Transaction Processing for Clustered Virtual Environments

Christian Schallhart
Institut für Informatik
Technische Universität München
schallha@cs.tum.edu

Abstract. This paper introduces Massively Multi-Player Online Role Games (MMORGS) which are currently a main focus of the gaming industry. MMORGS are Networked Virtual Environments (NVES) where a player can navigate his or her character through a large game-world which is populated by thousands of other players and non player characters. Based the problems arising in the context of MMORGS, we will motivate the development of a domain-independent middleware to support MMORG in particular and networked NVES in general. Finally, we will introduce APEIRON, a middleware which is based on a flexible transaction processing framework. APEIRON is designed to serve as basis for the next generation MMORG.

1 Introduction

In this paper we will motivate the development of a middleware which is projected as basis for networked virtual environments (NVES). Today, applications from diverse domains such as cooperative work and military simulations are collectively called NVES. To cope with the differing requirements which are associated with these applications, a couple of frameworks have been proposed and implemented, see [7] for a categorization of these approaches and an overview over of corresponding implementations.
Each of these frameworks was built without any separation between the domain-specific and domain-independent issues. Therefore, these solutions tend to be relatively inflexible.

Currently, the entertainment industry is developing so-called Massively Multi-Player Online Role Games (MMORGS). A MMORG simulates a game-world which is populated by thousands of different characters. Some of these characters are controlled by human players while others are controlled by the MMORG system. The user is connected to the virtual game world through the internet and observes his or her character from the third person perspective. MMORGS are challenging for several reasons such as high scalability and realism.

We propose a transaction based middleware, called APEIRON, to support MMORGS. Because of the real-time character of such simulation, the common transaction semantics must be expanded to guarantee non-blocking access. We choose transactions as the central abstraction in our middleware, since they allow to state the required synchronization guarantees in an abstract and uniform way. Thus, we do not confront the application developer with a number of low-level primitives but with a single method to organize the synchronization of different applications. Also a number of different implementations can be used with the same principal interface such that different platforms can be supported effectively and more importantly, an incremental development is manageable in clean way.

The paper starts with a brief introduction of NVES. In the following section, MMORGS are described and presented as an interesting research object. Section 4 gives an overview on the domain specific services of a MMORG. Finally, Section 5 proposes a domain-independent middleware solution for MMORGS in particular (and NVES in general) followed by some concluding remarks.

2 Networked Virtual Environments

The terms virtual reality and virtual environment have been used with varying meanings. The ambiguity of these terms is rooted in the wide range of applications which are based upon virtual reality. Examples for such applications include military simulations, educational systems, as well as entertainment applications. Thus, many definitions on different flavors of virtual reality systems have been introduced, to distinguish between these varying technologies which are related to virtual environments. For example, in augmented reality, the real environment is seen through a display which shows some overlaid virtual environment. As a common denominator, we will use the following definition [10] which summarizes the most common characteristics of virtual environments.

> A virtual environment (VE) is an interactive, immersive, multi-sensory, 3 dimensional, synthetic environment.

The appearance of the modern PC with its high-resolution graphic displays and the widespread use of the Internet allow the development and broad deployment of networked virtual environments (NVES) supporting multi-user applications such as large-scale simulations or collaboration tools. In the context of virtual environments, networking is employed for two ends.

- First, networking can be used to distribute the load of the computations which are associated with a large-scale virtual environment over a number of computers.
- Second, networking allows a number of geographically dispersed persons to interact with each other within the same virtual environment.

Both types of distribution with respect to virtual environments share a number of problems. NVES are usually soft real-time applications [5], i.e., the average response time of the NVE should be low. Because of the complex interactions which arise between the objects of an NVE, this requires a comprehensive synchronization and communication layer.

In addition, both aspects of networking have specific problems. In the first case, a relatively small number of nodes is usually interconnected within a cluster, i.e., the complete environment is dedicated to the NVE. Ideally, such a cluster should mimic the behavior of a single machine, so for example, the different tasks should be tightly synchronized and the load should be dynamically balanced. SMP become more and more attractive and should be utilized effectively.

In the second case, if the Internet is used to allow a geographically dispersed group of people to interact within a virtual environment, the main problem is that the connection between the server system and client is not under the control of the NVE-system. Consequently, the central problem is to provide a responsive and consistent environment to the user in the presence of long transmission times and frequent packet loss [9].

