Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2008, Article ID 327387, 16 pages
doi:10.1155/2008/327387

Research Article

High-Level Development of Multiserver Online Games

Frank Glinka, Alexander Ploss, Sergei Gorlatch, and Jens Müller-Iden

Department of Mathematics and Computer Science, University of Münster, 48149 Münster, Germany

Correspondence should be addressed to Frank Glinka, glinkaf@uni-muenster.de

Received 2 February 2008; Accepted 17 April 2008

Recommended by Jouni Smed

Multiplayer online games with support for high user numbers must provide mechanisms to support an increasing amount of players by using additional resources. This paper provides a comprehensive analysis of the practically proven multiserver distribution mechanisms, zoning, instancing, and replication, and the tasks for the game developer implied by them. We propose a novel, high-level development approach which integrates the three distribution mechanisms seamlessly in today's online games. As a possible base for this high-level approach, we describe the real-time framework (RTF) middleware system which liberates the developer from low-level tasks and allows him to stay at high level of design abstraction. We explain how RTF supports the implementation of single-server online games and how RTF allows to incorporate the three multiserver distribution mechanisms during the development process. Finally, we describe briefly how RTF provides manageability and maintenance functionality for online games in a grid context with dynamic resource allocation scenarios.

1. INTRODUCTION

The development of large-scale, *massively multiplayer online games* (MMOGs) is significantly more complex as compared to the development of casual online games for small user numbers. Since a single server is not capable of serving the high user numbers and the related game processing of contemporary MMOGs, a multiserver design becomes mandatory. This leads to a number of new, complicated design issues, including efficient multiserver communication, object migration between servers, distributed client connection handling, synchronization of data across servers, load balancing, latency issues, and others. A proper addressing of these aspects within the game development process requires high in-house expertise and employs low-level programming and network tools, which makes it time-consuming, risky, and often expensive.

This paper first analyses the basic structure of today's online games and develops a taxonomy of the currently employed development approaches. Based on this analysis, we then describe a novel, high-level development approach that aims at simplifying the development process and improving its productivity while maintaining a high level of flexibility for the developer. Our approach suits a wide spectrum of online games: from traditional single-server games to multiserver MMOGs, with the possibility to enhance a single-server design to a multiserver, multiplayer game. We describe in the detail *Real-Time Framework* (RTF)—a middleware system that supports the proposed high-level approach to game development. RTF, whose first architecture approach and offered development methodology has been introduced in [1, 2], provides integrated solutions for a variety of development and run-time problems. These solutions include the basic communication for a variety of development and run-time problems, ranging from the basic communication functionality to single-server online games to sophisticated solutions for the distribution management of MMOGs. This includes in particular the object-oriented design and efficient transmission of game data structures, interfaces and services that allow the developer to efficiently process an MMOG on multiple servers, integrated monitoring and controlling functionality for games, and management possibilities for the multiserver distribution of MMOGs.

Although there has been previous work targeting most of these problems individually [3, 4], there is a high demand of integrated solutions available as high-level libraries and middleware systems for broad classes of online games. Our RTF-based approach addresses a large variety of online game types, ranging from fast-paced and small action

games to large-scale MMOGs. Moreover, our development approach employs modern Grid computing technologies [5] to facilitate the dynamic usage of system resources, accordingly to changing user demands. This paper presents a comprehensive overview of RTF, motivating and describing its high-level development approach. We present in detail the different parallel processing models offered by RTF for scaling MMOGs, and describe use cases for RTF and how it can be used in a grid environment for on-demand provision of resources. Furthermore, we introduce several new application demonstrators built on top of RTF and report the results of first performance and scalability tests.

The contributions and the structure of the paper are as follows. We describe the basic design of online games that are based on a *real-time loop* in Section 2 and provide a comprehensive analysis of the current game development approaches, with respect to their complexity and flexibility in Section 3. We outline our high-level game development approach, describe the concepts of the supporting RTF middleware, and give an overview over RTF as a development tool in Sections 4 and 5. We show how RTF is employed for multiserver game processing and give an extensive overview of practically proven multiserver distribution mechanisms and the tasks implied for their management in Section 6. We demonstrate how RTF allows to realize a seamless virtual environment and the seamless transfer of game parts between resources in Section 7 and explain how RTF could be used in a grid context with dynamic resource management in Section 8. The paper is concluded by a detailed report on several prototype applications including experimental scalability tests in Section 9 and by summarizing our contributions in the context of related work in Section 10.

2. BASIC DESIGN OF ONLINE GAMES

The majority of today's online games typically simulate a spatial virtual world which is conceptually separated into a static part and a dynamic part. The static part covers, for example, environmental properties like the landscape, buildings, and other nonchangeable objects. Since the static part is preknown, no information exchange about it is required between servers and players. The dynamic part covers objects like avatars, *nonplaying characters* (NPCs) controlled by the computer, items that can be collected by players or, generally, objects that can change their state. These objects are called entities and the sum of all entities is the dynamic part of the game world. Both parts, together, build the game state which represents the game world at a certain point of time.

For the creation of a continuously progressing game, the game state is repeatedly updated in real time in an endless loop, called *real-time loop*. Figure 1 shows one iteration of the server real-time loops for multiplayer games based on the client-server architecture. The figure shows one server, but in a multiserver scenario this may be a group of server processes distributed among several machines. A loop iteration consists of three major steps. At first the clients process the users input and transmit them to the server (step 1 in the figure). The server then calculates a new game state

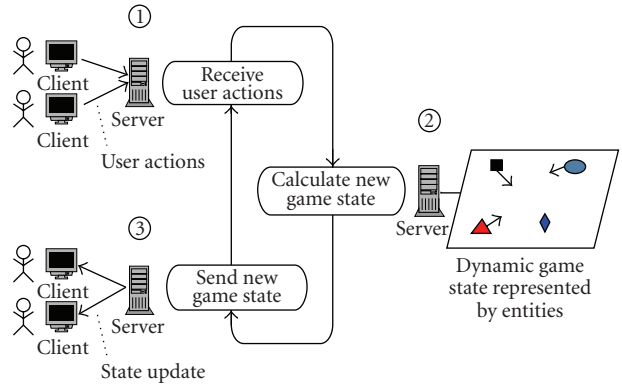


Figure 1: One iteration of the server real-time loop.

by applying the received user actions and the game logic, including the *artificial intelligence* (AI) of NPCs and the environmental simulation, to the current game state (step 2). As the result of this calculation, the states of several dynamic entities have changed. The final step 3 transfers the new game state back to the clients.

When realizing the real-time loop in a particular game, the developer has to deal with several tasks. In steps 1 and 3, the developer has to transfer the data structures realizing user actions and entities over the network. If the server is distributed among multiple machines, then step 2 also implies the distribution of the game state and computations for its update. This brings up the task of selecting and implementing appropriate distribution concepts.

3. GAME DEVELOPMENT APPROACHES

The central part of a game software system consists of the *game state* and the continuous *processing* of the game state. In this paper, we focus on the development of the game state and its processing, rather than on the game user interface, that is, the representation of the virtual environment the player interacts with.

In order to compare different development approaches of the overall distributed architecture of online games, we identify the following three major aspects of online game software systems:

(i) game logic: *entities*, *events* (data structures), and processing rules describing the virtual environment;

(ii) game engine: *real-time loop* which continuously processes (user) events, according to the rules of the game logic, to compute a new game state;

(iii) game distribution: logical partitioning of the game world among multiple servers, computation distribution management according to actual game state, and communication.

The third aspect, game distribution, can be further split up into two levels of distribution: (a) distribution of the user interface and game state processing between client and server, and (b) distribution of game state processing in the multiserver architecture.

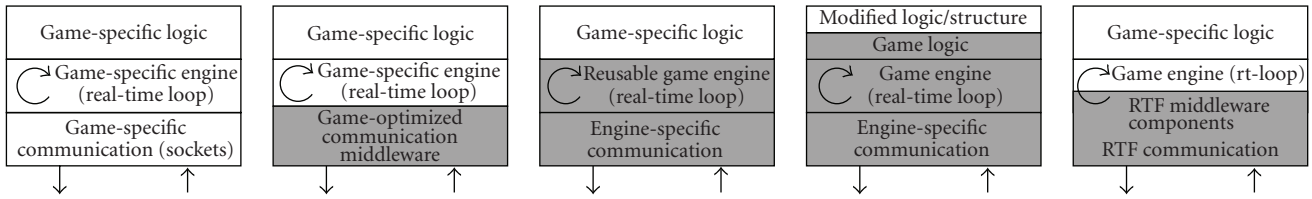


Figure 2: Main approaches to game development. White: self-developed; Gray: using existing components.

