contacts the Management Service. Depending on the region in which the player starts (e.g. the same region as he was in when he left), the player's node receives the list of Region Controllers that are currently responsible for that region. The RCs are informed about the new Game Client and from now on both sides can start communicating with each other. His node is also added to the Region Controller pool. This means that whenever a new RC is needed (e.g. to replace one that left or has failed), it can be taken from the pool. Usually we need significantly less Region Controllers than there are nodes in the system (an issue that will be discussed later). For this reason, there is no danger of running out of pooled RCs. Only if the total number of nodes is very low, some initial Region Controllers have to be provided.

We assume that players only join the system when they actually want to play, i.e. their node always becomes a Game Client. However, it is thinkable that players offer to act as an Region Controller even if they do not play themselves currently. They simply grant their unused resources to the system. The Management Service could account for the time that a node spends as an RC.

As a reward, players could get discounts on their subscriptions or — even more tempting for hardcore players and free of cost for the game publisher — exclusive items and abilities for their avatar. This way a game publisher can easily make the donation of computing power to the system very attractive.

### 4.3.3 Replica Selection

Whenever a node that acts as a Region Controller leaves the system, it needs to be replaced to maintain the desired number of replicas per region. For this purpose, the Management Service keeps a pool of available nodes that may become a Region Controller. Choosing nodes from this pool could simply be done in a random fashion, but there exist more sophisticated selection methods. In the literature, this problem is often referred to as the *Referee Selection Problem* [106, 105], where referee is a synonym for Region Controller.

There are several criteria which may be considered when selecting a Region Controller:

**Security.** First, we want to avoid that a node is a Game Client and a Region Controller for the same region at the same time. This way we can prevent a cheater from directly accessing data of the region where his avatar currently resides. Furthermore, we want to keep the probability that colluding cheaters become Region Controllers for the same region as low as possible.

**Responsiveness.** The network delay between a Game Client and any of its Region Controllers should be as low as possible.

**Fairness.** There should be no large variation of the delay between different Game Clients and the Region Controller.

Note that fairness and responsiveness are conflicting goals. If one of the Game Clients has a connection with a high delay, fairness would mean to artificially delay updates to the other Game Clients and thus lowering the overall responsiveness. Webb et al. [107] have proposed two algorithms for selecting Region Controllers out of pool of available nodes, namely SRS-1 and SRS-2. The first aims for responsiveness while the second aims for fairness. Depending on the requirements of the game, a game developer has to choose between the two algorithms.

Both algorithms determine a set of nodes with the chance that a majority of them is corrupt being very low. For this purpose, they use a selection protocol proposed by Corman et al. that allows to minimize this probability to less than  $10^{-5}$  [29]. This protocols ensures that our basic assumption, that the majority of Region Controllers for a region is honest, holds true. SRS-1 tries to find Region Controllers that have a low average delay to their clients, accepting that the delay variation may be high at the cost of fairness. In contrast, SRS-2 tries to minimize the delay variation at the cost of a higher average delay, thus reducing responsiveness. Both algorithms offer to artificially inflate the delays between nodes for a fine-tuning of the trade-off between fairness and responsiveness.

### 4.3.4 Consistency

In section 4.2 we described that the correct game state is determined by the Game Clients through a voting mechanism. All Region Controllers of a region send their updates to each Game Client and the one that holds the majority is taken as the correct one. Yet the question remains, how do the Region Controllers arrive at the same state? A *consistency model* defines what a node in a distributed system gets when it performs a read operation on its local replica of the global state.

Many different consistency models have been presented in the literature, [92] contains a rather comprehensive overview. Ideally, one would expect that any change that is made to the global state is reflected immediately on all replicas. However, because of the latency incurred by the underlying network, remote operations are always performed with a certain delay. This wouldn't be a problem by itself, since most applications could accept this small delay. However, whenever two nodes perform a remote write operation on the same data item nearly at the same time, the order of execution could depend on how long the appropriate message takes to arrive at the receiver. If write messages arrive in different orders at different nodes, a subsequent read operation on that data could return a different value on each node. Therefore, a consistency model has to determine the order of execution of read and write operations.

The most intuitive model, *Strict Consistency*, requires that any read operation on a data item returns the value of the most recent write operation. Put in other words, all operations are executed exactly in the order they were issued. This requires that every operation can be assigned an unambiguous timestamp according to some global clock. Unfortunately, the clocks of network nodes cannot be perfectly synchronized to a global reference time. This is the reason why achieving strict consistency is not feasible in a distributed system [96]. For this reason, weaker consistency models have been proposed. *Weaker* refers to the more relaxed assumption that there has to be *some* global order of execution which is not necessarily the same order as it would be seen by a global clock.

Lamport proposed the *Sequential Consistency Model* [64] which can be described by two requirements. First, all operations issued by a certain node have to be executed in the order they were issued. Second, operations issued by different nodes must be executed in some global order. Together, both requirements guarantee that there is a total ordering of all operations in the system. Thus, all nodes will perceive the same global state.

*Linearizability* [49] (also known as *Atomic Consistency*) is similar to sequential consistency. As stated above, it is not possible to synchronize clocks of different nodes exactly. Because of this, linearizability only requires that every operation is assigned a time interval instead of an exact point in time. The size of the time interval can be chosen depending on the accuracy of the clock synchronization that is in use. To ensure linearizability, sequential consistency must be ensured and the resulting sequential total order must correspond to an order that can be achieved by placing each operation at a single point in time within its time interval. Essentially, if two operations' time spans do not overlap they must be executed in the correct timely order. Linearizability is a slightly stronger model than Sequential Consistency but also more expensive in terms of worst-case response time [5].

There exist even weaker consistency models that only guarantee a partial

global ordering of operations. For example, the *Causal Consistency Model* [1] only guarantees that causally related operations are executed in the same order on each replica. Writes that are not causally related (so called *concurrent writes*) may be executed in different orders. Only if a program meets certain conditions (please consult the paper for details) the causal consistency model produces histories that are also sequentially consistent.

Our voting mechanism requires that Region Controllers arrive at the same state. A consistency model that does not enforce a total global ordering of operations is not sufficient. Thus, our system must maintain the replicas at least sequentially consistent. As described above, sequential consistency requires that all requests sent by a single node are processed in the order they were issued by that node and that there is some global sequence of all requests. To achieve this, we can exploit the fact that all games break down time into discrete slices, called *frames*. Sometimes games use frames of variable length but in our system the length of each frame is equal. Every client may issue a single request per frame and transmits the frame number within each request. Since the frame number is simply incremented for each frame, the sequence of requests sent by a single node can easily be maintained. To achieve a total global ordering of requests sent by all nodes we only need to find a global order for each frame. Within a frame they can be processed according to a fixed and unique client id that is assigned by the Management Service. Of course this does not retain the original order the requests were made by the clients. But this would mean striving for strict consistency which is not feasible in a distributed system anyway as explained above. Since frame sizes are usually very small — much less than a second — reordering requests within a frame will not be noticeable to players. The order of request processing can be changed every frame (e.g. rotating the order). Otherwise the player with the lowest id is always the first to act, which would probably affect fairness.

Now that we know how Region Controllers can determine a fixed order for incoming requests, we still need to describe how all nodes of a region can keep their frame advancement synchronized. At the beginning of a frame, the Game Clients send their requests. After processing these requests, the Region Controllers finish the frame by sending their updates. Of course we cannot wait until all Game Clients have made their requests or all Region Controllers have answered with an update. Otherwise we would enable a single malicious node to slow down the game arbitrarily. That is why we chose frames with a fixed length. After the time of a frame has elapsed, each nodes starts with the next frame. To keep the frame advancement synchronized, the clock deviation of the nodes may not exceed a certain limit. For this reason we make use of the *Network Time Protocol (NTP)* [72]. In section 5 we evaluate the effects of clock skew on our system and show that the deviation error introduced by using NTP is acceptable for our system.

Until now we have only talked about consistency among replicas. Since the client software running on the player’s node only visualizes the update sent by the majority of replicas it is always consistent with them. The only thing we have to care about on the client-side is the delay caused by the request/reply round-trip. Since the quality of the players’ game experience is affected by this delay [83], it can be hidden from the player using techniques like *Dead Reckoning* [82] or *Pre-Reckoning* [34].

### 4.3.5 Update Propagation

The approaches to update propagation can be differentiated into *Active Replication* and the *Primary-Backup Approach* [17]. Active replication means that requests made by clients are sent directly to all replicas. Following the primary-backup approach, requests are sent to a single replica, called primary replica (PR), which processes the request. The remaining secondary replicas (SRs) receive a state update message from the PR after the request has been processed.

The primary-backup approach bears the danger that a malicious primary replica could send falsified updates. Although this could be detected later, we want to prevent this right from the start. We could allow the PR to relay only client requests instead of sending complete updates to the secondary replicas. Since requests sent by clients are signed they cannot be modified by the PR. The only option left for a malicious PR is to drop client requests instead of relaying them to the SRs. If primary-backup update propagation is applied, sequential consistency can be achieved easily, since all updates are applied directly to the primary replica. The SRs will perform updates with a delay but this does not affect gameplay as long as the PR is working correctly. If the PR fails, one of the SRs has to take over. Any updates that have not already been applied to the SRs have to be resent by the clients which could delay gameplay significantly. Probably the biggest problem is that an SR with a corrupt game state (because of cheating or other faults) may become the PR in case of a failure. It is not yet clear how a new PR can be elected without opening new loopholes for cheaters.

Our system proposed in this paper uses the active replication since there are no solutions yet for the problems related to the primary-backup approach. But future research will probably yield solutions for these problems, providing an alternative system model that can be evaluated and compared to the current system.

## 4.4 Normal Operation

Up to now, we have described the individual parts and concepts our system is composed of. In this section we describe how the parts of the system work together under the assumption that no malicious nodes are present.

### 4.4.1 Bootstrapping

The starting point of our system is the central Management Service that is provided by the game publisher. It has a public Internet address that is known to all nodes in the system. Whenever a node wants to join, it starts by contacting the Management Service.

Initially, the game world is empty, i.e. it is not populated by any players. When the first player joins the game, he will start in one of the game world's regions. Since there are no Region Controllers available yet and a node may not become RC and Game Client of the same region at the same time, the Management Service will provide an initial RC instance for his starting region. Remember that a single initial Region Controller is sufficient, since it is run by the game publisher and thus trustworthy. As more players join the game, other regions of the game will be populated in the same manner. However, more players mean more nodes that can act as Region Controllers and serve

the regions of the initial RCs. At a certain point in time, there will be enough Region Controllers provided by player nodes so that an initial RC can withdraw from its region. Usually the number of RCs per region will be lower than ten, while the number of players may reach several dozens or even several hundreds. This ensures that the number of nodes in the system far outnumbers the number of required Region Controllers and the system will not run short of available RCs. Only if the number of nodes drops to a very low level, RCs provided by the game publisher have to take over some regions again.

#### 4.4.2 Game Client Login

The process of a Game Client logging into the system is shown in figure 4.1 as an UML sequence diagram. Note that this and the following sequence diagrams show only two exemplary Region Controllers and Game Clients respectively.

1. Since all players are subscribers of the game service, they are registered at the Management Service. Whenever a player starts his Game Client, it contacts this service.
2. The client receives all the necessary information, e.g. a unique player id, a pair of cryptographic keys to sign its messages, the public keys of its responsible Region Controllers and any other necessary information.
3. The Management Service informs every RC of the region in which the player starts about the new player. This includes at least the Internet address of the new node, its public key and any initial state of the player (if he doesn't start from scratch).
4. After the Region Controllers have learned about the new player, each of them sends the state of the player's immediate surroundings to his node.
5. The Game Client of the player waits until it has received the same initial game state from a majority of Region Controllers.
6. Now it can start the request-update cycle that is repeated until the player leaves the game again. The Game Client receives the commands from the player and send an appropriate request to all RCs. Each of the Region Controllers process the request and generates an update. This update is sent back to all interested Game Clients, which compare all of them and choose the one that holds the majority.

#### 4.4.3 Game Client Logout

Figure 4.2 shows the process of a Game Client logging out of the system.

1. When the player wants to exit the game, the Game Client sends a log-out message to the Management Service.
2. The service informs all responsible Region Controller about the leaving of the player and they stop sending updates to his node.
3. If any state of the player needs to be saved, the RCs may send it to the Management Service. As soon as the service received the majority of player state messages it adopts the contained changes.

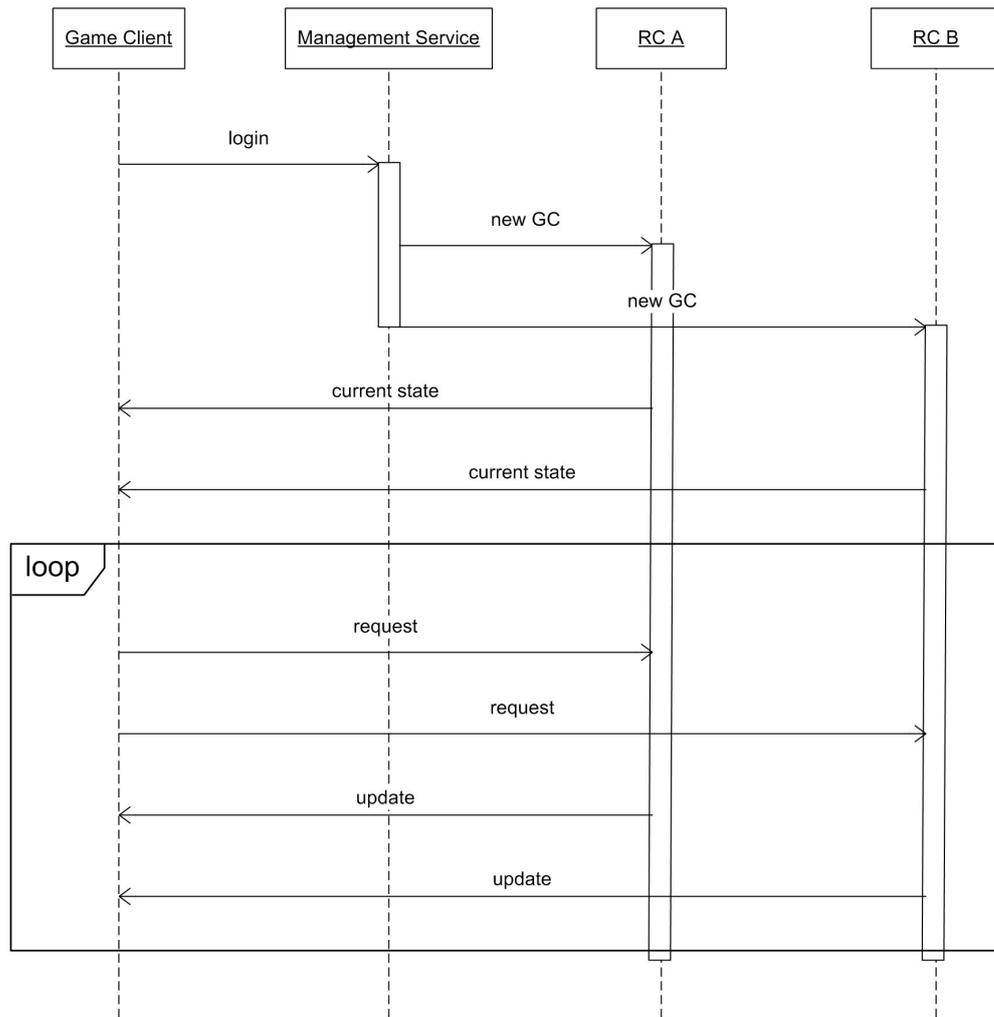


Figure 4.1: Game Client login procedure

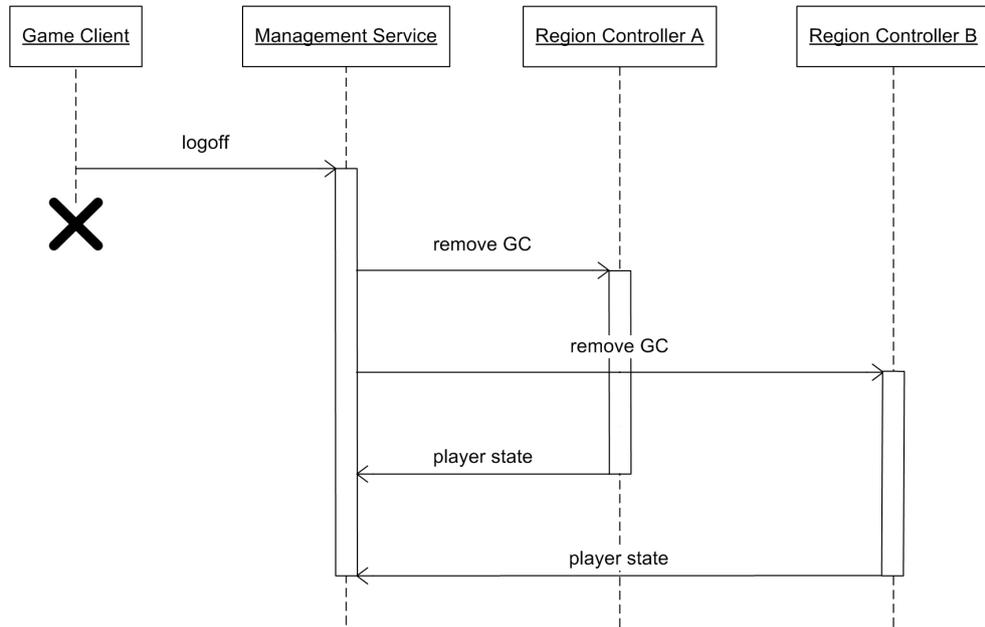


Figure 4.2: Game Client logout procedure

#### 4.4.4 Region Controller Login

Figure 4.3 shows the process of a Region Controller logging into the system.

