Scalability via Multi-Server Replication for Online Games: A Case Study of Quake 2

Jens Müller*, Tobias Schröter†, Stefan Fischer‡ and Sergei Gorlatch*

*University of Münster, Germany, jmueller|gorlatch@uni-muenster.de
†University of Brunswick, Germany, Dipl.Inf.T.Schroeter@web.de
‡University of Lübeck, Germany, fischer@itm.uni-luebeck.de

Abstract—Multiplayer online games are an increasingly popular class of distributed applications that require scalable network architectures and parallelization approaches. Existing Massively Multiplayer Online Role-Playing Games (MMORPG) already allow thousands of users to concurrently participate in a single application instance. However, there are several other types of genres, in particular action and strategy games, which have not been successfully scaled to the massively multiplayer realm so far. The main reason is that these games have different requirements in terms of scalability than the already scalable role-playing genre: in particular, the density of players has to be scalable. In this paper, we discuss different scalability dimensions of online games and summarize our approach of a multi-server replication mechanism for scaling the density of players. We present our recent work on porting the First Person Shooter action game QFusion, based on the famous Quake 2, onto our proxy-server network architecture using the replication approach. We discuss how to maintain data consistency of distributed replicated game states and how the existing game implementation has been enhanced to support multi-server replication. The scalability experiments with the ported QFusion game show that the maximum number of participating players can be doubled, while at the same time improving responsiveness, using our proxy-server network.

I. INTRODUCTION

Massively multiplayer online games (MMOG) involving thousands of users in a single game environment have recently become a very successful domain of Internet-based computer games. In order to provide the required massive amount of processing power to run large game sessions, scalable network architectures are necessary. Different game designs and genres require several dimensions of scalability: Massively Multiplayer Online Role-Playing Games (MMORPG), for example, require a network architecture which is able to scale the overall number of participating players and, additionally, to scale the overall size of the virtual game world. First Person Shooter (FPS) and Real-Time Strategy Games (RTS) in contrast usually take place in a much smaller game world and therefore do not require a scalable game world size as much as an MMORPG. An MMO version of an FPS rather requires the ability of a network architecture to scale the density of players, because in action games users tend to go where the action is and therefore cluster together with a high density.

In this paper, we conceptually discuss different dimensions of scalability required by massively multiplayer online games. We then analyze the needs of FPS and RTS games, for which we present and discuss our replicating proxy-network topology [1] as an alternative to conventional architectures. We already evaluated and demonstrated the capabilities of our architecture for Massively Multiplayer RTS games by designing and implementing the game Rokkatan [2], which can be played by several hundreds of users in a single game session. As the main contribution of this paper, we demonstrate the feasibility of our replication approach for the important and large class of FPS games by presenting our work on porting the open source FPS QFusion [3] onto our multi-server approach. The QFusion-Engine is an enhanced version of the well-known FPS Quake 2 [4] which has been released as open source by ID Software in 2001 [5].

II. SCALABILITY OF INTERACTIVE REAL-TIME ENVIRONMENTS

The target of our study are interactive real-time environments – an application class which includes
several different application types like training simulations, distributed network operating systems and online games. Several characteristics are common for these application types: They are distributed, highly interactive in the sense that human users frequently provide input, and are required to react to these user inputs in real-time.

For online games, the real-time property of the underlying distributed system means that the state of the game has to be frequently updated. These updates have to incorporate user inputs and the general simulation of the virtual world and have to be visualized at all participating game clients. The game state consists of the states of all dynamic game entities like avatars (the virtual representations of human users) and other elements users can interact with. Each state of such an entity contains attributes like position, orientation, or current action, which have to be updated frequently and have to be consistent: All clients connected to the same session have to present the same game state to their users.

