A Case Study on Using RTF for Developing Multi-player Online Games[⋆]

Alexander Ploss, Frank Glinka, and Sergei Gorlatch

University of Münster, Germany
{a.ploss,glinkaf,gorlatch}@uni-muenster.de

Abstract. Real-Time Online Interactive Applications (ROIA) include a broad spectrum of online computer games, as well as challenging distributed e-learning applications, like virtual classrooms and collaborative environments. Development of ROIA poses several complex tasks that currently are addressed at a low level of abstraction. In our previous work, we presented the Real-Time Framework (RTF) - a novel middleware for a high-level development and execution of ROIA in single- and multi-server environments. This paper describes a case study in which a simple but representative online computer game is developed using RTF. We explain how RTF supports the design of data structures and their automatic serialization for network transmission, as well as determining and processing user actions when computing a new game state; the challenge is to provide the state updates to all players in real time at a very high frequency.

1 Introduction

Real-Time Online Interactive Applications (ROIA) form a novel class of technically challenging distributed applications. They include for example e-learning applications, like virtual classrooms, as well as a broad spectrum of online computer games reaching from fast-paced action games to large-scale massively multiplayer online games (MMOG). In order to support high numbers of users, the processing of the application state needs to be implemented in an efficient, scalable manner, e.g., via parallelization and distribution on multiple servers. The communication among participating processes in a ROIA session (clients and servers) needs to be efficient and optimized for highly frequent data transfers. Since generic development approaches able to handle multiple aspects of scalable ROIAs are still lacking, developers implement ROIAs from scratch and at a low level of abstraction, which is error-prone and time-consuming.

The *Real-Time Framework (RTF)* [5] is a novel middleware developed at the University of Münster as part of the European edutain@grid [2] project. RTF simplifies the development process of ROIAs in which users continuously interact and concurrently modify a shared application state. RTF is implemented as a C++ library which is optimized for efficient processing and supports the

[⋆] This work is supported by the EU through IST-034601 edutain@grid project.

eventual consistency update model and light-weight UDP communication. RTF enables a high-level development of scalable ROIAs and transparently integrates monitoring and controlling functionality for dynamic resource management. Furthermore, RTF supports various distribution concepts (zoning, instancing and replication) which allow to overcome the saturation of computational and network resources caused by a growing number and/or increasing density of online users.

Our previous work [4] described the high-level concepts of RTF, showed how its distribution mechanisms can be used to implement scalable online games and how the RTF-based approach compares to existing development methods. The process of application development using RTF comprises two groups of tasks: 1) basic tasks like designing data structures to model the application state, distributing the processing between client and server including communication, and introducing new entities, and 2) parallelization tasks of organizing a scalable distributed processing when using multiple servers.

In this paper, we present a case study on using RTF for developing a simple but still quite challenging example ROIA – a multiplayer online computer game. Because of lack of space, we omit the developer tasks needed for the multi-server case; they are left for a future publication. We describe RTF from the developer perspective, in order to show how a particular application can be designed on a high level of abstraction.

The remainder of the paper is as follows. Section 2 describes the fundamental Real-Time-Loop processing model for ROIA and gives a short overview of RTF. Section 3 provides an in-depth view of a development use case for an online computer game. Section 4 describes both the case study and RTF in the context of dynamic resource management. Finally, Section 5 concludes the single-server development case using RTF and outlines the multi-server aspect.

2 Real-Time Loop in Multiplayer 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, e. g., environmental properties like the landscape, buildings and other non-changeable objects. Since the static part is pre-known, no information exchange about it is required between servers and players. The dynamic part covers objects like 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 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 an endless *real-time loop* [1,9]. Figure 1 shows one iteration of the server real-time loop for multiplayer games based on the client-server architecture. 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).

