computing@computingonline.net
www.computingonline.net

ISSN 1727-6209
International Journal of Computing

TOWARDS DATA PERSISTENCY IN REAL-TIME ONLINE INTERACTIVE APPLICATIONS

Max Knemeyer, Mohammed Nsaif, Frank Glinka, Alexander Ploss, Sergei Gorlatch

University of Muenster
Einsteinstr. 62, D-48149 Muenster, Germany
mohammed.nsaif@uni-muenster.de, gorlatch@uni-muenster.de

Abstract: *The class of distributed Real-time Online Interactive Applications (ROIA) includes such important applications as Massively Multiplayer Online Games (MMOGs), as well as interactive e-Learning and simulation systems. These applications usually work in a persistent environment (also called world) which continues to exist and evolve also while the user is offline and away from the application. The challenge is how to efficiently make the world and the player characters persistent in the system over time. In this paper, we deal with storing persistent data of real-time interactive applications in modern relational databases. We analyze the major requirements to a system for persistency and we describe a preliminary design of the Entity Persistence Module (EPM) middleware which liberates the application developer from writing and maintaining complex and error-prone code for persistent data management. EPM automatically performs the mapping operations to store/retrieve the complex data to/from different types of relational databases, supports the management of persistent data in memory, and integrates it into the main loop of the ROIA client-server architecture.*

Keywords: *Massively multiplayer online games (MMOG); persistency; virtual worlds; object-relational mapping; real-time applications.*

1. INTRODUCTION

Distributed real-time online interactive applications (ROIA) can potentially be used simultaneously by thousands of users. They make high demands on availability, responsiveness and scalability. The probably most demanding applications of this type are Massively Multiplayer Online Games (MMOGs). The number of users in this area increases sharply in recent years: the most successful Massively Multiplayer Online Role Playing Game (MMORPG) is the "World of Warcraft"[1] with more than 11 millions of active individual players. To manage the huge amount of involved data, data in such games are often stored in relational databases which are based on a solid and mature technology. For example, World of Warcraft[1], Guild Wars[2]and the virtual world Second Life[3] employ this technology.

In a MMOG, players stay together in a large virtual world to communicate and interact with each other. The players are being represented by virtual characters called *avatars*. In addition to interacting with other players, an important incentive of a player is to develop his avatar: e.g., the avatar can become equipped with new objects or learn new skills. To make this development persistently, i.e. such that changes are not lost when the game is interrupted, they must be saved (persisted). To increase the reliability of a gaming application, not only the states of avatars should be stored, but also the global state of the game world is usually stored permanently.

Through the actions of the user in the game, his avatar changes or evolves. To avoid losing the recent development of the avatar and the new states of the game world, there is a need to store these data persistently.

To store the changed entities, a system for persistent data storage is required. The persistent data storage is used when the player enters the virtual world at arbitrary time, such that all previous changes become available again. The saving is usually made at so-called *key points* of the game, for example, when the avatar completes a specific mission, acquires new objects or learns new skills. To increase reliability, it must be possible to save the changes of the game world continuously. Persistent data management takes a significant part of the code of a game, up to 40%[4]. Writing and maintaining this code is complex and error-prone, especially if new features are added to the game. The persistence code is usually tailored to a specific application use case, and thus poorly reusable.

Therefore, providing generic solution that supports the game developers in this task is desirable.

In this paper, we discuss the problem of efficiently storing the persistent data of real-time interactive applications. We target applications which a) are developed in C++, the programming language used for most ROIA, and b) store their complex data in relational database management systems (RDMS). As the result of our analysis, we present a preliminary design of our persistency system – the Entity Persistence Module (EPM) – which we design as a middleware, i.e. a software layer that connect the application with different types of relational databases. We also describe how EPM provides the application developer with a programming interface (API) in order to simplify the use of the presented persistency system.

In this paper, we discuss the problem of efficiently storing the persistent data of real-time interactive applications. We target applications which a) are developed in C++, the programming language used for most ROIA, and b) store their complex data in relational database management systems (RDMS). As the result of our analysis, we present a preliminary design of our persistency system – the *Entity Persistence Module* (EPM) – which we design as a middleware, i.e. a software layer that connect the application with different types of relational databases. We also describe how EPM provides the application developer with a programming interface (API) in order to simplify the use of the presented persistency system.

The paper is organized as follows. In Section II, we present basic fundamentals about the MMOG architecture. Section III describes how persistent data can be represented in relational database management systems. In Section IV, we describe and analyze the common approaches of persistence layers. Section V describes the preliminary design of EPM and explains how it works as a middleware software layer.

2. PROPERTIES OF MULTIPLAYER ONLINE GAMES

MMOGs are a class of online games in which thousands of players participate simultaneously in a game by communicating and interacting with each other. This game class has been growing in several distinct categories, such as: Role-Playing Games (RPG), First Person Shooters (FPS), Real-Time Strategy Games, and others. Although each category has its specific game logic, they basically have a similar structure as follows:

- The game comprises a virtual world where players reside and operate.
- The actions of players change the state of the game world, including player avatars, according to the rules of the game logic.
- The game logic dictates what actions are possible and how they affect the game world.

In MMOG, a player with his character, called avatar, moves and interacts with other objects in the game world. All changeable world objects are called dynamic objects or *entities*. These include, for example, computer-controlled characters, weapons, and the avatars of other participants. The entities have different *attributes* which describe them or their state. For example, an avatar may has information about its position in the virtual environment, its life force, its name and carried items. In role-playing games a user can, for example, move an avatar, collect items, and trade with other avatars. Through the actions of the user in the game, his avatar may change or evolve. To avoid losing the recent development of the avatar and the new states of the game world, there is an essential need to storing these data persistently.