3 MMORGS

3.1 The World of MMORGS

Massively Multi-Player Online Role Games (MMORGS) enable a large number of players to explore a persistently stored virtual game world using a fictionary character. Although the background stories of these games include different genres such as the middle-ages, fantasy, and science fiction, the principal appearance is always the same: A typically human-like character controlled by the player and observed by the player on the computer screen is acting in a virtual game world which may feature 3-dimensional geographical and architectural entities such as open landscapes, buildings and dungeons. The virtual game world is inhibited by a number of human controlled characters and computer controlled characters (non player characters or NPCS). From the player's perspective, one important goal of his/her interactions with the virtual game world is the development ("improvement") of the fictionary character by gaining experience, e.g., by exploring certain areas, by victories in fights with computer or human controlled characters. The other central motivation to play a MMORG is the interaction with other players within the virtual environment. Chatting, exchanging items of the game world, and exploring the game world collaboratively are essential to the experience of a MMORG player. Usually, MMORG players maintain their evolving characters over relatively long periods of time. It is not uncommon that players use the same the character over more than a year. To this end, a MMORG must maintain the game world itself and the characters persistently.

3.2 Research Interest into MMORGS

MMORGS are generally considered to be the most important recent development in the computer game industry. The first economically successful MMORG was *Ultima-Online*. The success of *Ultima Online* was inspiring – other MMORGS entered the emerging market, most notably *Everquest*, *Asheron's Call*, and *Anarachy Online*. In comparison to games which did not feature networked gaming (or only LAN-based gaming), these early MMORGS had a relatively weak game quality, especially with respect to graphics. The new generation of MMORGS, such as *Everquest II* and *Asheron's Call II*, presents to the player a world of almost the same quality as a traditional game. The economic interest into this genre is underlined by the fact that major companies in the IT-industry such as Sony, AT&T, and Microsoft show a strong interest in MMORGS and are actively developing products in this domain. The reason for the strong interest of the gaming industry into MMORGS is rooted in the fact that MMORGS are client-server applications which require players to authenticate themselves. Therefore the possibility to use illegal copies is eliminated. Moreover, for the same reason, it is possible to charge each player a monthly subscription fee.

From the technical and organizational point of view, MMORGS offer a new challenge to the gaming industry. So for example, the computational resources necessary to host a MMORG are by far larger than for any other kind of game. Also, the components which run server-side must be more reliable than usual game code. In general, the software qualities necessary to build a MMORG successfully, cannot be achieved by traditional game development strategies. For a long time, the game development community had been relatively uninterested into academic developments. However, as an article of the *Game Developer Magazine* titled "In Defense of Academe" (November 2002) shows, the gaming community becomes more and more interested in a more academic and methodologically grounded approach. More specifically, the following goals are shared by today's MMORG projects – and none of them is able to reach all of them.

- Today's MMORGS are supposed to allow 3.000 or more human players to be in the same simulated world. Roughly 30.000 NPCS must be maintained in the same world. This requires significant computational resources.
- MMORGS have to offer the player a logically consistent world, i.e., the world should follow its inherent game logic while the underlying implementation should not cause any unmotivated events. Of particular importance is the *seamlessness* of a MMORG. A MMORG is called seamless, if the distribution of a game world over a number of server-nodes does not lead to synchronization artifacts which can be observed by players.
- The world of any MMORG has to evolve over time. Changes on MMORGS include small changes such as the tuning of certain parameters to keep the game in balance[1], expansions such as adding a new area or a new non playing character, and global changes such as adding a new skill to all characters in the game. Such changes should happen in a fair manner, i.e., they should happen simultaneously for all characters. And ideally – changes should be executed without down-times of the MMORG server system.
- MMORGS are soft real-time services [5], i.e., a MMORG-system is required to respond to requests within a given time-bound on average with a small variance.
- Achieving high reliability is a prominent goal in the MMORG-community. Most of today's MMORGS display weak reliability which is degrading consumer satisfaction.
- Clients of MMORGS must be considered hostile since a certain fraction of players will always try to cheat. Any possibility to obtain some advantage in a MMORG will be exploited by cheating players.[2]

Summarized, MMORGS combine several features which make them an interesting object for research. First, MMORGS are a growing market which is currently conquered by industry. Second, MMORGS are technically very challenging and today's tools are not sufficient to build a MMORG which satisfies all these design goals. Third, the gaming industry has acknowledged the need for a more methodological approach and for academic support. Fourth, MMORGS are still games, i.e., while the goals of any MMORG-project are ambitious, partial failures are tolerable. Finally, we believe that MMORGS subsume many properties of other NVES. Therefore, a middleware which is able to satisfy the needs of a MMORG should be usable in a much broader context.