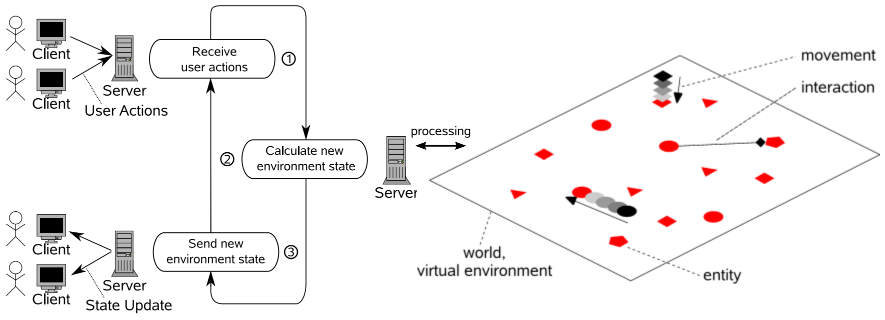

Fig. 1. Real-Time Loop for entity-based ROIAs

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. The figure shows one server involved in each step, but in a multi-server scenario this may be a group of server processes distributed among several machines.

Figure 2 shows an overview of the use of RTF in a session of a ROIA. The developer implements the application-specific processing following the Real-Time-Loop processing model. This application-specific part can use other application-specific components, like graphics engine, depending on the purpose of the software (client- or server side). To implement the processing, the application developer uses the parallelization and communication functionality provided by RTF. RTF automatically deals with the distribution, both, between client/ server and among multiple servers, and with the communication among all processes participating in a session of the ROIA.

RTF transparently implements the transmission of events between clients and servers and of state updates to clients and other servers (right-hand side of the figure). RTF automatically gains introspection to the application state (performance characteristics) and is thus aware of the current distribution status. This information can be provided to an external, application-independent

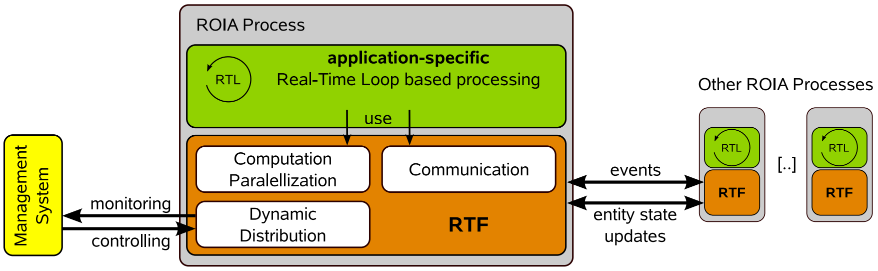

Fig. 2. Overview of the Real-Time Framework used in a ROIA

management system (left-hand side of the figure), which is then able to perform dynamic resource management for the ROIA via RTF. This monitoring and controlling by a resource management system is transparent for the application.

3 Case Study: Development of an Online Game

As a development example we will use a simple online game (*RTFDemo*), which, however, incorporates all fundamental (technical) features of a ROIA:

- Game world is simulated as a 3D virtual environment;
- Each player has one avatar;
- Players can move their avatars (using keyboard and mouse);
- Players interact by shooting other player's avatars (direct hit);
- Entities are solely controlled by the game logic.

To implement the basic ROIA state processing, the developer addresses the following tasks: 1) *data structure design* to model the application state, 2) *application state processing* to distribute the computations between clients and server using events and state updates, and 3) *Area of Interest management*, as well as some general tasks, like creating and introducing new entities. In the following we will describe how these tasks are addressed using RTF.

3.1 Task: Data Structure Design

The dynamic state of a ROIA is usually described as a set of *entities* which represent avatars or non-player characters in the game world. Besides entities, *events* are the other important structure in a virtual environment for representing user inputs and game world actions. Hierarchical data structures for events and entities in complex virtual worlds have to be serializable for efficient network communication.

Describing the Entity State. 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 are predetermined by RTF is the information about their position in the virtual 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.

In order to enable platform independence, RTF defines primitive data types to be used (e. g., `gcf_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.

We start to develop our application RTFDemo from a class to model the state of a player's avatar:

```
 1 class Avatar : public emf::Local {
 2 public:
 3   /* process a new avatar state */
 4   void think(const double& passedSec);
 5   void move(emf::Vector movement);
 6   [..]
 7   DECLARE_SERIALIZABLE_PUBLIC(Avatar, TypeAvatar)
 8 private:
 9   AvatarType avatarType; // type of the Avatar (enum)
10   emf::Vector velocity; // movement
11   emf::Vector orientation; // direction
12   gcf_uint16 health; // cur hitpoints
13   gcf_uint16 maxHealth; // max hitpoints
14   gcf::Annotation annotations; // State changes, Actions
15   DECLARE_SERIALIZABLE_PRIVATE(Avatar) };
```

Listing 1. Class `Avatar` models the state of a player's avatar

Listing 1 shows our class `Avatar` inheriting from `Local` (line 1) in which the position and dimension of the entity are described. Other attributes describe the game-dependent state of the avatar (lines 9-14). Attributes can be primitive types (`health` , `maxHealth`) including enumerations (`AvatarType` in line 9), classes (`velocity` and `dimension`), or even more complex containers of classes (`annotations`). Methods `think` and `move` (lines 5 and 6) implement the modification of the avatar state.

