A Platform for Dynamic Microcell Redeployment in Massively Multiplayer Online Games

Bruno Van Den Bossche
brvdboss@intec.ugent.be

Tom Verdickt
tverdick@intec.ugent.be

Bart De Vleeschauwer
bdevlees@intec.ugent.be

Stein Desmet

Stijn De Mulder

Filip De Turck

Bart Dhoedt

Piet Demeester

Ghent University - IBBT - IMEC, Department of Information Technology
Gaston Crommenlaan 8 bus 201, 9050 Gent, Belgium
Tel: +3293314900, Fax: +3293314899

ABSTRACT

As Massively Multiplayer Online Games enjoy a huge popularity and are played by tens of thousands of players simultaneously, an efficient software architecture is needed to cope with the dynamically changing loads at the server side. In this paper we discuss a novel way to support this kind of application by dividing the virtual world into several parts, called microcells. Every server is assigned a number of microcells and by dynamically redeploying these microcells when the load in a region of the world suddenly increases, the platform is able to adapt to changing load distributions. The software architecture for this system is described and we also provide some evaluation results that indicate the performance of our platform.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Performance, Design

Keywords

MMOG, Game Server Architecture, Load Balancing, Microcell Distribution

1. INTRODUCTION

With the advance of technology and the availability of broadband Internet access, interactive multimedia applications are becoming increasingly popular. One class of these applications is online games, especially Massively Multiplayer Online Games (MMOGs). These applications offer their users a huge virtual world in which they can interact with tens of thousands of other players and build new virtual identities. Examples of these applications include World of Warcraft and Lineage 2, each boasting millions of subscriptions and hundreds of thousands of simultaneous users. One of the main characteristics of these applications is that they are highly interactive and generate an enormous load due to their massive scale.

An efficient architecture is needed to manage the virtual worlds and provide a continuous service to its players. Two traditional ways to support an MMOG are the approaches followed in World of Warcraft and in Second Life. In the former, the virtual world is divided into several realms. By not allowing players to communicate between realms, the load is divided over a number of server clusters that may even be geographically dispersed. In Second Life, the world is divided into several "cells". A cell is a part of the virtual world and each of these cells is managed by one single server. Players can move freely from one cell to another. However, both of these approaches are not able to cope with an uneven and dynamically changing player distribution. When a lot of players are concentrated in one realm or cell, the responsible server gets overloaded, resulting in severe degradation of the game experience for the players in these regions.

To cope with the dynamics of MMOGs, we developed a novel technique to dynamically redistribute the load over a set of servers [4]. We split the virtual world into several small parts, called microcells. Every server of the system is responsible for a set of microcells. When some server is experiencing a high load, due to a high player density in the microcells it is responsible for, some of its microcells are assigned to another server, thereby reducing the load the server is experiencing. In this way, we are able to dynamically alter the portions of the virtual world the servers are responsible for. This enables us to evenly distribute the total system load over the available servers, thereby eliminating potential bottlenecks before they occur. Figure 1 illustrates the concepts of using microcells to divide the virtual world and to distribute the load, taking into account the player densities in the regions.

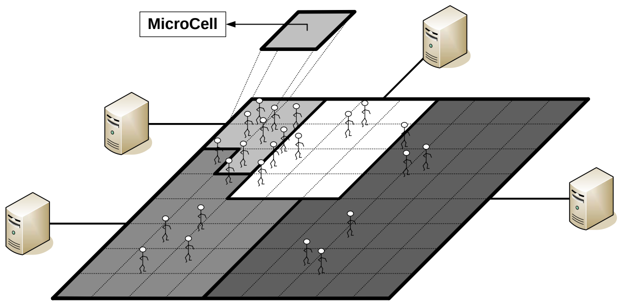

Figure 1: The virtual world is divided into several microcells. The microcells are assigned to a number of servers, where each server gets an similar amount of load.

In this paper we describe and evaluate the software architecture for our microcell based MMOG platform. The platform is built using the J2EE programming platform since it provides several features that enable us to design and deploy our application quickly.

This paper is outlined as follows: section 3 contains a detailed description of the architecture of our MMOG platform. Section 4 focuses on the migration of microcells from one server to another. The evaluation of some of the important characteristics of our architecture is presented in 5 and conclusions and future work are discussed in section 6.

2. RELATED WORK

The idea of dividing the game world of an MMOG in smaller parts to make it more manageable is not new and has already been investigated. In [5] the publish-subscribe paradigm is applied to an MMOG architecture where the world is divided into several cells. They also investigate the influence of the shape of the cells on the communication overhead that is induced by inter cell communication. Using hexagonal cells proves to be the most efficient shape for a cell. The architecture that we present in this paper does not put any restrictions on the shape or layout of a cell and could therefore be easily modified to use hexagonal cells.