[1]A game is balanced, if there is no dominating strategy. For example, if a certain character type, such as a warrior, is always superior to another character type (e.g., a priest), then the game is unbalanced. To make a game interesting to players, it is important to balance a game as to ensure a diversity of characters and game approaches.

[2]If there is any advantage in crashing the MMORG servers, cheating players will do so. In 2001, *Asheron's Call* has been a victim of such an attack. The attackers' goal was to duplicate a set of given items. To achieve this, they moved these items from one server to another. But with a fine-tuned timing, the attackers were able to crash the first server just after leaving it. Therefore, the item was still stored in the backup of this server and at the same time in the hands of the player's character, thus it has been duplicated. Once the reboot of the crashed machine had finished, they attackers returned to obtain the duplicate.

4 MMORG Services

In this section, we will present a model for the domain specific services of a MMORG. To do so, we subdivide the domain of a MMORG-system into three sub-domains, namely the *gaming domain*, the *maintenance domain*, and the *production domain*, see Figure **??**. The gaming domain contains all services which are needed by the clients to play the game while the maintenance domain consists of those services which are needed for the management of the produced content, subscription management and other administrative tasks. Finally, the content-production domain embodies all services which are necessary to produce the content itself. Here, we will consider the content-production domain only in so far as we need to describe the gaming and maintenance domain.

In the following, we will briefly introduce these three sub-domains with their respective services.

4.1 Gaming Domain

The services of the gaming domain collaboratively maintain the game state and allow the client to access this state. These services have to operate under soft-realtime constraints, must be highly available, and have to scale well. In total there are the following four services.

Game-State Server: A MMORG's game-state is held and continuously processed by a cluster of game-state servers. A specific game-state server handles a set of zones, i.e., the server executed the control loop of all active objects, such as non playing characters, which are located in this zone. Therefore, a single game-state server must access the set of distributed, persistent, and active objects which constitute each of its zones and process their control loop.

Connection Server: Each connection server for a MMORG is responsible for managing the connections for a number of clients, i.e., to mediate the messages between its clients and the game-state servers which are currently holding the characters of these clients. To do so, the connection server has to select, prioritize and bulk the data which is sent to the client. This data is provided by the game-state and chat servers. On the other hand, the connection server has to validate each (potentially hostile) datum sent from the client to the server at the application level. In the case of a positive validation it has to forward each datum to the responsible server.

Chat Server: Because of the large number of chat messages which typically occur in MMORGS, dedicated chat servers are required. A chat server should mimic features of a commonly accepted chat system, such as the Internet Relay Chat (IRC). In addition to these standard features, there must be a special chat channel which is geographically segmented, i.e., which allows geographically near objects to exchange messages. To achieve this goal, the chat servers need to communicate with the game-state servers. The chat servers also need to communicate with the connection servers which constitute the interface to the client.

Game-Control Server: The game-control server of a MMORG governs all services of the gaming domain of the MMORG. It controls their start-up, shut-down, patching, and balancing. In general, the game-control server acts as facade of the gaming domain with respect to the maintenance domain.

The gaming services are tightly bound to each other, since they all access the game state. The game-state server is maintaining the objects which constitute the game-state, i.e., it keeps them active by executing their control loops. The connection server needs to map changes in the game-state to messages to be sent to the client and has to send the client-requests to these objects. The chat server has to know the geographic position of the characters, and finally, the game-control server has to execute global transactions on the game-state, such as expanding it. All these operations are required to be synchronized and thus all these services should be based on the same common middleware services and distribution middleware.