RTF Serialization. RTF provides automatic serialization of the entities and events defined in C++, implements marshalling and unmarshalling of data types and optimizes the bandwidth consumption of the messages. While the developer specifies entities and events as usual C++ classes, RTF provides 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 serialization mechanism provided by RTF which is generated using convenient pre-processor macros. For all entities and events implemented in this manner, RTF automatically generates network-transmittable representations and uses them at runtime.

Non-entity classes, like actions, are directly derived from `Serializable`, whereas the `Avatar` automatically inherits the `Serializable` interface via `Local`. The `DECLARE_SERIALIZABLE_*` statements (lines 7 and 15 in Listing 1) generate code for the implementation of the `Serializable` interface. `TypeAvatar` (line 7) is a system-wide unique integer to distinguish `Avatar` from other Serializables.

```
[..] // application-specific code goes here
#include <gcf/GenericSerializerImpl.cpp>
IMPLEMENT_SERIALIZABLE_DERIVED(Avatar, emf::Local,
    ADD_ATTRIBUTE(Avatar, velocity, Unreliable, Public)
    [..] // dito for all network-transmittable attributes
    ADD_ATTRIBUTE_DEFAULT(Avatar, annotations) )
```

Listing 2. Implementation of the `avatar`

Listing 2 shows the use of RTF's automatic serialization mechanism. The `IMPLEMENT_SERIALIZABLE` statement (line 3) generates the implementation of the `Serializable` interface. The developer needs to describe attributes that should be transmitted over the network. For example, the `ADD_ATTRIBUTE` statement (line 4) adds the velocity attribute to the description of the serialized form of the `Avatar`. The automatic serialization mechanism can handle delta updates, i.e., only transmitting changed information, and differentiated updates for different processes. In order to use delta updates, the developer tracks modification to attributes in a mask provided by RTF. To use differentiated updates, the developer can specify different types of visibilities for attributes (`Public` in line 4). The developer also specifies for each process its level of visibility.

3.2 Task: Application State Processing

The central aspect of the development approach using RTF is the *real-time loop model* (Figure 1). Most contemporary multiplayer games are based on such a loop whose iterative updates are called *ticks*. RTF allows the game developer to implement his own real-time loop in the well-understood manner and, moreover, provides him a substantial support for implementing and running this loop on both the server- and client side.

The client side needs to 1) determine the user actions, and 2) display the current game state. The server side has to perform 1) processing of the events, and 2) updating of the game state.

Client: Determine User Actions. At first, we read the user's input (from keyboard/mouse). Then we determine the desired action and send it to the server, using the ClientCCPModule (Listing 3, line 5). Serialization and transmission are done by RTF transparently.

```
1    void ClientActionFactory::sendActionMove(){
2    serverPos = mAvatar->getLocation().getPos();
3    emf::Vector newPos = mGraphicManager->getPlayersPosition
         ();
4    ActionMove moveEvent(newPos-serverPos);
5    mClientCCP->sendEvent(moveEvent); }
```

Listing 3. Send a user action via `ClientCCPModule` from client to server.

Server: Process User Actions On the server side, events are automatically received by RTF and appended to the event queue. We process these events and calculate the new game state as shown in Listing 4.

```
 1 // emf::EventManager& em = ccpModule.getEventManager();
 2 // emf::ClientManager& cm = ccpModule.getClientManager();
 3 void  Server::processEvents() {
 4 for(emf::Event* e=em.popEvent(); e!=NULL; e=em.popEvent()
       ) {
 5   switch(e->getEvent().getType()) {
 6   case ActionMove::TYPE: {
 7     Avatar &actor = (Avatar&)
 8         cm.findClient(e->getSender())->getAvatar());
 9     ActionMove& actionMove = (ActionMove&)e->getEvent();
10     actor.move(actionMove.getMovement());
11 } break; } } }
```

Listing 4. Process Events

We access RTF's event queue via the `EventManager` (line 4) and use the RTF type identification to determine the correct class of the event (lines 5 and 6). The move action refers to a specific entity (line 7) which can be retrieved from RTF via the `ClientManager`, which allows to determine the client which has sent an event and get the avatar from that client (line 8). After having determined the event's type and actor, the action is applied to the game state (line 10).

