RTF: A Real-Time Framework for Developing Scalable Multiplayer Online Games

Frank Glinka, Alexander Ploß, Jens Müller-Iden, and Sergei Gorlatch
University of Münster, Germany
{glinkaf,a.ploss,jmueller,gorlatch}@uni-muenster.de

ABSTRACT

This paper presents a middleware system called *RTF* (Real-Time Framework) which aims at supporting the development of modern real-time online games. The RTF automatically distributes the game state among participating servers, and supports parallel state update computations, as well as efficient communication and synchronization between game servers and clients. The game developers access the RTF via a comfortable interface that offers a considerably higher level of abstraction than the time-consuming and error-prone programming in C++ with socket-based communication. We describe the structure of the RTF system and its current proof-of-concept implementation which supports the multi-server parallelization concepts of zoning, instancing and replication for real-time online games. The novel feature of the RTF is the combination of these three concepts which allows the developer to create large, seamless virtual worlds and to migrate zones between servers.

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, Design, Performance, Scalability, Middleware

1. INTRODUCTION

The development of large-scale *massively multiplayer online games* (MMOGs) is complex and expensive. Besides the huge game content, a complicated technical architecture of the game must be designed to cope with a high number of users, huge variety of objects, and numerous, complex interactions. Compared to the traditional development of games for one or few users, MMOGs must provide mechanisms for advanced scalability to support an increasing amount of players by using additional resources. These mechanisms should allow a running application to scale up to many thousand users. The current approaches to the design and realization of scalability mechanisms within online games are challenging and error-prone, because socket-based communication and scalable distribution management in traditionally used programming languages (e. g., C++) are complex and code intensive. Nowadays, most popular multiplayer games are developed using C++, because modern games have high performance requirements which are best addressed with a hardware-near programming language. High performance is especially important for real-time games, the most common category in today's successful commercial MMOGs, which require a fast reflection of user actions in the game world, i.e. responsiveness. Tools and libraries that help the game developer to solve the complex development tasks within MMOGs in a generalized manner are, therefore, very important and highly demanded.

The *Real-Time Framework* (RTF) presented here is an object-oriented, C++-based middleware system for developing real-time MMOGs. When using the RTF, the game developer focuses his efforts on designing and implementing the game world and game logic, rather than on cumbersome low-level programming. The RTF provides to the developer an easy-to-use interface for abstract description of the game state distribution among multiple servers. The developer using RTF can choose among different distribution concepts: *zoning* [1], *instancing*, and *replication* [2, 3]. Most research and experimental work so far has focused on the design aspects for one of these techniques, restricted to a special category of games: for example, [4] presents a distribution concept for role-playing games. There are also projects working towards comprehensive middleware systems for MMOGs, like *Emergent Server Engine* [5] or *BigWorld* [6].

The main new contribution of this work is that our RTF system allows the developer to implement real-time online games using multi-server parallelization with the combination of for zoning, instancing and replication concepts by providing an efficient support for distributed computation and communication. The distribution management is done automatically by the RTF, the developer only has to express distribution in terms of game content by, for instance, accounting new objects in the game state processing. Furthermore, the efficient communication solution in RTF takes

Netgames '07, September 19-20, 2007, Melbourne, Australia

care of all network-related details like forming communication links between distributed processes, sending and receiving of information through sockets, and data serialization.

We study in the following Section 2 the nature of game entities and present the techniques currently used within real-time MMOGs. In Section 2.1 we explain how game entities are modelled with the support of the RTF. Section 2.2 describes how the game computations are automatically distributed among the available resources. Section 3 presents a novel technique within the RTF that enables the seamless movement within the virtual game world when the world is partitioned among several servers. Section 4 summarizes the advantages of the RTF system and outlines the envisaged extensions of our system.