To store the changed entities, a system for persistent data storage is required. The persistent data storage is used when the player enters the virtual world at arbitrary time, such that all previous changes become available again. The game developers need such a system in order to save, load and delete entities. The saving is usually made at so-called *key points* of the game, for example, when the avatar completes a specific mission, acquires new objects or learns new skills. To increase reliability, it must be possible to save the changes of the game world continuously.

The basic architecture used for MMOGs is the traditional client-server architecture, enriched with multiple servers. A client is responsible for presenting the game world to a player and interacting with that player. The client takes the inputs from the player and initiates changes in the game world. The server is responsible for the simulation of the game world and updating its state; it is usually called *game server*.

Fig. 1 shows the main functions of the game server, which are realized in the following three steps:

- The game server manages all entities of the virtual game world by continually receiving the actions of the players from the clients and analyzing them (Step 1);
- The new game state is computed by applying the actions of the players and the rules of the game logic to the entities (step 2);
- The new state is sent to the players (step 3).

These three steps run within the game in a loop, called *mainloop*. A single iteration is called a *tick* and the number of cycles per second is called the *tick-rate* of the game. For a smooth gaming experience, it is essential that a certain tick-rate is kept. For example, Quake3 Arena[5] is a fast FPS

game which requires a tick-rate of minimum 20 ticks/second.

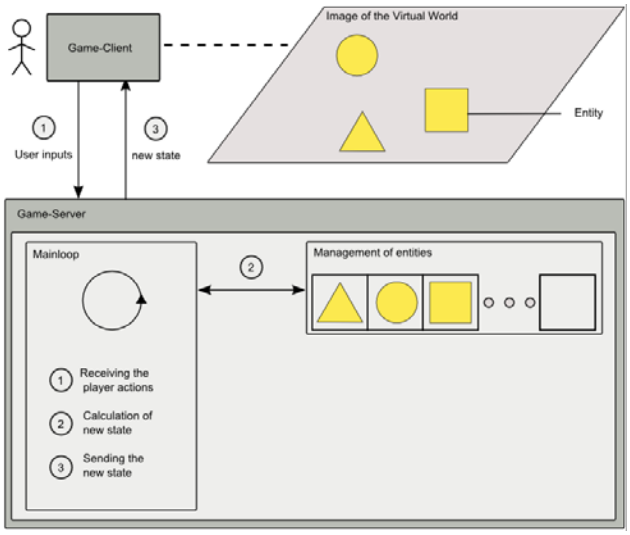

Fig. 1 – Steps of a ROIA mainloop

3. DATABASES FOR GAME PERSISTENCY

Database Management systems (DBMS) are classified according to the way of data representation, i.e., according to the data model of DBMS. The two most popular data models are record-oriented (i.e. relational data model) and object-oriented (i.e. object-oriented data model). The nature of the work environment and the requirements of an application determine which database model is more suitable[6].

An object-oriented DBMS supports complex data stored as objects; it employs a data model with object-oriented features: encapsulation, inheritance, and polymorphism. However, this data model lacks advanced searching facilities, therefore it sometimes called no-query. The underlying model in a Relational DBMS only supports simple data, rather than complex objects, but it strongly supports various advanced searching facilities[7], e.g., the relational SQL. To store the persistent objects of MMOG applications into a database and then retrieve them in an efficient way, we need both these facilities, complex data and query support.

In a relational DBMS, users can query any table in the database and combine related tables using special join functions to include relevant data contained in other tables into the results, and if needed, filter the results. We call this property the *ease of data retrieval*. The relational database model is naturally scalable and extensible, providing a flexible structure to meet changing requirements and increasing amounts of data. The relational model permits changes to the database structure which can be implemented easily without impacting the data or the rest of the database. There is theoretically no limit on the number of rows and columns of tables. In reality, growth and change are limited by the relational database management system (RDBMS) and the hardware used for implementation.

In order to create a relational database, it is necessary to define a *schema*, i.e. its structure described in a formal language supported by the DBMS. It refers to the organization of data and is a blueprint of how a database will be constructed (divided into database tables), i.e. it is a set of formulas (sentences) called *integrity constraints* imposed on a database. These constraints ensure compatibility between the parts of the schema. In relational databases, the schema defines tables, columns or fields, relationships, views, indexes, packages, types, database links, and other elements. In MMOGs, for example, two objects – the avatar and its inventory – are usually presented in two tables (relations), and their properties (attributes) are presented in the columns of these tables. Therefore, the properties of an avatar: AvatarID, Name, PositionX, PositionY, PositionZ, Energy, and InventoryID can be presented as columns in the avatar table, and the properties of inventory: InventoryID, Item1, Item2, and Item3 are presented as columns in the inventory table.

For defining and managing data and data structures in RDBMS, *Structured Query Language* (SQL) is used as a standardized special-purpose programming language. SQL acts as an interface to the RDBMS on the application development side.

Our approach is to develop a middleware for converting the complex data or complex objects of a MMOG application into simple data. Then we can use the relational DBMS as a database for MMOG applications. Relational DBMS are used in most popular MMOG applications, such as Second Life [3], and Guild Wars [2].

Nowadays, most popular multiplayer games, especially MMOGs, are developed using C++, because these modern games have high performance requirements which are best addressed with a relatively low-level, object-oriented programming language. Since the system for persistent data storage, which is presented in this paper, is used in the field of MMOGs, it is also developed in C++.