These three aspects are treated differently, depending on the requirements and properties of a particular game genre. For example, fast-paced action games rely on efficient communication and engine implementation while using only relatively simple game logic and content. The complexity of the game distribution aspect usually increases with the number and density of the participating users within a game and is thus particularly challenging for MMOGs.

Our classification in Figure 2 distinguishes common approaches (a)–(d) to game development, according to how they treat these three aspects. In each approach, the aspects shown in white are managed by the human developer, whereas the shaded areas are provided automatically by the development system.

(a) Custom development

The most direct approach used for game development is to design and implement the entire software system individually. The development team designs and implements all three major aspects of the game software system: game logic, game engine, and game distribution. This allows the developers to have full control over their code and optimized implementation with focus on the individual performance needs of the game. While the custom development of an entire game is very complex, hence cost-intensive and error-prone, it is sometimes the only way to achieve the particular objectives of the game design because of its flexibility.

(b) Game communication middleware

This approach uses special communication libraries and middleware systems (like Quazal Net-Z [6]) for game development. As shown in Figure 2(b), the game developer employs the middleware to realize the communication between clients and servers in a distributed game while implementing the game engine and logic on his own. Using this approach, the developer has enough flexibility to design and implement the aspects of game logic and game engine while the middleware deals with the game distribution. However, available libraries usually focus on a particular architecture setup, decreasing flexibility of the engine development. Furthermore, this approach has been used only rarely for the development of multiserver-based MMOGs since a pure communication library is not sufficient for these games. A middleware for MMOGs also has to deal with the difficult task of distributing the game processing among multiple servers, for which only a few middleware systems are available (e.g., Emergent Server Engine [7] or BigWorld [8]).

(c) Using existing engine

With this approach, shown in Figure 2(c), an existing game engine, that is, the processing component of a game, is reused to develop a completely new game. This reduces the complexity of development. Some game studios design their game engines primarily for the purpose of reselling and licensing the engine afterwards. Examples of popular and often used engines are the *Quake 3* engine or the *Unreal* engine. However, a particular engine is quite inflexible because it is usually closely tied to a specific game genre.

(d) Game modding

Figure 2(d) outlines the approach of game modding (community jargon for modifying an existing game) via a dedicated interface for programming the game logic. This was first done by hobby developers who modified the actual game content. Nowadays, the creation of mods is based on high-level tools created and also used by the game development studios themselves. Such tools allow the creation of game content by designers with minimal programming effort. The primary aspect of modding is the creation of new game content within the constraints of an existing game logic; hence it is rather inflexible. Nevertheless, modding allows to develop innovative game concepts, and sometimes a mod becomes even more popular than the original game as, for example, the mod *counter-strike* based on Valve's game *Half-Life*.

(e) RTF multiserver middleware

Our real-time framework, as illustrated in Figure 2(e), allows a novel game development approach which provides more processing support than using only a communication middleware, but does not constitute a complete game engine, allowing higher flexibility. Thus, RTF can be classified in between the approaches (b) and (c). The characteristics and usage of RTF, justifying this classification, are discussed in the next sections.

Figure 3 illustrates our taxonomy of the five discussed development approaches with respect to their flexibility and complexity. The most simplicity in terms of distributed

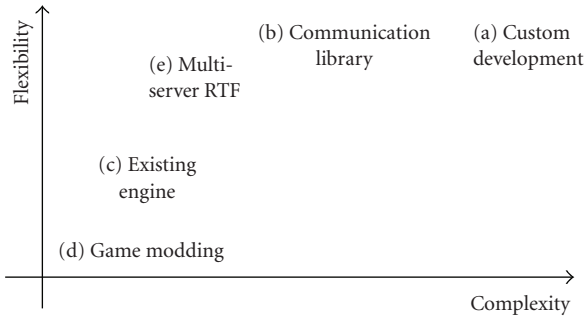


Figure 3: Taxonomy of game development approaches.

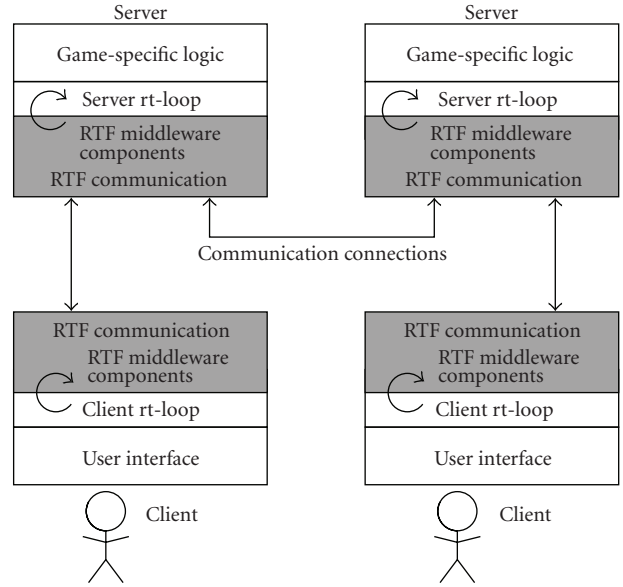


Figure 4: RTF multiserver middleware.

software infrastructure is offered by existing game engines (c) or modding toolkits (d). However, these approaches have the remarkable drawback of being quite inflexible. Obviously, the fully custom development (a) offers most flexibility while being rather complex. The use of special middleware (b) is a promising alternative for particular tasks: its use reduces the complexity of game development. Pure communication support is not enough for MMOGs: for such large distributed systems, the multiserver management is quite extensive and increases the development complexity. As indicated in the taxonomy, RTF is designed to provide the developer with the highest possible flexibility in game design while freeing him from complex low-level implementation tasks in the game development process.

4. RTF OVERVIEW

The real-time framework provides a high-level communication and computation middleware for single-server and multiserver online games. RTF supports both the server-side and client-side processings of an online game with a dedicated set of services which allows developers to implement their optimized engine at a high, entity-based level of abstraction in a flexible manner. Figure 4 shows a generic multiserver example of a game developed on top of RTF.

The RTF middleware deals with entity and event handling in the real-time client loop and the continuous game state processing in the real-time server loop, and the distribution of the game state processing across multiple real-time server loops. The developer implements the game-specific real-time loop on client and server, as well as the game logic, using the RTF middleware to exchange information between the processes.

RTF is based on a modular approach and provides additional components for other aspects of distributed games besides the game processing aspect. Figure 5 shows the components that exist beside the *communication and computation parallelization (CCP) module* which handles the game processing part. The shown components include a module for the persistent storage of game-related information, which is especially relevant for MMORPGs. The *persistency module* allows storing and retrieving entities specified by the application developer in and from a relational database. An *audio streaming module* supports Voice over IP (VoIP) audio communication over RTP (real-time protocol) and provides an interface for setting up channels, switching users between them and to unmute users. Other important aspects of MMOGs, which typically have long-running game sessions, are monitoring and controlling possibilities. The *controlling and monitoring module* is the middleware-endpoint for application developers to receive commands for steering the application at runtime and to report internals about the application status. The developer can define profiles which reflect game-specific controlling and monitoring characteristics on top of this module. There also exist predefined profiles for typical monitoring metrics and controlling tasks in games, and RTF supports some of these predefined modules, for example, reports RTF-internal values like communication characteristics (bandwidth consumption, latency, packet rate) and distribution characteristics (number of clients, number of entities, number of exchanged events), such that the application developer is not required to report such information explicitly.

A modular approach allows RTF to be extensible without a major impact on the existing parts of the RTF middleware. Furthermore, the developer only gets in touch with the modules he wants to use. The remainder focuses on the most important CCP module and explains how the game processing is realized with this module.

5. GENERAL DEVELOPMENT TASKS

The development of the game state processing in online games consists of several tasks, as shown in Figure 6. Regardless of developing a multiserver MMOG or a single-server, small-scale action game, the developer has to care about three *general tasks*, AoI management, game state processing, and data-structure design, when building the game on top of RTF. If the game uses multiple servers, then multiserver

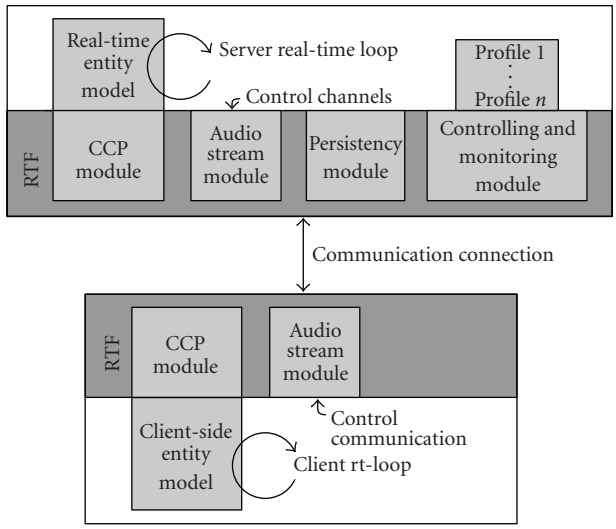


Figure 5: Modular approach of RTF.

parallelization and distribution also have to be taken care of by developers. Underneath these tasks for the developer, RTF provides a variety of low-level functionality like optimized event and entity serialization and communication, management of the game state and its possibly distributed processing. Overall, this separation of tasks among the developer and RTF reflects the overall approach of RTF sketched in Figure 2(e). Providing high-level game-engine-related functionality on top of an optimized communication middleware. The following subsections focus on the developer tasks and present the overall development methodology provided by RTF.