2. MIDDLEWARE INTERFACE

From the content point of view, real-time MMOGs typically simulate a spatial virtual world where a particular player moves and acts through an avatar, which represents the player within the game. During the development, the state of the virtual world is conceptually separated into a static part and a dynamic part. The static part covers the elements of the game world that never change, e. g., environmental properties like the landscape, buildings and other non-changeable objects. Since the static part is pre-knwon, no information exchange about it is required between the game participants. The dynamic part of the game state covers objects like the players' avatars, *non playing 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 state, such that dynamic information need to be exchanged between the game participants.

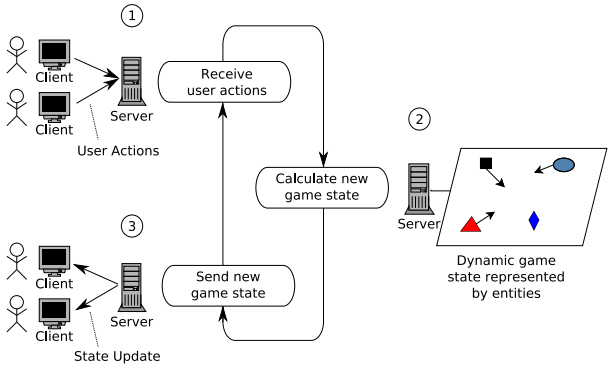


Figure 1: One iteration of the server real-time loop.

Contemporary multiplayer online real-time games implement an endless loop, called *real-time loop*, which repeatedly updates the game state in real time. Figure 1 shows one iteration of the server real-time loop for multiplayer games based on the client-server architecture. The figure shows one server, but in general this may be a group of server processes distributed among several machines. A loop iteration consists of three steps: At first the clients process the users' input and transmit (step ① in the figure) them to the server. The server then calculates a new game state 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 step 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. The remainder of this section will describe the interface of the RTF and how it copes with these tasks.

2.1 Modeling and handling the game state

When implementing the data structures to model the game state, the developer traditionally has to consider several aspects of socket communication. Because C++ has no further support for communication over a network than sending and receiving raw memory buffers over sockets, the developer normally has to implement individual communication protocols for the entities. Implementing efficient communication protocols for complex data structures is time-consuming, error-prone, and repetitive. The RTF liberates the developer from the details of network programming, by including a full implementation for transmission of complex data structures.

The RTF pursues an object-oriented way to describe game entities: the developer models the game state in a flexible way by designing hierarchies of entities. For all transmittable objects like entities, events and messages, the RTF provides an automatic serialization mechanism. Since the distribution management and hence the connections between processes are handled within the RTF, the network communication is completely transparent to the developer. The developer implements entities as usual C++ classes following the specifications of the RTF. For handling object transmission at runtime, the RTF needs some introspection to the class internals, e. g., the types of attributes, therefore, the developer has to describe the class attributes in a suitable way. In particular, the developer specifies which attributes should be transmitted by labeling them with the prefix `_ser_`. By means of this explicit labeling of attributes, it is possible to exclude some attributes from transmission, for instance to use local variables for intermediate results or extrapolation. To distribute and transmit objects with the RTF, the classes have to be derived from a particular base class `Serializable` of the RTF. Events, messages or other distributed objects inherit from `Serializable`, whereas entities are derived from `Local`, which is itself derived from `Serializable` and has additional attributes describing the position of an entity in the game world.

The introspection code, required by the RTF to transmit objects at runtime, is generated during the development of a game by a pre-processor tool called *scot* (*S*erialization *Co*de *T*ool) which is also part of the RTF. The developer invokes this pre-processor during the development process, i. e. prior to the normal compiler, for every class which is derived from `Serializable`. The game-specific code is implemented by the developer in usual C++ class implementation and definition files. The pre-processor generates and adds additional code to those files. This process is necessary once a class def-

inition changed, more precisely once a serializable attribute is added or changed. The pre-processor *scot* automatically generates appropriate getter and setter methods for each serializable attribute. The developer must use them to access the attributes even in class-internal implementations. This is due to additional maintenance like tracking state changes etc. which is needed for introspection by the serialization mechanism and would be skipped otherwise.