1. At the same time when the player's Game Client logs into the system, the node registers itself as an available Region Controller. The Management Service adds the node to the list of pooled RCs.
2. Whenever a new Region Controller is needed, e.g. if currently active Region Controller leaves the system or an initial RC needs to be replaced, an RC from the pool is activated. The Management Service sends a message to the new RC containing a list of Game Clients that it needs to serve.
3. In case that the new Region Controller replaces another, the Management Service informs the other RCs that still manage the region. They start to send the current state of the region to the new RC.
4. The Management Service informs all Game Clients of the region about their new Region Controller. The GCs start sending requests to the new RC. At this point, the Game Clients do not expect to receive updates from the new RC yet.
5. After some time, the new Region Controller has received the state of the region from the majority of the other RCs. This data is already outdated, because it reflects the state of the game at the time the RC was activated. However, since the new RC received all the client requests since its activation, it can apply them to the outdated state and bring itself up to date.

6. Now the Region Controller can start to serve the Game Clients of the region through the normal request-update cycle as it is described in step 6 in section 4.4.2.

#### 4.4.5 Region Controller Logout

Figure 4.4 shows the process of a Region Controller logging out of the system.

1. Whenever a player leaves the game, it unregisters the Region Controller running on its node.
2. The Management Service sends a message to each Game Client that is served by this Region Controller. They stop sending requests to this RC.
3. If the RC has just been a pooled RC, it is simply removed from the pool. If it has been an active RC, the activation of a new RC is triggered (see step 2 in section 4.4.4).

#### 4.4.6 Player Changing Regions

Figure 4.5 shows process of a player changing from one region to another.

1. Leaving a region, as any other action, starts with a request of the player's Game Client to all Region Controllers. Depending on the actual game, it can mean that the player's avatar enters a portal or crosses a certain border that represents the connection point between two regions.
2. All Region Controllers transmit the current state of the player to the Management Service. As usual, the correct player state is determined by a voting process.
3. The player state that has been elected is now transmitted to all Region Controllers of the region the player wants to enter.
4. All new Region Controllers transmit the current state of the region to the Game Client.
5. As soon as the majority of RCs sent the correct state, the Game Client can start the usual request-update cycle.

### 4.5 Cheating Attack Scenarios

In this section we discuss the various attacks that may be performed by Game Clients, Region Controllers or a combination of both. We show that attacks which procure a direct benefit for the performing malicious node can be prevented in the first place. Attacks which do not benefit the attacker but just negatively affect the experience of other players (e.g. interrupting the game) can at least be detected.

It is very important that the game publisher can trace malicious nodes in order to penalize them. For this purpose, we require that all messages exchanged

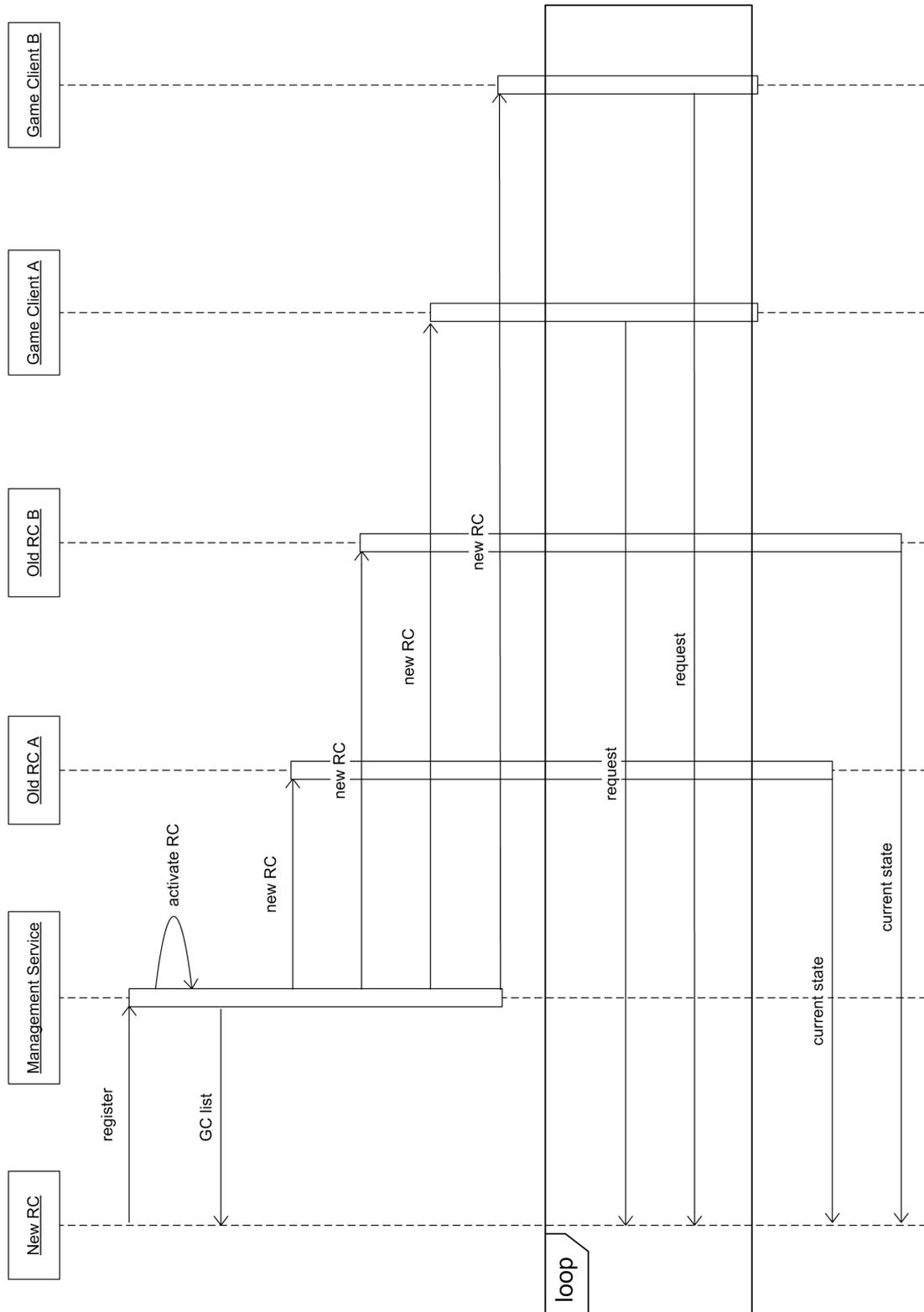


Figure 4.3: Region Controller login procedure

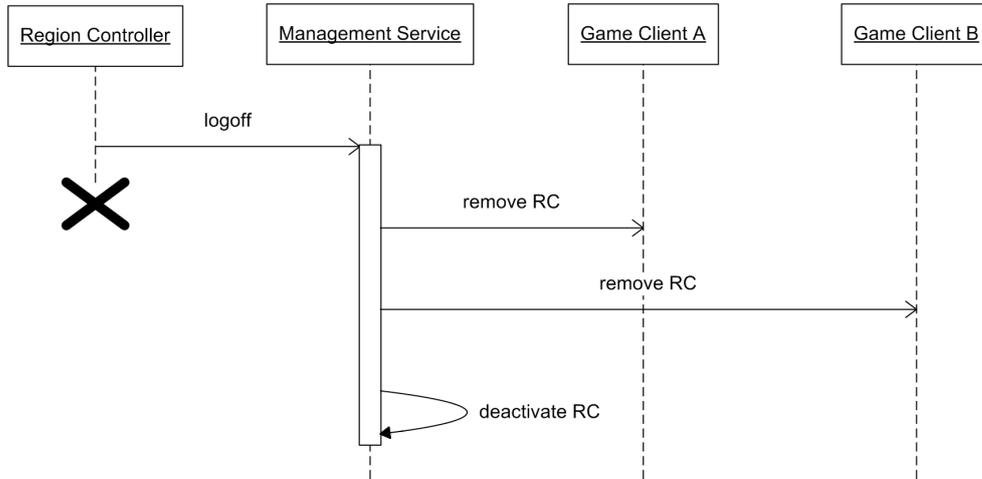


Figure 4.4: Region Controller logout procedure

within the system are signed by the sender. Messages which are not properly signed won't be accepted by the receiver. The signature enables honest participants of the system to prove the origin of illegal requests or falsified updates. The fact that a cheater will eventually be identified makes such attempts very risky. Penalties for performing cheats usually include that the fraudulent players will be banned from the system, losing all their achievements and the fees they have already paid.

#### 4.5.1 Attacks performed by Game Clients

The possibilities for a Game Client to cheat are rather limited. A manipulation of a client's local game state would be futile because it never transmits its state to anyone else. The only option left is to manipulate the requests it sends. A request may contain raw player input (e.g. mouse clicks, button presses) or more abstract commands (e.g. "move to (x,y)", "attack object z"). In any case the Region Controllers will perform a sanity check on the request: the state transition caused by the request must be legal according to the rules of the game. If a request would cause an illegal state transition it is simply dropped by all honest RCs.

In the following examples, nodes are shown as boxes that contain their name (e.g. GC1 for a Game Client) and below their current state as a string. The state string consists of the initial state (e.g. S) and all requests that have been applied to this initial state (e.g.  $S\alpha\beta$  means that the requests  $\alpha$  and  $\beta$  have been applied). Honest nodes are marked green while nodes are marked red. Note that we only show as many nodes as are necessary to illustrate the example. Usually, there are many more nodes per region and the number of Game Clients far outnumbers the number of Region Controllers.

In figure 4.6, we show an example where a malicious Game Client sends an illegal request to all RCs of its region. GC1 is the malicious node sending the illegal request  $\alpha$ , GC2 is an honest node sending the legal request  $\beta$ . After reception, all Region Controllers process the requests. Because request  $\alpha$  is

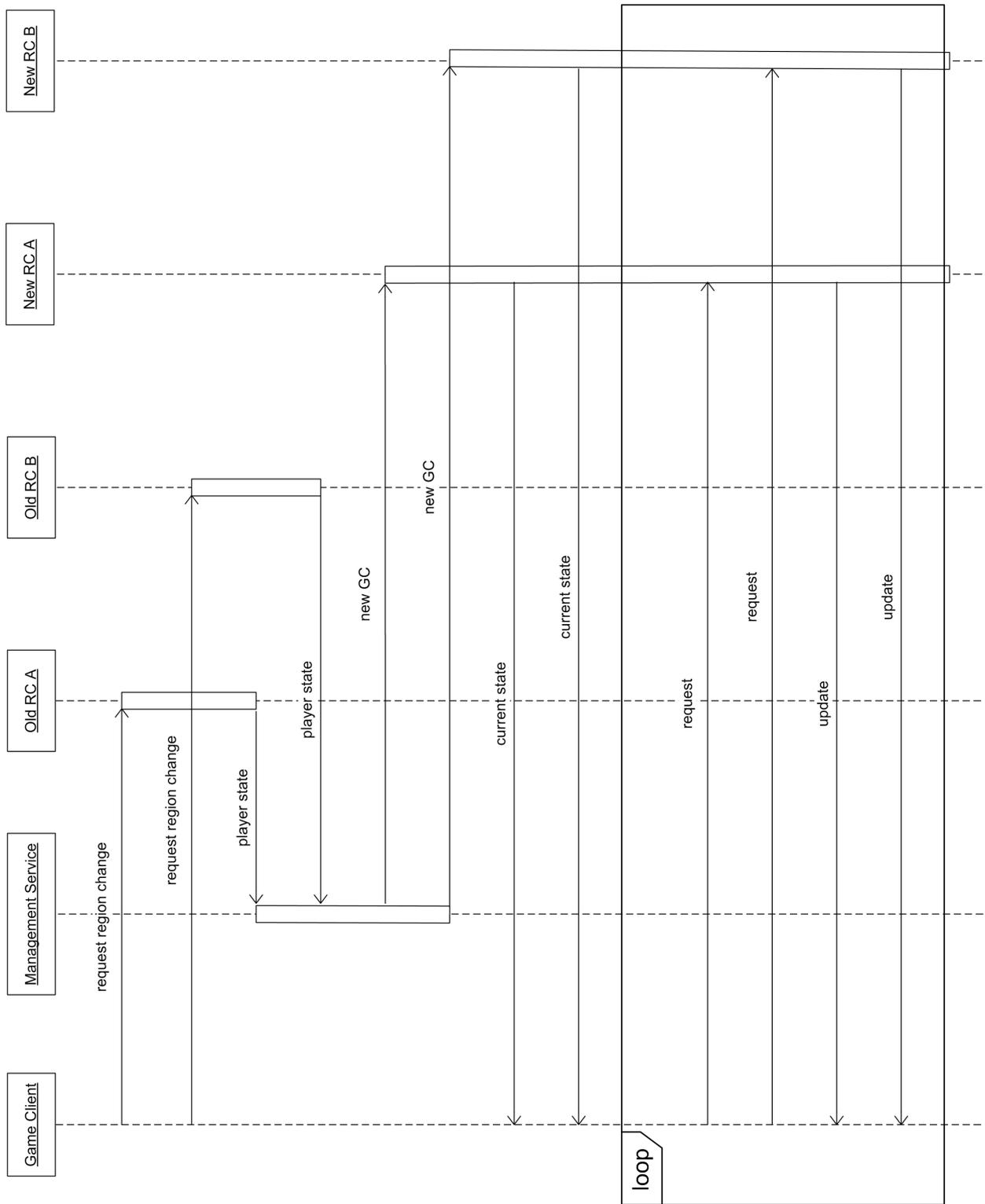


Figure 4.5: Player region change procedure

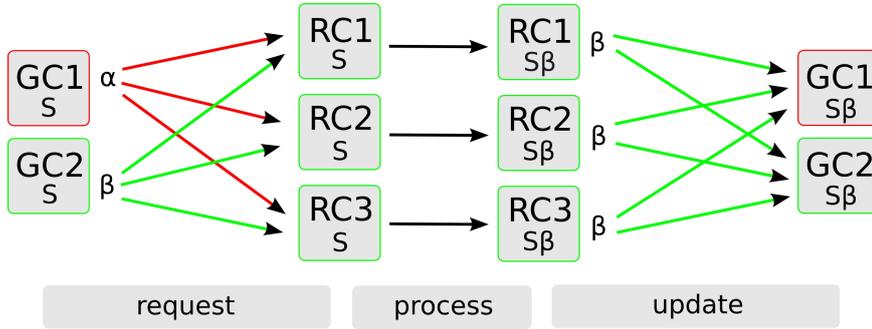


Figure 4.6: Game Client sends forged request to all Region Controllers

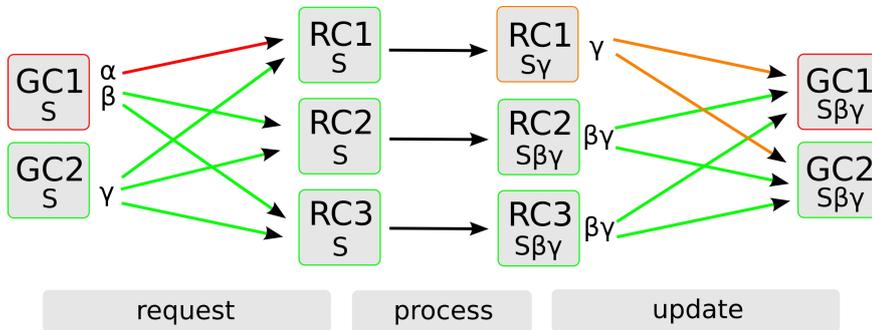


Figure 4.7: Game Client sends forged request to a minority of Region Controllers

illegal, they simply drop it and only apply update  $\beta$ , arriving at state  $S\beta$ . The updates sent by the Region Controllers thus reflect only request  $\beta$  and the attack of GC1 has been prevented.

Although a single client cannot enforce a fraudulent game state change, it may cause other trouble. By sending an illegal action request only to a subset of Region Controllers, while sending a legal one to the others, a client may cause two groups of RCs to arrive at different states and thus go out of sync.

In figure 4.7 a Game Client sends the illegal request  $\alpha$  to a minority of Region Controllers, while the majority receives the legal request  $\beta$ . Of course, request  $\alpha$  will be rejected by this minority which only applies the legal request  $\gamma$  from the honest Game Clients. However, the majority will receive the two legal requests  $\beta$  and  $\gamma$  and process both of them. The two Region Controller groups arrive at different states. Since the majority determines which state is correct, the smaller group will be considered out of sync (the nodes are marked orange) and the Game Clients will request their replacement at the Management Service. This way a malicious client can trigger the replacement of arbitrary Region Controllers even if they work correctly. This does not provide a direct benefit to the client but may be done as a preparation for later cheating attempts, as a denial-of-service attack or simply as a way for griefers to cause mischief. However, due to the message's signature, the source of this attack can be identified later and appropriate actions be taken.