5.1. Task (1)—data structure design

The dynamic state of an online game is usually described as a set of *entities* representing avatars, NPCs, or items in the virtual world. Beside entities, *events* are the other important structure in an online game engine for representing user inputs and game world actions. Hierarchical data structures for events and entities in complex game worlds have to be serializable in an optimized manner for efficient network communication. When using only a communication middleware, developers have to build data structures and serialization mechanisms from scratch, while using an existing engine requires the use of predefined entities and events, which reduces flexibility.

RTF provides an optimized high-level entity and event concept enabling automatic serialization while still providing full design flexibility. When using RTF, entities and events are implemented as object-oriented C++ classes. The developer defines the semantics of the data structures according to the game logic. The only semantics of entities that is predetermined by RTF is the information about their position in the game world. Entities, therefore, are derived from a particular base class `Local` of RTF that defines the representation of a position for entities. This is necessary since the distribution of the game state processing across multiple servers is based upon the location of an entity in the game world. Besides the requirement of inheriting from `Local`, the design of the data structures is completely customizable to the particular game logic, as illustrated by the example of a racing car entity shown in Algorithm 1.

In order to enable platform independence, RTF defines primitive data types to be used (e.g., `rtf_int8`). Also, easy-to-use complex data types for vectors and collections are provided to the developer. Overall, more complex entity and event data structures can be easily defined using these primitives.

5.1.1. Automatic serialization and network transmission

In online games, entities and events are continuously transmitted over a network. Therefore, these hierarchical data structures have to be serialized in an optimized manner. However, there is no standard serialization mechanism in C++, such that the developer has to define and implement a network-transmittable representation for each entity and event of a game when using a traditional communication middleware. As an alternative, most engines provide high-level scripting capabilities with automatic serialization, but they decrease flexibility and possibly also performance due to the abstraction overhead from native C/C++.

RTF provides automatic and native serialization of the entities and events defined in C++, implements marshalling and unmarshalling of data types, and optimizes the bandwidth consumption of the messages. RTF solves this problem for the developer by providing a generic communication protocol implementation for all data structures following a special class hierarchy. All network-transmittable classes inherit from the base class `Serializable` of RTF. The `Serializable` interface can be (a) implemented by the developer, or (b) automatically implemented using the generic serialization mechanism provided by RTF. This automatic implementation is generated using convenient pre-processor macros provided by RTF. For all entities and events implemented in this manner, RTF automatically generates network-transmittable representations and uses them at runtime.

The generic serialization mechanism of RTF supports the following hierarchies:

(i) primitive data types (e.g., `float` or `std::string`);

(ii) serializable objects;

(iii) pointer to Serializable objects;

(iv) containers of Serializable pointers (e.g., std::vector),

(v) inheritances hierarchies of Serializables.

Serializable classes to be implemented with the generic serialization mechanism can have primitive data types as attributes. Serializable classes can be used for further derivation, for example, to form hierarchies of entities. Furthermore, classes derived from `Serializable` as well as pointers to Serializables can be used as attributes, see `_engine` or `_item` in Algorithm 1. The support for serializing a pointer to Serializable objects allows to realize

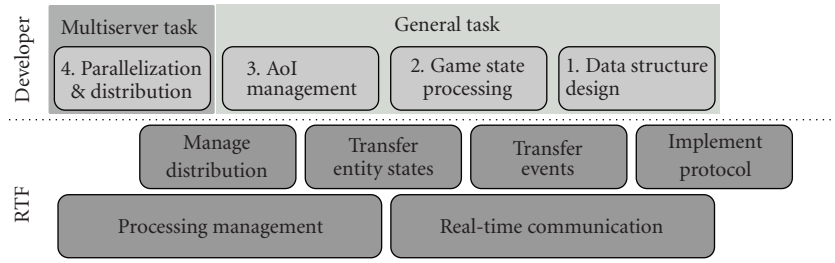


Figure 6: Development tasks for a multiserver game: distribution between RTF and developer.

```
class Engine        :public RTF::Serializable;
class Team          :public RTF::Serializable;
class PowerupItem   :public RTF::Local;

class MovingEntity  :RTF::Local{
    RTF::Vector   _velocity;
    RTF::Vector   _orientation;
};

class RacingCar     :public MovingEntity   {
    std::string    _driver;   // name of driver
    rtf_int8       _fuel;   // fuel level
    Engine         _engine;   // car's engine
    PowerupItem* _powerup;   // a specific item
    Team*     _team; // associated entity
};
```

Algorithm 1: An entity written in the manner of RTF.

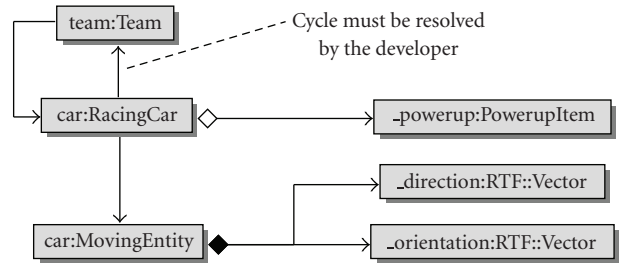


Figure 7: Example of an object graph supported by the generic serialization mechanism of RTF.

aggregation and associations in general. Support for using `Serializable` objects directly allows to map *composition*, that is, aggregation by value. It is also possible to use STL containers of `Serializable*` to express one-to-many associations. The number and depth of associations in the object graph is not limited by the generic serialization mechanism. However, a problem occurs when the object graph contains cycles. Such cycles are not automatically detected by the serialization mechanism, such that the developer must explicitly resolve them by denoting such associations as an *entity pointer*. An entity pointer is only serialized by the generic serialization mechanism as the ID of the referenced Serializable. For the developer of the entity hierarchy, it is not difficult to identify such associations. An object referenced as entity pointer needs to be treated like an entity: it must be manually subscribed to the AoI of a client, as described in Section 5.3.

The object graph shown in Figure 7 depicts associations for the example entity of Algorithm 1 and illustrates the object association treated by RTF's generic serialization mechanism. An entity of type `RacingCar` may have a `PowerupItem`, which will be serialized along with the `car` object. When serializing the `car` object, the generic serialization mechanism automatically deals also with the attributes of the `MovingEntity` base class. The graph contains a cycle as the example has a bidirectional association of `Team` and `RacingCar`—a car belongs to a certain `Team` whereas a `Team` is likely to have a list of cars. The `Team` itself needs to be available on the client independently from the player car. Therefore, it is easy for the developer to identify this fact and treat these associations as entity pointers, thus resolving the cycle.

For scenarios that forbid inheritance from the `Serializable` base class, the generic serialization mechanism provides the possibility to wrap entities within a generated, serializable version of themselves. This is, for example, necessary for the integration of RTF into an existing game engine with a specific inheritance structure. Although this wrapping imposes a slight code overhead to the default mechanism, it is still easy to use by developers and only has a minimal performance overhead.

Overall, this approach allows to use native C++ data structures for entities and events, while avoiding to implement the cumbersome, network-specific serialization by hand. Additionally, our approach is open to be combined with custom, engine-specific scripting capabilities, for example, LUA-bindings [9] for high-level behavior scripting can easily be added into the C++-based core data structures.