The serialization mechanism of the RTF can handle various data types and class associations: primitive attributes (including `std::string`), classes derived from `Serializable`, pointers to serializables, and fixed-size arrays of primitive types. Furthermore, the developer can use hierarchies of classes which inherit from `Serializable` as base class. The RTF uses unique typedefs for primitive types to ensure uniform representations on different platforms. Additionally the developer can utilize the convenience classes `Vector`, `Dimension` and `Space` defined in the RTF to implement geometry-related attributes.

At runtime, the RTF handles entities derived from `Local`: it automatically manages the distribution of objects accordingly to the distribution mechanisms described in Section 2.2. To introduce such entities to the RTF, the developer has to create a normal instance and register it with the RTF, as the following code example shows:

```
Avatar *avatar = new Avatar();
objectManager.registerLocal(*avatar);
```

A real-world example: Quake 3

The following example demonstrates how game entities can be mapped onto the RTF entity concept. Listing 1 shows the definition of a player entity used in the popular game *Quake 3* [7], where entities are realized as plain C- Structs.

```
typedef struct playerState_t {
  int    pm_flags; // ducked, jump_held, etc
  vec3_t origin;   //vec3_t is a typedef for float[3]
  vec3_t velocity;
  int   eventSequence; // index of active event
  int   events[MAX_PS_EVENTS];
  int   eventParms[MAX_PS_EVENTS];
  int   damageCount;       // health value
  int   weapon;            // ID of weapontype
  int   ammo[MAX_WEAPONS]; // ammo per weapontype
  [...]
} playerState_t;
```

Listing 1: Example code of an entity in Quake 3.

To work in the object-oriented manner of the RTF, the developer has to rewrite the `playerState_t` struct as a class which inherits from `Local`, as shown in Listing 2. This base class of all dynamic entities has an attribute describing the position of the entity, hence the `origin` attribute of the original `playerState_t` is no longer needed. This example demonstrates that only minor changes are required to the original C-Structs. Therefore, developers can make use of the advanced RTF-functionality easily, without having to learn a completely entity new description methodology.

```
class playerState_s : public RTF::Local {
private:
  rtf_int32  _ser_pm_flags; // ducked, jump_held, etc
  // position is now part of RTF::Local, replacing:
  //vec3_t   origin;
  RTF::Vector _ser_velocity;
  rtf_int8   _ser_eventSequence; // index in events
  rtf_int32  _ser_events[MAX_PS_EVENTS];
  rtf_int32  _ser_eventParms[MAX_PS_EVENTS];
  rtf_int8   _ser_damageCount; // health value
  rtf_int32  _ser_weapon;      // ID of weapontype
  rtf_int32  _ser_ammo[MAX_WEAPONS]; // ammo
  [...]};
```

Listing 2: The Quake 3 entity rewritten for RTF.

Optimized object transmission

When parts of the game state are replicated to the clients or the game state is mirrored to other servers, RTF uses delta updates, i.e. only those attributes of an object are transmitted which have actually changed since the last update. This is especially useful for game processing, because in a single tick of the real-time loop only few attributes of a particular entity usually change. Using delta updates, therefore, considerably reduces the amount of data transmitted over the network when the rate of attribute changes per tick is low. The code generated by the pre-processor *scot* keeps track of changes happening between two state updates. The memory overhead for ensuring delta updates is about one bit per attribute, which is fairly acceptable. Currently, RTF assumes a reliable communication; we will include mechanisms for unreliable communication in future versions.

2.2 Distributing the game state

The RTF supports three basic distribution concepts for real-time multiplayer games within a multi-server architecture: *zoning*, *instancing*, *replication*. The novel feature of the RTF is that it allows to combine these three concepts within one game design. Figure 2, explained in the following, shows how the distribution concepts can be combined to adapt to the needs of a particular game.