Figure 4.8 shows a similar scenario. This time, however, a majority of Region

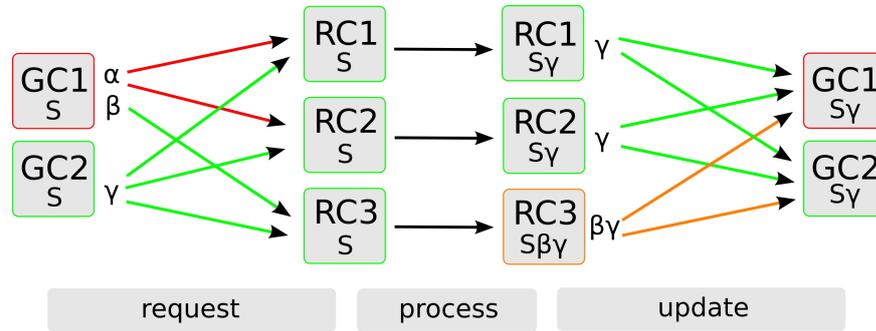


Figure 4.8: Game Client sends forged request to a majority of Region Controllers

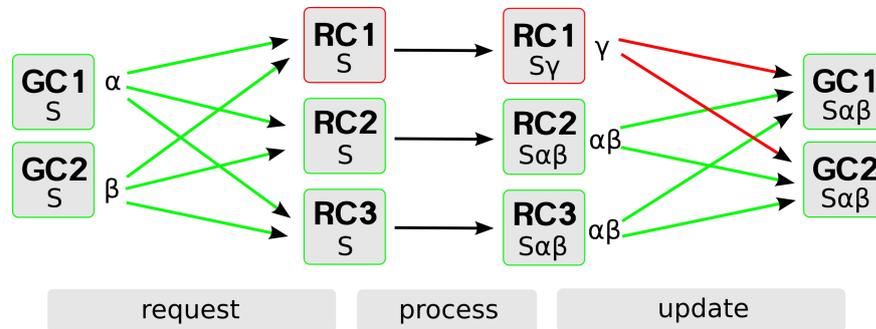


Figure 4.9: Region Controller sends forged update to Game Clients

Controllers receives the illegal request  $\beta$ . As a result, they only process request  $\gamma$  and thus state  $S\gamma$  is regarded as the correct one. As before, this attack can be detected and the attacker identified.

Since Game Clients only request changes for game entities that are under their control (e.g. their own avatar or units that are part of the player's troops), there is no benefit for them in colluding with other Game Clients. Region Controllers determine the correctness of requests only according to the game rules. Since there is no voting for requests, multiple Game Clients cannot convince a Region Controller to accept an illegal request.

### 4.5.2 Attacks performed by Region Controllers

In contrast to Game Clients, Region Controllers can manipulate the game state directly and try to spread these changes.

In figure 4.9, two honest Game Clients send their requests  $\alpha$  and  $\beta$  to the Region Controllers. The first RC is malicious: instead of processing the requests correctly and arriving at state  $S\alpha\beta$  the RC applies the change  $\gamma$  and thus arrives at state  $S\gamma$ . The other Region Controllers process the requests correctly and thus arrive at state  $S\alpha\beta$ . Since the majority of updates that are received by the Game Clients represent the change  $\alpha\beta$ , the update  $\gamma$  is dropped and the attack prevented. The Game Clients can inform the game publisher about the cheating attempt and prove its origin through the message signature.

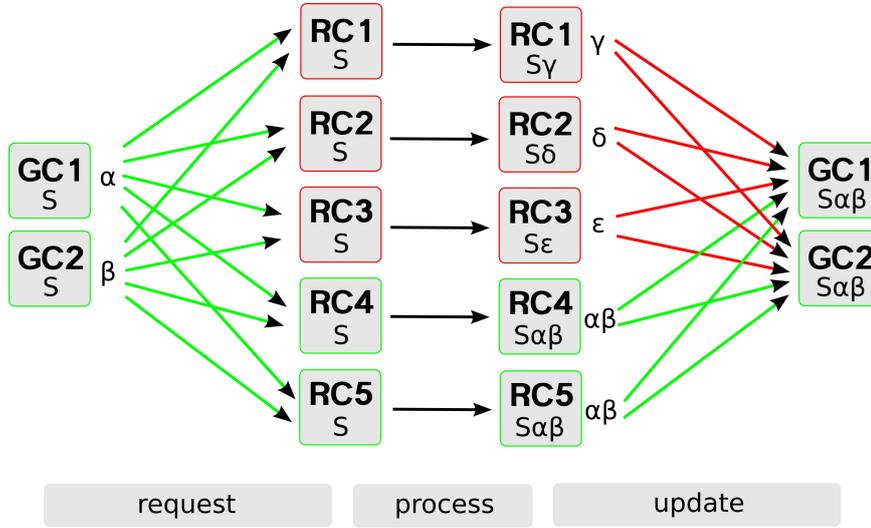


Figure 4.10: Multiple Region Controllers send forged update to Game Clients

The case where multiple malicious Region Controllers collude in performing an attack is very similar. As long as the group of colluding RCs is outnumbered by honest ones, it will not be able to win the voting procedure. Remember that our basic assumption is that the majority of Region Controllers per region is honest. But even if the malicious colluding RCs are in the majority, the attack could be detected later. Any honest Game Client could send the received updates to an auditing service which reconstructs the course of the game. Again, because all messages are signed, the attackers can be traced. See section 6.1 for more details about this approach.

Another case is when there are multiple malicious Region Controllers which do not collude. Figure 4.10 shows this situation. As before, two honest Game Clients send their requests  $\alpha$  and  $\beta$  to the Region Controllers. Some of the RCs are malicious and apply a falsified change to their local state. The remaining Region Controllers process the requests correctly and thus arrive at state  $S_{\alpha\beta}$ . Even if a majority of RCs is malicious, the Game Clients can determine the correct state as long as there are at least two honest RCs. If the malicious ones do not collude the honest RCs will achieve a majority, although no absolute one. However, for the reasons stated above, Game Clients should only trust an absolute majority.

Finally, there can be a group of colluding malicious Region Controllers and additional independent ones. As long as the honest ones outnumber the colluding group, the majority of updates will be correct. Again, however, to be on the safe side we should only trust an absolute majority.

Unlike a Game Client, a Region Controller cannot disturb the system by sending different deviating updates to the Game Clients. Through the voting procedure every Game Client sorts out incorrect updates sent by single Region Controllers, no matter whether the updates other Game Clients received from this RC were different or not.

### 4.5.3 Collusion Attacks

Up to now we have only looked at groups of colluding nodes that only consist of a single type: either all of them are Game Clients or all of them are Region Controllers. Next, we will have a look at the possibilities of Game Clients to colluding with Region Controllers.

One way to collude would be to accept forged messages as if they were legal ones. If a Game Client accepts a forged update from a Region Controller it will taint its own game state. However, because it has no way of further spreading this tainted state, doing so would be pointless. The same is true if a Region Controller accepts an illegal request from a Game Client. This will also result in an illegal state transition and consequently the RC will send incorrect updates. For other Game Clients receiving these updates it is irrelevant whether the RC manipulated the state itself or not. All that they can perceive is that the Region Controller issues state updates which deviate from the ones received by other RCs. Thus, this scenario is equivalent to that where a Region Controller issues forged updates (see section above).

Since attacks that just rely on accepting forged message from a colluding node do not work, cheaters must combine different attacks. First, a Game Client confuses honest Region Controllers by sending legal but different action requests to each of them. Then a group of malicious RCs that collude with the Game Client can agree on an illegal state transition and send corresponding updates. All updates sent by the honest Region Controllers are different so the one sent by the malicious RCs now holds the majority.

This is illustrated in figure 4.11. A malicious Game Client sends an illegal request to a group of colluding Region Controllers. To each honest RC, it sends a different but legal request. The malicious Region Controllers accept and process the illegal request, arriving at the same falsified state. Because all honest RCs arrive at a different state, they cannot achieve a majority. Instead, the majority is achieved by updates of the malicious ones although it is not an absolute majority. Note that in our example the malicious Region Controllers still accept and process the legal request sent by the honest Game Client. However, the group of colluding RCs may agree on any falsified state. This state does not necessarily have to be caused by an illegal request or include requests from honest clients.

This example shows clearly why only absolute majorities should be accepted, even if in some cases (see sections above) a simple majority would be sufficient. As long as our basic assumption that more than half of the Region Controllers of a region are honest holds true, the colluding RCs cannot achieve an absolute majority. The voting procedure will simply fail and the malicious nodes can be traced by their signatures.

### 4.5.4 Message Omission

So far we have discussed attacks that are based on sending falsified or aberrant messages. However, an attacker may also try to disturb the game by omitting certain messages. Fortunately, by not sending a message, a cheater cannot gain a direct advantage regarding the gameplay. Thus, the incentive to perform an attack this way may be rather low but nevertheless we discuss this scenario.

If a Game Client does not send an action request for a certain frame it

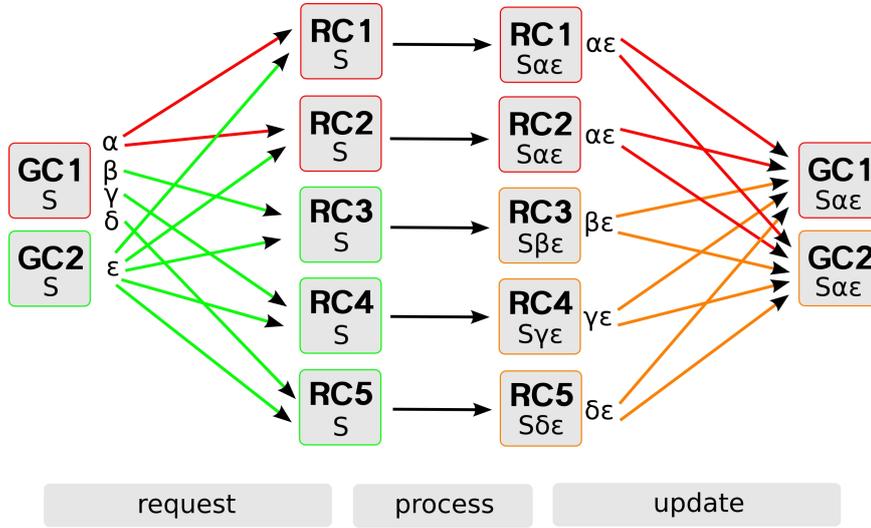


Figure 4.11: Game Client colludes with Region Controllers

means that the player simply does nothing. Depending on the implementation, we can require a Game Client to send a "No Operation" (NOP) request when the player issues no command. No matter whether a Region Controller receives a NOP or no request at all, it will perform some default state transition for the corresponding player entities, e.g. continue with the current action or stop the current action. Obviously, a cheater cannot gain any benefit from not sending a request. However, a Game Client may send its request only to a subset of Region Controllers. This is similar to the attack, where a player sends different legal requests to the RCs. The group which receives the request will perform a corresponding state transition, while the rest will perform the default action. Depending on which group is larger, the majority will vote for an update that either contains the request or the default action. Hence, a Game Client that does not send a request to some Region Controllers cannot disrupt the voting procedure. However, it can cause a minority of Region Controllers to become out-of-sync, since their state deviates from the majority.

In contrast to a Game Client, a Region Controller cannot cause any trouble by not sending updates. From a Game Client's point of view, it does not matter whether an RC sends an incorrect update or no update at all. Both will be sorted out during the voting procedure. The only difference is that a Game Client cannot prove that a Region Controller did not send an update.

Some griefers may find it worthwhile to omit requests in order to cause some Region Controllers go out-of-sync, even if they cannot cause the voting procedure to fail. Though the Region Controllers cannot prove that a certain Game Client did not send a request, they should report this to the game publisher anyway. If a certain number of nodes accuses a Game Client of omitting requests, the game publisher may trust the accusation and penalize the node. This can be seen as a very basic form of a reputation system [87]. Another possibility is to use a multicast routing mechanism that discloses the list of recipients to all receivers of the message. If some recipients were intentionally omitted, every

node that receives the message can detect this and report it to the game publisher. Unfortunately, multicast routing is not commonly supported by today's Internet routers.

### 4.5.5 Region Controller Replacement

In the sections above we talk about the replacement of Region Controllers if they become inconsistent. Replacing RCs is necessary to keep the degree of replication at the desired level in order to maintain the robustness against cheating attempts. Figure 4.12 shows the replacement procedure of inconsistent Region Controllers.

1. We start with a Region Controller that has an incorrect internal state. The reason for this may be that a malicious client sent a deviating request or the RC manipulated the state itself. Because of its inconsistent state, the Region Controller sends incorrect updates to its Game Clients.
2. We assume that the other region controllers work correctly, so they send correct updates to the Game Clients.
3. After receiving the updates, the Game Clients perform the voting procedure. They detect that the update from the incorrect RC deviates from the updates sent by the other RCs.
4. The Game Clients request the removal of the Region Controller that sends incorrect updates.
5. If the Management Service receives enough removal requests, it starts the RC removal procedure. It sends a message to the Game Clients containing a notification that this RC is not responsible for them anymore. Since it is possible that the incorrect RC is not malicious but became inconsistent because of network latency, it informs the RC about its removal and requests it to stop.

Note that a client will not immediately request the removal of a Region Controller in all cases. For example, it is possible that a Region Controller cannot send an update because it has not received all Game Client requests in time. It is also possible that a Region Controller's update does not reach the Game Client in time before it starts its voting procedure. In these cases, a Game Client will wait a certain number of frames before requesting the RC's removal. This gives the inconsistent Region Controller some time so it has the possibility to catch up and avoid its replacement.

## 4.6 Scalability

By shifting most of the computational effort and the bandwidth requirements onto the client side, game publishers can be relieved of the burden of providing huge amounts of back-end hardware. However, dedicated servers are usually very powerful machines leaving some scope to scale vertically, i.e. support more simultaneous users per server. Although most players of computer games possess

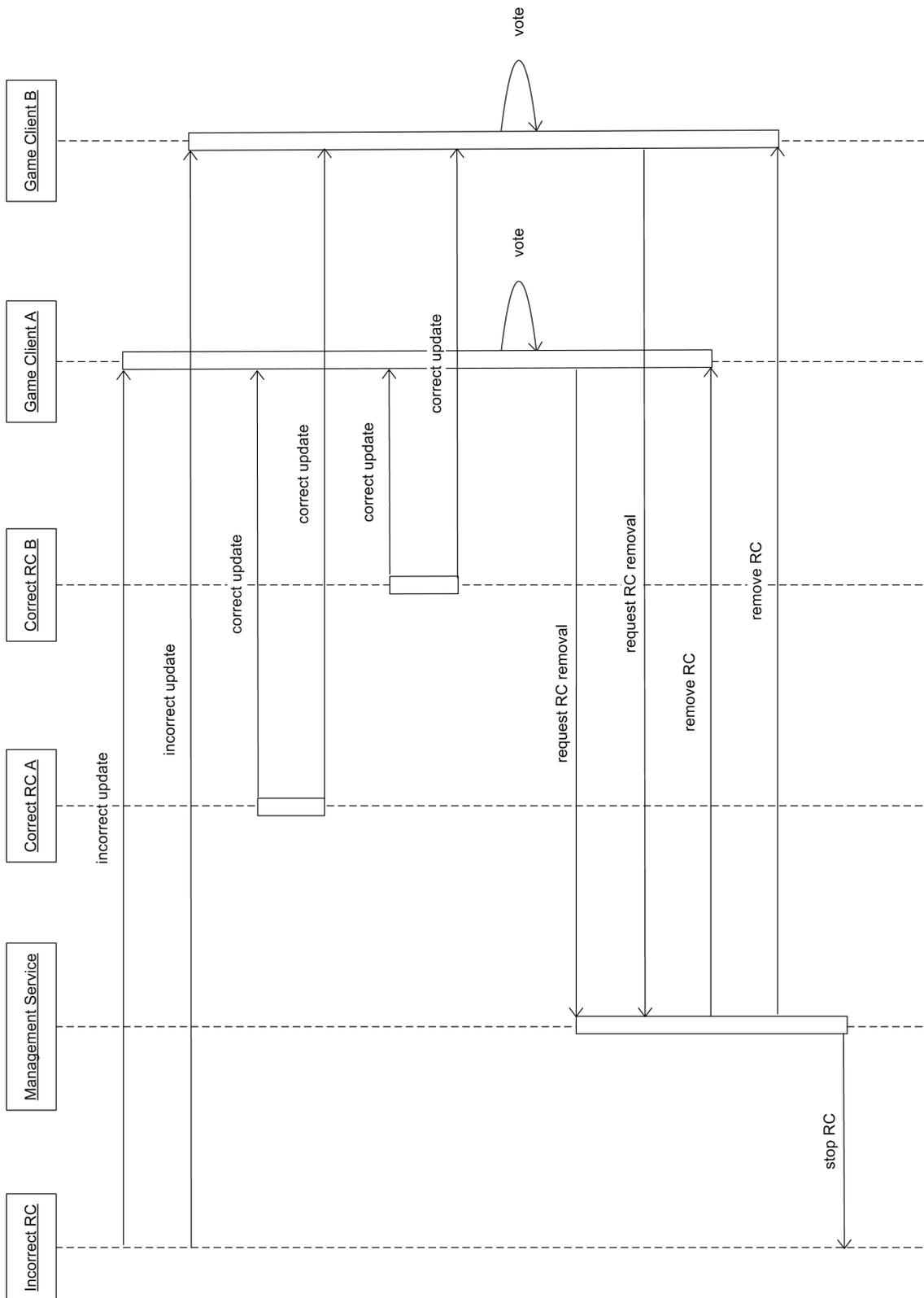


Figure 4.12: Region Controller replacement procedure

rather powerful computers as well, they cannot compete with high-end server hardware.

Fortunately, nearly all large-scale online games (i.e. MMOGs) today follow a different approach. As was mentioned in section 4.3.1, the game world of MMOGs is split into smaller sized regions which can be hosted on different servers. We also mentioned the concept of game world instances. These are special game world regions which are instantiated on demand for groups of players, which usually have between five and forty members. The partitioning into regions and instances allow the system to scale horizontally.

This horizontal scalability accommodates our approach. Our system only limits the amount of players within a single region or instance. This limit is determined by the computational resources provided by the players' computers. To be more precise, the limiting factor is usually the upload bandwidth since players usually connect to the Internet via asynchronous broadband connections. In chapter 5 we develop an analytical model that allows us to determine the possible number of players within a region depending on the players' Internet connection. Since regions are nearly independent of each other, the game world may be composed of an arbitrary number of regions. The only interaction between regions is the transfer of avatar objects whenever a player wants to change the region. However, the number of regions an avatar can switch to from a certain region is limited (usually the surrounding areas) and does not depend on the total number of game regions. Thus, the number of players entering a certain region depends only on the number of players per region and not on the total number of players within the game. Only the resource demand on the Management Service grows linearly with the total number of players. Since avatar objects are first transferred from the source region to the Management Service and then forwarded to the destination region, this server is involved in every region change.

