

On Correctness of Scalable Multi-server State Replication in Online Games

Jens Müller, Andreas Gössling, and Sergei Gorlatch

University of Münster, Germany

{jmueller|gorlatch}@math.uni-muenster.de

ABSTRACT

Massively Multiplayer Online Games (MMOG) require novel, scalable network architectures for a high amount of participating players in huge game worlds. Consequently, new and complex multi-server parallelization approaches have been proposed to provide responsive, massively multiplayer gameplay for different game genres. Besides scalability and performance, the issue of correctness of the game state processing is vital for providing a failure-free gameplay as expected by the users. In this paper, we first introduce the concept of correctness for multi-server replication architectures as the ability to preserve the order of user inputs in the virtual processing. We then present two correctness mechanisms optimized for multi-server replication: pessimistic lag and optimistic timewarp. We experimentally show that by implementing the lag mechanism for correctness in our multi-server implementation of the QFusion/Quake2-game the amount of incorrectly ordered actions can be reduced from 50 % to 10 %.

1. INTRODUCTION AND MOTIVATION

Multiplayer real-time games have become an important part of a broad entertainment market. There is a huge variety of gamestyles, from competitive action games to collaborative role-playing games. All these games are distributed applications, in which users issue non-deterministic inputs like moving, firing a weapon or healing each other. Game applications have to handle these user inputs in real-time, i.e., within a certain, game-dependent time interval in order to provide responsive gameplay.

A critical aspect in real-time games is the *correctness* of user action processing: it is expected by the users that their actions are observed in the virtual game world in the same order in which they actually have been initiated. Correctness, therefore, is the property of the distributed game architecture to preserve the real-world order in the processing of user inputs in the virtual world. The major obstacle for achieving the correctly ordered processing of user actions,

especially in Internet-based sessions, is delay and jitter in the network communication. In order to guarantee the correct processing order in conventional single-server network architectures, there have been different correctness mechanisms presented. In particular, pessimistic delay mechanisms like *Local Lag* as well as optimistic repair mechanisms like *Timewarp* [7] have been discussed and actually implemented in single-server games, for example in the game *Half Life* [3]. However, correctness in scalable multi-server network architectures suitable for *Massively Multiplayer Online Games (MMOG)* poses a much stronger challenge due to additional inter-server communication and synchronization and has only been addressed little so far.

This paper addresses the correctness issues in multi-server networks which replicate the game state among servers in order to support large numbers of participating players in *Massively Multiplayer Online Games (MMOG)*. While *Massively Multiplayer Online Role-Playing Games (MMORPG)* are the main MMOG genre, other game types like *First Person Shooters (FPS)* can be scaled to the massively multiplayer realm as well, as for example in *Planetside* [15] or the announced and much anticipated games *Tabula Rasa* [11] and *Huxley* [18]. In order to scale game sessions for MMORPG or FPS games, the game processing has to be effectively parallelized using the participating servers. There are two main concepts of game parallelization: The *zoning concept* for MMORPG partitions the huge game world, while the *replication concept* distributes copies of the game state onto different servers. Regarding correctness, the zoning concept does not differ from the conventional single-server processing, because for each zone, there is still only a single server involved in the processing. The replication approach, in contrast, poses a much stronger challenge for ensuring correctly ordered action processing, because interdependent user actions are processed at possibly different servers.

The paper specifically deals with correctness in the replication approach for multi-server online games. We briefly discuss the scalability concepts for online games using multiple servers, and present our multi-server replication approach using the *eventual consistency* model and the *proxy-server network architecture*. As the main contribution, we define the notion of correctness in our replication approach and present two novel correctness mechanisms based on the general pessimistic artificial lag and optimistic timewarp approaches. We evaluate the characteristics of our new mechanisms and report experimental results of implementing the

pessimistic lag mechanism into our replication-based, multi-server implementation of the QFusion/Quake2-engine.

2. SCALING ONLINE GAMES USING MULTIPLE SERVERS AND REPLICATION

Real-time multiplayer games are in fact soft real-time systems, in which the game state is updated at a high frequency. The actual frequency usually ranges from about 5 to 35 updates per second depending on the game type. The processing components of the distributed game system, i.e., the processes calculating a new state from user inputs and the old state, have to finish the state computation in the time given for a single update as the reciprocal value of the update frequency. If the processing components are not able to finish the calculations in time, then the timing constraints of the real-time system are violated and the game application becomes unresponsive, stalls and is not playable any more.

2.1 Scalability

The required computations for a single game state update grow with an increasing number of game entities which are processed in the update. Therefore, more users in a game session require more computation resources, because each user controls at least one game entity. For multiplayer online games, three different types of *scalability dimensions* depending on user behaviour and game design can be distinguished as we discussed in detail in [10]:

- 1 Scalability of *total player numbers*
- 2 Scalability of *game world size*
- 3 Scalability of *player density*

The conclusion from our discussion of scalability dimensions is that the “horizontal” parallelization of the commonly in MMORPG used *zoning approach* scales the total game world size and the total player numbers (dimensions 1 and 2). However, zoning is not able to scale the third scalability dimension, i.e., the density of players. It does not scale the density because there is always only a single server and therefore a fixed amount of processing power available for a particular region of the virtual game world, in which a high number of users might come together. Especially action games, in which users tend to go where the action is, require a parallelization approach with the ability to scale the density of players. An alternative parallelization which actually scales the density of players is the *replication approach*. The replication approach, which “vertically” parallelizes game computation over several servers is able to increase the available processing power for a fixed region of the game world. Replication is, therefore, a suitable mechanism for action games which can scale the density of players in huge fights.

The replication approach increases the available processing power for a particular region of the game world, because several servers vertically overlap their regions to be computed. However, this overlap of computation regions requires servers to synchronize their game states in order to

fulfill a certain level of consistency. In general, there are different possible synchronization mechanisms and consistency models. Another important issue resulting in a variety of different possible approaches is, what actual data should be replicated and how. There already have been different flavours of replication concepts evaluated and presented. For example, the *Colyseus* architecture [4] creates replicas depending on the game situation dynamically, while our own *Proxy-Server replication* approach [8] replicates all dynamic data directly at the beginning, to avoid setup times. We will summarize the synchronization and consistency of our replication approach in the following, before we, as the novel contribution concerning our approach in this paper, discuss mechanisms to ensure *processing correctness* in Section 3.