The frequency of the game state updates (so-called ticks) defines the responsiveness of a distributed online game, i.e., the shortness of the processing delay for a certain user input into a new game state at all participating processes. The required responsiveness depends on the type of the real-time online game: Fast-paced First Person Shooter games require a very high responsiveness due to the direct controlling of the player’s avatar in a very tight feedback loop. These shooter games usually run at update rates ranging from 10 up to 35 and more updates per second depending on the actual implementation. In contrast, the game genres of Real-Time Strategy (RTS) and Role-Playing Games (RPG) including MMORPG are played more indirectly by giving commands to units or the avatar in a more relaxed feedback loop. Therefore, these games usually run at lower tick-frequencies of 5 to 10 updates per second. However, all these games are soft real-time systems: A delay of a game state update beyond the length of a tick is a violation of the real-time constraints which immediately disrupts the steady flow of the game and leads to stuttering animations of entities’ actions. A lateness in game state computation or communication between the processes thus disrupts the users in their game immersion and has to be avoided.

The most common architecture setup to compute the game states and communicate them between all participating user clients is the well known client-server architecture. In this approach, which is commonly applied in small-scale FPS and RTS games, the single server is responsible to receive user actions from all participating clients, compute the new game state for every entity in the game and send the new state back to each client. If we abstract the virtual game world to be a two-dimensional area, the single server is responsible for this complete area and all entities as depicted by Figure 1.

The single-server concept is easy to implement, ensures consistency of the distributed game state and works well for small scale game sessions. However, the single server constitutes a single point of failure and, more important when it comes to massively multiplayer gaming, is not able to scale a game session for an increasing amount of entities. If the amount of entities in the game is increased by adding more NPCs or connecting additional game clients to the session, the required computation for a single game state update grows. The available amount of time for this computation, however, is constant and given by the time interval a certain state is valid. The server, therefore, will eventually congest and violate the game’s real-time constraint if more game clients are added to the session.

In order to scale a game session for a high number of users as required for massively multiplayer gaming, the constant computation power of a single server is not sufficient to ensure timely state computations for an increasing actual computation time. Therefore, the game network architecture must be able to increase the computing power, which generally means that the game state processing must be executed in a parallel or distributed way using several CPUs and/or servers. There has been
some work on game state processing on SMP-Servers with several CPUs [6] as we will discuss in Section VII. However, the much more widely used concept is to distribute the computation on several server machines.

A. Scalability Dimensions

Applications in the class of interactive real-time environments have several different requirements w.r.t. what particular characteristic should be actually scalable. For example, in multi-user training simulations for catastrophe scenarios, physical simulations of fire spread or behaviour simulation of virtual crowds in panic are usually required to be scaled. In virtual residence environments, as another example, the overall world size and the storage space for user-created models and textures should be a dimension of scalability.

In the following, we identify three general scalability dimensions of Massively Multiplayer Online Games by analyzing different game genres:

1. **Overall number of participating users**: the absolute number of users participating in a single game session. In MMORPGs, this is the number of users connected to a single virtual world (realm or shard) which therefore are able to interact with each other. In an FPS or RTS, this is the number of users playing in the same game session. Of these different game genres, an MMORPG has the strongest requirement regarding this scalability dimension, because the usually huge game world has to be actually populated. However, MMO versions of FPS and RTS require this scalability dimension as well, in order to provide game sessions for more than the current maximum of about 100 users.

2. **Game World Size**: In order to scale the game world size, two resources are required. The first resource is processing power for computing actions for more actively computer-controlled entities. In a larger game world, more of such entities like Non Player Characters (NPCs) or mobile enemies (mobs) have to exist and controlled by the servers. The second resource is main memory for storing an increasing amount of static terrain geometry and dynamic entities populating the game world. This dimension of scalability is especially required by MMORPG, where users can adventure in very large virtual worlds.

3. **Maximum Player Density**: The density of players refers to the number of players located near to each other in a comparatively small region of the game world. If more players move together in such a region, the density increases. This dimension of scalability is urgently required by FPS and RTS games. Because the main goal of these games is to defeat other human players, users move their players to where some action is going on, and thus dynamically create local player clusters with a high density. For MMO versions of FPS and RTS, player density has to be scalable in order to provide responsive gameplay for such situations with a lot of action. In MMORPG, this dimension may be required as well: For example, users come together at a high density in big cities to interact with each other (trade or chat) or form large armies to attack each other in player-vs-player scenarios.