## 4.7 General Applicability

In this section we want to point out shortly to what kind of games our approach can be applied and what the limiting factors are. The focus of our system is to prevent malicious manipulations of the game's state. In contrast to attacks addressed by other approaches (see chapter 2), this kind of attack is relevant to any kind of game.

Our replication mechanism does not impose any restrictions on the functionality of game objects, so in principle any game design can be realized with our system. However, in practice the implementation of specific genres may not result in an enjoyable game experience under certain circumstances. The critical factor is the rate at which updates of game objects are disseminated (usually referred to as the *frame rate*). The frame rate in our system is limited by the players' Internet connection and the number of players per region. Thus, fast-paced action games can only be realized if the player nodes are equipped with high-bandwidth connections or the number of player per region is kept low. For a detailed discussion on the connection between the frame rate, the connection parameters and the number of Game Clients per region please refer to the following chapter.

## 4.8 Conclusion

In this chapter we presented a Peer-to-Peer gaming system that distributes the management of game state and logic among the nodes of the players. This way it relieves the game publisher's servers of this resource intensive task. Although the state of the game world is managed locally on players' machines, our system provides appropriate countermeasures against malicious tampering. For this we rely on active replication of the game state on the players' nodes. Since there exist multiple (probably dissenting) state replicas, the correct one is determined by a majority voting.

In order to provide scalability, the game world is split into smaller sized regions which are hosted on player nodes. The assignment of region replicas to nodes is managed by a central service. This service has to be run by the game publisher and may also manage player subscriptions (i.e. only allow paying customers to join the game). Since these regions are nearly independent of each other, the amount of regions is only limited by what the central management service can handle.



# Chapter 5

## Evaluation

### 5.1 Introduction

The probably most important factor for the game experience of networked games is the speed at which updates of the game world are propagated to the clients. For fast-paced actions games like First-Person-Shooters (FPS), where players need to react on changes of their environment quickly (e.g. dodge attacks or shoot at moving targets), it is crucial that updates of the game world are reflected on the client as soon as possible. In contrast to that, games that rely on long-term planning like Real-Time Strategy Games (RTS) can cope with significantly higher delays. For example, players of the commercially successful FPS *Quake 3* tend to look for servers where their connection has a delay below a hundred milliseconds as this gives them a noticeable advantage over players with higher latencies [3]. Instead, players of the also very successful RTS *Warcraft III* can easily live with delays in the range of several seconds [28]. On the one hand, since the network is a limiting factor, it doesn't make sense to send updates at a rate higher than the network can handle. On the other, to widen the range of supported games, we want to keep the time between updates as small as possible. This chapter discusses how to estimate appropriate update rates in consideration of parameters like network latency and bandwidth, number of Game Clients, etc. For this purpose we develop an analytical model which is later compared to the results of a simulation.

### 5.2 The Request-Update Cycle

The speed at which updates are sent to the Game Clients is called the *frame rate* and is the inverse of the length of a *frame*. The frame length is the amount of time that passes between the issuing of a change request by a Game Client until the reception of an update that reflects the requested change. Figure 5.1 shows what happens in a typical frame on both sides, the Game Client and the Region Controller, and the network in-between.

As the figure shows, a frame can be divided into three phases:

1. A frame starts with a request phase where each Game Client sends a request to all Region Controllers. A request contains the desired changes

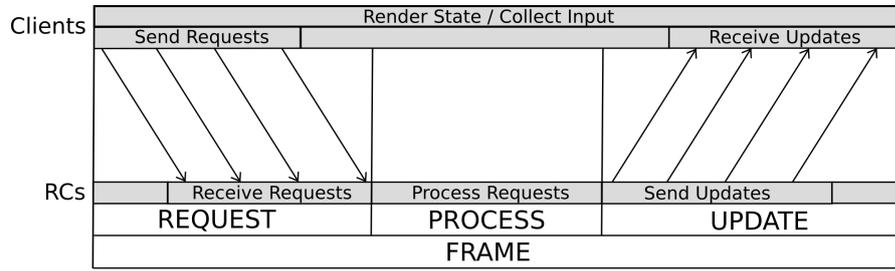


Figure 5.1: Sequence of actions within a frame

of the game world that result from a player's actions (hence they are also called *action requests*). These actions are usually determined by input events like button presses or mouse movements.

2. When a Region Controller has received all requests from the Game Clients, it may start processing these requests. All requests are validated against the game rules and the corresponding changes are applied to the game state. Finally, an update message is generated for each Game Client that contains only the changes that are relevant for this GC.
3. Each Region Controller sends the generated updates back to the Game Clients.

Note that reading player input and rendering the game world onto the screen are usually processes that are performed asynchronously. At first glance it may seem sensible to read the player input directly after an update has been rendered to the screen. But due to the human reaction time it may happen (especially in fast games) that multiple updates may happen before the player actually reacts. Waiting after each time for the reaction of the player would slow down the game unnecessarily. Moreover, the rendering of the game world happens on specialized graphics hardware which runs in parallel to the main processor. Thus, the screen may get rendered more often than the game world is updated and both events are usually not synchronized. Updating the screen faster than the game state can lead to a smoother visualization of the game. E.g. about 25 rendering frames per second are necessary to make a movement look smooth for the human eye. If the game state itself is updated less frequently, specialized algorithms are used to interpolate the movement of game objects.

### 5.3 Correctness

Before we analyze each phase and its execution time in more detail, we have to define how a correctly executed frame looks like and what kind of errors may occur.

We first assume that all Game Clients and Region Controllers are working correctly. At the beginning of a frame, each Game Client sends the same request to all Region Controllers. After receiving the request from all Game Clients, each Region Controller processes the requests, updates its state and sends an update to every GC. These updates are usually individual for each GC, since they only contain changes that are relevant for this GC.

A possible source of error is the sending of request messages to the Region Controllers. In order to arrive at the same state, all Region Controllers must receive the same requests. Otherwise, the state of a Region Controller may become inconsistent. There are two causes of inconsistencies:

1. **A request arrives too late.** Since the network latency is exposed to a certain amount of jitter, the request may take longer than expected. Note that a Region Controller cannot simply wait until it has received all requests. If it would do so, a faulty or malicious GC that never sends a request would cause the Region Controller to wait forever. Instead, after a certain timeout has passed, it must start processing the requests it received so far. If the request arrives later, the Region Controller may recover by rolling back its state and processing the requests again. In this case, the inconsistency will exist only for a limited amount of time.
2. **A malicious GC sends a deviating (but legal) request or omits sending a request.** This case has been already discussed in the previous chapter.

Since there is no direct communication between Region Controllers, an RC cannot detect whether it arrived at a different state as the others. Instead every Region Controller continues executing frames, assuming that its local state is the correct one.

After processing all requests, a Region Controller sends an update to each Game Client. The GC then compares the updates received by the different Region Controllers. Only at this point, Region Controllers in an inconsistent state may be detected. As long as the majority of all Region Controllers agrees on a certain state, inconsistent RCs do not affect the GCs. They can start a new frame, again sending requests to the RCs. Of course, inconsistent Region Controllers that cannot recover must be replaced after some time. Otherwise, more and more RCs may become inconsistent and eventually make it impossible to achieve a majority. Thus, if a Region Controller has not recovered after a certain number of frames have passed, the Game Client requests from the management service to remove the Region Controller. If enough GCs request the removal of an RC, the management service replaces it with one from the pool.

In contrast to a real-life system, in a simulation environment it is possible to inspect the global state of the network at any time. Thus, for each frame we can directly determine which Region Controllers are in an inconsistent state after processing requests. This is not only convenient, but necessary to correctly identify inconsistent RCs. Obviously, not only requests but updates as well may arrive too late. If a Game Client does not receive an update from a certain Region Controller in time, it treats the RC as being inconsistent. If the Region Controller does not send updates for a certain number of frames, the Game Client assumes that the RC has crashed and requests its removal.

The number of inconsistent Region Controllers within a certain frame is an appropriate measure of correctness. As long as this number is less than half of the total number of RCs, a majority can be achieved and the system works correctly. Obviously, the frame length and the number of inconsistent Region Controllers are reciprocal. Shorter frame lengths leave less time for transferring messages, raising the probability that requests arrive too late and thus RCs

become inconsistent. Bigger frame lengths allow to compensate network jitter more easily, lowering the chances of an RC becoming inconsistent. In order to allow a comparison of different scenarios, we also want to know what the probability is that a majority of RCs become inconsistent within the same frame. With this probability, we can determine the average time that game needs to run until a voting failure occurs.

## 5.4 Frame Execution Time

In order to determine the appropriate length a frame, we must make an estimation of the length of each phase. We start with the second phase, the processing phase, which is the simplest. At the heart of every computer game engine is a loop that reads player input, updates the game state accordingly and renders a visualization on the screen. In order to be playable, the game needs to run at a certain minimum speed. Thus, the updating of the game state may only take a limited time. This is the reason why all computer games have certain minimum hardware requirements. If the requirements are not met, the game runs too slow and is simply not playable. But if the hardware exceeds the performance requirements, the game should not run faster. Instead, either the additional performance may be used to perform smaller updates at a higher rate (leading to a smoother game experience) or to more idle time that may be used by other processes running on the machine. Thus, the time needed for the processing phase is the maximum time required for processing all updates on a Region Controller. If a Region Controller needs less time updating its game state, it could immediately start sending updates to the Game Clients. This doesn't lead to shorter frames since the GC needs to wait until the other RCs sent their updates. But sending earlier leaves more time for the transmission of the update messages, which lowers the probability that updates arrive too late because of network jitter. In the model and the simulator we use a fixed time interval for the processing phase. Implementing and analyzing the benefits of advanced update sending is left as future work.

Estimating the lengths of the first and the third phase is more complex. The time that it takes to send the necessary messages depends on many factors. First, the number of Game Clients and Region Controllers determines the number of request and update messages that need to be sent. Next, the number of messages and their size determine the amount of data which is transmitted. Finally, the amount of data, the available bandwidth on the link between two nodes and the network latency determine the transmission time. Unfortunately, the network delay is not fixed but shows a certain variance, called the *network jitter*. A mathematical model that pays respect to all of these factors is presented in the following.

### 5.4.1 Modeling Message Transmission Time

In this section we develop a model that allows us to estimate the amount of time that is necessary for transferring a certain amount of messages over a network. This model is then used to determine the length of the request and update phases of a frame. The *message transmission time*, i.e. the time to transmit a message from one node to another, is the sum of the *transmission delay* and

the *propagation delay*. The transmission delay is the time that elapses between the arrival of the first and the last bit of information at the receiving node. The propagation delay is the amount of time that elapses between the point when the first bit of information is sent by the sending node until the point when it arrives at the receiving node.

### 5.4.2 Transmission Delay

The transmission delay can be determined by dividing the amount of transmitted information by the available bandwidth. The available bandwidth itself is determined by the slowest link between the sending and the receiving node. In our model the connection between two nodes consists of three links: the sender's upload link, the intermediate network and the receiver's download link. The bandwidths of the upload and download links are fixed. However, determining the available bandwidth of the intermediate network is not trivial. One has to take into account that the maximum available bandwidth is affected by packet loss and, since we apply the TCP protocol, effects of the TCP Congestion Avoidance algorithm [55, 94]. Mathis et al. [67] have shown, through simulation and live observations, that equation 5.1 is an adequate model to predict bandwidth under these conditions.

$$Bandwidth = \frac{MTU}{RTT} \sqrt{\frac{3/2}{p}} \quad (5.1)$$

**MTU** The maximum transfer unit (in our simulation the MTU is 1500 Bytes).

**RTT** The average roundtrip time between the sending and the receiving node.

**p** The random packet loss probability.

Using commonly available ADSL lines, the uplink is usually the slowest link and thus determines the overall bandwidth.

After determining the available bandwidth, we can calculate the time that is necessary for the message transfer. Dividing the total amount of transmitted information by the available bandwidth, as described above, yields only the amount of time that passes until all messages have arrived at the receiver. But as we will see later, it is necessary to determine the arrival time of every single message. From an abstract point of view, all requests are sent at the same time. But on the network level, packets are sent sequentially and not truly in parallel. Fortunately, all request and update messages used by our system fit into a single packet. Thus, we do not have to worry whether packets of different messages are interleaved during the transmission. Since our simulator assumes a packet overhead of 40 bytes (20 bytes TCP header plus 20 bytes IP header) the size of a message  $Size_m$  is determined by equation 5.2 where  $Payload_m$  is the payload the message contains.

$$Size_m = Payload_m + \left( \left\lceil \frac{Payload_m}{MTU - 40} \right\rceil * 40 \right) \quad (5.2)$$

But in case that the payload fits into a single packet the equation is reduced to equation 5.3.

$$Size_m = Payload_m + 40 \quad (5.3)$$

The transmission delay  $TransDelay_m$  of a single message is

$$TransDelay_m = \frac{Size_m}{Bandwidth} \quad (5.4)$$

and the time  $TransDelay_{m_i}$  that passes from sending the first message until the  $i$ th message arrives at the receiver can be calculated as follows:

$$TransDelay_{m_i} = i * TransDelay_m \quad (5.5)$$

### 5.4.3 Propagation Delay

The propagation delay is the sum of time that the information takes to travel over the physical medium and the propagation and queueing delays on intermediate routers. It may be split into a fixed part, the *minimum propagation delay*, and a random part, the *jitter*. The random part obviously cannot be calculated by a function that returns a single value. Instead, it can be modeled by a log-normal distribution function [51] which gives the probability distribution of the jitter values. The corresponding cumulative distribution function allows us to determine the probability that the jitter stays within a given value. Equation 5.6 shows the log-normal probability density function and equation 5.7 the cumulative distribution function.

$$f_1(x; \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{\ln(x) - \mu}{\sigma}} \quad (5.6)$$

$$f_2(x; \mu, \sigma) = \frac{1}{2} + \frac{1}{2} erf \left[ \frac{\ln(x) - \mu}{\sigma\sqrt{2}} \right] \quad (5.7)$$

The parameters  $\mu$  and  $\sigma$  can be fitted with a *downhill simplex algorithm* [78] using real data from the PingER project. Note that  $x$  in  $f_2(x; \mu, \sigma)$  is the jitter of a roundtrip packet. Assuming that the roundtrip jitter is equally distributed on both hops, the probability that a single-hop jitter is less or equal than  $x$  can be calculated with  $f_2(2x; \mu, \sigma)$ .

### 5.4.4 Total Message Transmission Time

The time that it takes to transmit a message  $m_i$ , is the sum of the transmission delay (see equation 5.5), the propagation delay and the jitter:

$$TransTime_{m_i} = TransDelay_{m_i} + PropDelay + Jitter$$

### 5.4.5 Probability of an Inconsistency

In order to arrive in time, the transmission time of a message must be smaller than the time interval reserved for the request phase:

$$TransTime_{m_i} \leq ReqInt$$

$$\Leftrightarrow TransDelay_{m_i} + PropDelay + Jitter \leq ReqInt$$

$$\Leftrightarrow Jitter \leq ReqInt - TransDelay_{m_i} - PropDelay$$

Using  $f_2(2x; \mu, \sigma)$  (see equation 5.7) we can determine the probability that the one-way jitter is less or equal than a certain value  $x$ . The probability  $P_{InTime}(m_i)$  that message  $m_i$  arrives at the receiver in time is

$$P_{InTime}(m_i) = f_2(2 * (ReqInt - TransDelay_{m_i} - PropDelay); \mu, \sigma) \quad (5.8)$$

Remember that the number of request sent by a single Game Client is different from the number of request received by a Region Controller. Therefore we first need to calculate the average probability that any single request arrives in time. Then we can derive the probability for any Region Controller being inconsistent in a certain frame. The average probability for any single request out of  $n$  to arrive in time is

$$P(InTime_m) = \frac{\sum_{i=1}^n P_{InTime}(m_i)}{n} \quad (5.9)$$

The probability that all  $r$  requests sent to a region in a frame arrive in time is

$$P(InTime_{all}) = P(InTime_m)^r \quad (5.10)$$

Therefore, the probability that any of the requests in a frame does *not* arrive in time at a certain Region Controller and thus causes this RC to be inconsistent is

$$P(RCInconsistent) = 1 - P(InTime_{all}) \quad (5.11)$$

To determine the number of inconsistencies that occur within a certain time, we have to multiply  $P(RCInconsistent)$  with the number of active RCs per frame and the number of frames that are executed within the amount of time. Finally, we determine the probability that the voting fails because no majority can be achieved in a certain frame. In order to make the voting fail, more than half of the active RCs need to be inconsistent. If  $r$  is the number of active RCs, then equation 5.12 determines the probability of a voting failure occurring within any frame.

$$P(VotingFailure) = \sum_{i=\lfloor \frac{r}{2} + 1 \rfloor}^r P(RCInconsistent)^i \quad (5.12)$$

## 5.5 Simulation Setup

As our simulation environment, we use *PeerfactSim.KOM* [63] which is a discrete-event based Peer-to-Peer simulator written in Java. The main goal of *PeerfactSim.KOM* is to provide a general benchmarking platform for P2P systems. Its architecture consists of multiple layers and each of these layers encapsulates an important aspect of a P2P system. For example, the simulator supports the modeling of user behavior, application logic and overlay networks. But the main reason for choosing *PeerfactSim.KOM* is that it provides an accurate latency model for simulating message delivery times for the Internet. This model accounts for details of the OSI layers that have to be traversed when sending messages over end-to-end connections. These details include geographical distance between peers, processing delays, congestion, packet loss and retransmission. Real-life data from the *CAIDA Macroscopic Topology Measurements Project* [19] and the *Ping End-to-end Reporting Project (PingER)* [68] is used to simulate the realistic network delays which also include jitter. The validity of the simulator's latency model has been proven in [48].