2.2 Replication and Eventual Consistency

In our replication approach, each participating game server holds a full copy of the complete game state. For each dynamic game entity like a player avatar, a server-controlled NPC (Non-Player character) or a passive game item (e.g., a health pack) there is one dedicated server which has write access and therefore “owns” that *active entity*. All other servers have a passive replicate *shadow* copy of that entity and update its state according to synchronization messages sent by the owning server, as illustrated in Fig. 1.

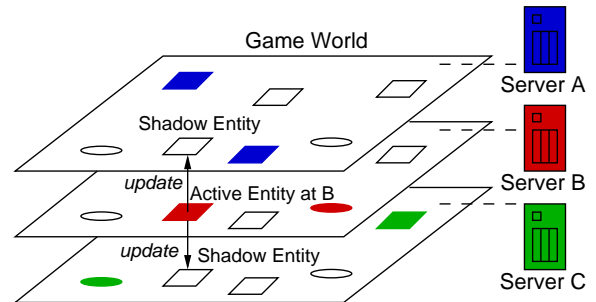


Figure 1: Our Full Replication Approach

The write-access distribution among servers as active entities and the synchronization messages implement the *eventual consistency* [17] model, which means that:

- Only one process is allowed to write per entity
- Owner process sends asynchronous update message to other processes on entity changes

This way, each replicate shadow entity copy at other processes will become updated when the asynchronous update message eventually arrives.

There are alternative, more strong consistency models which could be used in the replication approach. However, stronger models usually allow several processes to write to data and therefore require a mechanism to ensure mutual exclusion of write access. Using pessimistic locking would drastically decrease the game responsiveness, because servers would be required to communicate for establishing a distributed lock before user action could actually be processed and acknowledged. An alternative would be to implement an optimistic

mechanism which only becomes active if a write conflict actually occurred. However, such a post mortem recovery of consistency requires the game to undo state updates which might be already presented to users: An item which was picked up just half a second ago would have to be taken away again from the player or, even worse, an opponent which just has been beaten in a close fight would suddenly stand up again and continue to fight. Optimistic concurrency control, therefore, is not feasible for most game situations: the undo of already presented, game-decisive actions would be tremendously disturbing and game-breaking for users.

The eventual consistency used in our replication approach avoids the drawbacks of optimistic and pessimistic concurrency control mechanisms by allowing only a single server to have the write access for a particular entity. Our approach results in lower processing flexibility because not all processes are allowed to write to game state data. However, high flexibility is not required in our replication approach because the static write access for entities can be distributed in a load-balanced way among servers. This way, our replication approach scales the density of players very well, as briefly discussed for our two demonstrator games *Rokkatan* and *QFusion/Quake 2* in the next section covering our actual architecture implementation. Overall, from the different alternatives of concurrency control, our eventual consistency approach is a good compromise in the trade-off between the degree of state consistency and the responsiveness of the game.

2.3 The Proxy-Server Topology

We developed the proxy-server topology [8] as a multi-server network architecture for implementing our replication approach. Each participating server holds a full copy of the game state, for each game entity one of the servers is allowed to write to. The servers are called *proxies*, because in typical Internet-based game sessions, the servers can and should be placed at different Internet service providers (ISP), such that clients can connect to the “nearest” server in terms of communication latency as illustrated in Fig. 2.

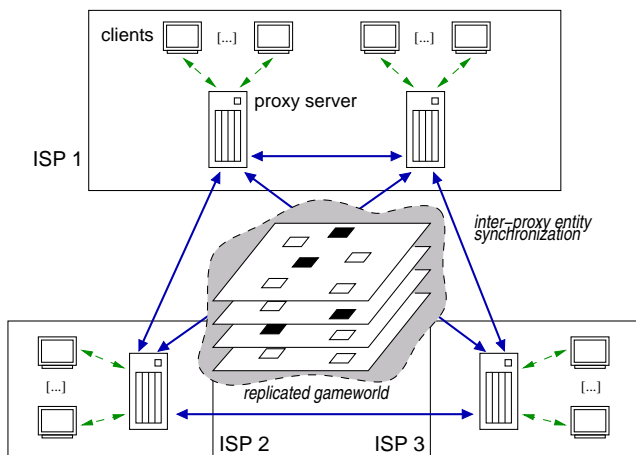


Figure 2: The Proxy-Server Architecture

The proxy servers are connected in a peer-to-peer manner. If one proxy updates a local active entity, it will send an update message to all other participating servers, which then

update their local shadow replicate copies accordingly. For the actual distribution of which entities are active on which proxies, we distinguish two kinds of game entities:

1. *User-dependent entities*: These are entities which are directly associated with a particular client. These entities are usually the avatars controlled by the user at a client, as well as status and game score information corresponding to that client.
2. *User-independent entities*: These entities are not directly associated with a particular client; they are game objects, with which users can interact without directly controlling them. Examples are non-player characters (NPCs), or items like health potions, ammunition packs, etc. which can be picked up by users.

We implemented two different games on top of the proxy-server architecture incorporating our full replication approach with eventual consistency. The first game is *Rokkatan* [9] – a massively multiplayer strategy game. Each player in *Rokkatan* controls a single avatar and fights other players to occupy flags placed in a comparatively small game world. Players are organized in teams which receive score points from occupied flags. This design results in tight collaboration between team members and huge fights at flag points. A screenshot of such a big fight in *Rokkatan* is depicted in Fig. 3. *Rokkatan* runs at a high update rate of 25 updates per second and provides a very responsive gameplay. The proxy-server architecture allows to scale the number of concurrent players in *Rokkatan* up to 500 players and more (using ten replicating proxy servers) with a very high player density on a small gamemap.



Figure 3: Game Screenshot of Rokkatan

Our second demonstrator game is the *QFusion Proxy-Architecture (QPA)* [10], a ported version of the *QFusion-engine* [12] which bases on *Quake 2* [14] from ID Software. Our *QPA* game is a fast-paced FPS game which uses the proxy-server architecture to scale the total number of participating players as well as the player density. The screenshot in Fig 4 shows a large fight of many players clustered together: this game situation could only be run responsively by using several proxy servers. As discussed in [10] in detail, the

original QFusion single-server implementation is only able to maintain the update rate of 10 updates per second and therefore the game’s real-time requirements for a maximum of 30 players when using a Pentium 4 1.7 GHz server host. Using four proxy servers for a single sessions, our QFusion Proxy-Architecture port allows more than 80 players to participate in a still responsive game.



Figure 4: Game Screenshot of QFusion

As the main contribution of this paper, we will discuss in the following the correctness of our replication approach using the proxy-server topology. Additionally, we will present an experimental implementation of a Local Lag algorithm in the QFusion Proxy-Architecture game for improving processing correctness in our proxy server approach.