There are several other dimensions which are more special to specific game designs: for example, a virtual housing environment requires scalable disc space for storing the enormous amount of user-created and uploaded models and textures. However, the three dimensions identified by us are general for all massively multiplayer games and required by the three main real-time game genres of MMORPG, RTS and FPS.

In the following subsections, we discuss the characteristics of two different multi-server parallelization approaches for massively multiplayer online games with regard to these three scalability dimensions: The **zoning concept** which is already used for MMORPG, while our own **replication concept** serves as an alternative approach suitable for fast-paced FPS and RTS games.

The zoning approach partitions the game world of an MMOG into several zones. The game state updates for these zones can be computed independently of each other on a cluster of servers. The maximum processing power in this setup will be reached if each server handles only a single zone of the game world. In this case, the overall architecture can be viewed as an independent collection of single-server topologies, corresponding to the number of zones in the game world. Each of these single-server topologies computes the new state only for the game entities (avatars, non-player characters, items, etc.) residing in its zone.
B. Game World Zoning

Figure 2 depicts an example of a game world partitioned into four zones, where each zone-server handles its entities independently of the rest of the game world. Communication between servers only occurs when a user moves his avatar from one zone into another: The game client has to establish a new connection to the server responsible for the new zone and the servers have to handle the client migration in a consistent way.

There are several possible enhancements to this approach like dynamic resizing of zones, processing several lightly populated zones on a single server or slightly overlapping zones for a transparent migration of clients. The usual cluster-setup of servers implementing this zoning concept is discussed in [7], while [8] presents a different, peer-to-peer-based setup of the zoning approach.

The zoning approach allows to scale the total number of players (first dimension) very well if the players are distributed among the different independent zones. In MMORPG, where zoning is commonly applied, the users are encouraged to spread out in the game world: the zones are usually different in difficulty and obtainable game rewards, such that players of different experience levels are separated. Additionally, zoning scales the game world size (second dimension) because additional zones and servers can be added arbitrarily. However, zoning does not work well for FPS and RTS games, because it does not scale the density of players (third dimension): In competitive player-vs-player games like FPS, users are encouraged to cluster together to fight each other and will actively search for ongoing fights in the game world. This behaviour results in “hot spots” of combat with a high density of players, which congests the single server of that zone.

C. Our Approach: Game World Replication

We developed our approach of game world replication as an alternative to the zoning concept for scaling the maximum user density of an interactive real-time application. In our multi-server replication concept, each server holds a complete copy of the world and all entities of the game. However, the responsibility for entities is equally distributed among all participating servers, such that a single server computes the new state only for a particular subset of entities. The entities a particular server is responsible for are called active entities for this server. The entities being updated at other servers are denoted shadow entities. Each server regularly, i.e., after each update, sends the new state of its active entities to all other participating servers which then update their shadow replicate copies of these entities. Figure 3 shows how the replicas at the different servers can be viewed as layers, in which the active entities of a particular server are depicted as filled, while the shadow entities maintained at another server are outlined.

Only a single particular server is allowed to alter the state of an entity, i.e., has write access, and propagates updates to all other replicate copies at the other servers. This synchronization approach guarantees eventual consistency of the replicated game state. In general, there is a trade-off between the responsiveness of the distributed state processing and the degree of consistency. We chose to implement eventual consistency because it enables high responsiveness of the distributed state processing (no communication for entity locking required) at a reasonable degree of consistency.

