From a Single- to Multi-Server Online Game: A Quake 3 Case Study Using RTF

Alexander Ploss, Stefan Wichmann, Frank Glinka, and Sergei Gorlatch
University of Münster, Germany
{a.ploss,swichmann,glinkaf,gorlatch}@uni-muenster.de

ABSTRACT

Fast-paced action online games like First Person Shooters (FPS) pose high demands on resources and thus require multi-server architectures in order to scale to higher player numbers. However, their multi-server implementation is a challenging task: the game processing needs to be parallelized and the synchronization of the distributed game state needs to be efficiently implemented. As part of the European edutain@grid project, we are developing Real-Time Framework (RTF) – a middleware that provides high-level support for the development of multi-server online games. This paper describes a case study on porting the open-source, single-server Quake 3 Arena game engine to a multi-server architecture using RTF and its state replication approach. We conducted extensive scalability and responsiveness experiments with the ported version of Quake 3 to evaluate the performance of our middleware. The experiments show that the responsiveness of RTF implementation can compete with the original Quake engine, and that the replication support allows to efficiently scale FPS games using multi-server processing.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems—Distributed Applications; C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems

General Terms
Distributed Architecture, Real-Time Online Interactive Applications, Online Games, Middleware

Keywords
Massively Multiplayer Online Games, Responsiveness, Performance, Scalability, Quake 3 Arena, RTF, Network Middleware

1. INTRODUCTION

Fast-paced action games pose high demands on the game hard- and software due to the high rate of interactions of a player with the game environment and other players. A game session evolves as the players constantly and independently issue commands (movements, attacks, etc.) that influence the game world. The challenge for the game software is to process these commands and manipulate the game state, such that the players continuously and timely (so-called soft real-time) see the evolution of the game state based on their actions. A prominent genre for this type of games are First Person Shooters (FPS) where the player directly controls his avatar from a first-person perspective and interacts with other players by shooting on their avatars.

To coordinate the interaction of the players, the processing of the global state in action games is often organized in a classical client-server architecture. However, the single server limits the total number of players that can participate in one session. Total player numbers can be increased by incorporating multi-server processing. The remarkable challenge of multi-server processing is the significantly increased complexity of the game design and implementation.

We are developing the Real-Time Framework (RTF) which provides a high-level support for real-time interactive applications in general and multiplayer online games in particular: it simplifies game parallelization using different distribution strategies and implements automatic management and communication for the game state synchronization.

In this paper, the development of multi-server games using RTF is illustrated using a real-world case study – the popular Quake 3 Engine, which was originally designed for a single server. We use the replication-based distribution [8], i.e. the game state is available on all servers. We test the responsiveness (time for the user to perceive the result of his action) and scalability (increasing player numbers when using additional servers) of the RTF-based multi-server version of Quake 3 against the original single-server Quake.

The remainder of the paper is organized as follows. In Section 2, we describe the original Quake 3 Engine. Section 3 presents the basics of the Real-Time Framework and the concept of replicated multi-server processing. Section 4 describes our multi-server implementation of Quake 3 using RTF. Sections 5 and 6 finally present our experimental performance study of the multi-server Quake.
2. THE QUAKE 3 ENGINE

Quake 3 Arena (Quake in the sequel) [11] was first released by ID Software in 1999 and has been one of the most popular and successful FPS games since then.

The main challenges of multi player action games can be observed for Quake: fast-paced gameplay, short actions' response time, very high rate of interaction, and high density of players [7]. Quake requires players to continuously move around and interact with the game environment, e.g., picking up powerups or attacking other players. Each map of the game is designed to support constant action where no player is likely to stay alone for a long time.

![Figure 1: Quake 3 Arena main software components.](image)

The Quake software consists of several components shown in Figure 1. The components Entities and Gameplay describe the game logic, i.e., the attributes and capabilities of entities in the game (position, hitpoints, etc.) and the rules for processing a new game state (movement speed, interactions, etc.). On the other hand, the components Entity Management, Memory Management, and Network Communication realize the game engine, i.e., the infrastructure for game state processing. For our work, we are only interested in the components involved in the game state processing (unshaded in the figure). The light gray components in the figure realize the interaction with the user.