3. CORRECTNESS: CONCEPTS AND MECHANISMS

We define the *processing correctness* of a distributed, multi-player real-time game as the property to preserve the real-world order in which actions have been issued by users in the game state computation. This general definition is independent of the actual network architecture used in a multiplayer game. We denote by $t_r(a)$ the point in time at which action a is issued by a user by clicking the mouse button or pressing a key with. Subsequently, the real-world order of two actions issued by different users a_1 and a_2 can be expressed as the relation $(a_1, a_2)_r \Leftrightarrow t_r(a_1) - t_r(a_2) < 0$, i.e., a_1 was issued before a_2 .

In the subsequent processing of an action issued by pressing a button or a key, the client game application has to process the key event and to construct and to send a corresponding network message to some server process. The server process then might process the action itself (with possibly waiting for a distributed write lock to relevant data), or, depending on the actual network architecture, forward the action to another server process. In any case, there is a point in time, $t_v(a)$, at which the action a is fully processed and its effects on the game state (health points of an opponent reduced in case of a weapon firing, new position of player character in case of a movement command, etc) are made permanent. Due to the required time for initial input event processing at the client and possible sending of network messages, it is safe to assume that the point in time at which an action a

is finally processed in the virtual game world is always some time after the initial real-world pressing of a key by the user, thus $t_v(a) > t_r(a)$. Analogously to the real-world order of actions, the order in which two actions a_1 and a_2 issued by different users are processed and permanently reflected in the state of the virtual game world can be defined as the relation $(a_1, a_2)_v \Leftrightarrow t_v(a_1) - t_v(a_2) < 0$, i.e., a_1 was processed before a_2 .

Based on the relations of real-world issuing and virtual world processing of user actions, we define the processing of two user actions a_1 and a_2 to be *correct in order*, iff $(a_1, a_2)_r \Rightarrow (a_1, a_2)_v$. In other words, the processing of these two actions reflects the actual order in which they were issued by users at different clients, as illustrated in Fig. 5(a) for a conventional single-server architecture. Consequently, the complete processing of a game is (*order*) *correct*, iff the pairwise issuing orders of all actions a_i, a_j are preserved by the game simulation, i.e., $(a_i, a_j)_r \Rightarrow (a_i, a_j)_v, i \neq j$.

Besides our definition of order correctness, other and also stronger correctness criteria can be defined basing on the points in time when actions are issued and processed. The strongest definition of correctness obviously is an *immediate processing* of all user actions a at the time they are issued, i.e., $t_v(a) = t_r(a)$. However, such an immediate processing can not be achieved in actual systems, because there are always latencies for communicating and synchronizing actions. Another definition of correctness we can give using our basic operations is *time correctness*: Not only the order, but also the distance in time between two actions a_1 and a_2 has to be preserved in the virtual world processing, i.e., $t_r(a_1) - t_r(a_2) = t_v(a_1) - t_v(a_2)$. However, such correctness requires QoS-enabled communication (using ATM, for example) and a very accurate clock synchronization of all hosts, which generally can not be achieved using the best-effort communication of the Internet. In the following, we therefore use the definition of order correctness for the discussion of distributed real-time processing and show that this correctness already provides an adequate degree of correctness in collaborative and competitive game situations.

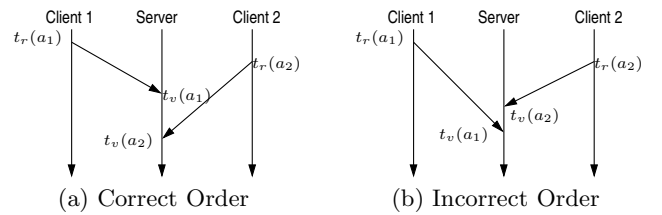


Figure 5: Correct and Incorrect Game Processing in the Single-Server Case

The main difficulties in guaranteeing order correctness are different network delays of the participating client and server processes. The time between real-world issuing and virtual world processing of two actions a_1 and a_2 , $t_v(a) - t_r(a)$, can be so different due to the first action delayed in the network, that the virtual processing order is the other way round than the real-world issuing order. Such a changed order of two interdependent user actions, as illustrated in Fig. 5(b), is an *incorrect* game processing and can affect the outcome of certain game situations in a decisive way.

As an example how incorrect game processing can decide the outcome of a particular situation or even a complete match, let us imagine a shootout between two players in an FPS. The first user aims at the other player with an “instant-hit” weapon and fires, while the other user anticipates this weapon firing and moves to evade the hit just a moment before the other user actually fires the weapon. If the virtual processing does not maintain the order of these user actions, the second player may be shot, although he moved out of the way before the firing of the first player. As another example, imagine two users standing next to each other with their players and concurrently trying to pick up a special and important item by actively clicking on it. This situation commonly occurs in RTS game matches, where hero units can carry items, or in MMORPG, where treasure chests can be opened or item can be looted from slain monsters. If the virtual processing of such two item pickup-actions changes the real-world issuing order, the wrong player receives the item.

3.1 Correctness in the Proxy Architecture

The active/shadow entity concept of our proxy-server architecture requires to distinguish between two main kinds of user inputs: The first input type are *independent actions* which only alter the state of entities associated with the user, like moving the own avatar, for example. The second input type are *interactions*, which possibly change the state of remote entities, like shooting another avatar with a weapon or casting a friendly healing spell on teammates. The following paragraphs explain how these two different user input types are handled in our proxy-server architecture and discuss the potential correctness problem resulting from their processing.

3.1.1 Independent Actions

Independent user actions only alter the state of the users’ own entities and characters and do not influence the state of other entities. Examples for such user actions are moving an own character or using a health-potion to restore the own health points. These actions can directly be executed at the proxy server the user client is connected to, because the server has write-access to the relevant, user-associated active entities. Fig. 6 depicts a independent movement action and the resulting processing in the proxy server architecture.