5.2. Task (2): game state processing

The central notion of our approach to the game development using RTF is the *real-time loop*, in which game states are updated. Most contemporary multiplayer games are based on such a loop, whose individual updates are called *ticks*. RTF allows the game developer, on the one hand, to implement his own real-time loop in the well-understood manner and, on the other hand, delivers him a substantial support for

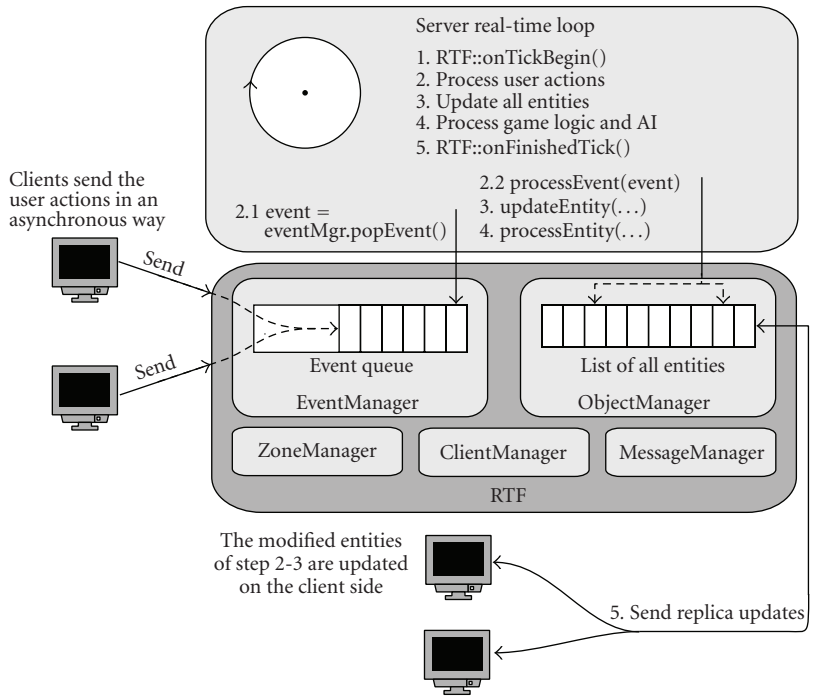


Figure 8: Server side game state processing integration with RTF.

implementing and running this loop on both the server- and client sides. RTF provides various manager classes accessible directly from his real-time loop as illustrated in Figure 8; these classes automatically manage several low-level aspects of an online game.

Figure 8 illustrates how the individual steps of the real-time loop are implemented by the game developer on the server side using RTF.

(1) First, the developer notifies RTF about the begin of a new tick by a call to `RTF::onTickBegin()`. Within this call, various low-level tasks are handled by RTF for the developer: incoming user actions are enqueued in the EventManager, new incoming client connections are dispatched and, if the game state is shared across multiple servers, then also game state updates from remote servers are applied automatically.

(2) The developer now processes the user actions received by the clients: the actions are retrieved from the EventManager, and the game state is updated as the reaction to the user actions according to the game rules. The ObjectManager keeps track of the game state, such that game entities are continuously added to or removed from the ObjectManager.

(3) In the third step, the entities are updated according to the game rules. In certain games, this may fall together with the next step.

(4) Artificial intelligence(AI), game logic and other update computations are performed. Some of these steps, like AI, might not happen in every tick.

(5) Finally, the developer notifies RTF about the end of the tick by a call `RTF::onFinishedTick()`. RTF executes most of its low-level runtime communication and distribution tasks within this call, including updates of the game state at remote clients. In the multiserver case, updates are also transferred to remote servers and modifications of the distributed game processing are handled; for example, connections to new servers are created and new servers are integrated into the game.

The real-time loop on the client side, illustrated in Figure 9, looks similar to the one on the server side, but works with a specific client side version. The developer has to perform the following steps in his client real-time loop.

(1) Similar to the server, a call to `RTF::onTickBegin()` is required at the begin of a tick. Incoming events sent by the server are enqueued and incoming game state updates are applied.

(2) The developer implements processing of the newly arrived server events, for example, incoming chat messages or notifications about in-game events.

(3) Typically, the server is not able to update the game state frequently enough to allow rendering of a fluent game progress. Therefore, game engines often use various interpolation and prediction techniques to compensate the low update rate of the server [10].

(4) The rendering step updates the visual screen of the player to show the updated game state. Also sound output and player input can be processed in this step.

(5) The developer notifies RTF about the end of the tick by a call `RTF::onFinishedTick()`.

This is the basic structure of the client real-time loop, although particular game designs may exclude some tasks from the real-time loop. Our approach works well with such

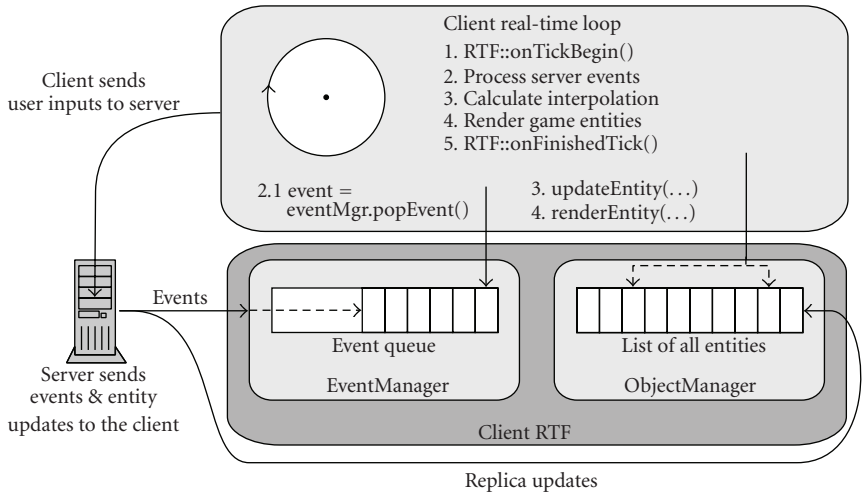


Figure 9: Client side game state processing integration with RTF.

games too, because loop-external tasks are synchronized with the game state which is still managed within the real-time loop.

This schema of integrating a communication and distribution middleware into the continuous processing of the game state is an important finding of our studies on providing a distribution middleware for online games: it frees the developer from low-level network programming which is required when using a conventional communication middleware. At the same time, our solution still provides full design flexibility for the real-time loop as opposed to approaches that use an existing game engine with a predefined processing loop.

5.3. Task (3): AoI management

An *area of interest (AoI)* concept assigns each avatar in the game world a specific area where dynamic game information is relevant and thus has to be transmitted to the avatar client. AoI optimizes network bandwidth by omitting irrelevant information in the communication. If done in a fine-granular manner, it avoids wallhack-like cheating [11, 12] at the client side which makes walls semitransparent and reveals hidden opponents outside of the AoI. Unfortunately, determining the relevant set of entities for a particular client can be quite compute intense, such that the AoI management, for which different algorithms are compared in [13], has to be implemented in an efficient and optimized manner.

RTF supports the custom implementation of arbitrary AoI concepts by offering a generic publish/subscribe interface for interentity visibility. The engine determines continuously which entity is relevant for a client avatar and notifies RTF of each change of an "interested" relation through a `client.subscribe(···)` and `client.unsubscribe(···)` call. RTF automatically takes care that the entity is available and always updated at the client or is removed from the client, respectively. RTF also takes care that entities are removed from the AoI of all participating clients if an entity disappears at a certain server as a result of this entity's movement from one zone into another and thus may be leave the AoI of clients implicitly.

5.3.1. Transferring entity states

Every time the game engine has finished the processing of a new game state, the RTF automatically synchronizes the state of entities between the distributed processes depending on the indicated AoI relations. When an entity is replicated to another process (e.g., all the entities within the AoI of a particular avatar to its client), the state of the remote copy has to be updated. Since the computations are usually performed in repeatedly executed cycles (*ticks*), the best way to perform state updates is after a computation cycle has finished, thus preventing propagation of intermediate states and read-write conflicts between the middleware and game engine.

The use of RTF simplifies this task of transferring entity states for the developer. He only has to inform the middleware that a computation cycle of the game engine has ended by invoking `RTF::onFinishedTick()` (step 5 in Figure 8). The necessary flow of information to update the game state on all participating processes is determined by the RTF upon the specified distribution. At runtime, the middleware automatically creates messages for changed objects and transmits them. This is done using the network transmittable representations that were generated for the data structures by the RTF preprocessor macros during the development cycle. The RTF part of the receiving process of such an update message automatically determines the object related to the messages and writes the updated data directly to the right object. Since the data is directly written to the objects used inside the game engine, this writing step is again triggered by the developer, for example, directly before a

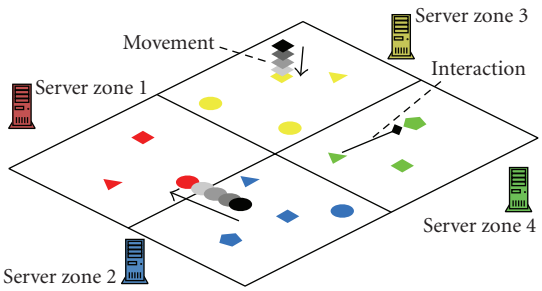


Figure 10: A zoned world distributed on four servers.

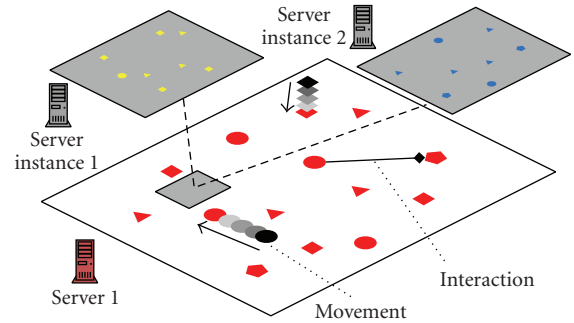


Figure 11: A virtual world with one instanced area processed independently by two servers.