![Figure 2: Quake 3 Arena client-server architecture.](image)

Like other action games, Quake traditionally uses the single-server architecture for the processing of the game state, see Figure 2. Usually, the processing of the game state is organized in multiple processing steps that are executed in a continuously running real-time loop (RTL in Figure 2). This computation of a new state for all participants of a game session is handled by a single process which ensures a consistent game state for all participants. In Quake, this processing server can run either as part of a graphical user client or as a dedicated server on a separate machine. A client in a Quake session mostly implements the interaction with the player, i.e., it reads the user's input and displays the current game state (graphics rendering and sound output), received from the server. Therefore, user actions and game state updates have to be continuously transmitted between client and server, which is realized via the Network Communication component.

Quake 3 Arena still uses the fundamental basics of the first engine of the Quake series [1]: A game session is operated by the server in a continuously running loop of frames\(^2\), whose three main steps are shown in Listing 1.

![Listing 1: Quake server frame (three main steps).](image)

In order to incorporate multi-server processing, the update of the game state (lines 6-7 in Listing 1), implemented in the game logic, needs to be parallelized. The challenge is to a) identify the parallelization criteria, and b) implement the algorithms of the game logic in a parallel manner. The entity management has to deal with the distribution of the game state. The processing algorithm for the gameplay rules has to be coordinated between the participating servers. This requires additional synchronization and communication between multiple servers, which should be implemented by a suitable network communication component.

Previous work [8] showed that the responsibility for processing of entities is a reasonable parallelization criterion for fast-paced games: all entities are partitioned among servers, such that each server applies the processing step only to the part of the game state assigned to it. Interaction between entities has to be coordinated if they are not processed by the same server. Therefore, the processing algorithm needs an access to all entities, must know for which entities it is responsible and must delegate the processing of other entities to their owners.

The next sections show how these challenges of parallelizing and implementing the distributed game state processing are handled with the high-level support of the RTF middleware. In particular, we describe how RTF helps in distributing the game state, coordinating the game logic processing algorithm and simplifying the network communication.

\(^2\)Quake refers to one processing cycle as frame, RTF as tick
3. THE RTF MIDDLEWARE AND ITS USE FOR QUAKE

We are developing the Real-Time Framework (RTF), implemented in C++, which provides a high-level communication and computation middleware for single- and multi-server online games. The RTF supports both the server-side and client-side processing of an online game with a dedicated set of services which allow game developers to implement their engine at a high, entity-based level of abstraction.

The RTF middleware deals with entity state updates and event transmissions to and from the clients and the distribution of the game state processing across multiple real-time server loops. The developer’s task remains to implement the game-specific real-time loop on the client and server, as well as the game logic, while relying on RTF for exchanging information between processes. In RTF Communication support is provided on a high, object-oriented level: it includes an automatic serialization mechanism for application-level objects, such that the application developer does not have to deal with the implementation of object (de)serialization.

RTF supports the three major concepts of parallel/distributed processing: zoning, replication and instancing. Zoning splits the game world into several spatial zones and assigns the processing of the entities inside a particular zone to a distinct server. It is possible for entities to move from one zone to another, but, in general, it is not possible to interact with entities in a different zone. Instancing creates separate copies of a particular zone of the game world; each copy is processed by a different server. The copies are, in general, independent of each other and it is not possible to interact with players in another copy. While zoning and instancing increase the overall player numbers in the game world, they also imply limitations for the interactions between players. A more detailed overview of the RTF capabilities in the game development process is given in [6, 10]. In the next section, we briefly describe how the novel replication concept of RTF is used to introduce multi-server processing in Quake and increase overall player numbers while allowing direct interactions of all entities.

3.1 Replication with RTF

Unlike the traditional zoning and instancing concepts, replication uses the processing responsibility for entities as the distribution criterion. The assignment of an entity to a particular server is not necessarily based upon the entity’s position in the game world, but rather on other aspects like the server’s processing load. In order to allow interactions between all entities of the game world, each server has knowledge about the whole game state, i.e., has a copy of all entities.