Server: Process New Game State. At this step, the active entities are updated accordingly to the game rules and game logic. As our implementation applies move actions directly to the entity state, we do not have to move players' avatars in this step anymore. But we have to update the rest of the game state, e. g., to move the non-player characters:

```
void Server::updateAllEntities() {
  std::map<gcf::DGObjectID, emf::Local*>::const_iterator
      it
    = om.getActiveObjects().begin();
  for(; it != om.getActiveObjects().end(); it++) {
  switch(it->second->getType()) {
    case Avatar::TYPE:
      Avatar& avatar = (Avatar&) *it->second;
      avatar.think(ticklength); // let every active
      [..] } } } // process other types of entities etc.
```

Listing 5. Update all Entities

Server Real-Time Loop. The complete processing cycle for the server is shown in Listing 6. During the `onBeforeTick` (Listing 6, line 2) and `onFinishedTick` (line 6) calls, RTF fills the event queue and sends the state updates to the clients. Therefore, the game application should not modify the game state concurrently to these calls. The call in line 6 processes the AoI Management which we will deal with in section 3.3.

```
while(!serverQuit) {
  ccpModule.onBeforeTick(); // inform RTF about begin of
     tick
    processEvents();
    updateAllEntities();
    interestManagement.update();
  ccpModule.onFinishedTick(); // inform RTF about end of
     tick
  [..] } // sleep, check for server quit etc.
```

Listing 6. Server Real-Time Loop

Client Real-Time Loop. The real-time loop on the client side looks similar to the one on the server side, but works with a specific client-side version:

```
1  while(mInputProcessor->mContinue) {
2    // sleep, calculate time since last tick
3    mClientCCP.onBeforeTick();
4      // Capture input and send actions (e.g.,
          sendActionMove)
5      mInputProcessor->capture(inputTimer.getTicklength());
6      updateEntities(timeSinceLastTick); // apply state
          updates
7      mGraphicManager->renderFrame(); // render a frame
8    mClientCCP.onFinishedTick(); }
```

Listing 7. Client Real-Time Loop

After determining user actions and sending them to the server (line 5), newly arrived updates from the server are processed (line 6) and the new game state is displayed on the screen (line 7). The client loop is completed by surrounding `onFinishedTick` (line 8) and `onBeforeTick` (line 3) calls, during which incoming events sent by the server are enqueued and game state updates are applied.

3.3 Task: 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's client. AoI optimizes network bandwidth by omitting irrelevant information in the communication. RTF supports the custom implementation of arbitrary AoI concepts by offering a generic publish/subscribe interface. The engine continuously determines which entity is relevant for a client and notifies RTF of each change of an "interested" relation through a `client.subscribe(entity)` and `client.unsubscribe(entity)` call. RTF automatically takes care that the entity is available/ updated at the client or removed from it. RTF automatically replicates a new entity to other processes (clients or servers) according to the AoI management. Clients will be informed about (dis-) appearing entities via the `ClientCCPModuleListener` interface.

```
1  void Client::objectAppeared(emf::Local& obj) {
2  switch(obj.getType()) {
3    case gcf::TypeAvatar: {
4      Avatar& avatar = static_cast<Avatar&>(obj);
5      mGraphicManager->AvatarAppeared(avatar); break; } } }
```

Listing 8. Notification about new entities (client-side)

Listing 8 shows the implementation of the `objectAppeared` callback for RTFDemo. During this callback the application can perform procedures to handle the newly appeared object, e. g., preparing the entity for introduction to the graphics engine (line 5).