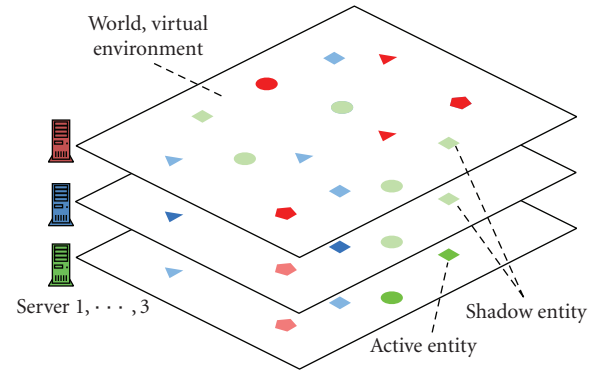


Figure 12: A zone that is cooperatively processed by three servers.

computation cycle, by invoking `RTF:onTickBegin()` (step 1 in Figure 8), to prevent read-write conflicts.

6. MULTISERVER TASKS

The general development tasks described in Section 5 are fundamental for any client-server-based game. However, in the case of massively multiplayer games, a multiserver approach is required for achieving high scalability. This section describes parallelization approaches supported by RTF and discusses how developers can easily use them for building MMO worlds.

6.1. *Parallelization concepts supported by RTF*

RTF currently supports three parallelization concepts and their free combination for scaling virtual world environments: *zoning*, *instancing*, and *replication*.

Zoning [14–16] partitions the virtual world into disjoint parts, called zones, and assigns each zone to one server. Figure 10 shows a virtual world with four zones on four servers and the clients and entities that move and interact within these zones. Although clients can move between zones, no interaction between clients in different zones is possible. Zoning is commonly used in contemporary MMORPGs like *EVE Online* [17] and *World of Warcraft* [18] and allows large player numbers in such games. The zoning approach fits best for games where the players are reasonably distributed in a very large virtual world.

The zoning concept requires that clients are always connected to their responsible server, that is, the one processing the client's zone. RTF performs run-time checks for all clients if this condition is met and transfers clients automatically to their responsible server. Such a transfer is completely transparent on the client-side RTF and only causes a notification on affected servers.

Instancing creates multiple copies of special parts of the game world. Figure 11 shows how a small area (the grey rectangle) is processed in separate copies by two different servers.

Instancing in online games can come in two flavors: complete zones can have several independent copies, which can be accessed by any player, and players that enter an instanced zone have to choose one particular copy. This type is usually not very appreciated by players, because it destroys the illusion of playing in a single world. The second flavor is to have instances for smaller areas in the game world and an instance is created simply by players, or groups of players, entering such an area. Such an area could be a very exciting dungeon and if a player enters it, he gets his own copy for the time he is in the dungeon. The copy is destroyed when the player leaves the dungeon. This second flavor is heavily used by MMORPGs.

Both flavors of instancing are supported within RTF. A client that enters an instanced area—depending on the instance flavor—either triggers an automatic instance creation within RTF or an existing instance must be specified as the transfer target. Subsequently, the client is automatically transferred by RTF to the server that is responsible for the new or the specified instance.

Replication [4, 19] is an alternative parallelization approach recently discussed in academia. Figure 12 shows three servers which cooperatively process the same virtual world zone. Each of the servers replicates his data in a symmetric manner and each server is responsible for some part of the overall data.

RTF supports the replication concept as it allows to add entities to a zone that is managed by multiple servers. A server which creates a new entity in a zone is automatically the responsible server for the entity which is called *active entity*. RTF automatically starts to replicate an active entity on all the remaining servers of the zone and these replicas are called *shadow entities* on the remaining servers.

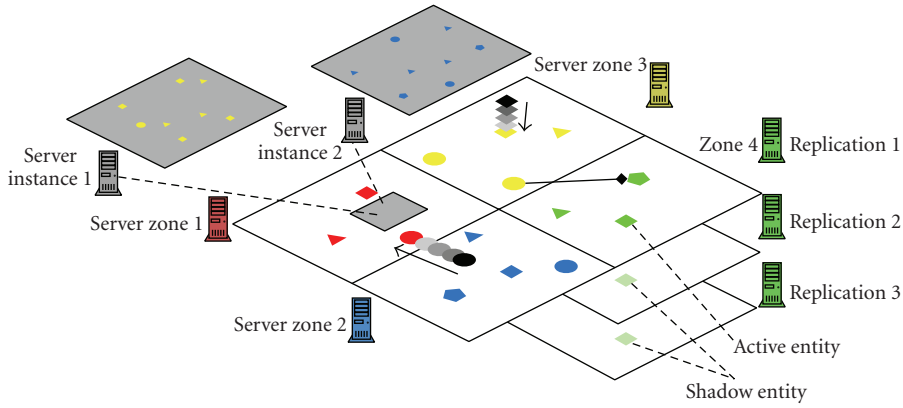


Figure 13: Combination of zoning, instancing, and replication for a single-game world in RTF.

All three orthogonal parallelization concepts aim at different scalability dimensions: zoning allows large user numbers in large MMORPG worlds, instancing runs a large number of game world areas independently in parallel, and replication targets high user density for action- and player-versus-player-oriented games. If the players are regularly distributed across the game world, then zoning is the best choice as it scales linearly with the number of zones. But if players tend to crowd at certain places and thus increase the player density at these places, instancing or replication is necessary. Instancing again scales linearly with the number of instances but introduces multiple copies and their conceptual drawbacks. If the creation of multiple copies of certain areas does not fit into the game concept, then only replication allows to scale the player density while providing a single, seamless world. The game concept usually determines the best possible combination of these concepts. For example, zoning could be used for a huge game world, while certain dungeons in this world are instanced for groups of players and cities are replicated to allow an increased player density.

A novel feature of RTF is the possibility to arbitrarily combine the three orthogonal distribution approaches depending on the requirements of a particular game design. Figure 13 illustrates a possible combination of these approaches in a single game as provided by RTF. This is an improvement as compared to MMORPGs which already combine zoning and instancing, but replication is currently not available for a combination with either of them.

The integration of all three concepts enables RTF to provide new manageability functionalities for games on RTF without a need for introducing a specific application support for these functionalities. Section 7 explains how the combination of zoning and replication enables RTF to transfer clients from one zone into another zone without a noticeable interruption on the client side. This combination also natively enables interactions across zone boarders, which must otherwise be implemented separately. The combination of instances and replication allows the reassignment of zones and instances during run time to new machines and the reaction on load increases in certain zones or instances. Adding an additional server to a zone or instance raises the supported number of clients for this zone or instance.

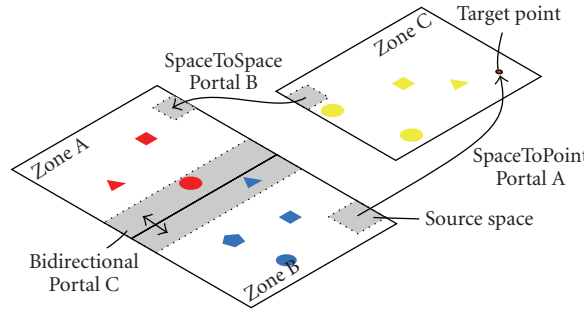
The overall goal of integrating these approaches into RTF, as discussed in detail in [20], is to provide general and dynamic scalability for all game genres within a single framework, which can be operated on demand in a grid computing environment. The following discussion sketches the envisaged methodology for developers for using these multiserver parallelization concepts.

6.2. Task (4): parallelization and distribution

If the multiserver capabilities of RTF are used, then, in addition to the general tasks 1–3 (data structure design, state processing, and AoI management), the developer has to segment the game world into zones, instances, and replication areas and to define the connections between them in form of portals. Using this information, RTF automatically assigns servers to each of the segments and connects each client to the particular segment the associated avatar resides in. If the user moves his avatar through a portal area, RTF will recognize this and automatically issue a connection transfer, making the server of the new segment responsible for processing the avatar. Each of the participating servers runs the normal server real-time loop discussed in the previous section for its associated segment—RTF internally handles connection migration and distributed entity management.