In order to access a relational database from C++, i.e. use SQL in a C++ program code, there are two main possibilities: native and general database libraries. Database vendors provide native libraries that can be used via a special API to establish a direct connection between the program code and a specific database without any mediation; this is what is called *native connection*. The native libraries that are represented as API wrappers include for example, MySQL++[8]for MySQL database, and libpqxx[9]for PostgreSql database. A native library

is usually better suited than general database libraries because of its ability to establish a direct connection with database, but the disadvantage is the restriction to a specific database. This may be a problem when using native libraries with MMOGs because the latter need to establish connections to not specified database types that are distributed on multiple servers. The solution is to use a database-independent library (or general database library).

Our approach relies on *Simple Oracle Call Interface*(SOCI) library[10] whichallows to access different databases, and at the same time it is a native database library. Fig. 2 shows the modular structure of the SOCI library allowing the integration of different database backends. SOCI makes SQL queries embedded in the regular C++ code, i.e. staying entirely within the standard C++. SOCI is integrated with databases via database backends. The backend forwards the data queries of an application into the appropriate database.

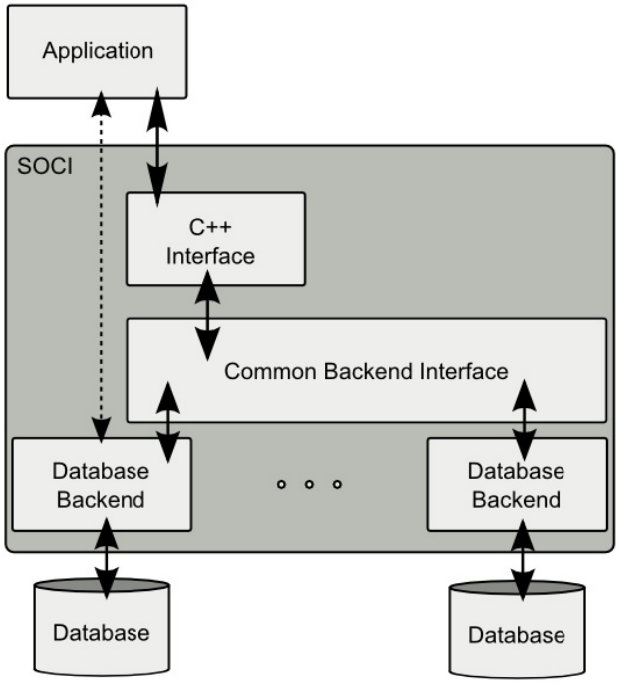

Fig. 2 – SOCI modular structure

The SOCI current version (3.1.0) supports various database types: Oracle, PostgreSQL, MySQL, SQLite3 and Firebird, as well as the generic backend: *Open Database Connectivity* (ODBC). Thus, by using SOCl we can combine the performance advantage of native libraries and the variety database access of ODBC. The SQL commands are passed to the RDBMS without conversion as it happens in ODBC. Additionally, SOCI offers a flexible support for user-defined data types and also an extensive integration with *Boost data types* of many Boost C++ Libraries, i.e. representation of nonstandard data type during storing and retrieving to/from databases. The Boost libraries of C++ are used to store arbitrary information in a variable, e.g. the *Boost.Tuple* library offers the *boost::tuple* class which offers the ability to store a virtually unlimited number of values in one variable in a C++ program.

4. KINDS OF PERSISTENCE SYSTEMS

There are three common approaches to persistence regarding the connection with database:

Database access by means of user classes

In this approach, particular methods for persistence are realized in the classes of users, i.e. the code is used for implementing the access to the database and to SQL commands directly by the classes written by the user. This approach is particularly suitable for small projects.

Database access by data access classes

Here, the code to access the database is placed in additional classes which separate the classes of users from the database. The class instances are called Data Access Objects[11] and are responsible for storing the persistent data of a class. To exchange data between user classes and the Data Access Objects, Data Transfer Objects are used to encapsulate the persistent data which is loaded from/to database. As a result, an additional code is needed in the classes of users to match the persistent data with the Data Transfer Objects. Furthermore, when replacing the employed database by a new one, all the data access classes should be adjusted.

Database access by an abstraction layer

In this approach, the user classes and the communication with database are strictly separated. Fig. 3 shows, as an example, a schematic representation of an abstraction layer to access and store a user class (persistent class) using mapping information. Communication with the database is performed at a central point of the abstraction layer used by all persistent classes of the user. With an abstraction layer, the developer does not write any additional code for database access, but rather defines meta-information which describes the mapping of objects to the database tables. For each user class required to be persistent, such mapping information must be specified. Using this mapping information, the code for database access and the necessary SQL commands are created by the abstraction layer.

This approach of using an abstraction layer (also called *persistency layer*) is used in our system presented in this paper. This approach has significant advantages over the two previous approaches: it is reusable for different projects, and furthermore, it is more easily extendible and customizable. These advantages are especially important for large projects.