4.2 Maintenance Domain

The maintenance services do not share the extensive scalability and realtime constraints of the gaming domain. In contrast, they provide standard services and might be based on commercial-off-the-shelf (COTS) products. To achieve the required availability, simple duplication of the corresponding services suffices.

Login Server: The login process into a MMORG can be divided into two steps. First, the subscriber's identity and permissions have to be checked. If this validation is successful, then in the second step the connection has to be established. The first step is entirely handled by the login server while the second step is mainly delegated to the game-control server which activates the subscriber's character and initiates the connection (by issuing the necessary operations on the game-state and connection servers). The login server receives the IP-address of the selected connection server and forwards it to the client.

Download Server: The download servers of a MMORG must provide all software and data which is needed by the clients to enter the game. Because of the continuous development of a MMORG, the download server must also provide incremental patches such that a client can update itself by downloading only a minimal amount of data.

Update Server: The update server is responsible for providing the patches which arise from the changes and expansions of the MMORG's world. In other words, the update server is the interface to the content-production domain. The finally produced content needs to be submitted to the update server in terms of a corresponding client and server patch. The client patches have to be deployed to the download servers and after an appropriate period of time the associated server patch has to be executed by the game-control server. Once the game-control server applies a server patch, the connection servers will be informed to require each connecting client to be patched with the associated client update. Thus the update server provides the patches but it does not control their application.

Configuration Server. The configuration server is responsible for maintaining the complete configuration of all the services which constitute the running MMORG, i.e., it will associate a gaming domain cluster with a set of login, download and update servers. It will also know the current configuration of the MMORG with respect to patching and issue the complete patching procedure, that is, it will upload a client patch onto the download server to make it publicly available, and later, it will command the game-control server to apply the corresponding server patch to the game-state.
By using a configuration server to establish an association between the different services, it is also possible to use the same services for several MMORG-instances. For example, it

will be necessary to maintain several test games which use the same update server as the publicly running game.

The services of the maintenance domain are relatively loosely coupled. The login server maintains its subscription database completely isolated from the other services, it only interacts with the game-control server to initiate the connection establishment. The download server is the data-source for the data distribution, the only occurring interaction with the other services is the upload of new patches from the update server. The update server is the interface between the content production domain and the maintenance domain and provides distinct interfaces to both domains. From the perspective of the maintenance domain, the update server provides a simple data-repository which is used by the game-control and download server. This interface might be a standard protocol such as FTP. The configuration server will employ a standard database product to manage its configuration data. Again, the configuration server can be designed as a classical stand-alone database application.

4.3 Content-Production Domain

The content-production domain can be further subdivided into the art-pipeline and the game-design. Both domains are populated by applications which are used to create and maintain the game world. We will not discuss the concrete applications of these two domains since any concrete production domain will be specific to the concrete game and especially to the graphical representation.

The **art-pipeline** is based on standard graphics tools for 3D-modeling. Typically, these tools are expanded by custom scripts and plug-ins, to form a pipeline. Because of the massive amount of work to be done by the art-team, the organization of this work is critical. Therefore, the content-production domain is characterized by the integration and modification of existing tools to form an effective production environment. In addition to the graphics-tools, there need to be extensive version control facilities for binary as well as for textual data. Finally, the raw-models produced by the art-pipeline must be transformed into the client's data-format and for each model a description of the corresponding object which is sufficient for the game-state servers must be generated.

The **game-design** uses these models to build, modify and expand existing areas. The main challenge is again the integration of different tools. In the ideal case, a game-designer can employ an environment which allows her or him to place different objects visually in an area and to program these objects both visually and in terms of a scripting language. Naturally, the game-designer must be able to test new areas. The integration of testing facilities can be done in a number of different forms, ranging from a special client which comes with a completely integrated game-logic to the utilization of the common client connecting to a test gaming environment.

4.4 Domain Independent Services

In this subsection, we will discuss which of the three sub-domains gaming, maintenance, and production can be supported effectively by a domain independent middleware. Furthermore, we will discuss the characteristics which should be met by such a middleware.