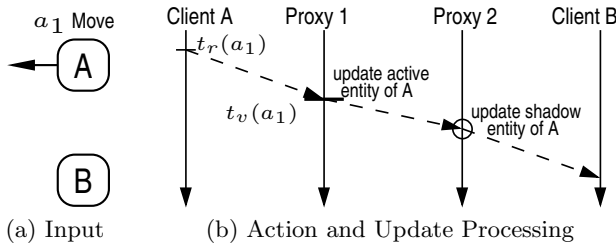


Figure 6: Independent Actions

The game situation example in Fig. 6(a) shows two avatars, A and B, of different users in a game situation. The user controlling avatar A issues an independent movement action a_1 at the time $t_r(a_1)$, while the user controlling B does nothing and just observes the movement of A. The associated processing depicted in Fig. 6(b) illustrates that the

client A is connected to Proxy 1, while client B is connected to a different server, Proxy 2. Proxy 1 receives the movement action a_1 from client A and can immediately execute it, which results in a new position state after the movement of the active avatar entity. This execution at Proxy 1 marks the point in time $t_v(a_1)$ when the action is finally included in the new game state. Furthermore, Proxy 1 forwards this new state of the avatar A to all other connected proxies, such that Proxy 2 receives the new position and updates its shadow copy of the avatar A accordingly. From the view of client A, its *movement action is processed locally* by the local Proxy 1. Finally, Proxy 2 sends the new position to avatar A to client B, because the entity A resides in B’s field of view.

3.1.2 Interactions

Interactions not only affect the state of the own entities and avatars, but also of other entities possibly associated with other users and maintained at other servers. For example, this can be firing a weapon at an enemy, healing another character or picking up an item. The proxy server, which receives such an interaction command from a local client, forwards the interaction to the other servers, which in turn process the interaction if one of their local active entities is the interaction target. Fig. 7 gives an example of an interaction and its processing in the proxy-server architecture.

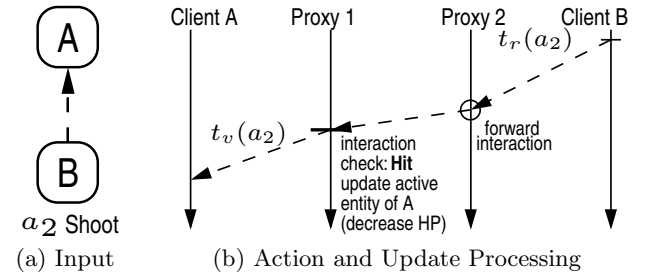


Figure 7: Interactions

In this interaction game situation, the user at client B issues a shoot-action a_2 at the time $t_r(a_2)$ towards the avatar of A as illustrated in Fig. 7(a). In the processing of this weapon firing shown in Fig. 7(b), Proxy 2 receives the interaction from client B, but can not immediately process the interaction because the target entity A is maintained at Proxy 1. The Proxy 2, therefore, forwards the interaction to all other proxies. Proxy 1 receives the weapon-firing and recognizes that avatar A should be hit by the attack. Therefore, the shot finally is executed by Proxy 1 at $t_v(a_2)$ decreasing the health points of the avatar A. From the view of client B, its *interaction is processed remotely* at the distant Proxy 1.

3.1.3 Interdependent Actions and Interactions

Each processing of independent actions and interactions in the proxy-server architecture results in a consistent game state. However, due to the two different processing locations (distant proxy for interactions, local proxy for actions), interdependent actions and interactions issued only a few milliseconds after each other might be processed incorrectly, i.e., in a different order from their real-world issuing order. Fig. 8 illustrates this problem by showing the combined situation from the two examples above.

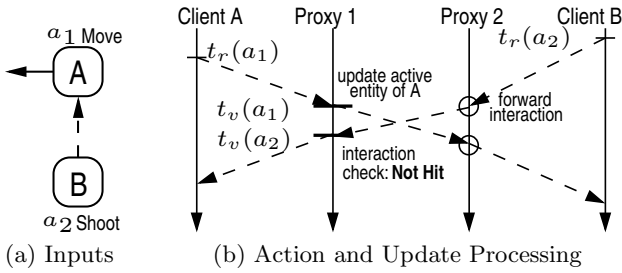


Figure 8: Concurrent Actions and Interactions

Fig. 8(a) shows the combined game situation of avatar A moving (user action a_1), and avatar B shooting at A (user action a_2). Let us assume that both actions are issued with only a little difference in time and the shot was issued just before the movement, i.e., $t_r(a_2) < t_r(a_1)$. In the processing depicted in Fig. 8(b), however, the movement command is incorrectly executed before the shot interaction: $t_v(a_2) > t_v(a_1)$. The combined communication latency of the forwarding of the shot interaction a_2 via client-proxy and inter-proxy communication leads to the situation that a_2 is later received at Proxy 1 than the movement action a_1 . Therefore, the user A is able to dodge the hit of B, although B shot first. Especially in competitive games this incorrect processing is not acceptable, because the wrong execution order discriminates users and might even decide the outcome of the game.

3.2 The Lag Mechanism

The main reason for the possibly wrong order in which user actions are executed in multiplayer game network architectures is that incoming actions are immediately executed. In order to overcome this problem and to maintain the real-world issuing order, a concept of introducing additional wait times for actions that have been sent faster through the network than slower ones has been applied in different fashions in previous work. For example, Mauve et al. discuss *Local Lag* [7] as a mechanism in which every peer, which can actively alter the game state, waits some time before locally issued actions are executed to give remote actions time to travel in the network. In single server game architectures, several games, like for example the shooter game *Unreal Tournament* [16], implement an optional lag mechanism in which the single server delays the execution of actions received over faster connections to provide equal communication latency for all clients. Generally, this approach is pessimistic, because it ensures the correct action order by letting each execution wait until the possible slowest action has been received and executed before.

For a multi-server replication network architecture it is even more important to implement mechanisms to ensure the correct processing order: interdependent user actions might be concurrently processed at different servers and therefore are much more prone to be executed in the wrong order than in single-server architectures. Such a mechanism is not only required by our particular proxy-server architecture, but for multi-server replication and parallelization architectures in general. In the following, we discuss the lag mechanism we designed for the proxy-server architecture in detail. How-

ever, the underlying correctness problem of concurrent distributed processing of interdependent user actions is general for the replication approach, such that the presented mechanism should apply with only little adjustments to other replication architectures as well.

As discussed in the previous section, there are two different types of user inputs, *independent actions* and *Interactions* in the proxy-server architecture, which have to be treated differently in our lag mechanism to ensure order correctness. The general idea is that proxies have to postpone the processing of an independent action received by a local client some time. This time has to be that long that interactions of remote clients which were issued before have enough time to be forwarded and to travel the network to be actually processed before.