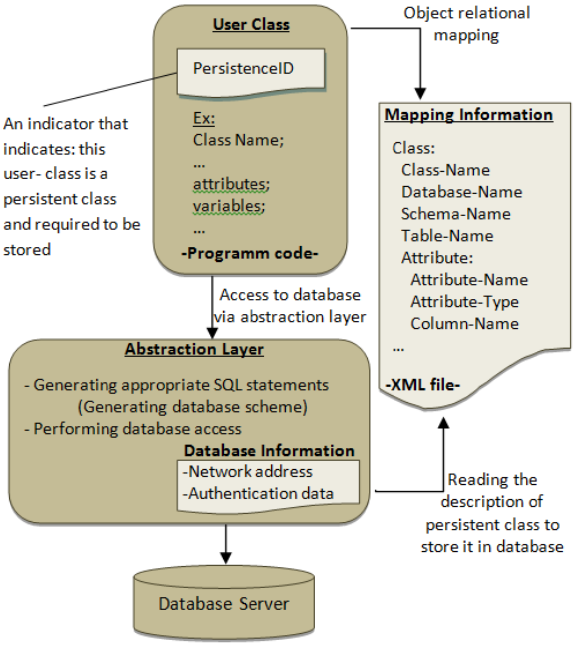

Fig. 3 – Schematic representation of the abstraction layer

The system of persistent data storage is a link between the entities of MMOGs (i.e., the objects of the application) and the RDBMS. Our analysis reveals the following basic requirements towards such a persistency system:

- The system should persist the state of the MMOG game world, i.e. the states of individual objects and entities of an application are continuously stored into a relational DBMS. One of the difficulties here is converting the data between incompatible type systems. The data type of objects is almost always a non-primitive value (composite value), while relational DBMS can only store and manipulate scalar values, (i.e., primitive data types), such as integers and strings that are organized in tables and stored as records. Therefore, the system should be able to convert the object values into groups of simpler values for storing in the database, and then, when the game logic requires it, convert them back upon retrieval from the database without mismatch. This task is usually called *Object-Relational Mapping*[12]. Particularly important is how the attributes of objects and the relationships between objects are stored. The system should support object-oriented concepts, such as inheritance and polymorphism.
- The system should be able to store entities continuously at certain times (e.g., when an avatar gets new objects or learns new skills, or completes a specific task in the game). Not always the entire entity is to be stored; it should be possible to define which particular attributes should be stored. For this purpose, the persistent data management system must provide an appropriate interface for the application developer. This interface will provide an abstraction from the direct interaction with the database, such that the developer does not need to write a database-specific code.
- The system should not be limited to a particular database, but rather be able to work with different RDBMS. Therefore, it must abstract from specific types of databases by providing general supporting interfaces for database connections.

5. THE EPM SYSTEM FOR DATA PERSISTENCY

This section describes the basic concepts and preliminary design of our system for persistent data storage called *Entity Persistence Module* (EPM). EPM serves as an interface between real-time interactive applications and the relational database management systems (RDBMS).

a. Integration of persistence in MMOG

We design our EPM system to work as a software layer between MMOG and RDBMS. To integrate persistence into the complex infrastructure of MMOG applications, EPM resides on two types of servers: login server and game server as shown in Fig 4. The login server checks whether a player is eligible to participate in the game. If necessary, corresponding data of the player, such as name, address, and list of avatars owned by the player is loaded from the database. To play the game, the player retrieves data from the account-database to the game-client; therefore, EPM works between the login server and the account-database. Based on this retrieved data, the game logic determines where the avatar has stopped in the previous game session. When the player exits the game, the game-client is logged out at the login server and the account data is stored in the account-database.

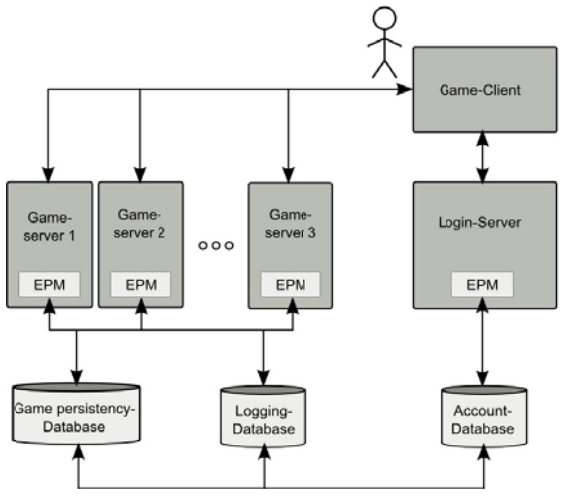

Fig. 4 – Integration of EPM in the architecture of an MMOG

Fig. 5 shows the integration of EPM with the mainloop performing the continuous processing of the game state. The mainloop receives the actions of players from the clients and analyzes them (Fig. 5 step 1), then it calculates the new state (step 2), which is persisted using EPM (step 3), and then sent back to the clients (step 4).

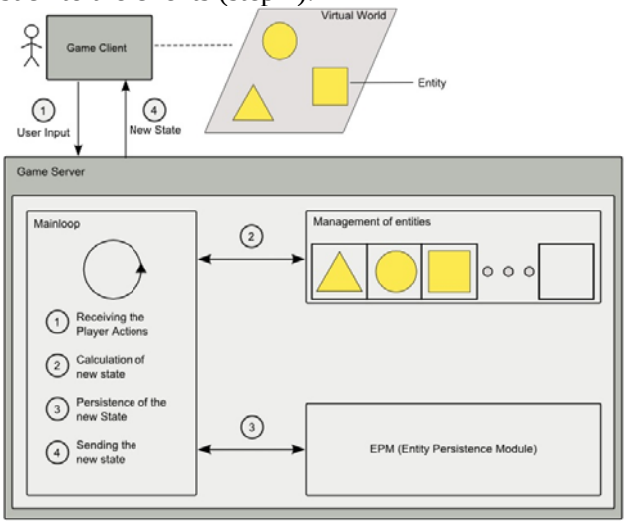

Fig. 5 – Integration of EPM in the application mainloop

b. Architecture of the persistence module

The essential part of the EPM architecture comprises the software components in the gray rectangle in the middle of Fig. 6.The bottom side of the figure represents the database side, while the upper side represents the application side as well as mapping and database information files. These files are the supplementary part of the EPM which allows the essential part of EPM to work in an efficient way. The architecture of EPM follows some ideas of an abstract design in[13].

c. Information about persistency

In order to allow for our persistency layer – the EPM system – to establish a connection between the persistent classes of MMOGs and one or more RDBMS, EPM needs meta-information about: 1) the classes which need persistency, and 2) the desirable database for storing. This meta-information is provided by the application developer and presented to EPM by: *mapping file, PersistenceID, datatype mapping, and database config.*

Mapping File

Mapping files describe where and how persistent objectsare stored. EPM provides its own XML representation of the data structures of mapping information as XML files. The mapping information specifies which classes and which attributes should be persistent and where they should be stored.