3.4 General Tasks: Client Connection and Entity Creation

A general task that occurs independently of the continuously state update is introducing new entities to the application state when they are created. A typical example for introducing new entities is a client connecting to the session or a newly spawned NPC. RTF informs the application about connecting clients with the `clientConnected` callback of the `ClientListener` interface.

```
1  // Place a walking NPC in the world.
2  Avatar& npc = *new Avatar(Avatar::ZONE_TRAVELER,
3      emf::Space(2400, FLAT_HEIGHT, 750, 40.0f, 85.0f, 40.0
           f),
4      emf::Vector(50,0,0), emf::Vector(1,0,0));
5  ccpModule.getObjectManager().registerActive(npc);
```

Listing 9. Introducing new entities to the application state

Listing 9 shows how new entities can be introduced to the application state. The server creates a new instance of the `Avatar` class (line 2) and registers this new entity with RTF by invoking the `registerActive` method of the `ObjectManager` (line 5).

4 Benefits of Using RTF

After all the described basic tasks are solved and the desired game logic is implemented, our RTFDemo game appplication is ready to operate online sessions with multiple users using a single server. Fortunately, RTF's development and runtime support for ROIA goes beyond this single-server case. RTF supports multi-server distribution of the application state processing to implement scalable ROIAs and, furthermore, dynamic monitoring and controling of ROIAs during runtime. This allows to operate ROIAs using distributed resources and, with the possibility of allocating additional resources on peak-loads and increasing resource usage efficiency, as described in [6].

RTF supports the ROIA distribution approaches *zoning*, *instancing*, and *replication* on a high level. Each of these approaches allows to scale a different aspect of a ROIA. for example, zoning scales the overall size of the application state, i.e., the number of users, by identifying independent parts of application state. First scalability experiments that demonstrate the performance of RTF's zoning support are covered in [6]. Another work [7] evaluates the scalability of the First Person Shooter game Quake 3 using RTF's replication support and compares the performance of the original Quake 3 with the version using RTF.

5 Conclusion and Related Work

The main novel features of our RTF middleware are as follows: (1) Highly optimized and dynamic real-time communication links adapt to changes in the dynamic distributed environment and can automatically and transparently redirect the communication to new servers; (2) Hidden background mechanisms allow the runtime transfer and redistribution of parts of a game onto additional resources without noticeable interruptions for the users; (3) A high-level interface for the game developer abstracts the game processing from the location of the participating resources; (4) Monitoring data are gathered in the background and used by a management system for capacity planning.

Our case study has demonstrated how an example online game can be developed using RTF on a high level of abstraction. Some game development studios re-use existing solutions, e. g., successful game engines like *Unreal* or *Quake*, or use optimized libraries for particular tasks like network communication. When using only a communication library, like *Torque Network-Library* [3], or *HawkNL* [8], 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. In contrast, RTF provides an optimized high-level entity and event concept enabling automatic serialization while still providing full design flexibility. In a future publication, we will cover in detail how RTF solves additional implementation tasks for multi-server processing, like distribution and scalable parallel processing, transparently for the user.

References

1. Dalmau, D.S.-C.: Core Techniques and Algorithms in Game Programming. New Riders Games (2003)
2. Fahringer, T., Anthes, C., Arragon, A., et al.: The edutain@grid Project. In: Veit, D.J., Altmann, J. (eds.) GECON 2007. LNCS, vol. 4685, pp. 182–187. Springer, Heidelberg (2007)
3. GarageGames. Torque network library, `http://www.opentnl.org/`
4. Glinka, F., Ploss, A., Gorlatch, S., Müller-Iden, J.: High-level Development of Multi-server Online Games. International Journal of Computer Games Technology Article ID 327387 (2008)
5. Glinka, F., Ploss, A., Müller-Iden, J., Gorlatch, S.: RTF: A Real-Time Framework for Developing Scalable Multiplayer Online Games. In: NetGames 2007, Melbourne, Australia, September 2007, pp. 81–86 (2007)
6. Gorlatch, S., Glinka, F., Ploss, A., et al.: Enhancing Grids for Massively Multiplayer Online Games. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 466–477. Springer, Heidelberg (2008)

7. Ploss, A., Wichmann, S., Glinka, F., Gorlatch, S.: From a Single- to Multi-Server Online Game: A Quake 3 Case Study using RTF. In: ACE 2008, Yokohama, Japan (December 2008) (to appear)
8. H. Software. HawkNL, `http://www.hawksoft.com/hawknl/`
9. Valente, L., Conci, A., Feij, B.: Real Time Game Loop Models for Single-Player Computer Games. In: SBGames 2005 (2005)