In order to calculate the actual length of time a local independent action has to be delayed, a proxy needs to know the maximum communication latency for a remote interaction. To simplify the discussion, we denote the communication latency between processes A and B with $l(A, B)$. Fig. 9 illustrates the lag mechanism in our proxy-server architecture for on the previous example depicted in Fig. 8 of two inter-dependending movement action and weapon firing interaction commands. Client A sends an independent action a_1 to its

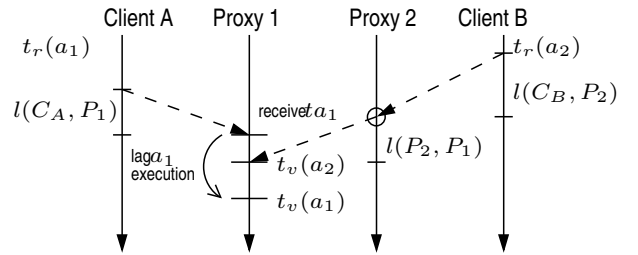


Figure 9: Delaying Independent Actions

Proxy 1, the message takes $l(C_A, P_1)$ to travel the network. Instead of immediately processing the action a_1 , proxy 1 calculates a delay for the action processing of $\max(l(C_x, P_y) + l(P_y, P_z)) - l(C_A, P_1)$, with x, y running through all connected clients and z, y through all proxies, respectively. In our example, this delay is $l(C_B, P_2) + l(P_2, P_1) - l(C_A, P_1)$, which compensates for the longer transmission delay for the not yet received interaction a_2 actually issued before a_1 , i.e., $t_r(a_2) < t_r(a_1)$. Proxy 2 immediately forwards the interaction a_2 to Proxy 1 which processes it as soon as it is received. The delayed action a_1 is executed after a_2 's processing: $t_v(a_2) < t_v(a_1)$, such that the order correctness is ensured.

The described delay mechanism conceptually even ensures time correctness, because it preserves the distance in time between the real-world issuing and the virtual execution of the actions as $t_r(a_1) - t_r(a_2) = t_v(a_1) - t_v(a_2)$. However, in a practical implementation this distance in time usually will not be preserved: the processing in game servers usually takes place at discrete points in time (once per tick) which may provide a small additional delay before the actual execution. Additionally, it is not possible to precisely obtain the communication latencies in actual implementa-

tions, which is problematic for ensuring order correctness as well. Especially communication with high jitter (variance of latency) may result in wrong execution orders even with the lag mechanism applied. However, such poor communication quality usually results in poor game performance and responsiveness anyways, due to the violation of the real-time requirements. In usual Internet-setups which provide good game experience, our proxy lag mechanism ensures order correctness as long as the difference in time between two actions is larger than the jitter of the network communication.

While the delaying of the execution of local actions results in order correctness for concurrent actions and interactions, two interactions issued at different clients still race for execution order. Imagine a new game situation depicted in Fig. 10, with two clients B and C which both interact with an entity E maintained at a remote proxy. That remote entity (which is passive in this situation for the sake of simplicity) could be another avatar both users heal or shoot at, or just an item that can be picked up. If the real-world order of these interactions is not maintained, the wrong user will get the credit for the successful interaction.

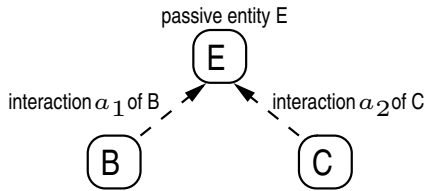


Figure 10: Situation of Competing Interactions

Let us assume for this example that client B issues its interaction a_1 shortly before C issues a_2 , such that $t_r(a_1) < t_r(a_2)$. Each client is connected to a different proxy, and each of these proxies has to forward the interaction to the proxy at which the target entity E is maintained. It is obvious that the actual processing order depends on the communication latencies between all these processes and that a_2 issued by C might overtake a_1 in the network, thus being incorrectly executed first. The delay of local actions presented above does not help here, because no local actions are involved. We therefore designed an additional *remote lag* mechanism for interactions at the processing proxy as depicted in Fig. 11 for this example.

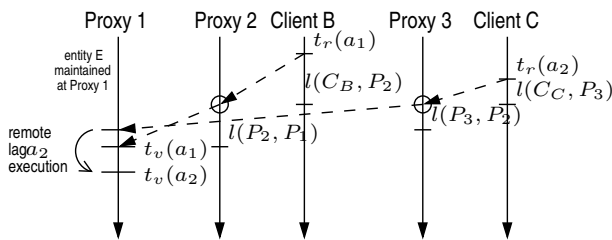


Figure 11: Delaying Competing Interactions with Remote Lag

The new *remote lag* mechanism delays the execution of interactions at the remote proxy server. In the example, the forwarding of action a_1 , which has been issued before a_2 , takes longer than the forwarding of action a_2 , such that

an immediate execution of both actions at proxy 1 would result in an incorrect execution order. Therefore, proxy 1 as the remote server processing both interactions remotely lags the execution of a_2 by $\max(l(C_x, P_y) + l(P_y, P_z)) - l(C_C, P_3) + l(P_3, P_1)$. The maximum forwarding latency in this example is the latency of forwarding the actions of B ($l(C_B, P_2) + l(P_2, P_1)$), such that the remote delay for the late interaction a_1 at proxy is actually 0. As the result, the execution of a_2 is delayed after the execution of a_1 , resulting in the correct execution order.

The presented lag mechanism for the proxy-server architecture consists of two different delay types: Local independent actions, on the one hand, have to be delayed by the local proxy maintaining the associated active entity. Interactions with other remote entities, on the other hand, have to be immediately forwarded by the local proxy and to be remotely lagged at the distant executing server. Both mechanisms delay user action execution to the maximum two-staged client-proxy/proxy-proxy latency of $\max(l(C_x, P_y) + l(P_y, P_z))$. This way, the complete proxy-network is pessimistically delayed to the level of the slowest connection and order correctness is ensured. Unfortunately, a massive artificial lag results in decreased responsiveness of the game. There is a general trade-off between responsiveness of the virtual world simulation and the processing correctness achieved by a pessimistic artificial lag mechanism. The workaround for this trade-off is to statically limit the maximum delay as $\min(\max(l(C_x, P_y) + l(P_y, P_z)), \text{maxdelay})$ such that the game still provides adequate responsiveness. The maximum amount of *maxdelay* has to be fine-tuned to the specific characteristics of the game genre and the actual game implementation. For Real-Time Strategy (RTS) games like *Warcraft 3*, for example, round-trip delays of several hundreds of milliseconds are tolerable [13], while this delay in fast-paces FPS games like *Quake 3* or *Unreal Tournament 2003* should not exceed 100 to 140 ms [1, 2]. Additionally, the maximum lag value of *maxdelay* avoids that a single very slow connection, which even could be actively slowed down by a malicious user to sabotage the game session, renders the game unplayable for all other participants.