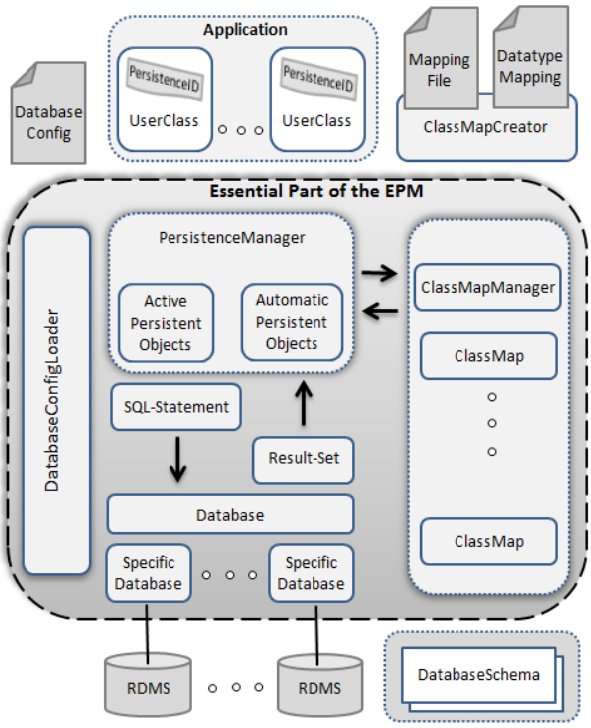

Fig. 6 – Architecture Overview of the Entity Persistence Module (EPM)

Listing 1 shows an example of XML mapping file that contains a part of the mapping information for a base class called Avatar with one of its attributes. This attribute in the example is the name of avatar. The first four lines define the mapping information of the avatar class, the given name for persistent class is specified in line 2, then the name of the database in line 3, the database schema in line 4, and the table in which the avatars will be stored in line 5. After we have accessed a specific table of a specific database depending on the first five lines, we need access to a specific column of that table (Avatar table) to store the specific name of the avatar (line 7). The data type of the name is specified in line 8 and the column in which the name should be stored is specified in line 9.

```
1  <class>
2      <name> Avatar </name>
3      <database> GameDatabase  </database>
4      <schema> gamedatabase </schema>
5      <table> Avatar </table>
6      <attribute>
7          <name> AVname </name>
8          <elementtype> std : : string </elementtype>
9          <columnname> name <columnname>
10     </attribute>
11 </class>
```

Listing 1- Part of an XML mapping file that describes an avatar class

Shadow information

Shadow information is an additional information added to a persistent object by the application developer, as shown in Fig. 6 within the *UserClass*. This information is required by EPM to manage the persistent objects and is not required by the actual application. An object within an application is uniquely identified by its address in the memory which is only valid as long as the object exists in the memory. Records of a table, in contrast, are uniquely identified by a primary-key that is valid as long as the database exists. Therefore, persistent objects require a unique ID to identify them in the application, as well as in the database. This ID is referred to as EPM PersistenceID, see Fig. 7; it consists of Universally Unique Identifier (UUID) and type information that indicates the type of the persistent object. Shadow information is an indicator of whether the object already exists as a record in the database or not. This information is used to generate an appropriate SQL command to store or update the object. The "insert" SQL command is used if the object has not been stored before, otherwise, the "update" SQL command is needed.

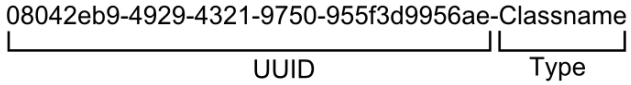

Fig. 7 – Example: presentation of PersistenceID as database key

d. Object-Relational Mapping

One of the most important issues in the design of EPM is *object-relational mapping* that converts complex data and objects of MMOG applications into simple data of primitive types for using a relational DBMS. The software components located on the right side of Fig. 6 are responsible for this. These components work across three stages of the persistence process:

(1) First, after reading the mapping information of persistent classes according to what the game developer specified in MappingFile and DatatypeMapping, the ClassMapCreator creates one ClassMap for each persistent class in the application; the ClassMap component works in the next stage.

(2) The MappingClasses are located within the essential part of EPM as shown in Fig. 6; they consist of two components: ClassMapManager and ClassMap. These components cooperate with the main EPM component (PersistenceManager) to store the persistent objects in an efficient manner. The ClassMapManager manages all the system's ClassMaps and ensures that they are initialized and made available to PersistenceManager. The ClassMap can access all data of an object at run time to generate the appropriate SQL commands, and thus the current state of an object becomes ready for storing in a relational database. These SQL commands are encapsulated within the essential part of EPM and used in the next stage.

(3) Finally, the persistent object is sent to permanent data storage (relational database). After obtaining the SQL statements from ClassMap, the DatabaseSchema defines the necessary tables, columns, relationships, data types, database links, and other elements which are necessary to store the persistent object.