In [7] a Distributed-organized Information Terra Platform (DoIT) is proposed, which offers a middleware platform to simplify the development, deployment and management of MMOGs. However, it does not offer a solution for dealing with *hotspots* where a large number of players migrate to the same part of the world, thus overloading the server responsible for it.

Another approach is taken by BigWorld [2] and EVE Online [3] where the load distribution is not handled by reallocating parts of the world to other servers but by shifting processing resources to the regions in need of extra processing power. Although this simplifies the development and management of an MMOG, it is less flexible than the software based solution proposed in this paper.

3. ARCHITECTURE

When designing a complex distributed application like an MMOG, measures must be taken to avoid architecture induced bottlenecks. Performance scalability needs to be achieved, as it must be possible to add extra servers and bandwidth, without the need to actually reconfigure the application or the deployment of the application itself.

When the concept of dividing the world into a fairly large number of microcells is used, it is beneficial to maintain a mapping between the logical world model and the architectural components of the application and thus have each microcell represented by a logical software component. In order to be able to actually influence the distribution of the microcells we need additional components to create, delete, move and basically manage microcells. One component, called the MicroCellController is present on every server in the cluster and takes care of the microcells assigned to it. A second (global) component, called the MicroCellManager takes care of managing the global deployment of the microcells over the available servers. The ActorController is responsible for all interactions between the application and the players per server.

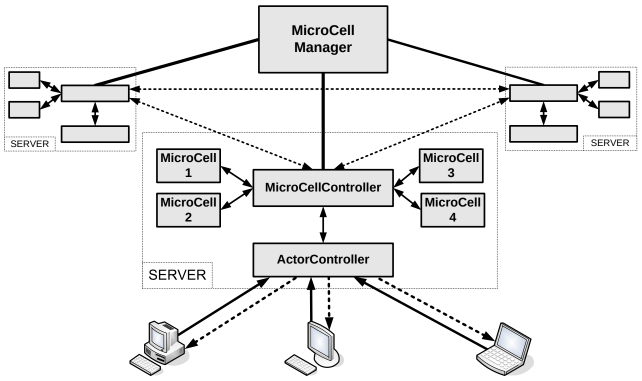

Figure 2: Global architecture of the microcell based MMOG platform. The microcells are deployed on the available servers and are managed by per server components and one centralized component.

Figure 2 shows the global architecture and how components interact. A more detailed explanation of the components and their role is presented in the following subsections. Furthermore an overview of the inter-component interactions is detailed to clarify the behavior of the application. But first we motivate the choice for J2EE as the underlying platform for the development of the application.

3.1 Why J2EE?

Developing the MMOG architecture with movable cells from scratch would be very tedious and error prone. Therefore existing architectures and software platforms were investigated and evaluated. J2EE, although not the most obvious choice, provides a large number of interesting features for the application in mind.

The development model of J2EE, with applications being composed of a number of smaller modular components fits the description of an architecture consisting of mobile components. Even more as J2EE offers a solid base for developing network enabled applications. Distributing components and accessing them over the network is handled transparently by the Application Server.

Another important feature of J2EE is the built-in support for asynchronous message-driven execution through the use of JMS [6]. This allows asynchronous processing of user and system generated events.

As J2EE is a Java technology it allows for platform independent development of applications. Even more, it allows the same application to be deployed on multiple J2EE application servers. Although platform independence might not be a primary issue when developing very specific applications such as an MMOG the use of J2EE does allow to easily switch application servers if the application server used would prove to have certain performance limitations.

3.2 Components

The MicroCell component is the virtual representation of the microcell. It contains all the data associated with a microcell, such as the items that can be found in that part of the world, the environment conditions and also all players and their respective information, such as their current position, the items they carry etc. The implementation of the MicroCell consists of an Entity Bean and a Session Facade [8]. The Entity Bean representing the microcell contains references to other Entity Beans representing the contents of the microcell, the players currently inhabiting it, neighboring microcells, etc.

The ActorController takes care of all world-user interactions. This component offers an interface to the clients that allows them to send events, such as movements, actions or interactions with the environment or other players. It manages all the events and messages that need to be sent to the players such as other players that come within viewing range, the actions of those other players and the properties of non-static game data. The ActorController is implemented as a Stateless Session Bean. There is exactly one ActorController running on each server of the MMOG.