3.3 The Timewarp Mechanism

Besides the pessimistic lag mechanism, an optimistic *timewarp* concept has been discussed for and implemented in different game architectures. Using timewarp, the processes responsible for calculating the game state immediately process incoming actions and hold a backlog of old states and already processed actions. If a process receives a user action which should have been executed before others which are already processed, then the process performs a rollback to the state just before the newly received action should be processed. The process executes the new action and the other already received actions again from the backlog and this way properly inserts the new action into the overall state processing sequence. Timewarp was generally discussed by Mauve et al. in [7] and is implemented for example in the game *Half-Life* [3] and therefore in the very popular FPS game *Counterstrike* which builds on top of *Half-Life*.

In our timewarp approach for the proxy-server architecture, each proxy holds a backlog of previous game states and processed user inputs. Each time a proxy receives a new action

a that has to be processed (both local independent actions or forwarded interactions received from other servers), it checks whether other actions a_i which have been issued after a , i.e., $t_r(a) < t_r(a_i)$ have already been processed. If so, the proxy performs a timewarp restoring the old state which was valid at $t_r(a)$, then it processes a and applies the other actions a_i again on the new starting state now incorporating a .

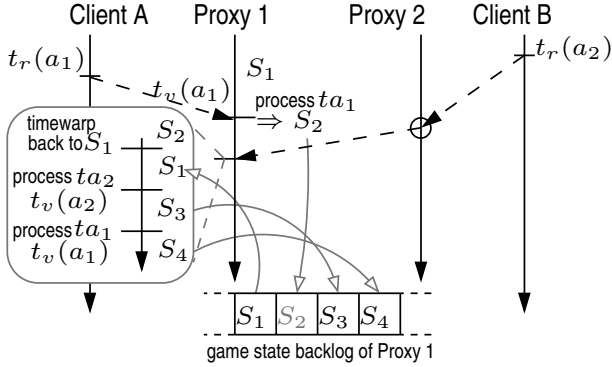


Figure 12: The Timewarp Mechanism

Let us reconsider the initial example of the movement of a user a_1 shortly after the firing of a weapon a_2 by a remote user, illustrated in Fig. 8(a). Proxy 1 receives the local independent action a_1 in state S_1 and immediately processes it resulting in the new state S_2 . Fig. 12 illustrates the game state backlog of Proxy 1 and the performed timewarp. Immediately after Proxy 1 receives a_2 , the state S_2 is recognized as being incorrect and S_1 is restored from the backlog. The server now processes a_2 before a_1 , producing the state S_4 , which now incorporates the correct execution order of the actions. However, there is still the problem that the incorrect intermediate state S_2 has been already broadcast to all other proxy servers.

We designed a new *undo* mechanism for active entity updates in the proxy-server architecture when using timewarp in order to overcome the problem of invalid states being broadcast among the proxy servers. Each time a proxy invalidates a state S_{old} sent to other proxies because it warps back to an old state and executes a different action sequence, it sends an undo-message for S_{old} to all other proxy servers. Such an undo-message can result in a timewarp at the other proxies: Imagine a situation where two users, A and B, competed to pick up a certain item (A issued the pickup slightly before B) and the responsible proxy of that item correctly gave it to user A. However, a third user C interacted with A before A's pickup action in a hostile way (e.g., shooting at A) such that the proxy of A has to timewarp to send an undo-message regarding A's pickup action. The server of the item now has to timewarp as well, sending an undo-message and now will give the item to user B. The undo-messages can cascade this way and decrease the scalability of the overall architecture, because the same actions have to be processed several times until a state becomes stable. The suitability of the timewarp concept depends on how much warps and undo-operations have to be performed and, therefore, on the game design, the game implementation and the actual latencies in a particular session setup.

The general problem of the timewarp mechanism is that there may be some user actions which can be made undone. This problem corresponds to the problem of optimistic consistency mechanisms as briefly discussed in Sect. 2.2. If certain game situations have already been presented to users, these can not be taken back without massively disturbing the flow of the game. If a user picks up an item visually confirmed and possibly accompanied with a corresponding sound effect, the user then expects the item not to be suddenly taken out again out of his inventory. Imagine that a healer in a collaborative fight against a computer-controlled boss visually confirmed casts a healing spell on someone getting attacked. The user expects that the amount healed is not taken back from the avatar healed. It is up to the game developer to game-specifically decide which incorrect action processing can be taken back and which not.

3.4 Combining Lag and Timewarp

Each of the approaches has some advantages and drawbacks: The pessimistic delaying, on the one hand, decreases the responsiveness of the game, but does not require additional computation power and communication bandwidth to undo invalid intermediate states. The optimistic timewarp, on the other hand, maintains the responsiveness especially for independent local actions like moving, but requires complex undo-messaging for certain warp-situations.

The combined use of the pessimistic lag and the optimistic warp concept, as already proposed in [7] for the general algorithms, is a promising approach for the proxy-server architecture as well. The lag mechanism ensures order correctness for user actions with only little difference in the points in time at which they were issued. The optimistic lag then afterwards repairs order incorrectness resulting from single delayed messages in networks with high jitter. Such a delay of single messages should not happen often compared to the total number of messages in session setups suitable for fast paced gaming, such that cascading undo-messages should not happen frequently. The exact parameters of these mechanisms like the maximum pessimistic lag delay or the size of the timewarp state backlog have to be fine-tuned for each specific game implementation. In order to quantitatively analyze the order correctness problem in the proxy-server architecture and the performance of our pessimistic lag mechanism, we designed a special test network architecture and implemented the lag mechanism into our *Quake2/QFusion* demonstrator FPS, as discussed in the next section.

4. EXPERIMENTAL IMPLEMENTATION AND QUANTITATIVE ANALYSIS

We implemented the pessimistic lag mechanism described in the previous section in our already existing proxy-server implementation of the FPS *QFusion* [12]. The *QFusion-Engine* is an enhanced version of the well-known FPS *Quake 2* [14], which has been released as open source by ID Software in 2001.

4.1 Implementation of the Lag Mechanism

In the implementation of the lag mechanism, each proxy server puts incoming user actions into a queue instead of immediately processing them. The queue is checked at each tick, which is 10 ms long, whether actions have waited long

enough to be processed. In comparison to the general concept in which actions are executed at the very moment there are due, the actual implementation with ticks results in a periodic discretized check of the queue. The periodic check of user action execution times, although running at a very high frequency of 100 checks per second, might add an additional delay of up to 10 ms in the worst case to the scheduled execution time of a particular action.