The common characteristic of the **gaming domain's services** is that they access a shared database in a highly concurrent manner requiring replication, persistence and synchronization primitives under the pressure of soft-realtime constraints. This database represents the game-world which consists of a set of active objects, i.e., objects which are changing their state pro-actively.
Not building the gaming domain on a shared general middleware will lead to the stovepipe system anti-pattern [2]. This anti-pattern arises when all interfaces between pairs of communicating subsystems are distinct and mutually incompatible. In such a situation usually multiple infrastructure mechanisms are used to integrate these subsystems. This leads to difficulties in modifying or even describing the architecture. The consequences of such a development approach are large semantic gaps between architecture documentation and implemented software. The software might even comply to the paper requirements but it does not meet the user expectations, system maintenance becomes surprisingly costly, the project takes more resources than expected for no obvious reason, the system complexity increases heavily on even slight expansions.
There are two situations which justify to follow this anti-pattern consciously – the exploration of a yet unknown domain and the quick development of a partly functional prototype. But in such a situation, it is usually necessary to restart the project from scratch afterwards.

In contrast, the **maintenance domain's services** are loosely coupled and might be assembled in a heterogenous manner by utilizing standard packages for the corresponding tasks. For example, the login-server might be developed based on the Apache Web-Server and a commercially available billing package, and the download server can be a simple FTP-server. The interface between the maintenance domain and the gaming domain is provided by the game-control server. This server will provide an interface to the maintenance domain which is not based on the middleware, therefore the maintenance domain is completely independent from the middleware.

The **content-production domain** will consist of a set of diverse applications. Many of these applications will not use a live gaming environment. Typically, they will be third-party provided graphical tools which are used to model the graphical appearance of the artifacts in the game-world. Thus, these applications can be kept completely independent from the middleware.
On the other hand, some of the applications in the content-production domain will require access to a (running) game-world. For example, there could be a world-editor, which allows to compose new zones from base elements such as terrain types, houses, items etc. Ideally, such an editor would not utilize some specific implementation of the game but would use the same implementation as the gaming domain itself, i.e., would be based upon the same middleware. However, game-editors display completely different access patterns than the services of the gaming domain. Also, a MMORG will not be built with a single editor but with a (growing) set of editors which provided for different and specific tasks. Thus such a shared middleware must provide the means to integrate new applications which follow different access patterns at ease, i.e., the middleware must allow to compose the complete system out of applications which are built as independently as possible. Consequently, there must be a single shared horizontal interface such that compliance to this interface ensures integrability. This situation is one more strong reason to seek a strong horizontal layer and to avoid the consequences of a stovepipe system.

Summarized, the gaming domain must be the main focus of a middleware for MMORGS, i.e., the middleware has to enable a cluster to maintain a shared and persistent database of active objects which provides a broad set of synchronization primitives and which supports replication for the sake of efficiency and fault-tolerance. Furthermore, the middleware must be designed to meet soft-realtime constraints, since the operations of the gaming domain require response times with small variances. The operations on the database which are executed by the gaming domain cause relatively small changes to the game state but might require to read a comparatively large set of objects. In addition to the services of the gaming-domain, various editing applications of the content-production domain must access the shared database. These applications are operating on the database as a whole, for example, they might need to change a complete zone in the world or might require to change a large set of objects simultaneously.

Thus, a middleware for MMORGS has to support a wide variety of different access patterns. In particular, different synchronization modes must be available, i.e., the applications must be able to choose between serializable access forms and less isolated access forms, such as reading a possibly outdated but consistent snapshot of the database.

Taking this starting point, we will propose a middleware which allows a cluster of SMP machines to maintain a database of active objects. The middleware will be based on an expanded transaction concept which enables its applications to access the database with a broad set of synchronization strategies.

5 Transaction-based Gaming Domain

5.1 Transactions with Weak Isolation Levels

NVES (and even MMORGS) have a very broad set of different requirements such that it appears impractical to build a specific solution directly. Thus, we are developing a middleware, called APEIRON, as a domain-independent middleware which is designed to support the domain-specific services introduced in Section 4 on page 5.

The most natural modeling of an MMORG (and VES in general) is a database of optionally active objects. Each active object is continuously updating its state based on the current state of its environment. Ideally, such an object is implemented without any knowledge on the underlying distribution and concurrency issues. However, this ideal is hardly achievable. While it is possible to handle the distribution of the database internally by the middleware, it is not possible to handle concurrency issues implicitly.