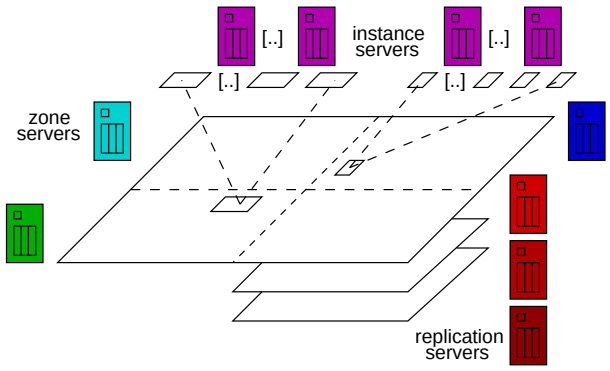


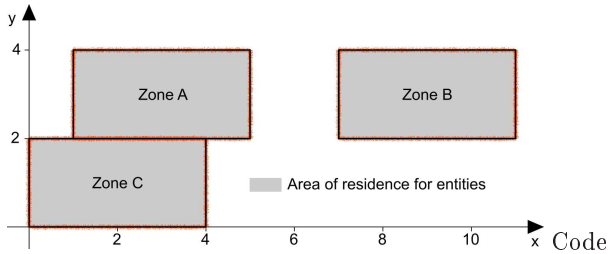
Figure 2: Combination of zoning, instancing, and replication concepts.

Zoning partitions the spatial world into disjoint parts, called zones (there are 4 zones in Figure 2). Clients can move between the zones, but no inter-zone events exist and calculations in the zones are completely independent from each other. The game developer must introduce runtime-checks for all entities if they have moved into another zone and therefore must be transferred to another server. The RTF enables the developer to specify the partition of the game world; it then automatically does all checks and transfers of the entities and clients as soon as they trigger a movement into another zone. Moreover, we show in section 3 how adjacent zones can be connected to each other in the RTF using a combination of *zoning* and *replication*.

Instancing is used to distribute the computational load by creating multiple copies of highly frequented subareas of the spatial world. In Figure 2 (top), some subareas are copied multiple times and assigned to instance servers. Each copy is processed completely independently of the others. The RTF enables the developer to specify instance areas within the game world. Upon request, new instances are created by the RTF on the available servers and clients and entities are automatically transferred into these new instances.

Replication uses the assignment of calculations to entities as a distribution criteria. The entities are distributed among all 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*. The shadow entities are replicated from other servers with only read-access. In Figure 2, the lower right zone is replicated across multiple servers.

The RTF allows the game developer to arbitrarily combine the described three distribution approaches depending on the requirements of a particular game design. The system provides methods for automatically routing messages to the active entity owner or the server responsible for a particular zone. Furthermore, the RTF automatically manages active and shadow entities and changes active and shadow states accordingly to the current distribution. The replication approach is comparatively new and more complex as compared to zoning and instancing, because interactions between active and shadow objects must be specially treated. However, it allows to scale the player density [3], i.e. place more players in a fixed-sized area by using additional servers. Furthermore, the combination of zoning and replication facilitates a flexible and seamless game world despite a partitioning into zones, which is covered in detail in Section3.



Code example:

```
Zone a = Zone(0, Space(1, 2, 0, 4, 2, 0), PLAIN);
Zone b = Zone(1, Space(7, 2, 0, 4, 2, 0), REPLICATE);
Zone c = Zone(2, Space(0, 0, 0, 4, 2, 0), PLAIN);
```

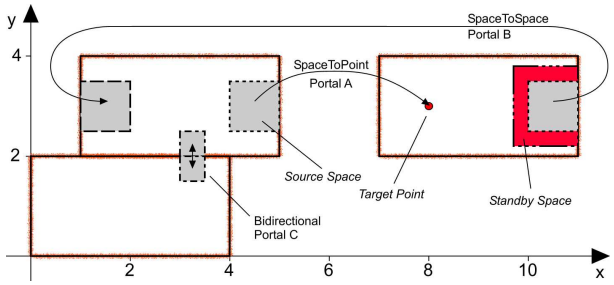
Figure 3: Partitioning of a two-dimensional world into three zones.