The proxy servers have to know all latencies between processes in the complete session in order to calculate the specific delay for a particular action. The servers frequently measure the latency to their clients and to other servers using special heartbeat-messages and broadcast the values to each other. This way, each proxy knows the approximate communication latency between all processes in the game session. Furthermore, for a received action a , a proxy needs to know the time $t_r(a)$ at which this action has been issued by the user. In our experimental implementation, we use a global session time which is periodically synchronized to identify $t_r(a)$ at clients and $t_v(a)$ at proxy servers, respectively. Another possibility would be to approximately calculate $t_r(a)$ at a proxy when a is received basing on the latency from that particular client. While this calculation at a proxy may be more inaccurate than the direct timestamp at a client especially with noticeable jitter, it avoids the problem of malicious clients purposely sending wrong timestamp information to sabotage the lag mechanism.

4.2 Experimental Setup

In order to be able to analyze the processing order of actions at runtime, we enriched the experimental proxy-server setup by adding an additional control server each process is connected to. The basic setup consists of two proxy servers and two clients, one connected to each proxy. The clients and the proxy servers run the QFusion game session in the usual way and, for the purpose of this experimental setup, also send action and time information to the control server. The overall setup is illustrated in Fig. 13.

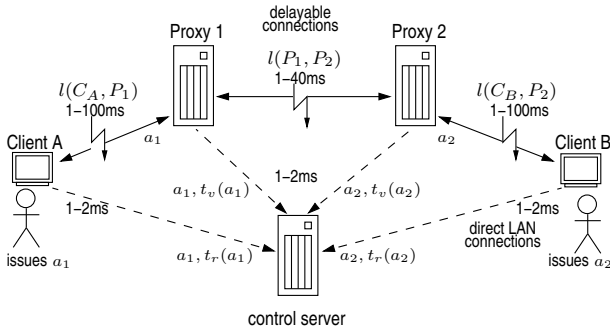


Figure 13: Experimental Setup

We developed the novel control server to actually investigate correctness issues in the proxy-server architecture and to quantitatively analyze the performance of the lag mechanism. The control server receives information about a and the real-world issuing timestamp $t_r(a)$ directly from the corresponding client. Additionally, the proxy servers send the timestamp $t_v(a)$ at which the action a actually has been processed to the control server, which, therefore, can compare the real-world issuing order of two actions a_1 and a_2

with their virtual processing order. In this test setup, we ran client-proxy and inter-proxy communication over routers using the NIST Net¹ network emulation software to delay communication and this way to induce different latencies. Altogether, the setup allowed us to analyze correctness issues and the performance of the lag mechanisms in a wide variety of Internet-like latency setups by just playing the game at two clients; the control server monitored all actions issued and processed in the sessions and continuously compared their real-world and virtual processing order.

4.3 Results of the Experiments

For a single experiment, we let two users play a competitive one-vs-one game of QFusion for five minutes. We ran a variety of experiments, each with a varied latency of the client-proxy and inter-proxy communication. The latencies between the clients and their relative proxy $l(C_A, P_1)$ and $l(C_B, P_2)$ varied from 0 to 100 ms, while the inter-proxy latency $l(P_1, P_2)$ varied from 0 to 40 ms. In each single experiment of five minutes length, from 30.000 to about 50.000 user actions were generated and processed. There were so many user actions, because already holding down a single movement key at one client generates 100 movement actions per second, i.e., one per tick. The control server continuously received these actions and their timestamps and compared the real-world issuing and virtual processing orders of subsequent actions. Even independent actions which do not interfere with each other, as for example independent movements of the players in different regions of the virtual game map, were compared. Table 1 shows a selection of more than 30 setups with different latencies that were tested.

	Latencies $l(\dots)$			Inc. Orders w/o Lag	Inc. Orders with Lag
	C_A, P_1	P_1, P_2	C_B, P_2		
1	1	1	1	46.1 %	9.9 %
2	40	20	40	50.3 %	43.5 %
3	60	20	60	49.9 %	10.6 %
4	40	20	60	50.4 %	11.0 %
5	20	20	80	64.0 %	49.3 %

Table 1: Experimental Results

The table shows examples of three different types of latency setups: in setup 1 (first row), there were no additional latencies introduced, the complete session therefore was a LAN game with fast connections. For this setup, more than 46 % of subsequent user actions were processed in the incorrect order if our lag mechanism was turned off. Turning it on, less than 10 % of actions were processed in the incorrect order. Our implementation, therefore, provides a serious improvement for LAN games. The two other types of setups are simulated Internet-based games with equal client-proxy latencies $l(C_A, P_1)$ and $l(C_B, P_2)$ (setups 2 and 3) and with very different client-proxy latencies (setups 4 and 5). These different classes of test setups represent different connection types and Internet Service Providers (ISP) of geographically distributed players in the same session. For these both types of test setups, the setups 3 and 4 show about 50 % incorrect action orders without the lag mechanism and about only 11 % incorrect order with the lag mechanism activated. The performance of the lag mechanism is as good as in the undelayed LAN case. However, the setups 2 and 5 show no significant improvement with the lag mechanism activated.

¹<http://www-x.antd.nist.gov/nistnet/>

4.4 Discussion of the Experiments

The experimental results show several interesting facts: First of all, without the lag mechanism applied, in all setups at least about 50 % of subsequent actions are processed in the incorrect order. We think that this problem does not only apply to our specific proxy-replication concept, but to all multi-server parallelization approaches based on server-side replication. Each architecture which distributes the processing of interdependent user actions on different servers is prone to mix up the processing order of actions. The performance of our novel lag mechanism for multi-server replication in the current implementation is very sensitive to the actual latencies in the distributed system. While for some setups the amount of incorrectly ordered processing is drastically decreased, the mechanism provides no significant improvement in other setups. However, there was no setup in which there was not at least a little improvement: the lag mechanism does not increase the number of incorrectly ordered processings.