If the implementation of the active objects has to deal with concurrency related issues explicitly, then the primitives for the synchronization should be as uniform as possible. Also, they should be abstract such that different concrete algorithms can be used to implement these primitives. The transaction is a concept which is flexible enough to offer different synchronization strategies but offers quite a uniform interface.

The classical notion of transactions are the ACID semantics. ACID is an acronym for atomicity, consistency, isolation, and durability. Atomicity refers to the fact, that all operations of an ACID transaction are executed completely or none of them. Consistency means that such a transaction transforms one consistent state of the database into another one. A transaction is isolated, if it is not affected by any other transaction, i.e., the transaction is accessing the database virtually exclusively. Finally, durability means that the effects of a transaction once executed are made durable. [3]

[3]The ACID semantics were introduced in [3]. For a general overview on classical transactions and their

Because of their serializability, ACID-transactions alone are insufficient. In addition to executing serializable transactions, the services of the gaming domain must be able to access objects in a non-blocking manner. For example, the decision what a non playing character is going to do next has to be made within soft-realtime, i.e., the time until the decision is found must be within a given bound almost all the time. Usually, the NPC has to know the state of the objects in its neighborhood to determine its next step. However, the decision of the NPC can be based on slightly outdated data.

To fit different synchronization scenarios, the transactions used in APEIRON allow the application to specify *isolation levels* for each accessed object [1]. The isolation-level of an accessed object describes the guarantees which are associated with this object with respect to all other objects which are locked within the same transaction. For example, the states of two different objects which are locked within a single transaction might be required to be mutually consistent.

The isolation levels of APEIRON form a hierarchy, i.e., every isolation-level includes the guarantees of the preceding ones. We list them below, starting at the weakest (committed) and finishing with the strongest (exclusive).

Committed: The object versions which are accessible under this isolation level are only guaranteed to be committed. Therefore, if two object are accessed under this isolation level, it is perfectly possible that the first object is accurately presented while the second one is already outdated.
If a transaction writes on an object under this isolation level, then lost updates might occur, i.e., the object might have been changed independently in the meantime.

Monotone: All objects which are locked under this isolation level (or a higher one) within the same transaction behave monotone with respect to the database development over time. More precisely, once an object state is made accessible to the application which incorporates modifications of a transaction T, then every object state which is locked afterwards must incorporate the changes of T. The behavior in case of a modifying access is the same as for the committed isolation level.

Consistent: This isolation level allows to lock a set of objects within a so-called consistency group, i.e., if one these objects reflects the effects of a transaction T, then all objects in this group must reflect the changes of T. If an object which is locked under the consistent isolation level is modified, then APEIRON enforces that no lost update occurs.

Accurate: If an object is locked under this isolation level, then the presented version must be up-to-date and no other transaction is allowed to change the object in the meantime. This scheme corresponds to classic ACID-transactions.

Pessimistic: The isolation level accurate and pessimistic are interchangeable – the correctness of a transaction is independent of this choice. However, such a change will have a strong impact on the performance of a transaction: If an object is locked with isolation-level pessimistic, then it is guaranteed that the corresponding transaction will succeed in validating the accesses this object. In other words, once the transaction has a pessimistic transactional lock on all locked objects, it knows it will be committable.
This isolation-level is useful for accessing high-contention data, i.e., objects which are

implementation, see [6].

modified by many transactions concurrently such that conflicts arise often. Naturally, obtaining a pessimistic lock might fail more often than acquiring an accurate lock.

Exclusive: This isolation-level gives exclusive access to an objects, thus it subsumes the guarantees of pessimistic. This isolation-level is mainly used as a synchronization primitive.

More precisely, APEIRON allows the application to specify an isolation-level which is required immediately and another one which is required directly before committing to the database.

We call the first isolation-level the **running isolation-level**. The application uses the running isolation-level to specify the isolation-level which is required such that the transaction itself does not crash. For example, a transaction might crash if the referential integrity of a set of objects is not guaranteed. In such a case, the application locks them under the running isolation-level consistent. Summarized, the running isolation-level is used to ensure that the transaction itself does not crash. However, it does not ensure that the transaction can be committed correctly.