With regard to the mainloop, our strategy with the mapping components focuses on separating the SQL generation from SQL execution. Thus, the ClassMap interrupts the mainloop to generate the SQL statements, and thereafter, the DatabaseSchema can execute the SQL statements concurrently with the mainloop (as a separate thread).

In sophisticated applications, most objects have one or more relationships with other objects. To avoid their separate storing, the ClassMap performs transitive persistence by storing the object with all of its associated objects to the database automatically. This recursive process of storing is called *cascading*. For example, the ClassMapAvatar of *avatar object* will cascade the storing operation to its associated object (*inventory object*). The (*save: Inventory*) task, included within ClassMapAvatar, holds mapping information that indicates a relationship between *avatar* and *inventory*. In EPM, any entity is automatically saved, loaded, and deleted to/from database, together with its associated objects.

e. Connection with databases

To connect with various RDBMS during the establishment of persistence, EPM provides a standard database interface, with different configuration possibilities for the game developer. The database interface accepts SQL statements and returns query results as result sets which are actually an object-oriented representation of relations. The RDBMS then stores the rows and columns that are represented in these result sets.

Listing 2 shows an excerpt from the EPM's database interface. The methods in line 3 and 4 initialize the database connection, i.e. opening and closing a connection with a specific database. The *getName()* method in line 5 identifies the specific name of the database with which the connection is established. The methods in lines 7 through 10 insert, retrieve, update, and delete data to/from database, correspondingly.

```
1  class database {
2  // methods to initialized the database connection
3  virtual void open ( ) = 0;
4  virtual void close ( ) = 0;
5  std : : string getName ( ) { return this -> name ; }
6  virtual Resultset * processSql (std : : string &) = 0 ;
7  virtual void processSql ( InsertSqlStatement &) = 0 ;
8  virtual ResultSet * processSql (SelectSqlStatement &) = 0 ;
9  virtual void processSql ( UpdateSqlStatement &) = 0 ;
10 virtual void processSql ( DeletetSqlStatement &) = 0 ; } ;
```

Listing 2 – Excerpt from the database interface

The DataTypeMapping and DatabaseConfig files are provided for the application developer as database configuration files as shown in Fig. 6. Since different databases use different data types to store data, DataTypeMapping file defines which C++ data type is mapped to which data types of the database. The DatabaseConfig file is used to configure a connection with a specific database. For example, this file can include: name of the database, database type, network address of the database server, and the authentication data for logging in; the file is configured during the initialization of the EPM. The database library used by EPM is also specified in this configuration file. Currently, EPM employs the SOCI library [10]which provides different backends supporting connection to various databases.

Fig. 8 shows the sequence of the database connection initialization. Here, the DatabaseConfig-Loader is the component located inside the essential part of EPM.

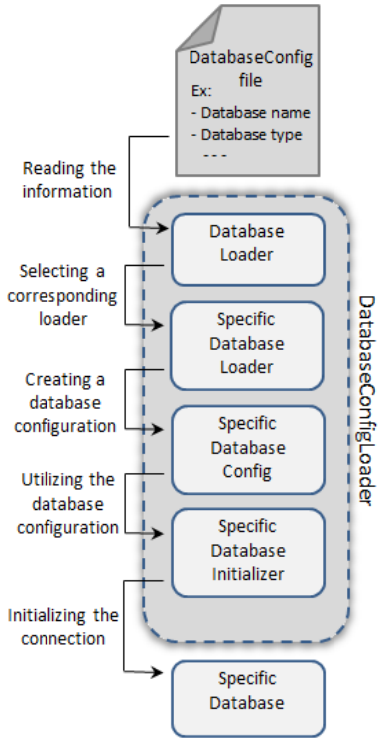

Fig. 8 – Initialization of database connection

f. Management of persistent objects

The components of EPM that support the management of persistent objects at runtime are: ClassMaps and SpecificDatabases. The main component of EPM (PersistenceManager) cooperates with these componentsand with shadow information to manage the persistent objects. The PersistenceManager provides the application developer with a programming interface to store, load and unload the persistent objects to/from database. We propose two methods within the PersistenceManager: to manage the active persistent objects in the main memory and to manage the registered persistent objects during the mainloop, as explained in the following.

Active Persistent Objects

To manipulate an object, it must be copied from the database into the main memory. The object that resides in the main memory is called *active object*. To manage the active state of the persistent objects, the EPM module checks if the persistent object is newly loaded to the main memory or already resides there. The PersistenceManager makes a list of IDs for persistent objects which currently reside in the memory. Here, the ID is the EPM persistenceID introduced at the beginning of this section. For instance, if EPM requests an active persistent object for a second time (Fig. 9 step 1), then the Persistence-Manager checks whether this object is already in main memory by checking the list of IDs (step 2), and after finding it, the PersistenceManager returns to the active object in the main memory (step 3). Therefore, the PersistenceManager can store the object copy which contains all changes without any data loss (step 4).