An example of replication is shown in Figure 3: entities are distributed among three servers, such that each server has a list of so-called active entities which it owns and is responsible for, and a list of shadow entities which are replicated from other servers, with read-only access. Each of the servers has a copy of the complete game world and each entity is marked as active in only one server; the other copies of this entity are shadow copies.

3.2 Multi-Server Architecture for Quake

According to the replication concept of RTF, the decision about the entity responsibility is made depending on the particular scenario of the application. Depending on the game rules and game play, entities may have relations to each other, which influence their interaction or require special coordination. Therefore, RTF leaves the task of entity distribution to the application developer who possesses the corresponding knowledge.

For Quake, this leads to a simple entity assignment: the server which creates an entity becomes responsible for it. This is the case for all player entities and also for dynamically created items (e.g., those dropped by players). The items that are a fixed part of the game world (powerups, weapons) and re-spawned, over time are created and handled by one server throughout the game. The rate of interaction in Quake is very high and entities are constantly moving around, so we decided not to optimize the assignment of entities based on their location.

Figure 4 shows our multi-server architecture for distributing the game state processing and the flow of information between distributed processes. Each client is connected to one dedicated server: the client sends the user commands to this server and receives from it the state updates for all (visible) entities. Thus, the distributed state processing is transparent for the clients. The use of multiple servers requires communication between them: state updates are sent to shadow copies of entities and the interaction between shadow and active copies needs to be coordinated. The next sections describe how we implemented this for Quake.
3.3 The Quake Game State Processing
The developer interface of the RTF middleware is designed to address the requirements of loop-based processing of online games. It, therefore, matches the original processing model of Quake as shown earlier in Listing 1.

Figure 5 shows the main steps of the Quake processing cycle in one server using RTF. In addition to the three main steps of the original Quake server frame (NET_Sleep, G_RunFrame, and SV_SendClientMessages in the figure), the engine has to inform the RTF about the begin (onBeforeTick) and end (onFinishedTick) of a processing cycle. The clients and other servers send events, e.g., user commands, via RTF, which enqueues them to a list of pending events. The processing of pending events is explicitly invoked from the engine (processIncomingServerEvents), and accesses the event queue (2.1 in the figure). To process the game state (3., G_RunFrame), the engine needs access to the distribution management for the game state (3.1). RTF provides a list of active and shadow entities (getObjects), allows to determine if a certain entity is active or shadow and also to create and introduce new entities (registerActive). The transmission of state updates is automatically handled by RTF during the onFinishedTick call, including creation and transmission of messages. The original Quake sends the state updates explicitly in step 4., SV_SendClientMessages. With RTF, this step is reduced to a simple light-weight notification message that broadcasts the current server frame time which is required by the client engine.

The assignment of an entity to a server is done by registering the entity as active with RTF (registerActive). RTF automatically handles the replication of entities: it provides shadow copies of all entities to the remaining servers. These shadow copies are automatically updated with changes from their active server. RTF explicitly informs the game developer about appearing and disappearing objects, which thus can be handled appropriately.

3.3.1 Distributed Game Rule Processing
Interaction between entities (attacking, touching items or teleports and jump pads) is – besides movement – the main content of one game frame (3. G_RunFrame in Figure 5). When such an interaction is detected, e.g., if entities collide with each other, the original Quake engine invokes a procedure to handle the interaction. For example, a combat interaction, e.g., a rocket hitting a player, is handled by a procedure called G_Damage. This procedure receives as arguments the target (hit player), inflictor (rocket), and attacker (player that fired the rocket) and the caused damage. It applies the damage, creates new entities like explosions, manages scores or issues splash damage for other entities. For the single-server case, this algorithm is quite simple.

The algorithm is more complicated for the distributed case. If the involved entities are all active on the current server, then the procedure can be applied as usual. If an involved entity is a shadow copy, then the interaction has to be handled also by the server responsible for this entity.