The MicroCellController is a key component in the architecture. It is responsible for the management of the microcells on the server and acts as the central access point for all actions that microcells participate in. The movement of a cell is, once initiated, under the total control of the MicroCellController. It contacts the destination server, moves the actual data, and takes all the necessary steps to arrange the redirection of message flows and clients. In order to be able to do all this, it is required that all communication between microcells and between microcells and clients passes through the MicroCellController. As this is a rather complicated logical unit, it is split into two subcomponents which are further detailed in subsection 3.3. A more thorough description of the role of the MicroCellController can be found in section 4. There is exactly one MicroCellController running on each server part of the MMOG.

The MicroCellManager is the component responsible for initiating the relocation of the microcells. It monitors the application components, the system load generated and determines more optimal microcell distributions if necessary. As this component does not contain any game logic, nor influences the game logic directly, it is currently designed to be one centralized component. However nothing prevents implementing a distributed or peer-to-peer version of this component. Furthermore, this component is not strictly necessary for the MMOG to function properly. A breakdown will only disable the redeployment of the microcells, resulting in a situation similar to current MMOGs with a static deployment. The MicroCellManager can be implemented using a Stateless Session Bean.

3.3 Communication

This section describes the interactions between the different components. An overview of all important interactions between the components is shown in figure 3. It is important to note there are two types of interactions used, namely synchronous communications, which correspond to method calls both local and remote, and asynchronous communications which correspond to the use of JMS.

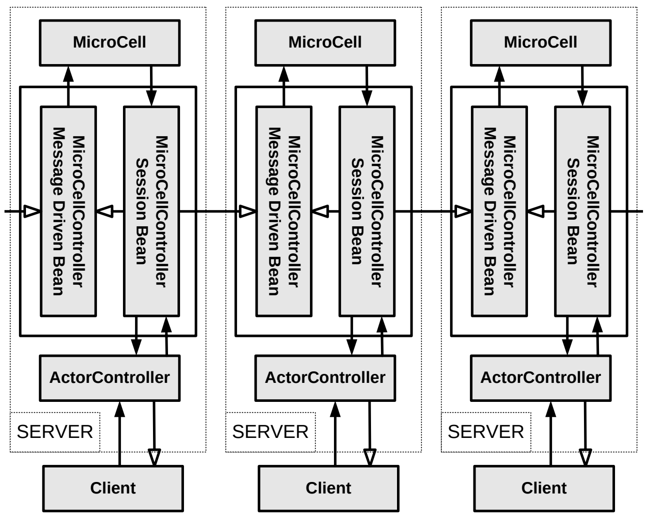
MicroCell
MicroCellController Message Driven Bean
MicroCellController Session Bean
ActorController
SERVER
Client

Figure 3: Overview of the communication patterns of the microcell based MMOG platform. The full arrow heads represent synchronous communication, the hollow arrow heads represent asynchronous communication.

This is why the MicroCellController is shown as a combination of two separate components. One part is a Message-Driven Bean which is responsible for the asynchronous processing of incoming messages and the other part is a Stateless Session Bean which allows the other components to generate the messages and send them to the correct destination. This design pattern is also known as the Message Facade [8]. Using this approach allows asynchronous processing and creates an abstraction layer to hide the use of JMS.

An example of a player generating an event and the processing of the event is detailed to explain the internal communications. When a client performs an action, he sends this action to the ActorController. The ActorController forwards this event to the MicroCellController which turns it into a JMS Message to allow for asynchronous processing and to greatly reduce blocking times which would prevent the client from sending other actions. The JMS Message containing the player action is then processed by the MicroCellController which performs the necessary logic on the MicroCell. As a result of this action, e.g. the player chopping down a tree, a MicroCell could generate one or more events of its own, e.g. the tree fell down and crushed another player. These are sent to the MicroCellController, converted into JMS messages and then processed. This means forwarded to the ActorController and then to the players close enough to the source of events or even forwarded to the neighboring microcells if the action took place close enough to the border for it to be seen in an adjacent microcell.

If neighboring cells are located on a different server this means the MicroCellController sends a message to the MicroCellController on that server, which accepts the event and takes care of the necessary processing on that server. Similarly, any MicroCellController can receive events from external MicroCellControllers responsible for neighboring microcells. When moving microcells between servers, similar actions are taken, although the initiating event is not a player action, but an action from the MicroCellManager and there will always be at least two MicroCellControllers involved.

3.4 Extensions

The architecture as detailed in this paper allows clients to interface directly with the application. In a real world situation this is not desirable and the use of extra proxy-servers would be preferred. These servers take care of the clients logging in and directing the requests to the correct ActorControllers. However, this does not change the core architecture as this would only add an extra layer between the clients and the application.