Our replication approach is suitable for FPS and RTS games because it does not require the
game environment to be segmented and works on arbitrarily small worlds. Additionally, it does scale the maximum density of players (third dimension) in a particular area, because all servers participate in the game state processing of the entities at a hot spot of clustered avatars. The main observation here is that the computation of the new state of an active entity (checking collisions and computing the new position, processing interaction targets, computing health points, etc.) takes more computation cycles than communicating and applying a remote update of a shadow entity which only requires some unchecked variable assignments. Additionally, the replication approach does not require clients to migrate the server connection during a game session. Each game server has a full copy of the game state and therefore can serve arbitrary clients regardless of its avatar position in the game environment. This characteristic allows the game to take place in a seamless world without zones and therefore is more suitable for small- to mid-sized worlds of FPS and RTS games. Additionally, our replication does not require clients to change the server connection when moving the avatar into other zones. A game client in the replication concept does not need additional functionality in comparison to the client of the single-server approach: the single replication server a client is connected to can provide information about the complete game world just as the single server in the conventional single-server setup.

In order to use this replication approach in an actual game implementation, we developed and implemented the Proxy-Server Network as an operational multi-server network architecture. In this architecture, several proxy servers are interconnected for a single online game session. Each proxy server holds a complete copy of the application state and can serve arbitrary clients. Servers therefore can be scattered over the Internet (at different Internet Service Providers, for example), and a client chooses to connect to a server with low communication latency: Such a server is close to the client in terms of communication delay and thus called proxy in our architecture. Figure 4 shows an example session with four proxy servers.

Each proxy is responsible to calculate the new game state for its active game entities and to communicate in to all other participating proxies. Thus, the proxy server architecture directly incorporates our multi-server replication approach (Section II-C). For the actual decision about which entities are active on which proxies, we distinguish two kinds of game entities:

1) **User-dependent entities** are directly associated to 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** are not directly associated with a particular client. Such entities are objects in the game world users can interact with, 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.

In order to balance the load on the servers, the active state of user-independent entities should be
distributed among all participating proxy servers, taking into account the actual processing power of the server machines. The user-dependent entities, however, should be active entities at the proxy server the particular game client is connected to. This way, all user actions which manipulate the user’s own entities (like moving its avatars, e.g.) can immediately be processed at the proxy server the client is directly connected to, resulting in high responsiveness for these frequently issued user actions. The proxy server can immediately change the entity state (e.g., change the position) and confirm the new state to the client. Additionally, the server sends an update message to all other proxies which update their corresponding local shadow entity of the changed game entity accordingly.

The processing of interactions can not be done solely by the proxy server of a particular client. Such an interaction issued by a user (firing a weapon, picking up an item, etc.) possibly affects not only the state of the user’s own avatar entity, but also other entities. In the general case, the interaction target entity will be active at another proxy, which requires sending the interaction to this remote proxy for evaluation. Fig. 5 depicts the processing of an user interaction example in the proxy-server architecture.

In Fig. 5, the user at client A issues an interaction affecting the avatar of client B. In the port of the QFusion-engine, for example, the user could fire a weapon onto the position of the avatar of B in the game environment. In step ①, client A submits the action to its proxy server which validates the received input in step ②. If the validation was successful, i.e., the state of the avatar allows to perform the attack, then the proxy sends an acknowledgement back to client A (step ③) and forwards the interaction to all other participating servers in step ④. The other proxies update their local game state, i.e., they update the avatar’s state of the attacking client A in step ⑤. Additionally, each remote proxy checks whether a game element it is responsible for is affected by the attack of avatar A. Let us assume that the avatar of client B is hit by the attack. The local proxy of client B updates the state of its avatar by decrementing health points and informs all other proxies about this state change (step ⑥). Finally, in step ⑦, all proxies inform local clients which are directly affected by the interaction (client A and B). Additionally, all clients whose avatar is located near to the interacting avatars, are notified about the interaction. For example, users at the clients D and E observe the interaction and the proxies inform the clients about it.

IV. THE QFUSION ENGINE

We ported the open source FPS QFusion [3] onto our proxy-server topology to demonstrate the feasibility of our approach to scale the maximum number of participating players in FPS games. However, before we discuss the details of the multi-server port of QFusion in Section V, we describe the original network processing. The QFusion-Engine, of which a client screenshot is shown in Fig. 6, has been developed as an enhanced version of the famous FPS Quake 2 [4] of ID Software. The source code of Quake 2 (implemented in C) has been made open source by ID Software in 2001 [5] and provides the basis for the QFusion-engine.