Figure 6 shows how the remote interaction is organized for the multi-server Quake. If a server detects a damage interaction, the original G_Damage procedure is invoked (1. in the figure). The procedure determines with the support of RTF if the involved entities are shadow or active entities (2.1, Trap_IsActive). The result of the interaction is directly applied to the active entity (3). For the shadow entity, an event (4.1, DamageEvent) describing the interaction is sent through RTF (4.2) to the server responsible for the entity: RTF automatically determines this server. On this server, the event is enqued by RTF until the Quake engine processes it during the processIncomingServerEvents step of the real-time loop cycle (5.2). The remote server invokes the same G_Damage procedure (6.) with an additional flag which denotes the call as triggered remotely. This is necessary to prevent a loop for shadow interaction handling since this time, the active/shadow roles are swapped. This pragmatic approach is suitable for the Quake engine, since the interaction between entities is deterministic: Each server will apply the same rules based on the arguments of the procedure. Hence, it is not necessary to coordinate the interaction between the servers.

4. QUAKE ENGINE MODIFICATIONS
To use the previously described multi-server architecture for Quake with RTF, we modified and/or exchanged the components of the original Quake (Figure 1). The classes realizing the entities were modified to fit the automatic (de)serialization mechanism of RTF. The Entity Management component was modified to use RTF for the distribution of the game state. Since RTF automatically handles the transmission of events and state updates, the Network Communication component of the original Quake has become obsolete and, therefore, was removed.
4.1 Replacing Network Communication

In the original Quake, communication must be explicitly invoked whenever processing steps create messages and use the network communication component to send them. The concept of RTF is different: while the sending of (user) commands is also invoked explicitly by the application, the sending of state updates is automatically handled by RTF. The RTF interface handles the transmission of application-level objects: RTF provides automatic (de)serialization for C++ objects. The application developer defines usual C++ objects and describes attributes that should be transmitted over the network directly in his code. The ported Quake engine uses this mechanism of RTF also for transmitting user commands. The synchronization of the game state between servers and clients is now handled automatically by the RTF. Like the original Quake – and like almost every online game – RTF uses UDP as transport protocol for the transmission of events and state updates.

4.2 Implementation Issues

We had to solve some technical issues when combining RTF with Quake. Quake was originally written in plain C while the RTF is designed for C++ which is used by most game companies nowadays. After some minor modifications that did not change the semantics of Quake, it was possible to compile the Quake sources with a C++ compiler. However, the separation of game engine and game logic, with the game logic compiled as binary code executed by the Quake Virtual Machine (QVM), could not be preserved. To circumvent this problem, we decided to compile game logic and engine together. Thereby, we lost the ability to use arbitrary modifications (mods) through pre-compiled plugins. However, the Quake modeling community has also decided recently to give up the intention of pre-compiled binary game logic modules: e.g., mods like Urban Terror [5] are delivered as whole binary package including the Quake engine.

Listing 2: Original Quake entity struct.

```c
struct trajectory_t;
struct {
    int number; // entity index
    entityType_t eType; // enum entityType_t
    trajectory_t pos; // for calculating position
    trajectory_t apos; // for calculating angles
    int time;
    float origin[3];
    float angles[3];
    entityType_t eType;
    int otherEntityNum; // shotgun sources, etc
    int groundEntityNum; // -1 = in air [...]
} entityState_t;
```

The Quake entities forming the game state are implemented as plain C structs (Listing 2), which had to be mapped onto RTF classes. The engine accesses and modifies the fields of these structs directly throughout the code. To support the transmission of application-level objects with an automatic serialization mechanism, RTF relies on introspection into the entities, such that changes to a particular entity’s attribute are tracked. In C++ classes created to be used with RTF and following the data encapsulation paradigm, such functionality can be provided by tracking changes in appropriate access methods. One challenge was to find a feasible solution for the modification tracking without changing all accesses to an entity attribute in the original Quake code.