6.2.1. Specify segmentation

RTF offers a dedicated interface for specifying how the overall game world is segmented into a combination of zones, instances, and/or replication areas. A *world*-loader creates the specification with this interface once during application startup and the specification is then static for the overall application session. We provide a default loader which uses game world definitions in XML files, but developers may implement their own loader and file formats. Figure 14(a) illustrates a two-dimensional game world example with three zones and portals of various types. The definition of a zone,



(a)

```
Zone A = Zone (0,   Space (0, 2, 0, 4, 2, 0),   PLAIN );
Zone B = Zone (1,   Space (0, 0, 0, 4, 2, 0),   PLAIN );
Zone C = Zone (2,   Space (5.5, 1, 0, 4, 2, 0),   REPLICATE );
Portal & pA = *new SpaceToPointPortal ( entranceA, destinationA );
Portal & pB = *new SpaceToSpacePortal ( entranceB, destinationB );
Portal & pC = *new BidirectionalPortal ( spaceOne, spaceTwo );
```

(b)

Figure 14: Segmentation specification example.

as illustrated in Figure 14(b), consists of an ID, the occupied space, and a flag if it is allowed to replicate the zone across multiple servers. For the portals, an entrance area and a connected destination area are given. During runtime, all zones are assigned to the set of available servers. Figure 14(b) shows different portal types supported by RTF for expressing various transfer relations (uni- bidirectional, space to space, space to point) and how they connect different areas of the game world.

6.2.2. *Implement segment-related game logic*

With the introduction of zones, replications and instances—the segmentation of the game world—also the game-world-related update processing should be segmented. For example, a server should place new NPCs only in the zone he is responsible for and he should only create in-game events that are related to this zone.

For the realization of necessary interzone and interserver events, RTF provides three mechanisms: a server can send events and messages to a certain zone; a server can send events to the owner of a certain shadow entity; and a server can create global objects. If an event is sent to a certain zone, RTF automatically determines and transmits the event to the responsible servers (could be multiple servers for a replicated or instanced zone). If an event is sent to the owner of a shadow entity, RTF determines the server that holds the corresponding active entity and transmits the event to this server. Finally, global objects are serializable objects that are replicated automatically to all participating servers of the game. A global object can contain, for example, the information of the current weather for the complete game world or a global scoreboard. RTF currently does not provide a distributed synchronization mechanism for the collaborative modification of global objects and currently the global object owner serializes the distributed write access. Efficient synchronization mechanisms which are appropriate for real-time online games are being investigated and will be incorporated into upcoming RTF versions.

6.2.3. *React upon distribution changes*

As clients and entities can move between zones and instances, the game-state distribution may change during runtime. Therefore, the developer is informed about clients and entities that have entered or left a zone. If a client or entity moves into a certain zone, the responsible server is notified about this event and must process the updates and events related to this client or entity.

Overall, game developers only have to implement mechanisms at a high level of abstraction in the RTF multiserver task. In particular, they can start developing any multiserver game engine as a single-server engine at begin and then easily switch over to a scalable multiserver engine. For this switch, developers, in most cases, only have to segment the game world into zones, instances, and replication areas, possibly implementing segment-related game logic mechanisms on top of the already existing specified entity and event data structures.

7. SEAMLESS GAME WORLD AND ZONE MIGRATION

Using zoning as a distribution concept is subject to two restrictions: (a) entities and clients must be transferred between the participating servers if they are moved between the zones, and (b) no interactions are allowed across zone borders. Traditionally, the game developer implements the transfer functionality by explicitly establishing a connection to the new server and communicating the entity view from this server to the client. To allow interactions, for example, attacking a remote entity across the zone border, special synchronization and interserver communication are

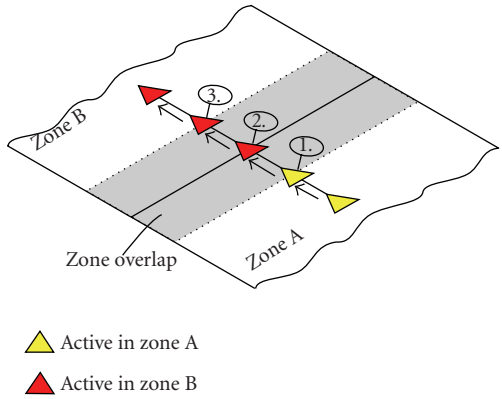


Figure 15: Overlapping zones for a seamless migration of an entity between two zones.

required, increasing therefore the overall complexity of the game architecture and reducing its scalability.

In this section, we describe how RTF provides a transparent solution for these problems and, furthermore, allows to move zones between servers. RTF allows a seamless interzone migration and interaction by creating an overlapping area between two or more adjacent zones. Since RTF allows to freely combine zoning with replication, this overlap is replicated across the servers.

Figure 15 illustrates how two zones are overlapped in the 2D case, thus creating a seamless game world. The bottom part of the figure demonstrates how the movement of an entity between these zones is handled.

(1) The entity moves within zone A into the overlap and is replicated as a shadow entity on the server of zone B (step 1 in the figure).

(2) After the entity covers half the distance of the overlap, RTF automatically changes its status in A from active to shadow and vice versa in B (step 2).

(3) As soon as the entity leaves the overlap, RTF removes it from zone A (step 3).

If the entity is a client's avatar, then the communication connection of the client to the server of zone A must be transferred during step 2 to the server of zone B. RTF manages this seamlessly if the developer makes the overlap bigger than the area of interest of the client: in this case, both servers responsible for A and B have the same view of the game world within the replicated area, such that no initial communication between the new server and the client is necessary. Furthermore, interactions across the two zones are now possible because they take place within the replicated overlap area and the client is placed in both overlapping zones at the same time. In summary, this leads to a completely seamless game world for the clients.

RTF also supports a migration of a zone to another server during runtime, by using a replication mechanism similar to the one shown in Figure 15. Since the migration is performed over an extended period of time, no interrupts are necessary, such that the players observe a smooth game flow. In addition, this allows to dynamically assign servers depending on the current system load and maintenance work, which is particularly interesting for the utilization of grid resources.

8. RTF IN GRID CONTEXT

RTF integrates solutions which enable games on top RTF to be easily deployable and executable in a grid system. Its interface realizes an abstraction of the participating resources, for example, events are sent by a developer to zones instead of particular hosts. This abstraction allows RTF to add hosts and relocate zones to different hosts transparently to the game developer. Furthermore, RTF provides an integrated in-application monitoring and controlling module, allowing to manage an application on top of RTF by external management consoles which connect to this module.

The work on RTF is part of the *edutain@grid* project funded by the EC IST, where it provides the fundamental real-time computation and communication middleware for interactive applications and online games operated in a grid computing infrastructure. It is developed with a strong emphasis on studying and optimizing mechanisms in the area of distributed real-time computation and communication, continuous processing parallelization and development methodology of distributed virtual environments and online games. Games implemented on RTF and operated in a grid environment can be easily and automatically adapted to changing user demands. Possible scenarios currently taken into account in edutain@grid include the following.

(1) *Daytime-dependent user load:* At prime time at night, each zone can be operated by a dedicated server for maximum performance, while a single server can be responsible for several zones during daytimes with low demand. This frees resources for different tasks or allows to efficiently share a pool of resources among several games or sessions at a data centre.

(2) *User hot spots:* Users of games with large worlds like MMORPG tend to cluster in particular zones for socializing, trading, or fighting with each other at large scale. This dynamic behavior leaves some zones nearly empty, while the hot spot zones may be congested. The replication approach allows to dedicate additional server resources to such a heavily frequented zone, thus scaling the maximum player density in that area for maintaining a fluent game experience. This replication can be dynamic: if users move over to an adjacent zone, the grid environment can automatically remove replication servers form the old zone and assign them to the new heavily utilized zone.

(3) *Server role change:* Especially in MMORPGs, a lot of large raids for player versus environment (PvE) gameplay often form at night and enter dungeon instances for fighting large boss mobs for several hours. This behavior implies a large demand for instance servers during that time, while at other times of the day instances might be barely frequented. The edutain@grid system is designed to switch the roles of RTF-based game servers: in this example, unutilized replication servers could be switched to become instance

servers during main raiding time, and be switched back to replication servers if only few instances are requested.

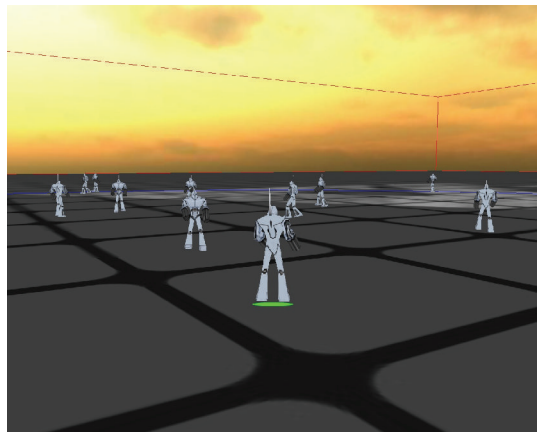
Although RTF is developed with the goal in mind to support these scenarios and to ease the development of grid-enabled interactive online applications in cooperation with grid management services, its major focus is to be a development tool on its own. RTF is used in the first place by developers to realize their online games on a high level of abstraction while RTF cares about the efficient serialization, communication, and distribution management of the game state and processing. The manageability features that RTF can provide beneath this high-level abstraction are particularly interesting for the dynamic usage of cluster- or grid resources for the game service provisioning.