QFusion provides much more sophisticated graphics than the original Quake 2 engine. However, the basic network code has not been changed by the QFusion-project team. Therefore, just as the unmodified Quake 2, QFusion constitutes a very fast-paced, small-scale multiplayer First Person Shooter enabling a very competitive style of gaming using a single game server. During a running game session, all participating clients are connected to a single server. Each client sends the user commands to the server which frequently (10 updates per second)
computes a new game state and sends them back to the clients. An example of this original setup is depicted in Figure 7.

![Game Screenshot of QFusion](image)

Fig. 6. Game Screenshot of QFusion

While the single-server network architecture works well for small-scale sessions, there is a particular maximum player number depending on the processing power of the server host: the server will not be able to maintain the required update rate of 10 updates per second if the number of clients is increased beyond this maximum. Our experiments, which will be discussed in detail in Section VI, showed that the original single QFusion-server running on a Pentium 4 1.7 GHz host can maintain the required update rate for 38 participating clients. Connecting more than these 38 clients to the session leads to server congestion, the game becomes unresponsive and movements of avatars and user interactions are constantly lagged.

In order to port the QFusion-engine to our proxy-server architecture for increased number of players, we thoroughly analyzed the game state processing inside the server update loop. Such a game state is called frame in QFusion. There are four blocks of computation in the main update function `SV_Frame()` which are processed during each update loop:

1.: Receiving, validating and processing of user actions in the function `ClientThink()`. The server receives these user actions in form of `usercmd_t` structs from the clients. For each of these messages, the server computes the new state of the corresponding user, which is stored as a `player_state_t` struct containing variables for all relevant game state parts like position, movement velocity, viewing direction, the weapon currently used etc. The update of these variables depending on the user actions and possible interactions form other users (being hit by a weapon, e.g.) takes place in the main player update function `ClientThink()` for each participating client.

2.: Updating user-independent entities of the game world in the function `G_RunFrame()`. All server-controlled items like weapons, armor, ammunition packs etc. are processed at this step. In this function, the server calculates a new state for each of these entities depending on the freshly passed time of 100 ms of the currently processed frame represents at the standard tickrate of 10 ticks per second. For example, items usually reappear after a certain amount of time to be picked up by users again. This reappearing (orRespawning) is handled inside of `G_RunFrame()`.

3.: Waiting for the complete time of 100 ms of a single server frame to pass. If there is time left after the computations of steps 1 and 2, then the server will sleep for the rest of the duration of the server frame. Additionally, during this time the server still listens on incoming user actions from clients. If a user action is received during this time, then the server will awaken and will receive, validate and process this action by running the player update function `ClientThink()` for the particular client.

4.: Transmission of the new game state to clients in the function `SV_SendClientMessages()`. At the end of the game frame, all the new player
and server-entity states computed in the three steps before have to be sent to the clients. Instead of sending the complete state to all clients, the server determines and sends the relevant subset of the overall game state to each client. This filtering of information reduces the used network bandwidth and prevents the type of cheating in which hacked clients can make hidden entities visible through transparent walls, because the server does not sent recent data concerning these hidden entities in the first place.

For the port of the QFusion-engine onto the proxy-server architecture, this main loop for the computation of a single server frame has to be enriched to incorporate the active-/shadow-entity concept. Each server has to receive and process updates of shadow entities and to generate and send appropriate update messages for its own active entities. We discuss the required changes to the server processing in the next subsection in detail.

One main important feature of the proxy-server architecture is that a client, compared to the single-server approach, does not need any additional functionality. Therefore, although the original QFusion-server of course had to be adjusted to incorporate the replication concept, no changes were required to the QFusion game client.