class EntityState : public RTF::Local {
public:
    RTF::ManagedType<int> number; // entity index
    RTF::ManagedType<eType> eType; // entityType_t
    Trajectory pos; // for calculating position
    Trajectory apos; // for calculating angles
    RTF::ManagedType<int> time;
    float origin[3];
    float angles[3];
    RTF::ManagedType<int> otherEntityNum; // shotgun sources, etc
    RTF::ManagedType<int> groundEntityNum; // -1 = in air [...]);
}
```

Listing 3: Quake entity rewritten for RTF.

Our solution is the template class ManagedType that can be used with several primitive types like int; it overloads all necessary operators, such that an instance of ManagedType<int> can be used like a usual int variable with tracked changes (Listing 3).

5. RESPONSIVENESS ANALYSIS

Fast-paced action games like Quake are highly sensitive to the system’s responsiveness [2, 9]: it is critical for players to observe state changes caused by their own or other players’ actions as soon as possible.

5.1 Response Time Model

Our model of response time splits one iteration of the game’s real-time loop (RTL) into several phases accomplished in the game processing system: sending of the user command, processing of the command by the game logic in the server real-time loop (RTL), preparing and transmitting the information over the system network, and notifying the clients of the result. In order to analyze the game responsiveness as precisely as possible, we study the duration of these processing phases. In order to analyze the performance of the software independently of effects caused by the network communication, we also consider the so-called network-adjusted response time, described in the next section.

Figure 7: Response time in Quake.

Figure 7 illustrates our perspective of the response time in Quake: The response time ($t_{resp}$) is the timespan between the moment the user triggers a particular action (e.g., a shot with a rocket launcher in the left screenshot in the figure) and the moment the result is displayed on the client after one iteration of the real-time loop (RTL) has been executed (e.g., the fired rocket becomes visible in the right screenshot).

The absolute response time $t_{resp}$ consists of several phases:

- **User command**
- **Processing of the command**
- **Scheduling**
- **Synchronizing the server state**
- **Notifying the clients**
- **Serialization**
- **Transmission over the network**
- **Decompression on the client side**
- **Reconstruction of the server state**
- **Sending the result**
- **Notifying the clients**
- **Deserialization on the client side**
- **Reconstruction of the local state**
- **Reconstruction of the local state**
- **Scheduling**
- **Processing of the command**

The network-adjusted response time $t_{resp,n}$ is defined as the response time minus the network component.

$$t_{resp,n} = t_{resp} - t_{network}$$
of Quake (right-hand side). Both sides of the figure consist of two parts: client and server. The breakpoints correspond to the method calls performed in the real-time loop; the methods’ names are different for the two implementations, but we enumerate them according to their semantics, such that we can compare the phases of the two implementations. We measure the time at the breakpoints in our experiments described in the sequel. The right-hand side of the figure demonstrates that RTF automatically takes care of important parts of the network communication in the game, which are traditionally managed “manually” by the game developer. For example, at breakpoint 2., the user command is sent to the server using the \texttt{CL\_CreateCmd} method in the original Quake client engine; in RTF, this is done automatically by \texttt{sendEvent}.

In the following, we briefly comment on the main breakpoints and phases in the model. The start of a user action is marked by the creation of the user command in Quake (1. \texttt{CL\_CreateCmd} in Figure 8); the first phase denoted by $t_{cmd}$ is the creation of the command message which includes wrapping of the command in a network-transmittable form; it ends with the dispatching of the message to the operating system network stack (2.). The next phase – transmission of the command message over the network – ends when the message is unwrapped after it is received from the OS network stack (3.). The phase starting from (3.) is denoted by $t_{event}$ and ends with the parsing of the user command (4.). The actual processing of the command, denoted by $t_{create}$, ends when the rocket entity is created (5.). Afterwards, the update of the game state for the client is prepared, $t_{update}$, and finished by the dispatching of the update message to the OS network stack (6.). On the client side, the update phase $t_{snap}$ begins upon receipt of the update message (7.) and ends with applying the content of the update message to the client’s game state (8.). The final breakpoint is the appearing of the rocket entity on the client display (9.).