On the application layer of the simulator, we implemented the system that has been presented in the previous chapter. All necessary services — Game Clients, Region Controllers and the Management Service — have been implemented as an application. Each player node runs both, the GC and RC application, while the Management Service runs on a separate node. All important interactions presented in chapter 4 have been implemented exactly as described. The bootstrapping, the login and logout of Game Clients and Region Controllers as well as the replacement procedure for inconsistent RCs.

Each simulation run starts with a small warm-up phase. In this phase, initial Region Controller instances are provided on separate nodes as described in section 4.4.1. As soon as regular player nodes join the system, the initial RC instances are successively replaced by RC instances that run on a player node. When all Region Controllers are hosted on player nodes, the actual simulation phase starts and lasts for approximately one hour of real-time. Naturally, the shorter the frame length is, the more frames can be executed within a certain time period. However, since we want to determine the mean time between voting failures, we keep the simulated real-time period fixed. All scenarios were simulated fifteen times using different random seeds.

All player nodes modeled in the simulator emulate the bandwidths of real Digital Subscriber Line (DSL) connections available from German Internet providers. We start with a widely available 2 MBit ADSL connection and continue with the faster but less commonly available 16 MBit ADSL connection. We conclude our scenarios with a high speed 50 MBit VDSL connection which at the moment is only available in certain cities. But we still talk about connections that are already available to regular customers.

As we increase the nodes' bandwidth, we first retain the number of Game Clients to see how the faster connection affects the possible frame length and the mean time between voting failures. We then increase the number of clients within a region to show how faster connections can handle larger region sizes. The number of Game Clients is increased from 25 to 100 and finally to 250. The number of Region Controllers is maintained through all scenarios since we want to keep the degree of replication equal.

The last value that is changed between different scenarios is the *buffer time*. As described above, the length of the request and update phases are determined by the time needed to transfer the request and update messages respectively. However, due to the network jitter there is a certain probability that the message transfer takes longer than expected. For this reason, we extend each phase by a certain buffer time. The larger this buffer time is, the less is the probability that the message transfer exceeds the phase length. As we will see in the second scenario, shorter frame lengths (which are possible on faster connections) lead to significantly smaller mean times between voting failures. Thus, in order to keep these mean times tolerable, we increase the buffer times whenever the frame lengths become smaller.

In the following section we start with the first scenario which also serves as an example of how our model can be used to predict the results of the simulation. In section 5.6, we extend our simulation with clock skew, node churn and crashes to achieve more realism. Finally, in section 5.7 we discuss the remaining scenarios.

### 5.5.1 Scenario 1 - 2 MBit ADSL Node, 25 Game Clients

We will now give an example how the model can be used to predict the results of the simulation. The following table gives an overview of the main parameters.

Number of GCs per Region	25
Maximum number of RCs	7
Node download bandwidth	2 MBit/s
Node upload bandwidth	192 KBit/s
GC Request Size	60 Bytes
RC Update Size	300 Bytes
RC processing time	100 ms
Buffer time	20ms
Node location	Germany
Simulation length	6921 Frames

This setting uses standard 2MBit ADSL nodes running a region with 25 Game Clients. This is a typical size for instance raid dungeons in online games. According to [40], the maximum packet sizes for updates are around 300 Bytes, while the maximum request size is about 60 Bytes. We assume that a RC needs at most a hundred milliseconds to update its state according to the GC requests. We add 20 ms as buffer time to both, the request and the update phases, to compensate for the network jitter. A maximum number of seven RCs means that there need to be at least four colluding malicious RCs within the same region in order to manipulate its state. We assume that all nodes are located within Germany. For these node PingER states a minimum roundtrip time of 6.0 ms and a one-way packet loss rate of 0.005 percent. The simulation ran for 6921 frames, not including the warm-up phase, simulating approximately one hour of real-time.

We start with estimating the amount of time that should be reserved for the request phase. In this phase, every Game Client sends seven requests to the Region Controllers and every Region Controller receives 25 requests from the GCs. Because the number of messages on the uplink is not the same as on the downlink, we cannot simply compare the bandwidths of both links. Instead, we have to calculate the time it takes to transfer all messages on each link and see on which link it takes longer. Formula 5.2 yields a total request message size of 100 Bytes.

$$60 + \left( \left\lceil \frac{60}{1460} \right\rceil * 40 \right) = 100$$

Transferring seven messages over the uplink of the Game Client takes approximately 28 ms.

$$\frac{7 * 100 \text{ Bytes}}{192 \text{ KBit/s}} = \frac{700 \text{ Bytes}}{24576 \text{ Bytes/s}} = 0.02848 \text{ s}$$

Transferring 25 requests over the downlink of a Region Controller takes approximately 10 ms.

$$\frac{25 * 100 \text{ Bytes}}{2 \text{ MBit/s}} = \frac{2500 \text{ Bytes}}{262144 \text{ Bytes/s}} = 0.00954 \text{ s}$$

Formula 5.1 yields that the network between those two nodes can easily transfer data above a rate of 250 MBit/s.

$$\frac{1500\text{Bytes}}{7.68\text{ms}} \sqrt{\frac{3/2}{0.00005}} = 258.1\text{MBit/s}$$

As a result, the limiting link for sending requests is the uplink of the GC and thus determines the transmission delay.

To the transmission delay of the request messages we have to add the propagation delay, which itself is the sum of the minimum propagation delay and the jitter. The minimum propagation delay is equal to half of the minimum roundtrip time, i.e. 3 ms. As mentioned before, the jitter is not a fixed value. The only thing we can do is add a certain buffer time to compensate for the jitter and calculate the probability that the requests will arrive within that time. We add the additional 20 ms buffer time and round the result to the next full millisecond.

$$28.48\text{ms} + 3\text{ms} + 20\text{ms} = 51.48\text{ms}$$

This means that for our request phase we chose a length of 51 ms.

The length of the update phase is estimated in the same manner. Formula 5.2 yields a total update message size of 340 Bytes.

$$300 + \left( \left\lceil \frac{300}{1460} \right\rceil * 40 \right) = 340$$

Transferring 25 updates over the uplink of the Region Controller takes approximately 346 ms.

$$\frac{25 * 340\text{Bytes}}{192\text{KBit/s}} = \frac{8500\text{Bytes}}{24576\text{Bytes/s}} = 0.34587\text{s}$$

Transferring seven requests over the downlink of a GC takes approximately 9 ms.

$$\frac{7 * 340\text{Bytes}}{2\text{MBit/s}} = \frac{2380\text{Bytes}}{262144\text{Bytes/s}} = 0.00908\text{s}$$

Again, the limiting link is the uplink. We add the minimum propagation delay and the buffer time and get a rounded update phase length of 369ms.

$$345.87\text{ms} + 3\text{ms} + 20\text{ms} = 368.87\text{ms}$$

The total frame length is the sum of the request, processing and update phase lengths:

$$51\text{ms} + 100\text{ms} + 369\text{ms} = 520\text{ms}$$

In order to determine the probability of inconsistencies to occur, we first need to fit the parameters  $\mu$  and  $\sigma$  for equation 5.7 to determine the distribution of the network jitter. We apply the *downhill simplex algorithm* mentioned above using real-life data from the PingER project. The algorithm yields the parameters  $\mu = 0.07776$  and  $\sigma = 1.08218$ . Figure 5.2(a) shows the probability density for the real data measured by PingER and for the approximated function. Figure 5.2(b) shows the cumulative distribution for the approximated function.

Using equation 5.8, we can determine the probability that a request sent by a GC within a certain frame arrives in time. The length of the request phase

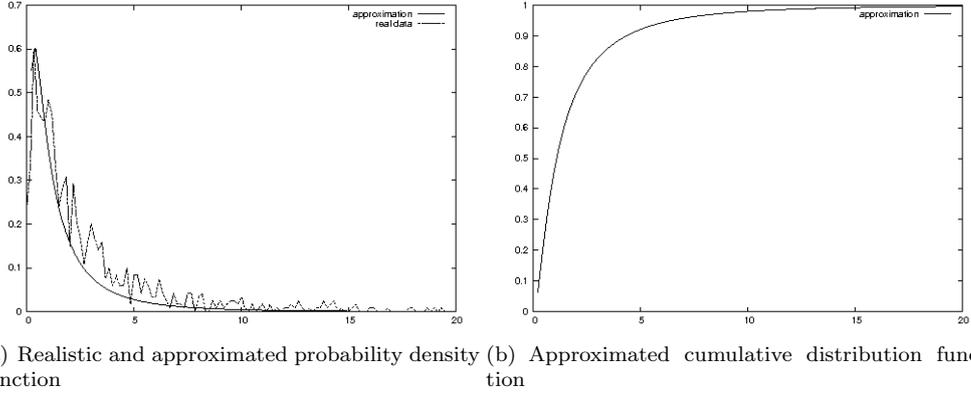


Figure 5.2: Density and distribution function for the network jitter

$ReqInt$  is 51 ms, the propagation delay  $PropDelay$  3 ms and the transmission delay  $TransDelay_{m_1}$  for the first message  $m_1$  is 4.07 ms. Therefore, the probability of arriving in time is for the first message

$$P_{InTime}(m_1) = f_2(2 * (51ms - 4.07ms - 3ms); \mu, \sigma) = 0.99998$$

The second message has a transmission delay of 8.14 ms, the third of 12.21 ms and so on. The probabilities of arriving in time for the messages  $m_2$  to  $m_7$  (each GC sends 7 requests) are

$$\begin{aligned} P_{InTime}(m_2) &= 0.99996 \\ P_{InTime}(m_3) &= 0.99994 \\ P_{InTime}(m_4) &= 0.99992 \\ P_{InTime}(m_5) &= 0.99986 \\ P_{InTime}(m_6) &= 0.99976 \\ P_{InTime}(m_7) &= 0.99954 \end{aligned}$$

The average probability for any single request to arrive in time (equation 5.9) is therefore

$$P(InTime_m) = 0.99985$$

This means that during 6921 frames, where in each 25 Game Clients send 7 requests, on average  $6921 * 25 * 7 * (1 - 0.99985) = 179.33253$  requests are too late. The probability for any Region Controller (each of them receives  $r = 25$  requests) being inconsistent (equations 5.10 and 5.11) in a frame is

$$P(RCInconsistent) = 1 - P(InTime_m)^r = 1 - (0.99985)^{25} = 0.00370$$

Having 7 Region Controllers running for 6921 frames means that, on average, approximately 179.01426 inconsistencies occur. Note that the average number of inconsistencies is slightly less than the average number of delayed requests. This is because there is a small probability that multiple requests arrive too late at the same RC within the same frame, causing only a single inconsistency.

Simulation Run	Delayed Requests	RC Inconsistencies
1	182	182
2	174	172
3	192	191
4	148	148
5	193	193
6	183	181
7	186	186
8	186	186
9	193	192
10	176	174
11	174	174
12	186	183
13	163	161
14	182	181
15	177	177
Average	179.6667	178.7333
Expected	179.3325	179.0142

Table 5.1: Simulation results for scenario 1

Finally, the probability that a voting failure within any frame occurs (equation 5.12) is

$$P(\text{VotingFailure}) = \sum_{i=4}^7 0.00370^i = 1.87107 * 10^{-10}$$

This means, that within 5.345 billion frames (approximately 88 years) only a single voting failure will occur.

In order to compare the results of the model to those of the simulation, we performed the simulation with the parameters above fifteen times using each time a different random seed. The results are shown in table 5.1. If we take the average result of the simulation as the reference, the error of the model is only 0.00186 for the number of delayed requests and 0.00157 for the number of inconsistent RCs.

## 5.6 Adding Realism to Scenario 1

Up to now we have only simulated the system being in an ideal state. No nodes join or leave the system, no nodes crash and all clocks are perfectly synchronized. We now extend our simulation with node churn, node crashes and clock skew and see how each of these extensions affects the correctness of the system.

### 5.6.1 Clock Skew

Our system relies on the fact that tasks are performed at certain times. For example, the Game Clients start to send requests at the beginning of the request phase and the Region Controllers start processing these requests at the beginning of the processing phase. Unfortunately, the clocks on different nodes

usually don't show exactly the same time. If the clock on a GC is late, then the sending of requests will also happen later (with respect to some reference time). If the clock on a Region Controller is ahead of time, it will start processing requests earlier. In both cases, the time that is left for sending the requests is shorter than intended. This may raise the probability that requests arrive too late and therefore more RCs may become inconsistent.

Many different approaches for synchronizing the clocks of nodes in a distributed system exist. For example, by using the *Global Positioning System (GPS)* clocks can be synchronized with an error of 10 nanoseconds and less [91, 30]. GPS uses a multitude of satellites that emit a radio signal which can be received by a special GPS receiver device. Out of the different signals, the receiver can calculate its own position and the current UTC time. GPS receivers have become affordable and are widely-used in civil vehicles today as navigational aids. There exist also many purely software-based solutions. In [101] a synchronization method based on the *TimeStamp Counter (TSC)* register which is found in nearly all modern CPUs. This register simply counts clock cycles of the CPU. Because CPU oscillators show a very high stability, it is possible to synchronize clocks with a precision in the order of 30 microseconds. Another TSC-based approach is presented in [84] and provides an even better accuracy of about one microsecond.

All approaches mentioned above exhibit an offset that is well below a millisecond and thus could easily be ignored by our system. However, the most popular synchronization method that is used today is the *Network Time Protocol (NTP)*. NTP uses special time servers that are queried by the network nodes. Its accuracy is bounded by the round-trip time between the time server and the node and lies usually in the order of 15 milliseconds and less on Internet connections [73]. In order to measure the actual performance of the NTP synchronization, we have used the standard NTP query program *ntpq*. On a standard ADSL node with 2 MBit downlink and 192 KBit uplink that was connected to three different time servers, we collected nearly ten thousand samples of the clock offset. The samples were normally distributed with a mean of 0.00020224 and a standard deviation of 0.9437702. Figure 5.6.1 shows the distribution of the offset samples.

Our simulator uses this normal distribution to generate offsets for the clocks on the nodes. Every time a node schedules an event to proceed to the next phase (e.g. from the request phase to the processing phase) a value from this distribution is added to the scheduling time. Note that this is a worst case scenario: because the random offsets are independent from each other, it can change instantly from a high negative offset to a high positive one. In reality, it is very unlikely that the offset performs large jumps.

Table 5.2 shows the results of the simulation runs.

As we can see, the average number of delayed requests did not change significantly. This is because we measure if the network transmission time of the requests exceeds the request phase and not if the request arrives too late at the receiver. On the contrary, the number of RC inconsistencies reflects which RCs became inconsistent because of requests that arrived too late. It does not matter whether the request was delayed on the network or the request was sent too

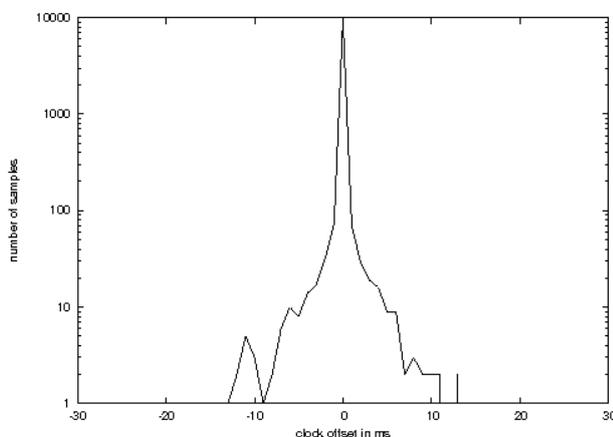


Figure 5.3: Distribution of NTP time offset samples

late because the senders clock is late. This explains why the number of inconsistencies may actually be higher than the number of delayed requests. The probability of a Region Controller being inconsistent per frame is  $\frac{181.1333}{6921 \times 7} = 0.00374$ . Consequently, the probability of a voting failure per frame is raised

$$\sum_{i=4}^7 0.00374^i = 1.96134 * 10^{-10}$$

and results in one voting failure in 84.1 years on average.

## 5.6.2 Node Churn

The next extension to our simulator is the joining and leaving of nodes while the system is running. Depending on the type of game, the churn patterns can be very different. For example, in Real-Time-Strategy Games the situation is similar to a board game. Players usually come together to play one or more sessions of the game and during a session, players rarely leave. Whenever a game session is finished, its state is discarded and the next session starts from scratch. In a Massively Multiplayer Online Game there is usually a persistent world which is continuously online for years. There are no sessions with a restricted lifetime and players enter and leave the world whenever they like.

To our simulator we added the more challenging churn pattern of an MMOG. In [39], a 3-year long-term study of the MMOG *EVE Online* [21] was performed. During this time, it had nearly one million unique players, 67 million player sessions and 17 thousand player years of gameplay. The authors show that the player session times can be modeled with a Weibull distribution with the parameters  $\beta = 0.456$  and  $\eta = 11.7$ . Figure 5.4 shows a plot of this distribution. For our simulation, all session times were generated according to this distribution. In order to keep the utilization of the system always close to the maximum, every node that logs out of the system is replaced shortly after.

Of course we expect that node churn has a negative impact on the correctness of the system. If the node that had left was an active Region Controller, an inactive one from the pool has to take its place. The remaining active RCs

Simulation Run	Delayed Requests	RC Inconsistencies
1	149	152
2	186	191
3	173	178
4	174	171
5	161	168
6	182	184
7	188	189
8	182	188
9	193	187
10	186	193
11	183	183
12	195	200
13	175	179
14	188	187
15	203	197
Average	181.2000	181.1333
Expected	179.3325	179.0142

Table 5.2: Simulation results for scenario 1 including clock skew

provide the new one with the current region state. This takes away a fraction of the bandwidth and thus may lead to slightly more delayed request messages. The system could be improved in such a way that it interrupts the sending of the region state during the request phase. However, the replacement of Region Controllers would then also take more time.