V. Porting the QFusion-Engine on the Proxy-Server Architecture

We ported the QFusion-engine onto the proxy-server architecture in order to demonstrate the feasibility of the replication approach to scale the maximum number of players for fast-paced FPS games. In the remainder of this paper, the new ported engine will be called QFusion Proxy-Architecture (QPA). We expected that using several proxy servers for a single QFusion session would result in higher possible player numbers without violating the game’s real-time constraints, because the computation for the new game states is split up among several servers. Additionally, the port serves as a demanding case study on how the eventual consistency synchronization affects the game’s responsiveness: Since QFusion is a very fast-paced shooter game, the gameplay must not be delayed by the inter-proxy synchronization communication. The actual performance of the QFusion Proxy-Architecture will be discussed in Section VI; in the following subsections we will describe how the ported engine operates.

A. Incorporating the Active-/Shadow-Entity Concept

The two different entity types of user-dependent and user-independent entities discussed in Section III can be found in the QFusion-engine as well. Our general concept for the distribution of the active state of these entities is that each proxy is responsible for the user-dependent entities of its locally connected clients for maximum responsiveness and that the active state of user-independent entities is distributed among participating proxies in a load-balancing way.

For our QFusion Proxy-Architecture, we implemented the authority distribution of user-dependent entities as discussed in Section III. The player state of connected clients can only be changed by the proxy the client is directly connected to. This distribution ensures eventual consistency of these entities following the general concept of the replication approach and the proxy-server architecture.

The user-independent entities, however, are active on every proxy server in the QPA. Therefore, each proxy can alter the state of these entities, and has to send an appropriate update message to other servers if the state is changed. Although the concurrent access to user-independent entities is not synchronized by the architecture itself this way, consistency is still ensured because the game logic prevents such a concurrent access: All user-independent entities in QFusion are passive items (weapons, ammunition, health packs or armor) which can be picked up. In order to pick up such an item, a user has to move the player model of its avatar directly over the item. Due to the collision handling of player models based on the models’ bounding boxes, only one model at a time can walk over an item to pick it up.

We implemented the processing of an item pick-up action in QPA as follows: the responsible proxy of the player model that moved over an item removes the item from the map, updates the player state (for example, increasing health points in case of a health pack being picked up) and sends an update notification concerning the removed item and the updated player state to the other servers.
If another player model maintained at a different proxy tries to walk over the same item at the same time, this proxy will detect the collision with the player model already “standing on” the item and will not execute that movement of its player. Overall, this optimized synchronization of pick-up actions of passive items using the already existing collision detection reduces network communication. Additionally, it results in highly responsive execution of such item pick-ups, because the local proxy executing the movement can immediately execute the pick-up action as well.

B. Main Loop of QPA

We enriched the original server update function `SV_Frame()` to incorporate the active-/shadow-entity concept and to handle incoming and outgoing synchronization messages among participating servers for the eventual consistency of the replicated game state. Figure 8 illustrates an example setup of the QFusion Proxy-Architecture, the communication between the participating processes and the processing in a single call to the enriched `SV_Frame()` function.

As illustrated in Figure 8, each of the ported QFusion-servers participating in a single session has a full copy of the game state. During the computation of a new game state, each server has to process its local active entities and to update the remote shadow entities; the resulting processing steps for the different types of entities are printed in bold face in the Figure.

In the following, we discuss the main steps of the QFusion-Proxy-Server update loop in detail referring to Figure 8:

1.: Receiving, validating and processing of user actions from local clients. Just as in the original, single-server version, each proxy server receives user actions from the directly connected clients and computes the new state of the corresponding active player entities using the player update function `ClientThink()`. In addition to the single server implementation, the proxy server will immediately forward the new `player_state_t` structs to all other participating proxy servers if the action of that particular user is an interaction. The other proxy servers need this information about the interaction in order to check whether one of their local clients is affected. This forwarding of interactions occurs directly in this first step in the new function `QPA_UpdateRemoteClientState()`.

2.: Receiving and processing updates of shadow player entities from other proxy servers. If the state of a remotely connected client has been updated by the responsible proxy, the corresponding update message including the new `player_state_t` struct has to be processed. In this step, the proxy processes all received and pending updates in the function `QPA_UpdateRemoteClientState()`.
by assigning the received updated values of position, health points etc. to its local shadow data structure of that player entity.