A second extension is the use of a different messaging system than JMS. The actual use of messaging is confined to a limited number of components and abstracted through the use of the Message-Facade pattern. Therefore it can be easily replaced with another messaging or event based system.

4. MICROCELL MIGRATION

As explained in section 1, the goal of migrating microcells between servers is to evenly distribute server load, or to prevent servers from becoming overloaded. Thus, when the load on a server exceeds a certain threshold, one or more microcells are moved from that server to another server.

An important requirement of this microcell migration is that it should happen in mid-play, without hindering gameplay. This not only means that the overhead for moving a microcell should be minimized. It also means that the game state should be kept consistent throughout the migration and that play should not be interrupted for players residing in or close to the microcell being moved. This is made even more complex by the fact that, apart from the microcell itself and its data, the connections between the microcell and its neighbors need to be moved, as well as the connections between the microcell and the players in the microcell, or rather between the ActorController of the microcell and the players.

Therefore, a two-phase migration process is used to ensure continuing and correct game-play during and after the microcell migration. First, a copy of the microcell is made on the new server. The “old” microcell remains active and the new microcell copy is kept up-to-date when it is created. This is the *relocation* phase. During the second phase, the *rerouting* phase, control is passed on to the new microcell copy, and a message is sent to the players in the microcell and to the neighboring microcells, asking them to reconnect to the new copy. Finally, after all players are reconnected, the old copy can be removed and the microcell migration is completed. A more detailed description of the different migration phases will be provided in the remainder of this section.

The microcell migration process is largely independent of lower-level aspects of the platform. Therefore, a conceptual overview of the migration process is provided here, without mentioning JMS. Similarly, the MicroCellController is described as one solid component in this section, while in reality it consists of two separate components, as explained in section 3.3 and figure 3.

4.1 The relocation phase

The first part of moving a microcell from one server to another is moving the microcell data (figure 4). This is done by creating a new microcell on the target server, and loading that new microcell with the data of the microcell to be moved. Meanwhile, the current copy of the microcell can continue operating, ensuring uninterrupted gameplay.

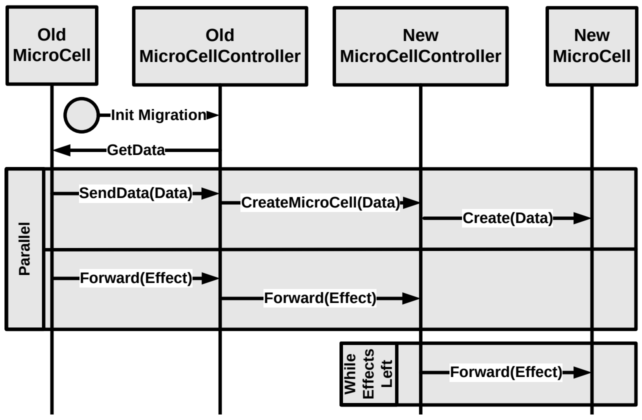

Figure 4: The relocation phase of the microcell migration, consisting of 2 parallel parts: copying the microcell data and forwarding the effects of arriving actions.

While copying the microcell and its data to the new location, new player actions will be arriving at the microcell. When the microcell processes these actions, its internal state might change, e.g. when the player cuts down a tree or picks up a rock. This in turn causes inconsistencies between the current microcell copy and the newly created copy, since the results of actions performed during the copying are not present in the new microcell copy.

To solve this problem, the new microcell copy must be updated again after it has been created. This is performed as follows: when the microcell relocation starts, the MicroCellController warns the microcell that it is being relocated. From that moment on, each time the microcell processes an action, the result of the action (its effect) is sent to the MicroCellController, who forwards it to the MicroCellController of the new microcell copy. These effects are processed by the new microcell copy once its creation is finished, bringing the new copy up to date with the “old”, still functioning, copy.

Note that it is important to forward the results of player actions to the new microcell copy, instead of the actions themselves. If the actions would be forwarded directly, and thus would be processed both by the old and the new microcell copy, inconsistencies between both versions might arise if the result of the action depends on some random value, the exact timing, or other factors which might be different for both versions.

Once the backlog of effects is processed and the new copy is up to date, the MicroCellController of the new copy sends a message to that of the old copy, asking for the next phase of the migration to be started. It is possible for further effects to arrive at the new cell copy after this time, e.g. because they were still traveling through the network when the request for the next phase was sent and had not yet been received by the MicroCellController. This is not a problem however, as they are still forwarded to the new microcell copy, to bring it further up to date.