The developer must specify the partitions of the game world to let the RTF know how entities should be distributed. He describes the game world by a set of `Zone` and `Instance` objects which are defined by an ID and the cuboid space they occupy, and specifies whether a particular zone or instance can be replicated among multiple servers. Zones do not necessarily lie next to each other, as Figure 3 illustrates: it defines three 2D zones with height zero. They are assigned to servers during the game start-up and then each server knows which server is responsible for which zones and thus can determine to which server a client must be transferred if it enters a new zone.

Portal Definition

For the movement of players and entities between zones, especially between zones that do not lie next to each other, game developers use so-called *portals*. The RTF supports three types of portals, illustrated in Figure 4:

- The *SpaceToPointPortal* moves an entity entering the source space of the portal to a fixed target point.
- The *SpaceToSpacePortal* is an extension which owns a target area that is of the same size as the source area. An entity is moved to the same position in the target area that it had previously in the source space.
- The *BidirectionalPortal* is the combination of two SpaceToSpacePortals and thus is two-way. It has the additional constraint that entities first have to leave a source space if they were previously moved there - to prevent infinitive forward and backward movements.



Code example:

```
Dimension sizeAB = Dimension(1, 1, 0);
Dimension sizeC = Dimension(0.5, 0.5, 0);
AbstractPortal& portalA = *new SpaceToPointPortal(
    Space(4, 2.5, 0, sizeAB), Vector(8, 3, 0));
AbstractPortal& portalB = *new SpaceToSpacePortal(
    Space(10, 2.5, 0, sizeAB), Space(1, 2.5, 0,
        sizeAB));
AbstractPortal& portalC = *new BidirectionalPortal(
    Space(3, 1.5, 0, sizeC), Space(3, 2, 0, sizeC));
```

Figure 4: Example for interconnection portals of the zones showed in Figure 3.

A common but undesirable situation with portals is that the client is interrupted during the movement process, which can take a long time as a connection transfer must be done and new data must be received from the new server. To reduce delays, portals in the RTF are enhanced by an additional area, the *standby space*. In Figure 4, the standby space is shown for the right "SpaceToSpace Portal B". If an entity enters the standby space of a portal, the RTF supposes that a movement will be triggered soon and performs preparations for the actual movement process: preconnection to the potential destination server, replication of the entity at that server, and also checks whether the movement is allowed at all – which could take a long time if an external database must be searched for the necessary credentials, if not prepared by the standby space.

Interest Management

For each client, the responsible server must ensure that the client has access to the part of the game state needed to display the game properly on the client's screen. This part is called client's *area of interest* and its determination is

called *interest management*. Different techniques for interest management were studied and compared, e.g., in [8].

The RTF supports the publish-subscribe abstraction for interest management that allows the developer to subscribe a client to an entity of the game world using the `subscribe(Local&)` and `unsubscribe(Local&)` methods of the `Client` object, which represents a connected client within the RTF. This enables the developer to realize a highly optimized, game-specific area of interest management which exploits unique features of a particular game. The RTF also provides a default implementation of the *Euclidean distance algorithm* upon the publish-subscribe abstraction, which frees the developer completely from the cumbersome programming of the interest management, though this implementation has the known drawbacks regarding scalability. More sophisticated implementations, e.g. tile-based, are planned.

If a `Client` object is subscribed by the interest management to an entity, it is transfered from the server to the client and a callback is performed by the RTF at the client that informs about the new available entity which then, e.g., is displayed on the screen. The RTF also informs the client by a callback about disappearing entities, which are no longer sent by the server to the client because the interest management on the server has decided that an entity is no longer required at the client.