Additionally, the proxy will check if a received interaction like the firing of a weapon affects a local active player entity. If there was a successful hit, for example, the proxy will update its local active player entity (decreasing health points, etc.) and send a corresponding update to the other proxies. This way, our port of the QFusion-engine incorporates the general interaction processing scheme of the proxy-server architecture discussed in Section III and illustrated in Figure 5.

3.: Updating the user-independent, passive entities of the game world (pickup-items like weapons and armor). As discussed in the previous subsection, each proxy server checks whether such an item is picked up by one of its players in the function G_RunFrame().

4.: Waiting for the complete time of 100 ms of a single server frame to pass. Just as in the unmodified QFusion-engine, the server sleeps the remaining time of the frame (if there is any left) and processes incoming user actions during that time by running the ClientThink() function.

5.: Transmission of the new game state to clients. This part has not been changed in comparison to the original engine: each proxy server determines what new state data has to be sent to which directly connected client and transmits this information in the function SV_SendClientMessages().

6.: Final synchronization transmission of all changes of active entities which have not been sent in the current frame yet. While interactions already have been forwarded immediately in the very first step, non-interactive user actions like movements, changes of viewing directions or changes of the activated weapon have not been sent yet. These entity updates are sent in this final step using the function QPA_SV_SendRemoteCLUpdate() to be processed by other servers as soon as they run step 2 of the frame computation next time.

This main loop of our QFusion Proxy-Architecture port incorporates the general ideal concept of the replication approach. The two most compute-intensive functions of the state computations of all active player entities (ClientThink()) in step 1. and the filtering of the relevant state information (SV_SendClientMessages()) in step 5. are distributed among all participating proxy servers. However, we introduced some additional computation for the sending and processing of entity synchronization messages (QPA_UpdateRemoteClientState() and QPA_SV_SendRemoteCLUpdate()). We found out in our experimental measurements as discussed in Section VI that this overhead for the inter-proxy synchronization is about an additional 19% of the unmodified sequential QFusion implementation. However, our implementation to be scalable w.r.t the maximum number and the density of players, because the new functionality requires much less computation time (mostly some variable assignments) than the processing parts we distributed among servers.

C. Communication in QPA

The unmodified QFusion-engine uses its own network subsystem, in which clients and the server directly communicate over the sockets API as illustrated in Figure 9. This QFusion network layer consists for the most part of the unmodified Quake 2 network system.

![Fig. 9. Original QFusion Networking](image)

In order to synchronize the replicated game state of a QFusion Proxy-Architecture session, additional inter-server communication functionality had to be incorporated into the QFusion network layer. However, instead of adding this functionality to the original network system, we replaced the complete communication layer of the QFusion-engine with our Game Proxy-Architecture (GPA) communication
library as illustrated in Figure 10. This library has been implemented in our previous work using C++ to ease the use of the proxy-server architecture by providing general, socket-based communication channels for the client-server and inter-server communication. For each of these communication channels, the processes can send and receive character-arrays using several degrees of communication reliability over an convenient API. The inter-server communication uses IP-Multicast and automatically falls back to unicast communication to server hosts which are not able to join the multicast channel because some intermediate router does not support multicasting. Figure 10 shows the altered server main loop and the exchanged network subsystem shadowed, while the client logic of QFusion was not required to be changed at all.

![Fig. 10. Ported QFusion-engine using the Game Proxy Architecture (GPA) Library](image)

Besides the core communication functionality, the GPA provides utility functions for starting up and finding a game session, for determining communication latency between processes and for managing distributed game entities by providing session-wide unique identifiers. The GPA has already been successfully used as the network subsystem in our RTS demonstration game Rokkatan and allowed us to add the required additional communication functionality into the QFusion-engine with comparably little work.

VI. Experiments

We conducted extensive tests of the presented QFusion Proxy-Architecture in order to verify its functionality and to evaluate the actual increase of possible player numbers in a single session.

A. Functionality Test

Regarding the functionality test, we set up a distributed session with two QFusion Proxy-Servers located in Muenster and Luebeck, as illustrated in Figure 11.