Regarding correctness, it doesn't matter whether a Region Controller is inconsistent or not available. In both cases, the number of consistent RCs is lowered by one, raising the probability of a voting failure. We need to estimate how many active RCs leave within a certain time and how long it takes to replace them. The churn pattern, generated using the session time distribution function, reveals that of all nodes that leave during the simulation on average eighteen are active RCs. To replace an active Region Controller, the new RC needs to be informed about the current state. As described in section 4.5.5, the remaining RCs each send a message that contains the current state. Thus, the time the replacement takes corresponds to the time that this message needs to be transferred. In our scenario, the upload link of the sending RC is the limiting factor. However, not the whole bandwidth is available for sending. During the update phase, the RC also needs to send its updates to the Game Clients. As shown above, the sending of 25 update messages results in a total of 8500 Bytes. A frame has a length of 520ms and the upload bandwidth of the RC is 24576Bytes/s. Within a frame, an RC can upload 24576Bytes/s \* 520ms = 12780Bytes. For sending the state message, per frame 12780Bytes - 8500Bytes = 4280Bytes remain. In our simulation we assume that the state is represented by 1024Bytes per player, so we end up with

$$\frac{1024Bytes * 25}{4280Bytes/Frame} = 5.9813$$

frames to replace an active RCs. It takes 5.9813 frames on average to replace an

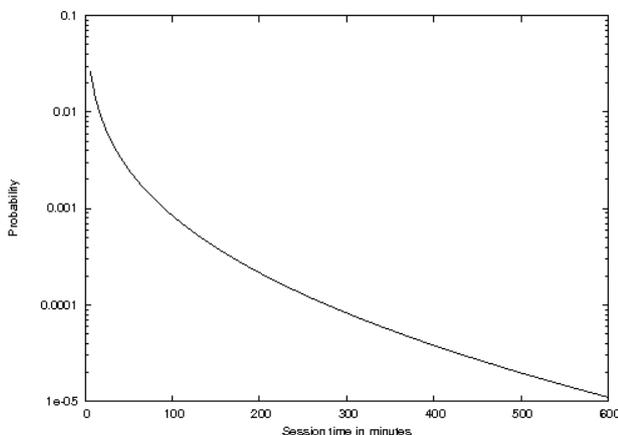


Figure 5.4: Distribution of player session times

RC with one from the pool, so we have  $18 * 5.9813 = 107.6634$  frames in total where an additional inconsistency occurs. We add this number to the 179.0142 inconsistencies that the model predicts due to delayed messages and end up with  $179.0142 + 107.6634 = 286.6776$  inconsistencies. Table 5.3 shows the results of the simulation runs.

It may seem surprising that the number of delayed requests is smaller than in the ideal world setting. To explain this, we must remember that the system in this scenario is not always fully utilized. Whenever a node leaves, we wait for approximately one minute of real-time until it is replaced by a new one. During this time, fewer Game Client requests are sent compared to the ideal scenario with no churn. Analyzing the log files of the simulation runs reveals that the system sends about 4.79 percent less request messages. Assuming that the number of delayed requests is also proportionally lower, we would expect on average  $179.3325 * 0.9621 = 170.7493$  delayed requests. However, as explained above, the actual number is slightly higher due to the bandwidth consumption of the RC initialization messages. Of course, the smaller number of delayed requests leads also to less inconsistencies than expected. The probability of a Region Controller being inconsistent per frame is  $\frac{286.6776}{6921 * 7} = 0.00578$ . Finally, probability of a voting failure per frame is

$$\sum_{i=4}^7 0.00578^i = 1.12009 * 10^{-9}$$

and results in one voting failure in 14.7 years on average. As we can see, adding churn significantly reduces the time between voting failures, but they still remain a very rare occasion.

### 5.6.3 Node crashes

In a realistic system, nodes do not always leave the system cleanly. They may crash and hence stop sending messages to other nodes. We thus extend the above node churn scenario with nodes that crash in a fail-stop manner. Since we could not find any real-life data on the probability of node crashes, we assumed

Simulation Run	Delayed Requests	RC Inconsistencies
1	164	272
2	194	301
3	195	300
4	155	262
5	184	290
6	176	279
7	171	276
8	179	285
9	172	276
10	170	273
11	158	262
12	179	283
13	171	275
14	179	287
15	172	277
Average	174.6000	279.8667
Expected	179.3325	286.6776

Table 5.3: Simulation results for scenario 1 including node churn

that ten percent of the leaving nodes do so without correctly unregistering themselves. Note that this is a very pessimistic assumption, we believe that this crash rate is most likely an order of magnitude larger than in a real system.

We expect the system to produce somewhat more inconsistencies than in the example above. The churn rate itself is the same, but since ten percent of the leaving nodes do not unregister themselves correctly it takes more time to replace them. Like in the scenario above, only nodes that were active Region Controllers at the time of their crashing affect the correctness of the system. Game Clients notice crashed RCs because they do not receive updates from them. After a certain timeout, the GC requests the removal of the RC at the management service. This timeout is a configuration parameter and set to three frames for our simulation. If the manager receives a certain amount (again, a configuration parameter which is usually set to the same value as the minimum number of RCs that are necessary for a majority vote) of such requests, it triggers the replacement process for the RC. On average, 1.8 nodes that crash are active RCs, resulting in  $3 * 1.8 = 5.4$  additional inconsistencies.

Table 5.4 shows the results of the simulation. As before, missing Region Controllers are treated as inconsistent. The number of delayed messages is nearly the same as in the scenario before. However, because the replacement of a crashed active RC takes three frames longer due to the replacement timeout, the number of inconsistencies grows slightly. The probability of a Region Controller being inconsistent per frame is  $\frac{288.6667}{6921 * 7} = 0.00596$ . Consequently, the probability of a voting failure per frame is

$$\sum_{i=4}^7 0.00596^i = 1.26799 * 10^{-9}$$

and results in one voting failure in 13.0 years on average.

Simulation Run	Delayed Requests	RC Inconsistencies
1	168	281
2	182	298
3	186	302
4	161	276
5	185	298
6	166	276
7	163	278
8	165	278
9	170	283
10	181	295
11	192	307
12	189	304
13	167	283
14	174	290
15	166	281
Average	174.3333	288.6667
Expected	179.3325	292.0776

Table 5.4: Simulation results for scenario 1 including node churn and node crashes

#### 5.6.4 Combination of all extensions

Finally, we performed the simulation including all three extensions described above. Table 5.5 shows the results of the simulation.

As expected, the number of delayed requests is nearly the same as in the last example. The additional clock skew has no effect on this number as explained in section 5.6.1. The number of Region Controller inconsistencies increases slightly, but not as significantly as from the first to the second scenario. The small effect of the clock skew submerges in the huge effect of the node churn. The probability of a Region Controller being inconsistent per frame is  $\frac{290.3333}{6921 \cdot 7} = 0.00599$ . Finally, the probability of a voting failure per frame is

$$\sum_{i=4}^7 0.00599^i = 1.29757 * 10^{-9}$$

and results in one voting failure in 12.7 years on average.

### 5.7 Further Scenarios

In this section, we analyzed different scenarios using nodes with varying bandwidths. The given up- and download bandwidths are realistic numbers based on current offerings by German Internet providers. As before, all scenarios are simulated for one hour of real-time. Obviously, if we keep the simulation length fixed, shorter frame lengths lead to a higher number of frames per simulation run. This means that the number of delayed requests and Region Controller inconsistencies also grows and thus these figures cannot be directly compared to those of the other scenarios. In order to compare different scenarios one must

Simulation Run	Delayed Requests	RC Inconsistencies
1	176	282
2	166	288
3	188	311
4	157	282
5	182	298
6	172	285
7	163	287
8	178	291
9	173	284
10	189	293
11	193	301
12	185	301
13	168	287
14	185	304
15	150	261
Average	175.0000	290.3333
Expected	179.3325	292.0776

Table 5.5: Simulation results for scenario 1 including clock skew, node churn and node crashes

look at the mean times between voting failures which are given at the end of each scenario.

### 5.7.1 Scenario 2 - 16 MBit ADSL Node, 25 Game Clients

This scenario is identical to the first one, except that we use a standard 16 MBit ADSL node.

Number of GCs per Region	25
Maximum number of RCs	7
Node download bandwidth	16 MBit/s
Node upload bandwidth	1 MBit/s
GC Request Size	60 Bytes
RC Update Size	300 Bytes
RC processing time	100 ms
Buffer time	20ms
Node location	Germany
Simulation length	16662 Frames

First, we use our model to predict the results of the ideal world setting. For the sending of requests, we get

$$\frac{7 * 100\text{Bytes}}{1\text{MBit/s}} = \frac{700\text{Bytes}}{131072\text{Bytes/s}} = 0.00534s$$

for the upload and

$$\frac{25 * 100\text{Bytes}}{16\text{MBit/s}} = \frac{2500\text{Bytes}}{2097152\text{Bytes/s}} = 0.00119s$$

for the download. For the sending of updates, we get

$$\frac{25 * 340\text{Bytes}}{1\text{MBit/s}} = \frac{8500\text{Bytes}}{131072\text{Bytes/s}} = 0.06485s$$

and

$$\frac{7 * 340\text{Bytes}}{16\text{MBit/s}} = \frac{2380\text{Bytes}}{2097152\text{Bytes/s}} = 0.00405s$$

respectively. In both cases the upload link is the limiting factor. We round the transmission times, add the buffer time and the minimum propagation delay to each of the two phases. Finally, we add the processing time, resulting in a frame length of

$$(5ms + 20ms + 3ms) + 100ms + (65ms + 20ms + 3ms) = 216ms$$

To speed up the calculation of the average probability of a request message to arrive in time, we derive a single equation. First, we combine equations 5.8 and 5.9:

$$P(InTime_m) = \frac{\sum_{i=1}^n f_2(2 * (ReqInt - TransDelay_{m_i} - PropDelay); \mu, \sigma)}{n}$$

Next, to calculate  $TransDelay_{m_i}$ , we use equation 5.5 and end up with

$$P(InTime_m) = \frac{\sum_{i=1}^n f_2(2 * (ReqInt - (i * TransDelay_m) - PropDelay); \mu, \sigma)}{n}$$

The size of the request interval  $ReqInt$  is  $28ms$ , the minimum propagation delay  $PropDelay$  is  $3ms$  and the transmission delay of a single message  $TransDelay_m$  can be calculated using equation 5.4:

$$TransDelay_{m_i} = i * TransDelay_m = i * \frac{100\text{Bytes}}{1\text{MBit/s}} = i * 0.76294ms$$

Since we send  $n = 7$  requests, we end up with

$$P(InTime_m) = \frac{\sum_{i=1}^7 f_2(2 * (28 - (i * 0.76294) - 3); \mu, \sigma)}{7} = 0.99968$$

The probability for any single request to arrive in time is 0.99968, which means that on average 935.1887 requests will be too late. The probability for any Region Controller being inconsistent in a frame is

$$1 - 0.99968^{25} = 0.00799$$

so we get 931.9057 inconsistencies on average. Finally, the probability that a voting failure within any frame occurs is

$$\sum_{i=4}^7 0.00799^i = 4.10296 * 10^{-9}$$

meaning that every 1.7 years a voting failure will occur. This is a surprisingly small time compared to our first setting. A small difference of 0.00018 in the probability of a single request to arrive in time doubles the probability of a

Simulation Run	Delayed Requests	RC Inconsistencies
1	949	945
2	973	969
3	935	931
4	891	886
5	924	920
6	937	932
7	901	895
8	912	911
9	964	962
10	942	938
11	943	938
12	899	894
13	961	959
14	933	930
15	953	948
Average	934.4667	930.5333
Expected	935.1887	931.9057

Table 5.6: Simulation results for scenario 2 with ideal-world setting and  $20ms$  buffer

single RC becoming inconsistent. Hereupon the probability of a voting failure is increased nearly by the factor 22, leading to a comparatively short time between voting failures. Table 5.6 shows the results of the corresponding simulation runs.

In order to lower the probability of a voting failure, we repeated the experiment with an increased buffer time. Raising the buffer time to  $25ms$ , resulting in a total frame length of  $226ms$ , showed the desired results. The number of delayed requests decreases to an average of 433.2808, the number of inconsistencies to 432.4733. This leads to one voting failure in about 31.5 years on average. Table 5.7 shows the results of the simulation runs with an extended buffer time of  $25ms$ .

We repeated the simulation in the real-world setting, including node churn, crashes and clock skew. Since we have the same number of nodes and the the same amount of simulated real time, the average number of leaving and crashing nodes is the same as before.

In the  $20ms$  buffer scenario, a frame has a length of  $216ms$ . The upload bandwidth of the RC is  $131072Bytes/s$ , so within a frame an RC can upload  $131072Bytes/s * 216ms = 28312Bytes$ . For sending the state message, per frame  $28312Bytes - 8500Bytes = 19812Bytes$  remain and the sending of the state takes

$$\frac{1024Bytes * 25}{19812Bytes/Frame} = 1.2921$$

frames on average. Since eighteen RCs need to be replaced of which ten percent crash, we end up with  $18 * 1.2921 + 5.4 = 28.6578$  additional inconsistencies.

In the  $25ms$  buffer scenario, a frame has a length of  $226ms$ . Within a frame, an RC can upload  $131072Bytes/s * 226ms = 29622Bytes$ . For sending the state message, per frame  $29622Bytes - 8500Bytes = 21122Bytes$  remain and the

Simulation Run	Delayed Requests	RC Inconsistencies
1	427	425
2	460	459
3	449	448
4	418	416
5	443	440
6	433	432
7	437	431
8	427	423
9	423	422
10	416	411
11	432	432
12	428	426
13	454	451
14	429	428
15	411	407
Average	432.4667	430.0667
Expected	433.2808	432.4732

Table 5.7: Simulation results for scenario 2 with ideal-world setting and 25ms buffer

sending of the state takes

$$\frac{1024\text{Bytes} * 25}{21122\text{Bytes}/\text{Frame}} = 1.2120$$

frames on average and we end up with  $18 * 1.2120 + 5.4 = 27.2160$  additional inconsistencies.

The results are shown in table 5.8 for the setting with 20ms buffer time and table 5.9 for 25ms buffer time respectively. The probability of a Region Controller being inconsistent per frame is in the first setting  $\frac{943.2667}{16662*7} = 0.00809$ . The probability of a voting failure per frame is

$$\sum_{i=4}^7 0.00809^i = 4.31285 * 10^{-9}$$

and results in one voting failure in 1.6 years on average. For the second setting, the probability of a Region Controller being inconsistent per frame is  $\frac{459.6892}{15925*7} = 0.00412$ . The probability of a voting failure per frame is

$$\sum_{i=4}^7 0.00412^i = 2.90363 * 10^{-10}$$

and results in one voting failure in 24.7 years on average.

### 5.7.2 Scenario 3 - 16 MBit ADSL Node, 100 Game Clients

We use the same ADSL nodes as in the last scenario, but now extend the number of Game Clients per region to one hundred. Because of the high number of GCs,

Simulation Run	Delayed Requests	RC Inconsistencies
1	952	979
2	880	905
3	938	965
4	945	971
5	905	930
6	941	967
7	905	933
8	933	960
9	908	934
10	933	960
11	938	962
12	849	874
13	926	951
14	881	907
15	925	951
Average	917.2666	943.2667
Expected	935.1887	960.5635

Table 5.8: Simulation results for scenario 2 with real-world setting and 20ms buffer

Simulation Run	Delayed Requests	RC Inconsistencies
1	454	478
2	411	432
3	431	452
4	449	473
5	402	423
6	432	454
7	434	455
8	423	448
9	453	477
10	390	413
11	435	457
12	431	455
13	436	458
14	442	463
15	401	424
Average	428.2667	450.8000
Expected	433.2808	459.6892

Table 5.9: Simulation results for scenario 2 with real-world setting and 25ms buffer

we further increase the buffer time by  $10ms$  to keep the number of inconsistencies low.

Number of GCs per Region	100
Maximum number of RCs	7
Node download bandwidth	16 MBit/s
Node upload bandwidth	1 MBit/s
GC Request Size	60 Bytes
RC Update Size	300 Bytes
RC processing time	100 ms
Buffer time	35ms
Node location	Germany
Simulation length	8179 Frames

For the sending of requests, our model yields

$$\frac{7 * 100Bytes}{1MBit/s} = \frac{700Bytes}{131072Bytes/s} = 0.00534s$$

for the upload and

$$\frac{100 * 100Bytes}{16MBit/s} = \frac{10000Bytes}{2097152Bytes/s} = 0.00477s$$

for the download. For the sending of updates, we get

$$\frac{100 * 340Bytes}{1MBit/s} = \frac{34000Bytes}{131072Bytes/s} = 0.25940s$$

and

$$\frac{7 * 340Bytes}{16MBit/s} = \frac{2380Bytes}{2097152Bytes/s} = 0.00405s$$

respectively. As before, in both cases the upload link is the limiting factor. For the frame length we get

$$(5ms + 35ms + 3ms) + 100ms + (259ms + 35ms + 3ms) = 440ms$$

The probability for any single request to arrive in time is

$$P(InTime_m) = \frac{\sum_{i=1}^n f_2(2 * (43 - (i * 0.76294) - 3); \mu, \sigma)}{7} = 0.99995$$

so on average 274.8737 requests will be too late. The probability for any Region Controller being inconsistent in a frame is

$$1 - 0.99995^{100} = 0.00479$$

so we get 274.2215 inconsistencies on average. Finally, the probability that a voting failure within any frame occurs is

$$\sum_{i=4}^7 0.00479^i = 5.28808 * 10^{-10}$$

Simulation Run	Delayed Requests	RC Inconsistencies
1	275	274
2	268	268
3	301	298
4	258	256
5	282	280
6	273	272
7	262	260
8	265	265
9	290	289
10	271	270
11	299	298
12	268	268
13	242	241
14	259	258
15	265	262
Average	271.8667	270.6000
Expected	274.8737	274.2215

Table 5.10: Simulation results for scenario 3 with ideal-world setting

meaning that every 26.4 years a voting failure will occur. Table 5.10 shows the results of the simulation.