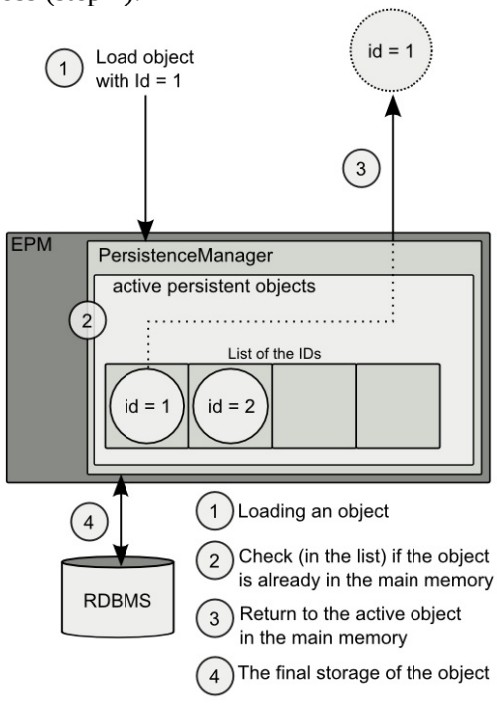

Fig. 9 – Loading an already active object

Automatic storage of objects

EPM allows for saving persistent entities during the mainloop. The API of the PersistenceManager offers a possibility for the application developer to register any persistent entity which needs to beautomatically and continuously stored.The complete storing of all persistent entities cannot be implemented efficiently at each iteration of the mainloop, rather the application developer should specify which attributes should be stored at what time: e.g., theattributes that do not change often can be stored at longer intervals than other attributes. Fig. 10 illustrates the process of automatic persistence for an object with two attributes. The first attribute (black) should be saved every 20 ticks, and the second attribute (gray) should be saved every 4 ticks. For this reason, the object is registered twice in the PersistenceManager for automatic storage (Fig. 10 step 1): once for storing the first attribute and once for storing the second attribute. The PersistenceManager holds a list that contains: the persistenceID of the registered objects, the attributes, and the tick-numbers. When the automatic persistence method is initiated (step 2), the PersistenceManager checks which object must be saved during the mainloop-tick. In the 4th tick, the second attribute is not stored, rather it is updated and remains in the main memory for a limited time. However, in the 20th tick, both attributes will be stored in the database because 20 is a multiple of 4. EPM combines all suspended updates of the persistent object and then stores them using only one access to the database (step 3).

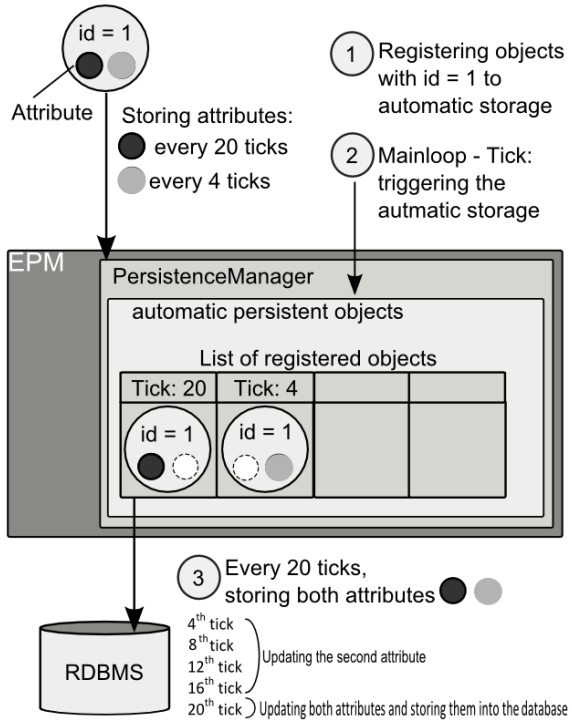

Fig. 10 – Automatic saving of an object

g. Persistence of entities

EPM aims at ensuring persistence for two kinds of objects: (1) states of game world, and (2) individual entities. These objects are mapped onto database tables for persistency. For this *Object-Relational mapping*[12], various kinds of information are required by EPM: the mapping information which specifies where and how the entities are stored, and the shadow information that is needed by EPM at runtime for managing persistent objects. The mapping strategy of EPM for dealing with the tables of the database is divided into three areas of mapping: (a) mapping of attributes, (b) mapping of hierarchies, and (c) mapping of relationships.

6. CONCLUSION AND RELATED WORK

Persistent data storage plays an important role in many distributed Real-time Online Interactive Applications (ROIA) such as modern Massively Multiplayer Online Games (MMOG). For the game developer, programming the connection between MMOG applications and RDBMS is not only time-consuming and error-prone, it is also poorly reusable. Therefore, a flexible and reusable solution is desirable.

In this paper, we analyze the problem of persistency for ROIA applications and present a preliminary design of the Entity Persistence Module (EPM) as a middle software layer to store the persistent data of MMOG applications. The game developer is provided by EPM with a comfortable API that relieves him from writing any additional code for both database access and object-relational mapping. The developer creates a configuration file to define which objects and attributes of the objects are persistent and in which database they should be stored. Depending on this information, the Mapping Classes and the required database schemas are automatically generated by EPM. The Mapping Classes then prepare the persistent data of the application and make it compatible with structures and data-types of RBBMS, as well as generate the required SQL commands to retrieve and store data from/to database.

The presented methods block the mainloop of ROIA as short as possible by generating the SQL commands to update the database, and executing them asynchronously, running in a separate thread. EPM provides a method for partial storage of objects because not always the whole object needs to be saved if only few attributes have changed. With this method, the time to build and run the SQL commands is shortened. After registering an object in the automatic storage method, the application

developer has the opportunity to store an object continuously in a database.

Although there are several sophisticated persistent data systems for Java such as Hibernate[14], or Java Data Objects[15], only few systems have been developed for C++. For example, *LiteSQL*[16]focuses on object-relational mapping by providing a layer that integrates C++ objects into a relational database; our persistence layer is specified to persist the state of real-time applications, and in addition to persist the C++ objects by our approach of object-relational mapping. *DataXtend CE*[17]has been used for applications with demanding real-time and object persistence requirements, particularly, in the fields of financial applications, flight booking, and courier delivery services. But it could not be applied in the field of MMOG applications, because the complexity of MMOG-architecture requires an efficient approach to manage the persistence of both objects and game state that are distributed across multiple game servers.