To calculate the network-adjusted response time, we measure the time at breakpoints: $T_1, \ldots, T_9$, and accumulate the above described phases as follows:

$$
\begin{align*}
    t_{cmd} &= T_2 - T_1, \quad t_{snap} = T_8 - T_7, \quad t_{disp} = T_9 - T_8 \\
    t_{event} &= T_4 - T_3, \quad t_{create} = T_5 - T_4, \quad t_{update} = T_6 - T_5 \\
    t_{client} &= t_{cmd} + t_{snap} + t_{disp} \\
    t_{server} &= t_{event} + t_{create} + t_{update} \\
    t_{process} &= t_{server} + t_{client}
\end{align*}
$$

Here, $t_{client}$ denotes the client processing portion and $t_{server}$ the server processing portion, which together constitute the network-adjusted response time $t_{process}$. The total response time $t_{resp}$ can obviously be measured as the difference between $T_9$ and $T_1$. The relation between $t_{resp}$ and $t_{process}$ can be approximated as doubled latency between client and server, denoted by $RTT$ (Round Trip Time):

$$
t_{resp} = T_9 - T_1 \approx t_{process} + RTT \quad (1)
$$

\subsection*{5.2 Experiments on responsiveness}

To compare the responsiveness of the original Quake with the RTF-based multi-server port of Quake, we examine the use case of a player firing a rocket. We measure the times $T_1$ till $T_9$ which constitute the model in Section 5.1.

We conducted two experiments: 1) server and client running on the same machine, and 2) server and client running on different machines connected either directly via LAN or via Internet. We used the first experiment to compare RTF and original Quake with respect to the network-adjusted response time ($t_{process}$). The second experiment allows to study the absolute response time ($t_{resp}$) in comparison to the network-adjusted.

\subsubsection*{5.2.1 Network-adjusted response time}

Figure 9 shows the measurement results for the phases of network-adjusted response time for client and server running on the same machine. For both implementations, the largest amount of time is spent in the phase $t_{disp}$ on the client side, where the client integrates the appeared entity to the graphics engine. For the server part, we observe a difference between the two implementations: the phase $t_{update}$ is comparatively long for the original Quake, whereas for the

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure8.png}
\caption{Network-adjusted response time model.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure9.png}
\caption{Response time, RTF vs. original.}
\end{figure}
ported version the \( t_{\text{event}} \) phase takes most time. This is due to the fact that the length of one processing frame in Quake is fixed (50ms). An update for the game state is sent at the end of the frame, and hence is independent from when the event actually arrived in the server tick. For the two implementations the point when the event is processed is different (RTF: at the end of the frame, original: directly upon arrival). However, for both versions the processing time in the server was always within one tick.

### 5.2.2 Absolute response time

To evaluate the impact of the network performance, we conducted a second test series where we observed the absolute response time over high-latency network connections.

**Figure 10:** \( t_{\text{resp}} \) for Quake using RTF over different networks.

Figure 10 shows the measured response time \( t_{\text{resp}} \) compared to the accumulated client and server processing time \((t_{\text{client}}, t_{\text{server}})\) obtained using our adjusted model and the round trip time (RTT) between client and server using RTF. The RTT was measured using the Linux `ping` tool. The results show that for the Quake port using RTF the measured response time is equal to the sum of the network-adjusted response time and the RTT, which verifies the estimation (1) obtained from the model in Section 5.1. Since we used different machines for the different network scenarios, the values for the processing time vary.

### 6. SCALABILITY ANALYSIS

While RTF is a generic middleware with high-level support for multi-server implementation of online games, our main objective in this paper is to use multiple servers for scalability, i.e., to increase the maximum number of players per session by increasing the number of servers. In this section, we study the scalability of the described multi-server Quake port using the RTF replication approach.

As a metric for the overall game performance, we use the saturation of the processing real-time loop, i.e., the percentage of time the server is active during a tick. This is a feasible metric to determine how well the server can cope with the current number of clients, while the frame rate of the server is fixed at 20Hz. To observe the saturation of the server real-time loop, we introduced several measuring points in this loop. As explained in Section 2, one iteration of the loop consists of three main steps: waiting till the next server frame starts (\( \text{NET} \_\text{Sleep} \)), processing of the game state (\( \text{G} \_\text{RunFrame} \)), and the update of the game state to the clients (\( \text{SV} \_\text{SendClientMessages} \)). Therefore, the active time of the server during a tick can be measured as the duration of the processing and update step. Since the measurement points show the accumulated execution time of all methods in the real-time loop, they are different from the measurement points of the responsiveness analysis model.