For predicting the real-world scenario, we have to recalculate the time necessary to replace an RC. The upload bandwidth of an RC is  $131072\text{Bytes}/s$ , so within a frame an RC can upload  $131072\text{Bytes}/s * 440ms = 57672\text{Bytes}$ . For sending the state message, per frame  $57672\text{Bytes} - 34000\text{Bytes} = 23672\text{Bytes}$  remain and the sending of the state takes

$$\frac{1024\text{Bytes} * 100}{23672\text{Bytes}/\text{Frame}} = 4.326$$

frames on average. Since we did not change the number of Region Controllers, there are still eighteen RCs that need to be replaced of which ten percent crash. We end up with  $18 * 4.326 + 5.4 = 83.268$  additional inconsistencies.

Table 5.11 shows the results of the simulation. The probability of a Region Controller being inconsistent per frame is  $\frac{349.1333}{8179 * 7} = 0.00809$ . The probability of a voting failure per frame is

$$\sum_{i=4}^7 0.00610^i = 1.39132 * 10^{-9}$$

and results in one voting failure in 10.3 years on average.

### 5.7.3 Scenario 4 - 50 MBit VDSL Node, 100 Game Clients

Our last two scenarios make use of so-called VDSL (which stands for *Very High Speed Digital Subscriber Line*) nodes. They are already available in major German cities and provide a downstream of up to  $50\text{MBit}/s$  and a downstream

Simulation Run	Delayed Requests	RC Inconsistencies
1	253	336
2	282	362
3	255	337
4	284	367
5	265	346
6	282	363
7	263	345
8	281	362
9	265	347
10	245	327
11	253	336
12	278	359
13	271	353
14	297	377
15	238	320
Average	267.4667	349.1333
Expected	274.8737	357.4895

Table 5.11: Simulation results for scenario 3 with real-world setting

of up to  $10\text{MBit/s}$ . In this scenario we investigate how the system performs on faster connections using the same number of nodes as in the last scenario.

Number of GCs per Region	100
Maximum number of RCs	7
Node download bandwidth	50 MBit/s
Node upload bandwidth	10 MBit/s
GC Request Size	60 Bytes
RC Update Size	300 Bytes
RC processing time	100 ms
Buffer time	35ms
Node location	Germany
Simulation length	17643 Frames

For the sending of requests, our model yields

$$\frac{7 * 100\text{Bytes}}{10\text{MBit/s}} = \frac{700\text{Bytes}}{1310720\text{Bytes/s}} = 0.00053s$$

for the upload and

$$\frac{100 * 100\text{Bytes}}{50\text{MBit/s}} = \frac{10000\text{Bytes}}{6553600\text{Bytes/s}} = 0.00153s$$

for the download. For the sending of updates, we get

$$\frac{100 * 340\text{Bytes}}{10\text{MBit/s}} = \frac{34000\text{Bytes}}{1310720\text{Bytes/s}} = 0.02594s$$

and

$$\frac{7 * 340\text{Bytes}}{50\text{MBit/s}} = \frac{2380\text{Bytes}}{6553600\text{Bytes/s}} = 0.00130s$$

Simulation Run	Delayed Requests	RC Inconsistencies
1	617	616
2	584	584
3	569	569
4	576	574
5	610	606
6	546	543
7	545	544
8	575	572
9	615	613
10	596	596
11	552	549
12	584	580
13	607	606
14	559	559
15	588	587
Average	581.5333	579.8667
Expected	581.3415	579.9891

Table 5.12: Simulation results for scenario 4 with ideal-world setting

respectively. For the first time, the download link of a Region Controller is the limiting factor for the request phase. However, the difference for the rounded transmission times is only one millisecond. As before, the upload link of an RC limits the sending of updates. For the frame length we get

$$(2ms + 35ms + 3ms) + 100ms + (26ms + 35ms + 3ms) = 204ms$$

The probability for any single request to arrive in time is

$$P(InTime_m) = \frac{\sum_{i=1}^n f_2(2 * (40 - (i * 0.00153) - 3); \mu, \sigma)}{7} = 0.999953$$

so on average 581.3415 requests will be too late. The probability for any Region Controller being inconsistent in a frame is

$$1 - 0.999953^{100} = 0.00470$$

so we get 579.9891 inconsistencies on average. Finally, the probability that a voting failure within any frame occurs is

$$\sum_{i=4}^7 0.00479^i = 4.88699 * 10^{-10}$$

meaning that every 13.2 years a voting failure will occur. Compared to the last scenario, the shorter frames length incurred by the faster connections of the nodes lead to a shorter mean time between a voting failure. Table 5.12 shows the results of the simulation.

We now calculate the time needed to replace an active Region Controller. The upload bandwidth of an RC is 1310720Bytes/s, so within a frame an RC

Simulation Run	Delayed Requests	RC Inconsistencies
1	601	614
2	597	611
3	569	582
4	589	599
5	576	587
6	531	545
7	588	600
8	610	621
9	567	580
10	548	559
11	588	600
12	575	588
13	542	556
14	573	584
15	563	576
Average	574.4667	586.8000
Expected	581.3415	593.2911

Table 5.13: Simulation results for scenario 4 with real-world setting

can upload  $1310720\text{Bytes}/s * 204\text{ms} = 267387\text{Bytes}$ . For sending the state message, per frame  $267387\text{Bytes} - 34000\text{Bytes} = 233387\text{Bytes}$  remain and the sending of the state takes

$$\frac{1024\text{Bytes} * 100}{233387\text{Bytes}/\text{Frame}} = 0.439$$

frames on average. Thus, we end up with  $18 * 0.439 + 5.4 = 13.302$  additional inconsistencies.

Table 5.13 shows the results of the simulation. The probability of a Region Controller being inconsistent per frame is  $\frac{586.8}{17643 * 7} = 0.00475$ . The probability of a voting failure per frame is

$$\sum_{i=4}^7 0.00475^i = 5.12091 * 10^{-10}$$

and results in one voting failure in 12.6 years on average.

#### 5.7.4 Scenario 5 - 50 MBit VDSL Node, 250 Game Clients

For our last scenario, we use the same VDSL nodes as in the last scenario, but extend the number of Game Clients per region to 250. Because of the higher number of GCs, we increase the buffer time again by  $10\text{ms}$ .

Number of GCs per Region	250
Maximum number of RCs	7
Node download bandwidth	50 MBit/s
Node upload bandwidth	10 MBit/s
GC Request Size	60 Bytes
RC Update Size	300 Bytes
RC processing time	100 ms
Buffer time	45ms
Node location	Germany
Simulation length	13580 Frames

For the sending of requests, our model yields

$$\frac{7 * 100Bytes}{10MBit/s} = \frac{700Bytes}{1310720Bytes/s} = 0.00053s$$

for the upload and

$$\frac{250 * 100Bytes}{50MBit/s} = \frac{25000Bytes}{6553600Bytes/s} = 0.00381s$$

for the download. For the sending of updates, we get

$$\frac{250 * 340Bytes}{10MBit/s} = \frac{85000Bytes}{1310720Bytes/s} = 0.06485s$$

and

$$\frac{7 * 340Bytes}{50MBit/s} = \frac{2380Bytes}{6553600Bytes/s} = 0.00130s$$

respectively. As in the last scenario, the download link of a Region Controller is the limiting factor for the request phase and the upload link of an RC limits the sending of updates. For the frame length we get

$$(4ms + 45ms + 3ms) + 100ms + (65ms + 45ms + 3ms) = 265ms$$

The probability for any single request to arrive in time is

$$P(InTime_m) = \frac{\sum_{i=1}^n f_2(2 * (52 - (i * 0.00152) - 3); \mu, \sigma)}{7} = 0.999984$$

so on average 370.2597 requests will be too late. The probability for any Region Controller being inconsistent in a frame is

$$1 - 0.999984^{250} = 0.00389$$

so we get 369.5424 inconsistencies on average. Finally, the probability that a voting failure within any frame occurs is

$$\sum_{i=4}^7 0.00389^i = 2.29275 * 10^{-10}$$

meaning that every 36.7 years a voting failure will occur. Compared to the last scenario, the shorter frames length incurred by the faster connections of the

Simulation Run	Delayed Requests	RC Inconsistencies
1	366	366
2	374	373
3	370	368
4	354	353
5	345	342
6	375	374
7	383	381
8	363	363
9	401	400
10	369	368
11	393	389
12	389	386
13	354	352
14	351	350
15	362	359
Average	369.9333	368.2667
Expected	370.2597	369.5424

Table 5.14: Simulation results for scenario 5 with ideal-world setting

nodes lead to a shorter mean time between a voting failure. Table 5.14 shows the results of the simulation.

We now calculate the time needed to replace an active Region Controller. The upload bandwidth of an RC is  $1310720\text{Bytes}/s$ , so within a frame an RC can upload  $1310720\text{Bytes}/s * 265ms = 347341\text{Bytes}$ . For sending the state message, per frame  $347341\text{Bytes} - 85000\text{Bytes} = 262341\text{Bytes}$  remain and the sending of the state takes

$$\frac{1024\text{Bytes} * 100}{262341\text{Bytes}/\text{Frame}} = 0.390$$

frames on average. Thus, we end up with  $18 * 0.390 + 5.4 = 12.420$  additional inconsistencies.

Table 5.15 shows the results of the simulation. The probability of a Region Controller being inconsistent per frame is  $\frac{376.2667}{13580 * 7} = 0.00396$ . The probability of a voting failure per frame is

$$\sum_{i=4}^7 0.00396^i = 2.46442 * 10^{-10}$$

and results in one voting failure in 26.2 years on average.

## 5.8 Results

In this section, we recapitulate the results of our calculations and the simulation runs. We start with the results for the different extensions of our first scenario which are shown in table 5.16. The table contains the expected number of inconsistencies determined by our analytical model, the average number

Simulation Run	Delayed Requests	RC Inconsistencies
1	369	382
2	382	395
3	387	399
4	363	378
5	345	356
6	380	394
7	357	369
8	354	366
9	338	352
10	395	407
11	381	395
12	338	352
13	359	372
14	337	352
15	361	375
Average	363.0667	376.2667
Expected	370.2597	381.9624

Table 5.15: Simulation results for scenario 5 with real-world setting

Scenario	Expected Inc.	Average Inc.	Error	MTBVF
ideal	179.0142	178.7333	0.0016	88.0
clock skew	179.0142	181.1333	0.0117	84.1
churn	286.6776	279.8667	0.0243	14.7
crashes	292.0776	288.6667	0.0118	13.0
combined	292.0776	290.3333	0.0060	12.7

Table 5.16: Results of the different extensions of scenario 1

of inconsistencies from the simulation runs, the error of the simulation results compared to the analytical results and the mean time between voting failures in years (MTBVF).

As we can see, the difference between the model and the simulation results is very low, at most 2.43 percent. Especially in the ideal-world scenario, the model very precisely predicts the average of the simulation results. The error slightly increases in the clock skew scenario, since the analytical model was not adapted to this extension. Only for the churn and crash extensions the model was also enhanced. Here it shows the largest deviations from the simulation results, but still the difference remains very low. For the most interesting scenario, the combination of all extensions, the error decreases again. However, when looking at the higher errors of the other real-world scenarios, this small error seems to be just coincidental.

For the ideal-world setting, we can expect just a single voting failure within 88 years of playing. Even the addition of clock skew doesn't change this significantly. However, as node churn comes into play, the probability of an voting failures increases considerably. Although the number of inconsistencies only increases by approximately 65 percent, the MTBVF drops to less than 18 percent. As expected, adding node crashes and clock skew further lowers the MT-

Scenario	Expected Inc.	Average Inc.	Error	MTBVF
1	292.0776	290.3333	0.0060	12.7
2a	960.5635	943.2667	0.0183	1.6
2b	459.6892	450.8000	0.0197	24.7
3	357.4895	349.1333	0.0239	10.3
4	593.2911	586.8000	0.0111	12.6
5	381.9624	376.2667	0.0151	26.2

Table 5.17: Results for the real-life scenarios

BVF. Compared to the significant decrease caused by adding node churn, these changes are rather small.

Table 5.17 shows the results for all real-life scenarios in the same format. The highest error that occurs is still below 2.5 percent, showing that our analytical model is very precise. Looking at the mean time between voting failures shows an interesting fact. Scenario 2a (the one with the 20ms buffer) uses the same settings as scenario 1, only the higher bandwidth leads to a shorter frame length (216ms instead of 520ms). Reducing the frame length by the factor 2.41 multiplies the number of inconsistencies by 3.25 and the MTBVF even decreases by the factor 7.94. This clearly shows, that if we just reduce the frame length proportionally to the additional bandwidth, this may lead to a significantly higher chance of a voting failure. In scenario 2b, which uses just a slightly larger buffer time of 25ms instead of 20ms, the situation is reversed. Due to the rather small buffer extension, the MTBVF is even twice as high as in scenario 1. Consequently, the buffer is raised in the following scenarios to 35ms and 45ms respectively. The MTBVF stays between 10.3 and 26.2 years, which we consider very acceptable.

## 5.9 Conclusion

In this chapter we have evaluated whether the approach presented in the previous chapter is feasible in a realistic setting. For this purpose, we developed an analytical model that allows us to determine the number of inconsistencies and the mean time between voting failures according to a range of parameters like the bandwidth of the player nodes, the number of Game Clients and Region Controllers and the length of a frame. To verify the results of the model, we implemented the system using a simulation framework. We simulated five different scenarios with realistic parameters and took the average of fifteen simulation runs per scenario. The results of the simulation runs show that the analytical model gives a rather precise prediction of the average simulation results. Furthermore, they show that our approach can be realized on player nodes that use currently available Internet connections.

A shortcoming of our model is that one cannot directly calculate the necessary buffer times for given scenario parameters and a desired mean time between voting failures. In order to do so, we would need the inverse of the cumulative distribution function of the log-normal distribution. Unfortunately, no simple analytical closed form of this function exists. However, the model can be used to approximate buffer sizes that ensure that the probability of a voting failures stays within certain bounds.

## Chapter 6

# Summary and Future Work

In this thesis we presented a novel network architecture for multiplayer online games that reduces the costs for providing online game services significantly by utilizing available computational and bandwidth resources on the customers' computers. Additionally, we evaluated whether the proposed architecture is suitable for real-life scenarios. Finally, we embedded this architecture into a development framework that reduces complexity while at the same time enhances reusability.

Chapter 3 presents the framework that provides a game developer with a complete abstraction from network related issues. With network implementation details hidden, game developers can focus more on game design rather than writing specialized code. Implementation details like data-driven game objects further emphasize this approach. The framework consists of three layers which hide the details of the respective lower layers. Usually, a regular game developer will only get in touch with the highest layer, the game layer. On this layer, standard components, like the game engine and components managing audiovisual feedback and player input, are located. The modular design allows to easily replace components with custom or off-the-shelf ones. This is also the place where the rules and the logic of a specific game are implemented. All components on this layer communicate with the layer below, the object layer. The object layer makes access to remote objects completely transparent. Since consistency and ownership management are handled automatically, game developers can create, manipulate and delete all game objects as if they were local. The lowest layer, the networking layer, hides all network related issues behind a Publish/Subscribe abstraction. If necessary, this layer can be customized for different quality requirements, like higher scalability or lower latency.

In chapter 4 we presented a Peer-to-Peer gaming system that distributes the management of game state and logic among the nodes of the players. This way it utilizes unused computing time and bandwidth on the players' computers and relieves the game publisher's servers of resource intensive tasks. Storing the game state on player nodes makes it vulnerable for being manipulated by malicious nodes. By not trusting a single node for managing the correct state but replicating it on multiple nodes we counteract tampering. Because there may now exist multiple dissenting versions of the game state, the correct one is determined by a majority voting. Since player nodes are not powerful enough to handle the complete state of the game world, it is split into smaller sized regions.

A single player node only manages a replica of a region which is assigned to the node by a central trusted service. This service is also responsible for handling sensitive player information, like subscription and credit card data.

In chapter 5 we provided a detailed evaluation of the network architecture proposed in the previous chapter. We developed an analytical model that allows us to determine the number of inconsistencies and the mean time between voting failures according to a range of parameters like the bandwidth of the player nodes, the number of Game Clients and Region Controllers and the length of a frame. The model was verified against a implementation of the system using a simulation framework. We evaluated five different scenarios with realistic parameters. For each scenario we performed fifteen simulation runs and compared the average to our model. The results of the simulation runs show that the analytical model gives a rather precise prediction of the average simulation results. Furthermore, they show that our approach can be realized on player nodes that use currently available Internet connections.

## 6.1 Future Work

Many multiplayer online games are session-based, i.e. the game runs only for a limited amount of time. E.g. a First-Person-Shooter deathmatch session may be started and runs for twenty minutes. Players may join and leave at any time during the session. At the end, the score for each remaining player is determined. On the contrary, a session of a Real-Time-Strategy game is started and runs until one of the players wins. Usually, no new players may join the session and nobody should leave.

For session-based games, persistence is not an issue. However, for Massively Multiplayer Online Games (MMOGs) which usually have persistent worlds, the situation is different. The "session" starts when the game is launched and is closed probably years later when the game service is discontinued. During this time, players may accumulate plenty of virtual achievements and possessions. The state of such persistent game worlds should be regularly written to persistent storage. This is necessary in case the system completely crashes or needs to be shut down for maintenance purposes. Moreover, if a region of the game world is currently empty, i.e. there are no avatars in it, the region can be shut down and the corresponding RCs can go back to the pool of free RCs.