9. IMPLEMENTATION CASE STUDIES

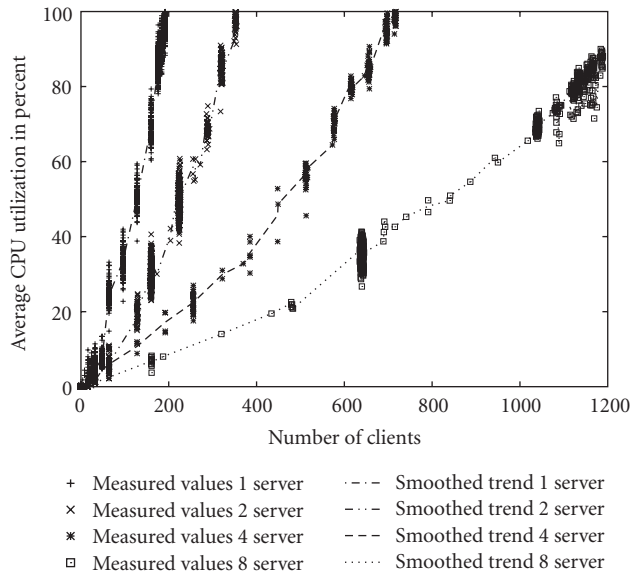
Several case studies are currently developed on top of the RTF prototype. The *RTFDemo* application is a fast-paced evaluation and demonstration game that takes place in a zoned 3D world. The game state is updated 25 times per second and the zones are overlapped and allow a seamless migration across zone boarders. Each client has control over a single avatar and can move it around and let it interact with the game world. Figure 16(a) shows a screenshot of one client in the game, looking at avatars of other clients. We used a heterogeneous setup of PCs with 2.66 GHz, CoreDuo 2 CPUs, and 4GB RAM in a LAN setup for preliminary performance and scalability tests. The average CPU utilization was measured as a metric for the test and multiple computer-controlled clients continuously sent inputs to their server. Clients could move freely between zones, but were initially distributed equally between the servers. Figure 16(b) shows the number of players that could participate fluently in the game for one, two, four, and eight zones. The current RTF version already achieves more than 160 clients on a single server with a high update rate of 25 Hz. RTF also meets the expectation that the zoning approach should scale nearly linearly if the clients are equally distributed.

For an evaluation of the overall development process and methodology based on RTF, we are developing an MMOG named *Offshore*, which takes place in an aquatic metropolis. Figure 17(a) illustrates the corresponding game world segmented into nine zones, while Figure 17(b) shows a screenshot of the current client prototype giving an overview of the game world from an elevated position.

A custom game engine, relying on RTF, was built for this MMOG by 12 developers in six-month, part-time student project and incorporates Ogre3D as a rendering and input engine, OgreAL for sound, and TinyXML for map loading. The integration of all components into a custom game engine went very well and all basic elements of a faster-paced MMORPG are present. RTF first supported the general development tasks for single-server operation, after which the game engine has been successfully switched over to multiserver processing by segmenting the game world. The prototype has about 58 K *lines of code* (LoC) whereas RTF itself has 35 K LoC and we estimate that the usage of RTF has saved about 25 K LoC in the prototype. This project mainly served as a test how understandable the development methodology of RTF is to nonexpert developers and the feedback was very positive and helped to improve the developer manual.



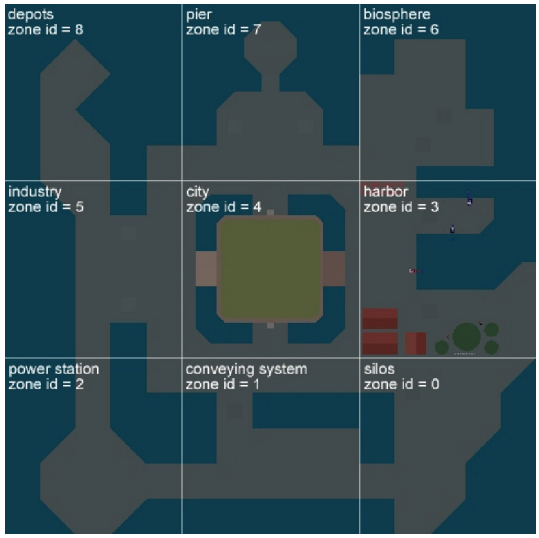
(a) In-game screenshot of RTFDemo game



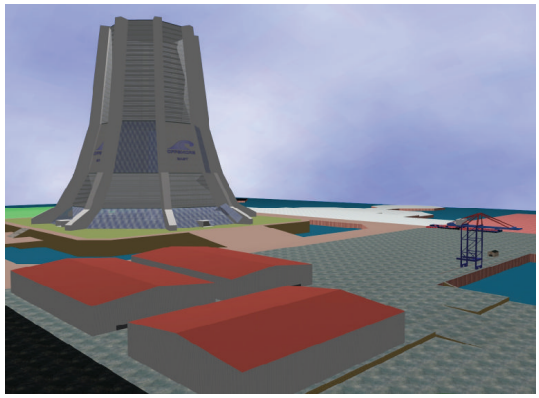
(b) CPU utilization for different numbers of servers

Figure 16: RTFDemo: the fast-paced evaluation game and the results of the experiment.

For an evaluation of the responsiveness of RTF, its flexibility and suitability for fully fledged fast-paced games, the publicly available *Quake 3* game engine is currently ported onto RTF. Quake 3, illustrated by the screenshot in Figure 18(a), is a very popular first-person shooter which is highly optimized for performance and low network traffic. RTF replaces the network module of the engine and the current beta state already allows to run Quake 3 on top RTF. Technically, Quake 3 is originally written in plain C and it was necessary to make Quake 3 C++ compliant, as RTF is designed and implemented in C++. Along the RTF integration, Quake 3 is also modified to support



(a) Offshore metropolis segmentation



(b) Overview screenshot of current prototype

Figure 17: Offshore—a basic MMOG.

the replication distribution concept. Although Quake 3 is extended for replication, the substitution of the original network module with RTF reduced the code amount from about 328 K LoC to 313 K LoC. A detailed report on this work of porting Quake 3 onto RTF is currently in preparation for publication.

Beside the case studies that we are developing, external partners incorporate RTF into their applications. Darkworks [21], a French game-development studio, is working on the integration of RTF into an upcoming professional game engine. The University of Linz incorporates RTF into their *Net'O'Drom* [22, 23] racing game. The game now uses a zoned world with seamless migration between the zones. Figure 18(b) shows a screenshot of the Net'O'Drom game from an elevated position.

Overall, these case studies and external developers provide a lot of valuable feedback. The successful and easy integration of RTF into all these projects supports the position that the RTF development process and methodology are suitable for a large class of online multiplayer games and especially complex multiserver online games.

(a) Screenshot of the quake 3 port

(b) Screenshot of the Net'O'Drom port

Figure 18: Existing games that are ported onto RTF.

10. CONCLUSION AND RELATED WORK

In this paper, we have studied and analyzed contemporary development methods for massively multiplayer online games and demonstrated to what extent the low-level custom development can be substituted by a high-level approach using game middleware for single-server and for scalable multiserver engines. Our particular new contributions are as follows.

(1) We provide a comprehensive taxonomy of contemporary game development approaches with respect to their flexibility and level of abstraction. Based on this taxonomy, we describe a new development approach that aims both at single-server and multiserver settings and still provides a very high degree of flexibility. Game developers are liberated from the low-level communication and distribution management tasks while being able to realize the remaining game development tasks without inappropriate restrictions.

(2) We describe in detail our RTF middleware system which is used to support the human developer in the development process. RTF's comprehensive distribution capabilities enable a smooth transition from a single-server to a multiserver game design. Since RTF focuses on the processing part of games, it puts no constraints on the remaining development tasks as, for example, graphics or game logic implementation.

(3) We sketch how RTF automatically enables the dynamic usage of grid resources for changing user demands

and describe common scenarios where the dynamic exploitation of grid resources could be interesting.

Zoning [3] and replication [4, 19] were already investigated successfully as independent distribution concepts for scalable multiplayer online games. Our high-level development approach integrates both concepts, including instancing, in a seamless way and we described and tested RTF as a development tool for this high-level approach.