In comparison to existing approaches in the field of object persistence middleware for MMOG applications like Versant [4], EPM provides more generic middleware which allows to store the persistent data to major types of relational databases, while[4] depends upon a native persistence for objects. Hence, the core database engine of [4] requires a specific database technology while our approach overcomes this drawback.

As future work, we plan to integrate EPM with the Real-Time Framework (RTF) [18]that was developed at the University of Münster [19]within the *edutain@grid* project. After integrating the features of EPM (object- and game state-persistence) with the features of RTF (high-level game design), we will obtain a comprehensive middleware for developing and running online games.

ACKNOWLEDGMENT

Mohammed Nsaif is supported by the cooperative program (BaghDAAD) for German-Iraqi academic exchange.

7. REFERENCES

[1] World of Warcraft – Homepage, Blizzard Entertainment, [Online]. http://www.wow-europe.com/de/index.xml

[2] The database technology of Guild Wars, [Online], http://www.dbms2.com/2007/06/09/the-database-technology-of-guild-wars

[3] Mitch Wagner, Inside Second Life's Data Centers. In: Information-Week. [Online]. http://www.informationweek.com/news/showArticle.jhtml?articleID=197800179

[4] Robert Green, Advanced Mata Management for MMOG – The Versant Object Database in MMOG Applications, Versant, White Paper Version 2008.

[5] Quake 3 Arena Homepage. [Online]. http://www.idsoftware.com

[6] Database Classifications and the Marketplace. [Online]. http://seqcc.icarnegie.com/content/SSD/SSD7/1.5.2/normal/pg-trends/pg-nonrdb/pg-dbclassifications/pg-dbclassifications.html

[7] Lisbeth Bergholt and Jacob Steen Due,.: The Centre of Object Technology (COT), 1998.

[8] MySQL++ Homepage. [Online]. http://www.tangentsoft.net/myaql++/

[9] Lipqxx Homepage. [Online]. http://pqxx.org/development/libpqxx/

[10] SOCI Homepage. [Online]. http://www.soci.sourceforge.net/

[11] Sun Microsystems – Core J2EE Patterns – Data Access Object, [Online]. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

[12] S.W. Ambler. (1998, Amby-Soft Inc. Version: May) Mapping Objects to Relational Databases: O/R Mapping In Detail. [Online]. http://www.agiledata.org/essays/mappingObjects.htm

[13] S.W. Ambler, The Design of a Robust Persistence Framework for Relational Databases / Amby-Soft Inc. [Online]. http://www.ambysoft.com/downloads/persistenceLayer.pdf

[14] Hibernate Homepage. [Online]. http://www.hibernate.org

[15] Java Data Objects Homepage. [Online]. http://java.sun.com/jdo/

[16] LiteSQL Homepage. [Online]. http://sourceforge.net/projects/litesql/

[17] DataXtend CE – Progress Software. [Online]. http://www.progress.com

[18] Frank Glinka, Alexander Ploss, Sergei Gorlatch, and Jens Müller-Iden, High-Level Development of Multiserver Online Games, International Journal of Computer Games Technology, no. Article ID 327387, pp. 16 pages doi: 10.1155/2008/327387, vol 2008.

[19] The Real-Time Framework (RTF). [Online]. http://www.real-time-framework.com/

Max Knemeyer *received MSc degree in 2009 in Computer Science from the University of Muenster (Germany). He worked in the group of parallel and distributed systems at the University of Muenster. His research area focuses on relational databases, object oriented application frameworks, Real-time Online Interactive Applications (ROIA), and Massively Multiplayer Online Games (MMOG).*

Mohammed Nsaif *received MSc degree in 2005 in Computer Science from Iraqi Commission for Computers and Informatics (ICCI) in Iraq. He is a PhD student with the Department of Computer Science (in the group of Prof. Sergei Gorlatch), University of Muenster in Germany. His research interests are relational database management systems, distributed systems, Real-time Online Interactive Applications (ROIA), and Massively Multiplayer Online Games (MMOG).*

Frank Glinka *received his computer science degree from the University of Muenster in 2006 and is now a research associate at the department of computer science in the group of Prof. Sergei Gorlatch. He has worked as a work package leader for the European research project edutain@grid covering the topic of real-time application services and is currently completing his PhD thesis titled "Developing Grid Middleware for a High-Level Programming of Real-Time Online Interactive Applications".*

Alexander Ploss *received his degree in mathematics from the University of Muenster in 2007 and was a research associate in the department of computer science in the group for parallel and distributed systems at Muenster until July 2011. During this time, he worked in the international research project edutain@grid and received his doctoral degree in 2011 for his dissertation on "Efficient Dynamic Communication for Real-Time Online Interactive Applications in Heterogeneous Environments". He has published about 20 papers in peer-reviewed conferences and journals as well as book chapters on the scalability of interactive applications, e.g., massively multiplayer online games, with the focus on communication aspects.*

Sergei Gorlatch *has been Full Professor of Computer Science at the University of Muenster (Germany) since 2003. Earlier he was Associate Professor at the Technical University of Berlin, Assistant Professor at the University of Passau, and Humboldt Research Fellow at the Technical University of Munich, all in Germany. Prof. Gorlatch published more than 150 peer reviewed papers and books. He has been principal investigator in various international research and development projects in the field of parallel, distributed, Grid and Cloud computing. Sergei Gorlatch holds MSc degree from the State University of Kiev, PhD degree from the Institute of Cybernetics of Ukraine, and the Habilitation degree from the University of Passau (Germany).*