In order to create a snapshot of the current region state, all RCs of that region send the changes since the last backup to a persistence service. It determines the correct state by choosing the one which holds the majority. Every time a player leaves, his avatar's data can be sent to the persistent service. Whenever he joins again, his data can directly be sent to the responsible RCs. This makes it unnecessary for RCs to store data of avatars which are currently not in the game. The persistence service may be provided by the game hoster or could be itself a P2P-based system running on players' nodes. [89]

There are also still opportunities for future research in detecting and persecuting cheating attacks. One is the implementation of a log auditing service [59] which enables the system to detect certain kind of attacks (see section 4.5) which cannot be prevented in the first place. The idea is that each node of the system

keeps a log of all received messages for a certain period. Since all messages in our system are required to be signed, it is possible to prove the origin of each message. Whenever a node detects a cheating attack, it can request an investigation of the case by a trusted authority. For this purpose, it sends a digest of the received messages (including their signatures) to the trusted authority which can analyze the messages. If an attack has been detected, appropriate measures can be taken.

Another opportunity for future work is the challenge of how to deal with omitted messages as described in section 4.5.4. As we have shown, a malicious node cannot gain unfair advantages from omitting messages. However, it can cause Region Controllers to go out-of-sync and thus degrade player experience. The problem is, that neither the sender can prove that it sent a message nor the receiver can prove that it did not receive a message. If the underlying network provided some reliable multicast mechanism that inserts a list of all recipients into every message, at least the honest recipients can prove that a certain node was omitted. This way clients that omit only certain Region Controllers from their requests can be detected. Another possibility would be the use of reputation systems [57]. Whenever a node does not receive a required message it may report the omitting node to the reputation service. This way, a long-term estimation about the trustworthiness of nodes can be achieved.



# Bibliography

- [1] Mustaque Ahamad, Phillip W. Hutto, Gil Neiger, James E. Burns, and Prince Kohli. Causal memory: Definitions, implementation and programming. Technical Report GIT-CC-93/55, Georgia Institute of Technology, 1994.
- [2] ArenaNet. Guild Wars. [www.guildwars.com](http://www.guildwars.com), 2008.
- [3] Grenville Armitage. Sensitivity of Quake3 players to network latency. ACM SIGCOMM Internet Measurement Workshop (Poster Session), 2001.
- [4] Marios Assiotis and Velin Tzanov. A distributed architecture for MMORPG. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, 2006.
- [5] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.
- [6] Nathaniel E. Baughman and Brian Neil Levine. Cheat-proof payout for centralized and distributed online games. In *Proceedings of the IEEE INFOCOM*, 2001.
- [7] Nathaniel E. Baughman, Marc Liberatore, and Brian Neil Levine. Cheat-proof payout for centralized and serverless online games. Technical report, University of Massachusetts Amherst, 2004.
- [8] Eric J. Berglund and David R. Cheriton. Amaze: A multiplayer computer game. *IEEE Software*, 2(3):30–39, 1985.
- [9] Paul Bettner and Mark Terrano. 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. In *Proceedings of the Game Developers Conference*, 2001.
- [10] Ashwin Bhamambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the ACM SIGCOMM*, September 2004.
- [11] Ashwin Bhamambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: A distributed architecture for online multiplayer games. In *Proceedings of the ACM NSDI*, 2006.
- [12] Ashwin R. Bhamambe, Sanjay Rao, and Srinivasan Seshan. Mercury: a scalable publish-subscribe system for internet games. In *Proceedings of the 1st workshop on Network and system support for games*, pages 3–9. ACM Press, 2002.

- [13] Scott Bilas. A data-driven game object system. In *Proceedings of the Games Developer Conference*, 2002.
- [14] Blizzard Entertainment. World of Warcraft. [www.worldofwarcraft.com](http://www.worldofwarcraft.com), 2008.
- [15] Jonathan Blow. Game development: Harder than you think. *ACM Queue* vol. 1, no. 10, 2004.
- [16] Yves Bresson. XBlasT. [www.xblast-center.com](http://www.xblast-center.com), 2008.
- [17] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. *Distributed systems (2nd Ed.)*, pages 199–216, 1993.
- [18] Michael Buro. ORTS: A hack-free RTS game environment. In *Proceedings of the International Joint Conference on AI*, 2003.
- [19] CAIDA. Macroscopic Topology Measurements. [www.caida.org/projects/macroscopic](http://www.caida.org/projects/macroscopic), 2008.
- [20] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication*, 20(8), 2002.
- [21] CCP Games. Eve Online. [www.eve-online.com](http://www.eve-online.com), 2008.
- [22] Fábio Reis Cecin, Rafael de Oliveira Jannone, Cláudio Fernando Resin Geyer, Márcio Garcia Martins, and Jorge Luis Victoria Barbosa. FreeMMG: A hybrid peer-to-peer and client-server model for massively multiplayer games. In *Proceedings of the 3th ACM SIGCOMM workshop on Network and system support for games*, Workshops on NetGames '04, pages 172–172. ACM Press, 2004.
- [23] Fábio Reis Cecin, Rodrigo Real, Márcio Garcia Martins, Rafael de Oliveira Jannone, Jorge Luis Victória Barbosa, and Cláudio Fernando Resin Geyer. FreeMMG: A Scalable and Cheat-Resistant Distribution Model for Internet Games. In *8th IEEE International Symposium on Distributed Simulation and Real Time Applications*, 2004.
- [24] Chris Chambers, Wu-chang Feng, Wu-chi Feng, and Debanjan Saha. Mitigating information expose to cheaters in real-time strategy games. In *Proceedings of the 15th international workshop on Network and operating systems support for digital audio and video*, 2005.
- [25] Angie Chandler and Joe Finney. On the effects of loose causal consistency in mobile multiplayer games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, 2005.
- [26] Alvin Chen and Richard R. Muntz. Peer clustering: A hybrid approach to distributed virtual environments. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, 2006.

- [27] Doug Church. Object systems: Methods for attaching data to objects and connecting behavior. In *Proceedings of the Game Developers Conference*, 2002.
- [28] Mark Claypool. The effect of latency on user performance in real-time strategy games. *Computer Networks*, 49(1):52–70, September 2005.
- [29] Amy Beth Corman, Peter Schachte, and Vanessa Teague. A Secure Group Agreement (SGA) Protocol for Peer-to-Peer Applications. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, 2007.
- [30] Peter H. Dana. Global Positioning System (GPS) Time Dissemination for Real-Time Applications. *Real-Time Systems*, 12(1):9–40, 1997.
- [31] Steven B. Davis. Why cheating matters - cheating, game security, and the future of global on-line gaming business. In *Proceedings of the 2003 Game Developers Conference*, March 2003.
- [32] DFC Intelligence. [www.dfciint.com/news/prjune62006.html](http://www.dfciint.com/news/prjune62006.html), 2006.
- [33] Michael Doherty. A software architecture for games. Technical report, University of the Pacific Department of Computer Science, 2003.
- [34] Thomas P. Duncan and Denis Gračanin. Algorithms and analyses: Pre-reckoning algorithm for distributed virtual environments. In *Proceedings of the 35th conference on Winter simulation*, 2003.
- [35] Alex Duran. Building object-systems: Features, tradeoffs and pitfalls. In *Proceedings of the Game Developers Conference*, 2003.
- [36] Hans Eriksson. MBONE: the multicast backbone. *Communications of the ACM*, 37(8):54–60, 1994.
- [37] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.
- [38] Lu Fan, Hamish Taylor, and Phil Trinder. Mediator: a design framework for P2P MMOGs. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, 2007.
- [39] Wu-chang Feng, David Brandt, and Debanjan Saha. A long-term study of a popular MMORPG. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, 2007.
- [40] Wu-chang Feng, Francis Chang, Wu-chi Feng, and Jonathan Walpole. A traffic characterization of popular on-line games. *IEEE/ACM Transactions on Networking*, 13(3):488–500, 2005.
- [41] Stefano Ferretti and Marco Rocchetti. AC/DC: an algorithm for cheating detection by cheating. In *Proceedings of the 16th international workshop on Network and operating systems support for digital audio and video*, 2006.

- [42] Stefan Fiedler, Michael Wallner, and Michael Weber. A communication architecture for massive multiplayer games. In *Proceedings of the 1st workshop on Network and system support for games*, 2002.
- [43] Epic Games. Unreal 3 engine. [www.unrealtechnology.com/html/technology/ue30.shtml](http://www.unrealtechnology.com/html/technology/ue30.shtml), 2007.
- [44] Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, 2004.
- [45] Laurent Gautier and Christophe Diot. Design and evaluation of MiMaze, a multi-player game on the internet. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, 1998.
- [46] Laurent Gautier and Christophe Diot. Distributed synchronization for multiplayer interactive applications on the internet. Unpublished, 1998.
- [47] F. Glinka, A. Ploß, J. Müller-Iden, and S. Gorlatch. RTF: A real-time framework for developing scalable multiplayer online games. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, 2007.
- [48] Oliver Heckmann. *A System-oriented Approach to Efficiency and Quality of Service for Internet Service Providers*. PhD thesis, Technische Universität Darmstadt, 2004.
- [49] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [50] Mojtaba Hosseini, Steve Pettifer, and Nicolas D. Georganas. Visibility-based interest management in collaborative virtual environments. In *Proceedings of the 4th international conference on Collaborative virtual environments*, 2002.
- [51] Tobias Hoßfeld, Andreas Mäder, Kurt Tutschku, Phuoc Tran-Gia, Frank-Uwe Andersen, Hermann de Meer, and Ivan Dedinski. Comparison of crawling strategies for an optimized mobile p2p architecture. In *19th International Teletraffic Congress (ITC19)*, Beijing, China, sep 2005.
- [52] ID Software. Quake 2. [www.idsoftware.com/games/quake/quake2](http://www.idsoftware.com/games/quake/quake2), 2008.
- [53] IEEE. Standard for distributed interactive simulation - communication services and profiles, 1996.
- [54] IEEE. Standard for distributed interactive simulation - application protocols, 1998.
- [55] V. Jacobson. Congestion avoidance and control. In *SIGCOMM Symposium proceedings on Communications architectures and protocols*, 1988.

- [56] Daniel James, Gordon Walton, Brian Robbins, Elonka Dunin, Greg Mills, John Welch, Jeferson Valadares, Jon Estanislao, and Steven DeBenedictis. IGDA Persistent Worlds Whitepaper, 2004.
- [57] Audun Josang, Roslan Ismail, and Colin Boyd. A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2):618–644, 2007.
- [58] Patric Kabus and Alejandro P. Buchmann. Design of a Cheat-Resistant P2P Online Gaming System. In *Proceedings of the 2nd International Conference on Digital Interactive Media in Entertainment and Arts*, 2007.
- [59] Patric Kabus, Wesley W. Terpstra, Mariano Cilia, and Alejandro P. Buchmann. Addressing cheating in distributed MMOGs. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, 2005.
- [60] Yugo Kaneda, Hitomi Takahashi, Masato Saito, Hiroto Aida, and Hideyuki Tokuda. A challenge for reusing multiplayer online games without modifying binaries. In *Proceedings of the 4th ACM SIGCOMM workshop on Network and system support for games*, 2005.
- [61] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *Proceedings of the IEEE INFOCOM*, 2004.
- [62] Kotaku. How Much Has WoW Cost Blizzard Since 2004? [kotaku.com/5050300/how-much-has-wow-cost-blizzard-since-2004](http://kotaku.com/5050300/how-much-has-wow-cost-blizzard-since-2004), 2008.
- [63] Aleksandra Kovacevic, Sebastian Kaune, Patrick Mukherjee, Liebau Liebau, and Ralf Steinmetz. Benchmarking Platform for Peer-to-Peer Systems. *it - Information Technology (Methods and Applications of Informatics and Information Technology)*, 49(5):312–319, 2007.
- [64] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28, Issue: 9:690–691, 1979.
- [65] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
- [66] Emmanuel Lety, Laurent Gautier, and Christophe Diot. MiMaze, a 3D multi-player game on the internet. In *Proceedings of the 4th International Conference on Virtual System and MultiMedia*, 1998.
- [67] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communications Review*, 27(3), 1997.
- [68] Warren Matthews and Les Cottrell. The PingER project: Active Internet performance. Monitoring for the HENP community. *IEEE Communications Magazine*, 38(5):130–136, 2000.

- [69] Martin Mauve. Consistency in replicated continuous interactive media. In *Proceedings of the ACM conference on Computer supported cooperative work*, 2000.
- [70] Martin Mauve, Jürgen Vogel, Volker Hilt, and Wolfgang Effelsberg. Local-lag and Timewarp: Providing Consistency for Replicated Continuous Applications. *IEEE Transactions on Multimedia*, 6(1):47–57, 2004.
- [71] Sun Microsystems. JSR-223 Scripting for the Java Platform, 2006.
- [72] David L. Mills. RFC 1305 Network Time Protocol (Version 3) specification, implementation and analysis. Network Working Group Report, 1992.
- [73] Nelson Minar. A survey of the NTP network, 1999.
- [74] Shunsuke Mogaki, Masaru Kamada, Tatsuhiro Yonekura, Shusuke Okamoto, Yasuhiro Ohtaki, and Mamun Bin Ibne Reaz. Time-stamp service makes real-time gaming cheat-free. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, 2007.
- [75] Christian Mönch, Gisle Grimen, and Roger Midstraum. Protecting on-line games against cheating. In *Proceedings of the 5th ACM SIGCOMM workshop on Network and system support for games*, 2006.
- [76] Jessica Mulligan, Bridgette Petrovsky, Bridgette Patrovsky, and Raph Koster. *Developing Online Games: An Insider's Guide*. Pearson Education, 2003.
- [77] Napster Network. [en.wikipedia.org/wiki/Napster](http://en.wikipedia.org/wiki/Napster).
- [78] J.A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.
- [79] NPD Group. [www.npd.com/press/releases/press\\_080131b.html](http://www.npd.com/press/releases/press_080131b.html), 2008.
- [80] World of Warcraft Forums. [forums.worldofwarcraft.com](http://forums.worldofwarcraft.com), 2006.
- [81] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus: an architecture for extensible distributed systems. In *Proceedings of the 14th ACM symposium on Operating systems principles*, 1993.
- [82] Wladimir Palant, Carsten Griwodz, and Pål Halvorsen. Evaluating dead reckoning variations with a multi-player game simulator. In *Proceedings of the ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2006.
- [83] Lothar Pantel and Lars C. Wolf. On the impact of delay on real-time multiplayer games. In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, 2002.
- [84] Attila Pásztor and Darryl Veitch. PC based precision timing without GPS. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 1–10, New York, NY, USA, 2002. ACM.

- [85] PricewaterhouseCoopers Ltd. Global entertainment and media outlook 2008-2012, 2008.
- [86] Matt Pritchard. How to hurt the hackers: The scoop on internet cheating and how you can combat it. *Gamasutra*, 2000.
- [87] Paul Resnick, Ko Kuwabara, Richard Zeckhauser, and Eric Friedman. Reputation systems. *Communications of the ACM*, 43(12):45–48, 2000.
- [88] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.
- [89] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *ACM SIGOPS Operating Systems Review*, 35(5):188–201, 2001.
- [90] RTP. RFC 1889 - RTP: A Transport Protocol for Real-Time Applications, 1996.
- [91] James R. Semler. Common-view GPS time transfer accuracy and stability results. In *Position Location and Navigation Symposium*, 1990.
- [92] Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, 2004.
- [93] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer, 2005.
- [94] William R. Stevens. RFC 2001 TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, 1997.
- [95] Sun Microsystems Inc. Java Message Service (JMS) Specification Version 1.1, 2002.
- [96] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems - Principles and Paradigms*. Prentice-Hall, 2002.
- [97] TCG. Trusted Computing Group. [www.trustedcomputinggroup.org](http://www.trustedcomputinggroup.org), 2008.
- [98] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. BubbleStorm: Resilient, Probabilistic, and Exhaustive Peer-to-Peer Search. In *Proceedings of the ACM SIGCOMM Conference*, 2007.
- [99] Wesley W. Terpstra, Christof Leng, and Alejandro P. Buchmann. BubbleStorm: Analysis of Probabilistic Exhaustive Search in a Heterogeneous Peer-to-Peer System. Technical report, Technische Universität Darmstadt, 2007.
- [100] Times Online. [www.timesonline.co.uk/tol/news/uk/scotland/article3821838.ece](http://www.timesonline.co.uk/tol/news/uk/scotland/article3821838.ece), 2008.

- [101] Darryl Veitch, Satish Babu, and Attila Pásztor. Robust synchronization of software clocks across the internet. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.
- [102] Jürgen Vogel and Martin Mauve. Consistency control for distributed interactive media. In *Proceedings of the 9th ACM international conference on Multimedia*, 2001.
- [103] Steven Daniel Webb and Sieteng Soh. Cheating in networked computer games - review. In *Proceedings of the 2nd International Conference on Digital Interactive Media in Entertainment and Arts*, 2007.
- [104] Steven Daniel Webb and Sieteng Soh. Round length optimisation for P2P network gaming. In *Postgraduate Electrical Engineering and Computing Symposium*, 2007.
- [105] Steven Daniel Webb, Sieteng Soh, and William Lau. Enhanced mirrored servers for network games. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, 2007.
- [106] Steven Daniel Webb, Sieteng Soh, and William Lau. RACS: a referee anti-cheat scheme for P2P gaming. In *Proceedings of the 17th international workshop on Network and operating systems support for digital audio and video*, 2007.
- [107] Steven Daniel Webb, Sieteng Soh, and Jerry Trahan. Secure referee selection for fair and responsive peer-to-peer gaming. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, 2008.
- [108] Wikipedia. Bridge. [en.wikipedia.org/wiki/Contract\\_bridge](http://en.wikipedia.org/wiki/Contract_bridge), 2008.
- [109] Bruce Sterling Woodcock. An Analysis of MMOG Subscription Growth. [www.mmogchart.com](http://www.mmogchart.com), 2008.
- [110] Jeff Yan and Brian Randell. A systematic classification of cheating in online games. In *Proceedings of the 4th ACM SIGCOMM workshop on Network and system support for games*, 2005.