Also various high-level game development approaches have been proposed, for example, [24, 25] investigate the abstraction of the overall game engine, making it possible to exchange an underlying game engine without modifying the game. This development approach represents an even higher level of abstraction, compared to ours, but requires the implementations for various network-related issues, graphics-rendering, input-processing, and so on. In our approach, RTF supports an easy realization of these issues in the context of online games. Project Darkstar [26] proposes a separation of all game-related processing into task objects that are freely distributable across multiple servers. An underlying run-time and global object store distributes the tasks and manages the distributed object access. However, a global object store and random distributed access might be difficult to manage efficiently in a very responsive and highly interactive game. Also [27, 28] present development approaches for multiplayer games. However, [27] discussed their novel high-level approach so far only for peer-to-peer-based or single-server-based online games and [28] focuses on monolithic, compile-time management of the complexity of an MMOG-architecture, while we provide a more incremental development approach with strong focus on runtime functionality of our RTF middleware.

In comparison to existing approaches in the field of basic communication middleware like *Net-Z* [6], *HawkNL* [29], or *RakNet* [30], RTF provides a much higher level of abstraction: this includes automatic entity serialization and hides nearly all of the technical network communication aspects. On the other hand, RTF is significantly more flexible than reusable game engines like the *Quake* or *Unreal* engines, because it is not bound to a specific graphics engine and leaves the real-time loop implementation to the developer, who is now supported by the high-level mechanisms of RTF for entity and event handling. The multiserver capability of RTF allows to easily incorporate three different parallelization and distribution approaches and is open to be extended in future game designs. This flexible support of different parallelization concepts allows RTF to be usable for a broader range of MMOG concepts than existing multiserver middleware like the Emergent Server Engine [7] or BigWorld [8] which focus mostly on the concept of zoning.

Our proposed high-level development approach is efficiently supported by the current RTF implementation, which both provides a high level of abstraction and preserves design flexibility for single- and multiserver game engines. We conducted several case studies which showed that RTF is indeed easy to use and it successfully shields the developer from the low-level tasks of online game implementation.

Summarizing, RTF offers the following integrated functionalities.

(1) Game data structures are specified as plain C++ entities. The serialization for the transfer over a network is done automatically by RTF, and the underlying communication implementation is optimized with delta updates to reduce the amount of data sent over the network.

(2) The game logic and entities are implemented in a usual object-oriented way and are open to be integrated with state-of-the-art scripting capabilities, like LUA [9], for example.

(3) The proven multiserver distribution concepts zoning, instancing, and replication, as well as their combinations, are supported by RTF. The corresponding segmentation and distribution of the game world are described on an abstract level.

(4) Distribution management and parallelization of the game state processing is fully handled by RTF. Segments can be reallocated to new servers during runtime and interserver client migrations are realized in a seamless way.

(5) Support for advanced monitoring and controlling capabilities simplifies the dynamic management of online games in a grid environment with resource management.

Besides the in-depth evaluation of RTF's performance characteristics, we are investigating the applicability of our approach to broader, nongame classes of applications that still exhibit a basic real-time loop structure, for example, e-learning and spatial physics simulation. Furthermore, we plan to integrate additional features into RTF in the future. In particular, additional grid-related monitoring and manageability functions are highly promising to be integrated for further enhancing RTF as a comprehensive middleware for online games.

ACKNOWLEDGMENTS

The authors like to thank the anonymous reviewers, whose comments helped them a lot to improve this paper. The work described in this paper is supported in part by the European Union through the IST 034601 project "edutain@grid."

REFERENCES

[1] F. Glinka, A. Ploss, J. Müller-Iden, and S. Gorlatch, "RTF: a real-time framework for developing scalable multiplayer online games," in *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '07)*, pp. 81–86, Melbourne, Australia, September 2007.

[2] A. Ploss, F. Glinka, S. Gorlatch, and J. Müller-Iden, "Towards a high-level design approach for multi-server online games," in *Prooceedings of the 8th International Conference on Intelligent Games and Simulation (GAMEON '07)*, pp. 10–17, Bologna, Italy, November 2007.

[3] M. Assiotis and V. Tzanov, "A distributed architecture for MMORPG," in *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '06)*, ACM, Singapore, October 2006.

[4] J. Müller-Iden, *Replication-based scalable parallelization of virtual environments*, Ph.D. thesis, Universität Münster, Münster, Germany, July 2007.

[5] R. Prodan, V. Nae, T. Fahringer, et al., "Toward a grid environment for real-time multiplayer online games," in *Proceedings of the CoreGRID Integration Workshop (CGIW '08)*, Crete, Greece, April 2008.

[6] Quazal net-z, 2006, http://www.quazal.com/.

[7] Emergent Game Tech., 2007, http://www.emergent.net/.

[8] BigWorld, "BigWorld Technology," http://www.bigworldtech.com/.

[9] W. Celes, L. H. de Figueiredo, and R. Ireusalimschy, "Binding c/c++ objects to lua," in *Game Programming Gems 6*, M. Dickheiser, Ed., pp. 341–355, Charles River Media, Rockland, Mass, USA, 2006.

[10] J. Smed and H. Hakonen, *Algorithms and Networking for Computer Games*, John Wiley & Sons, New York, NY, USA, 2006.

[11] J. Yan and B. Randell, "A systematic classification of cheating in online games," in *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '05)*, pp. 1–9, ACM, Hawthorne, NY, USA, October 2005.

[12] C. Choo, "Understanding cheating in Counterstrike," November 2001, http://www.fragnetics.com/articles/cscheat/print.html.

[13] J.-S. Boulanger, J. Kienzle, and C. Verbrugge, "Comparing interest management algorithms for massively multiplayer games," in *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '06)*, ACM, Singapore, October 2006.

[14] W. Cai, P. Xavier, S. J. Turner, and B.-S. Lee, "A scalable architecture for supporting interactive games on the internet," in *Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS '02)*, pp. 60–67, IEEE, Washington, DC, USA, May 2002.

[15] P. Rosedale and C. Ondrejka, "Enabling player-created online worlds with grid computing and streaming," September 2003, http://www.gamasutra.com/resource_guide/20030916/rosedale_01.shtml.

[16] M. R. Macedonia, M. J. Zyda, D. R. Pratt, P. T. Barham, and S. Zeswitz, "NPSNET: a network software architecture for large-scale virtual environments," *Presence*, vol. 3, no. 4, pp. 265–287, 1994.

[17] CCP. EVE, http://www.eve-online.com/.

[18] World of Warcraft, http://www.worldofwarcraft.com/.

[19] A. Bharambe, J. Pang, and S. Seshan, "Colyseus: a distributed architecture for online multiplayer games," in *Proceedings of the 3rd Symposium on Networked Systems Design & Implementation (NSDI '06)*, pp. 155–168, USENIX Association, San Jose, Calif, USA, May 2006.

[20] J. Müller and S. Gorlatch, "Scaling online games on the grid," in *Proceedings of the 4th International Game Design and Technology Workshop (GDTW '06)*, M. Merabti, N. Lee, K. Perlin, and A. El Rhalibi, Eds., pp. 6–10, Liverpool John Moores University, Liverpool, UK, November 2006.

[21] Darkworks s.a, 2008, http://www.darkworks.com/.

[22] C. Anthes, A. Wilhelm, R. Landertshamer, H. Bressler, and J. Volkert, "Net'O'Drom—an example for the development of networked immersive VR applications," in *Proceedings of the 7th International Conference on Computational Science (ICCS '07)*, pp. 752–759, Springer, Beijing, China, May 2007.

[23] H. Bressler, R. Landertshamer, C. Anthes, and J. Volkert, "An efficient physics engine for virtual worlds," in *Proceedings for the Medi@terra Art & Technology Festival*, pp. 152–158, Athens, Greece, October 2006.

[24] A. BinSubaih and S. C. Maddock, "Game portability using a service-oriented approach," *International Journal of Computer Games Technology*, vol. 2008, Article ID 378485, 7 pages, 2008.

[25] A. BinSubaih, S. C. Maddock, and D. Romano, "A survey of 'game' portability," Tech. Rep. CS-07-05, University of Sheffeld, Sheffield, UK, 2007.

[26] Inc. Sun Microsystems, Project darkstar, http://www.projectdarkstar.com/.

[27] P. Kabus and A. P. Buchmann, "A framework for network-agnostic mutliplayer games," in *Proceedings of the 8th International Conference on Intelligent Games and Simulation (GAMEON '07)*, Bologna, Italy, November 2007.

[28] V. Narayanasamy, K.-W. Wong, and C. C. Fung, "Complex systems-based high-level architecture for massively multiplayer games," in *Game Programming Gems 6*, M. Dickheiser, Ed., pp. 607–622, Charles River Media, Rockland, Mass, USA, 2006.

[29] Hawk Software, HawkNL, 2006, http://www.hawksoft.com/.

[30] Rakkarsoft, RakNet, 2003, http://www.rakkarsoft.com/.