2.3 Processing the game state

The ongoing game processing is done on the server in the real-time loop, and the RTF is designed to seamlessly integrate into this loop. The work to be implemented by the game developer consists of two sets of tasks:

1. Processing the incoming user actions, updating entities, executing the game logic, calculating the NPC behaviour and artificial intelligence (AI).

2. Reacting to the events triggered by RTF, e.g., newly connected clients, migrated entities and changed zone distribution.

The access to the corresponding part of the distributed game world is provided by the RTF's *ObjectManager*. It contains a list of all server's entities and is also the place where newly created entities are registered to introduce them to the RTF. The RTF's *EventManager* provides access to all incoming events from clients and servers, which are enqueued by the RTF at each loop iteration. There are also *ZoneManager*, *ClientManager* and *MessageManager* which are used to obtain information about the current distribution of the game world among the servers, allow the addition or removal of clients, and provide a general-purpose message exchange facility, correspondingly.

Figure 5 is an example of how the real-time-loop works in conjunction with the RTF (in some games optimizations are possible, e.g., AI processing is done only in each second iteration or concurrently). The clients send their user actions as events asynchronously to their server. The events are processed (indicated by 2.1 in the figure), which usually requires access to the ObjectManager's entity list (indicated by 2.2). After the event processing, the game world is updated and the game logic is executed. This finishes loop iteration, after which the developer calls the RTF's `onFinishedTick()` method, which transfers the updated entities in an asynchronous way to all interested clients and servers. Moreover,

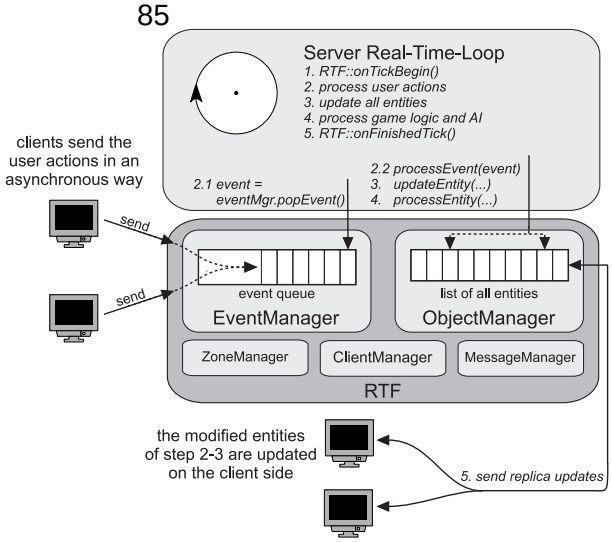


Figure 5: The data flow within the real-time loop.

this call redistributes entities and triggers callbacks that inform the developer about:

- Entities that appeared/disappeared locally or changed their state from active to shadow;
- Which shadow entities have changed their content due to an update from the active entity owner;
- Creation of new instances or the state change of a zone from plain zone to replicated zone and vice versa;
- New clients desiring to participate in the game, to disconnect or those disconnected due to network failure;
- Migration of zones and instances to other computing hosts, as described in Section 3.

The developer reacts to the events in a game-dependent way.

3. SEAMLESS GAME WORLD AND ZONE MIGRATION

Zoning allows to scale the game world size, but it brings 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 by explicitly establishing a connection to the new server and communicating the entity's view from this server to the client. To allow interactions, e.g., attacking a remote entity across the zone border, special synchronization and inter-server communication are required, increasing therefore the overall complexity of the game architecture and reducing its scalability. In this section, we describe how the RTF provides a transparent solution for these problems and, furthermore, allows to move zones between servers.