The second one is called the **commitment isolation-level**. This isolation-level is enforced during the commitment of the corresponding transaction. This isolation-level can only be stronger than the running isolation-level (since the isolation-level can only be expanded over time). The commitment isolation-level is used to describe the guarantees which are required for a transaction such that its outcome is correct. For example, a transaction might access only two objects under the running isolation-level committed but might require a commitment isolation-level accurate. In this case, the transaction must be programmed such that inter-object inconsistencies does not cause the transaction to crash, but the outcome of the transaction must only be correct, if the locked states of the two objects were up-to-date and are unmodified at the moment of commitment.

5.2 Services as Active Objects

As pointed out in [4], the migration of a character is comparable to the migration of an process. In both cases an active objects has to be suspended, transfered to another node, and resumed again. In the case of the migration of a character, the additional difficulty is to achieve migration as quickly as possible – the character should be seamlessly observable.

In an NVE-cluster, the objects which are observable from more than one node must be replicated in order to achieve the real time character of the simulation. The replicas will not be perfect but slightly outdated with respect to the master instance. The weaker isolation levels which allow to access slightly outdated versions of an object can use such replicated versions of an object. If a character moves from one node to another, typically it has been replicated at the new node for quite some time. In such a situation, the migration is only required to change a former slave replica into a master replica and vice versa.

Since each NVE-cluster solution must come up with a solution for character migration, it is actually providing a solution for process migration. Therefore, within APEIRON, services are modeled in terms of active objects. In consequence, all features associated with common objects are available for services. Particularly, services can be maintained in fault-tolerant manner and can be migrated for the sake of load-balancing.

5.3 Incremental Implementation

The transaction interface is the main interface for applications. As stated in the preceding subsections, we are using transactions as interface since they allow to state different synchronization strategies uniformly. At the same time, they are sufficiently abstract such that a variety of different implementations can be applied. In particular, it is possible to build the underlying implementation incrementally.

An implementation can be restricted in terms of a conservative implementation, i.e., instead of implementing an isolation level, the implementation can just use the next higher isolation level.

Another option for incremental development is to vary the implementation of the distribution mechanisms. The common application interface of APEIRON does not give allow to manipulate the distribution of objects. This interface only allows to constrain the replication of an object – the way these constraints are enforces is not observable through the interface.[4] Therefore, it is possible to start with a relatively simple implementation for SMP machines without any distribution, or to build an implementation which does not maintain any replicas for the sake of fault-tolerance.

6 Conclusion

We presented MMORGS and motivated the design of APEIRON by the requirements of MMORGS. However, APEIRON itself is domain-independent and provides a general and consistent set of functionality to the application developer. Therefore, we think that APEIRON can be used as basis for other types of NVES – at least to solve the problem of load distribution. APEIRON leaves the problem of geographical distribution to the domain-specific and/or application level since there is no general solution for it.

The key-concept of APEIRON is a transaction interface which is based on weak isolation levels. Such an interface is intuitive to application developers and clearly separates the applications from any low-level operations. Therefore, it becomes possible to develop applications atop of APEIRON using a uniform interface concept, based on varying implementations.

Currently, we are deriving a complete architecture for APEIRON and started to implement basic modules. We hope to have a running prototype for SMP-machines supporting at least two weak isolation levels by the end of 2004.

References

[1] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementation for Distributed Transactions*. PhD thesis, MIT, 1999.

[2] William J. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. *Anti Patterns*. Wiley & Sons, 1998.

[3] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *Computing Surveys*, 15(4), 1983.

[4] J. Huang, Y. Du, and C.-M. Wang. Design of the server cluster to support avatar migration. In *Virtual Reality*, pages 7–14, 2003.

[4]There is a separate interface which allows to access and manipulate the underlying meta-data which contains the underlying platform parameters and the configuration of the active databases.

[5] Hermann Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[6] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[7] Michael R. Macedonia and Michael J. Zyda. A taxonomy for networked virtual environments. *IEEE MultiMedia*, 4(1):48–56, – 1997.

[8] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. C++ In-Depth Series. Addison-Wesley, 2002.

[9] Sandeep Kishan Singhal. *Effective Remote Modelling in Large-Scale Distributed Simulation and Visualization Environments*. PhD thesis, Stanford University, 1996.

[10] R. Stuart. *The Design of Virtual Environments*. Computing McGraw-Hill, 1996.