The reason why there are always at least about 10 % incorrectly ordered actions, and why for some setups the lag mechanism in the current implementation does not provide a real improvement at all, is the discrete check of the delayed action queues. The proxies check only every 10 ms whether actions have been delayed long enough and are due to be processed. These checks at both proxy-servers are not synchronized, such that, although scheduled correctly in the queues, the distributed processing order is incorrect. Unfortunately there is no conceptual solution for this problem. Further increasing the tickrate and therefore decreasing the wait time between checks might help, but the game already runs at a very high rate (the normal tickrate of the game is 10 updates per second instead of our 100 updates). A higher rate will lead to server congestion much earlier and will decrease the total number of possible players. Instead of the more naive discrete polling of the delay queue, the complete main loop of a game should be already designed with the lag mechanism in mind, possibly using conditional-wait data queues which notify the game at the very moment when an action is due to be processed. Another reason is that the clock times for the timestamps can only be synchronized with a certain deviation and even in a Local Area Network there is still a small jitter in communication latency. Therefore, for user actions issued with only a very little time difference, the variance of the distributed clocks and the actual time for these particular packets to be sent via the network might result in an incorrect processing order, even with a very optimized game loop implementation in the servers.

In the implementation of the lag mechanism the server has to check in each tick which actions are due to be processed from the pre-sorted user action queue by comparing the current and the actions' execution timestamp. This determination of actions to be processed is of linear complexity regarding the number of clients and therefore grows with an increasing number of users. While this computational overhead is very low for small scale games with only a few players, large scale games with a lot of users might result in a much longer action queue to check each tick and therefore require a very optimized implementation of the lag mechanism. One possible optimization would be to have the queue know the time

interval it will be searched for actions and store a hint updated at each insert of an action up to which index position actions will be due at the next search. The communication bandwidth overhead of the lag mechanism is minimal, because it only requires timestamps being added to actions sent between processes. Adding these timestamps, if not already sent for game-specific synchronization issues, only adds a constant and small amount (four Bytes at maximum) of data to an action message. In conclusion, the implementation of the lag mechanism introduces some computation and communication overhead. However, while we only implemented the straightforward linear action queue processing for our small scale testing with two players, this processing can and should be optimized for large scale games. We think that, regarding the trade-off between correctness and required computation, the lag mechanism is a very feasible option for the proxy architecture.

5. RELATED WORK AND CONCLUSION

Multi-server replication for scaling real-time multi-user environments has been proposed in several different flavours like the bandwidth-optimized *RING* system [5], peer-to-peer style systems like in [6] or dynamic runtime replication as in *Colyseus* [4]. Similar to our own proxy-server approach, all these replication and parallel execution approaches share the problem of interdependent user actions being concurrently processed at different servers. In terms of processing correctness, these replication architectures pose a much harder challenge than conventional single-server approaches where the complete game state processing is being serialized and sequentially executed at a single game server. Our discussion of the correctness problem, although using our proxy-server architecture as the conceptual basis, is therefore not limited to just our approach but addresses problems of the replication approach in general.

The concepts of the pessimistic lag and optimistic timewarp mechanisms ensure processing correctness in our proxy-server architecture and should be adjustable for and transferable to other replication approaches. Our new *remote lag* mechanism as an extension to the existing local lag approach, addresses and solves a problem which in this way occurs in every replication scheme when two different interactions race the network for the execution at a remote server process.

Our concept and implementation of the control server to compare the order of real-world issuing and virtual processing of actions allowed us to quantitatively analyze and verify the processing correctness at runtime. The test setup with the possibility to delay each communication link in a fine-grained way allowed us to test a huge variety of actual Internet-like session setups. As one result, the tests using our existing QFusion FPS-engine port for the proxy-server architecture showed that there is actually an urgent need for incorporating correctness mechanism into distributed game state processing, as about 50 % of pairwise subsequent actions were processed in the wrong order. The experimental results of the lag mechanism in the QFusion engine showed that, while for some setups the number of incorrect processing is drastically reduced, there is no significant improvement for other setups. Therefore, in actual implementations a lot of technical details have to be taken into account, as the discrete tick-wise processing, for example. In order to

provide processing correctness for all actions in usual setups, the complete game and in particular the game loop should be already designed with the correctness problem and specific correctness mechanisms in mind, especially when using a complex multi-server replication architecture.

6. REFERENCES

- [1] G. Armitage. Sensitivity of Quake3 players to network latency, IMW 2001 poster, 2001.
- [2] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The effects of loss and latency on user performance in unreal tournament 2003. In *Proceedings of ACM Network and System Support for Games Workshop (NetGames)*, Portland, Oregon, USA, September 2004.
- [3] Y. W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Proceedings of Game Developers Conference'01*, 2001.
- [4] A. Bharambe, J. Pang, and S. Seshan. A distributed architecture for multiplayer games. In *PACM/USENIX NSDI*, San Jose, USA, 2006.
- [5] T. A. Funkhouser. RING: A client-server system for multi-user virtual environments. In *Symposium on Interactive 3D Graphics*, pages 85–92, 1995.
- [6] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE Infocom*, March 2004.
- [7] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: Providing consistency for replicated continuous applications. *IEEE Transactions on Multimedia*, 6(1):47–57, February 2004.
- [8] J. Müller, S. Fischer, S. Gorlatch, and M. Mauve. A proxy server-network for real-time computer games. In M. Danelutto, D. Laforenza, and M. Vanneschi, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 606–613, Pisa, Italy, Aug. 2004. Springer-Verlag.
- [9] J. Müller, J. H. Metzen, A. Ploss, M. Schellmann, and S. Gorlatch. Rokkatan: Scaling an RTS game design to the massively multiplayer realm. In *ACM SIGCHI International Conference on Advances in Computer Entertainment Technology (ACE 05)*, pages 125–132, Valencia, Spain, June 2005. ACM.
- [10] J. Müller, T. Schröter, S. Fischer, and S. Gorlatch. Porting quake2 onto a scalable multi-server network architecture (in preparation for publication). Technical report, Westfälische Wilhelms-Universität Münster, 2006.
- [11] NCSOFT. Tabula rasa
<<http://www.playtr.com/index.html>>.
- [12] Open Source Project. Qfusion engine
<<http://sourceforge.net/projects/l33t/>>.
- [13] N. Sheldon, E. Girard, S. Borg, M. Claypool, and E. Agu. The effect of latency on user performance in Warcraft III. In *Proceedings of ACM Network and System Support for Games Workshop (NetGames)*, Redwood City, California, USA, May 22-23 2003.
- [14] ID Software. Quake 2
<<http://www.idsoftware.com/games/quake/quake2/>>.
- [15] Sony Online Entertainment. Planetside
<<http://planetside.station.sony.com/>>.
- [16] T. Sweeney. Unreal networking architecture
<<http://unreal.epicgames.com/network.htm>>, July 1999.
- [17] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.
- [18] Webzen. Huxley <<http://www.huxleygame.com/>>.