The RTF allows an inter-zone 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 6 illustrates how two zones are overlapped in the 2D case, thus creating a seamless game world. The bottom

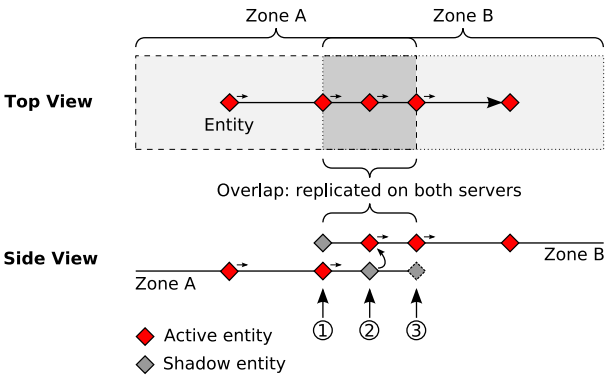


Figure 6: Overlapping zones for a seamless migration of an entity between two zones.

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 passes the half way 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 connection of the client must be transferred during step ② to the server of zone B. The RTF manages this seamlessly if the developer makes the overlap to be 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.

The RTF also supports a migration of a zone to another server during the runtime, by using a replication mechanism similar to the one shown in Figure 6. 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 the game service provider to dynamically assign servers depending on the current system load and maintenance work.

4. CONCLUSION AND FUTURE WORK

This paper presents a novel middleware system for developing massively multi-player online games. The interface of the RTF described in Section 2 is highly optimized for the typical patterns in modern game implementations. It seamlessly fits into the common real-time loop of games, and hence supports an efficient and comfortable design process for real-time MMOGs.

In particular, the RTF system has the following benefits:

- Partitioning and distribution of the game world are described on an abstract level in game design.
- The proven multi-server distribution concepts zoning, instancing and replication, as well as their combinations are supported.
- Distribution management and parallelization of the game state processing is fully handled by the RTF.
- The RTF serialization mechanism liberates the developer from the details of network programming.
- Communication is optimized with delta updates to reduce the amount of data sent over the network.
- The game logic and entities are implemented using C++ in a usual object-oriented way.

A further advantage of the RTF is the creation of seamless worlds as it was described in Section 3. This novel concept is very valuable for MMOGs and also tackled by other middleware systems, e. g., Emergent Game Technologies announced seamless virtual worlds for their Server Engine. The RTF supports seamless worlds without putting any constraints on the design of zones. By using the client migration and the replication concept to transfer the game state, it is possible to transparently reassign resources during runtime of a game. The RTF system is not bound to any specific hardware setup and targets heterogeneous systems.

The main features of RTF were validated using a simple game prototype. Our current work is to optimize RTF implementation and to study its use for developing sophisticated multi-player games with dynamic resource management. Future versions of the RTF will be extended with a mechanism for persisting the game state. We also plan to integrate an interface for audio and video streaming.

5. ACKNOWLEDGEMENTS

This work is partially supported by the European Commission through the IST 034601 project *edutain@grid* [9]. We are grateful to the anonymous referees for their helpful remarks on the preliminary version of the paper.

6. REFERENCES

[1] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Paul T. Barham, and Steven Zeswitz. NPSNET: A network software architecture for large-scale virtual environments. *Presence*, 3(4):265–287, 1994.

[2] Stefan Fischer, Martin Mauve, and Joerg Widmer. A generic proxy system for networked computer games. In *ACM NetGames '02*, pages 25–28, Bruinswick, 2002.

[3] Jens Müller and Sergei Gorlatch. Rokkatan: scaling an RTS game design to the massively multiplayer realm. *Computers in Entertainment*, 4(3):11, 2006.

[4] M. Assiotis and V. Tzanov. A distributed architecture for MMORPG. In *ACM NetGames '06*, Singapore, 2006.

[5] Emergent Game Technologies www.emergent.net, 2007.

[6] BigWorld Technology www.bigworldtech.com, 2006.

[7] Quake 3 sourcecode www.idsoftware.com/business/techdownloads, 1999.

[8] Jean-Sèbastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *ACM NetGames '06*, Singapore, 2006.

[9] edutain@grid <http://www.edutaingrid.eu/>, 2006.