![Fig. 11. Distributed Functionality Test Setup](image)

At each site in the distributed functionality test, several clients participated in the test sessions. The clients were connected to the server near to them in terms of communication latency, such that the proxy functionality of the servers was used according to the design of the architecture. Both proxy-servers fully replicated the game state of the single session and synchronized their entities according to our active/shadow approach over a public Internet connection with an inter-proxy latency of about 15-20 ms.

The distributed play test ran successfully, there was no noticeable delay of interactions between users at different sites. Even better, avatar movements were very fluent because the according user actions were processed at the local proxy with very little latency.
B. Scalability Tests

We tested the maximum number of participating players for different numbers of connected proxies in a single LAN environment in Muenster. For these tests, we used an autonomous non-graphical bot client of which several could be started at a single client host. The bot client generates as much actions as possible by continuously moving around and firing the weapon as fast as possible. This way, we constantly tested the worst case for the architecture in which no user is idle and, therefore, an action for each user has to be processed each tick. Using up to 20 hosts at our department for the servers and clients, we were able to test sessions with up to four participating proxies. The server machines in the tests were single-CPU Pentium 4, 1.7 GHz hosts. For each setup, we continuously added clients to the session until the servers became congested, i.e., were not able anymore to finish the complete tick calculation in the time of 100 ms given by the QFusion standard tickrate of 10 ticks per second. The maximum number of players up to the server saturation is illustrated in Figure 12.

![Fig. 12. Maximum Player Numbers for Increasing Proxy Servers](image)

The tests were run on a very small quadratic map without any additional separating walls, because we wanted to test the actual scalability of player density. In the single server setup of the QFusion Proxy Architecture up to 32 users were able to participate in an uncongested session. In comparison, the unmodified QFusion-engine with its single server running on the same server host was able to support 38 users. The overhead of the additional proxy-related processing, which already is performed if there is only one server in the session therefore is about 19%. However, the QPA version allows to increase the maximum number of participating users to up to 80 players when using four proxy servers. Although this scaling is not linear due to the overhead of the inter-proxy synchronization, we effectively were able to double the player density using four servers. The screenshot series of the test sessions in Figure 13 illustrates the achieved increase of density of the bot players gathered very closely together.

![QPA Screenshots illustrating Increasing Player Density](image)

Fig. 13. QPA Screenshots illustrating Increasing Player Density

VII. RELATED WORK AND CONCLUSION

Our replication approach using the proxy-server network allows to scale the total number of users
as well as the density of users in an online game session. While the zoning approach as discussed in [7], [8] is preferable for the large game worlds of MMORPGs, replication allows to scale small world games like RTS and FPS titles. The Colyseus system [9] uses a replication approach as well, but entities are replicated on demand which requires a setup time and targets more bigger game environments. Regarding our presented scalability dimensions and tick-model of an online game server, [10] and [11] provide a more basic discussion. [11] presents a detailed scalability analysis for the game Quake 1 as the predecessor of Quake 2 used in our implementation. Furthermore these authors presented a parallel implementation of Quake 1 for SMP server hosts in [6]. Although it requires expensive server machines, such a parallelized implementation of an FPS is an interesting alternative to our proxy-network for scaling the player density.

The presented port of the QFusion-engine demonstrates that our replication approach on top of the proxy-server network is a feasible method to scale the overall number of players as well as their density in FPS action games. The existing sequential QFusion server loop is already highly optimized, such that efficiently porting the server onto the proxy-network was quite difficult and introduced an overhead of about 19% for the additional processing. As we experienced while implementing our demonstrator game Rokkatan [2], the implementation can be more efficient when incorporating the replication approach from scratch, because the data structures can be designed and optimized accordingly. However, the QPA port does scale and allows to double the player numbers and density with reasonable effort of using four servers. In real game sessions, the users will usually spread out more than in our synthetic worst-case density tests, thus allowing for even better scalability.

The replication approach, therefore, is a suitable concept for FPS games, which allows to increase player numbers and density in a responsive way using the proxy-server network.

REFERENCES