In our experiments, for a particular number of servers, we examine the real-time loop saturation while connecting more and more clients. We tested our implementation with several different self-created maps and with original Quake maps. For the scalability test, most of the players were simulated using `simbots` which automatically target other players, move the own avatar and issue attack commands. The bots were configured to approximate typical Quake game play. Figure 11 shows a screenshot of our experimental game session with 38 clients running in a small testmap. We used P4 1.7Ghz processors with 500MB of RAM as servers. All machines were connected via LAN (100 MBit/s).

**Figure 11:** Screenshot of Quake using RTF.

In Figure 12, the saturation of the real-time loop is plotted against the number of clients per session. A saturation over 100% means that the server’s active time exceeded the predefined frame time of 50 ms. All values were measured on the same map (q3dm4). Similar values can be observed for other maps. The curves in the figure show the average saturation over several measurements. For one server using the RTF version, we also show the actually measured values, each as a single dot. For the multi-server case, we accumulated the average saturation over the 2, 3, and 4 servers while adding the total amount of clients. The clients were evenly distributed among the servers.

With the ported RTF version, it was possible to handle up to 38 clients using a single node until the real-time loop became saturated, and about 54 clients with 2 servers, 66 and 84 clients with 3 and 4 servers, correspondingly. With the original Quake engine, a single server handling 64 clients was about 90% saturated. The maximum number of participating clients is technically limited to 64 for the original Quake engine. In our ported version, each server can handle more than 64 clients, so we extended the technical limit for the overall maximum number of clients per session and with 4 servers.
servers it is possible to support more clients than with the original single-server version.

Figure 13 shows the scalability of the RTF implementation, i.e., the maximum number of players observed in our experiments. The solid line shows the overall number of clients for 1 to 4 servers; the dashed line shows the maximum number of clients per server. With 4 servers, a single server was capable to handle 21 clients instead of 32 clients with a single server. In our experiments, we were limited by the available number of computers, which did not allow us to start more than 85 clients simultaneously: since each bot client needs a high-performance graphical client interface, it was only possible to run few bots on a single machine.

7. RELATED WORK AND CONCLUSION

In comparison to existing approaches in the field of communication middleware like Net-Z [12] or RakNet [13], RTF provides a much higher level of abstraction featuring automatic entity serialization and hides nearly all of the technical network communication aspects. On the other hand, RTF leaves the real-time loop implementation to the developer, supported by the high-level entity and event handling mechanisms. RTF allows to easily incorporate three different parallelization and distribution approaches and is open to be extended to future approaches. This flexible support of different parallelization concepts allows RTF to be usable for a wider range of MMOG concepts than existing multi-server middleware like Emergent Server Engine [4] or BigWorld [3], which mostly support the concept of zoning.

We described a successful port of the Quake 3 engine using our RTF middleware. Besides the port itself, the main contribution of our work is the parallelization of the game processing which allows to employ multiple servers in the game and thereby improve the game’s performance. We use a new distribution method – replication – which extends the possibilities of traditional zoning and instancing. Moreover, we extensively studied the responsiveness and scalability of the multi-server solution. Our first experiments show that the responsiveness of the Quake port using RTF can compete with the original version. The results of the scalability analysis show that the multi-server approach allows to scale this game to higher player numbers. Our results are promising and will be used to further improve both the RTF middleware and the multi-server Quake 3 implementation.

8. ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their remarks on the preliminary version of the paper.

9. REFERENCES


Figure 12: Real-time loop saturation vs. number of clients.

Figure 13: Scalability of Quake on multiple servers.