4.2 The rerouting phase

The goal of the second phase of the microcell migration is to start using the new microcell copy. At that point, the currently used copy becomes redundant and can be erased, which was the goal of migrating the microcell. This can be done simply by making sure that all further actions and other messages sent to the microcell are sent to the new copy instead of to the old one. Since messages are sent by players and neighboring microcells (and their MicroCellControllers), these will all have to be informed of the new location of the microcell. There is no further need to "activate" the new microcell copy. Once it is up to date, it is ready to process further player actions, and can thus immediately take over from the current copy, as long as all further messages to the microcell arrive at the new copy instead of the old (current) one.

Informing the neighboring cells of the new microcell location is fairly straightforward. The dispatching of messages is performed by the MicroCellControllers, so only they need to know the microcell locations. The new location is simply sent to the MicroCellManager. The MicroCellManager then informs the MicroCellControllers of the microcells neighboring the migrated cell of its new location, so they can send their messages directly to the new microcell copy.

Rerouting the client connections is equally straightforward. The MicroCellController (of the old microcell) sends a message to its ActorController to inform the clients of all players in the microcell of the new microcell location. After a client receives such a message, it sends further actions to the new location, and thus to the new microcell copy (via the ActorController and the MicroCellController of the new microcell).

The difficulty of the rerouting phase however is that none of these things happen instantaneously. While the location update messages are sent to the clients, new player actions may arrive at the old microcell location, where a microcell is still present, but should no longer be used. Another issue is that the new microcell, which is already fully functional, might need to send a message to a client that is still connected to the old microcell copy. This situation could arise because of an action from a player that has already connected to the new copy.

These problems are solved as follows. Once the relocation phase finishes and the new microcell copy goes live, the old microcell copy still accepts player actions and other messages. Only instead of processing them, it forwards them to the new, currently active, microcell copy. The new copy then processes the actions, thus ensuring once again that all actions are processed only once.

However, this does not solve the entire problem. After the player action is processed, its results still need to be sent to the player itself, and possibly to a number of different players in the same cell. If a player is already connected to the new microcell copy, this is not a problem. The message with the result of the action can just be sent to the player. However, it is possible that the player has not yet connected to the new copy, and is instead still connected to the old copy. If that is the case, the microcell can not directly send a message to the player, as it doesn't know the client location. Therefore, before sending a message to a client, the microcell checks whether the client is connected to the new microcell copy or to the old one. More accurately, the MicroCellController performs this check, as all messages to and from the microcell pass through the controller. If the client is connected to the new copy, the message is sent directly to the client. If the client is not yet connected to the new copy, the message is forwarded to the MicroCellController of the old copy, to which the client is still connected, which can then send the message to the client (or more correctly, the ActorController will send the message to the client).

Checking whether a client has connected to the new server or not can be done by keeping a list of clients to connect to the new copy when the rerouting phase starts. Each time a client sends an action to the new copy, that client can be removed from the list. Once the list is empty, all clients have been connected to the new microcell copy. This concludes the rerouting phase, as the old copy will no longer receive further player actions, and can now safely be removed. To clarify to rerouting of the client connections when a microcell migrates between two servers, see figure 5.

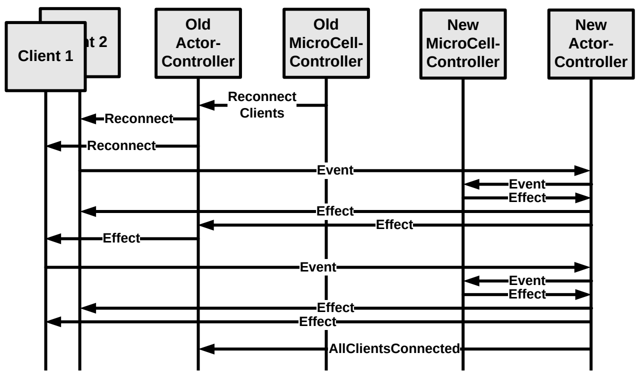

Figure 5: After receiving a request to connect to the new microcell Client 2 sends an event to the new microcell copy. As Client 1 is not yet reconnected it receives the resulting effect via the old ActorController. As soon as Client 1 is reconnected to the new microcell as indicated by it sending an event, it receives further effects directly.

Note that clients, after connecting to the new microcell copy, still need to listen to the connection with the old copy, because it is possible that a message to the client is underway via the old copy when the client reconnects to the new copy. In order not to miss this message, the client needs to listen to the old connection for a short time after connecting to the new microcell copy.

5. ARCHITECTURE EVALUATION

Performance is an important consideration for most distributed systems. Even more so for multiplayer games, where performance plays a crucial part in the way players experience the game. Large latencies prevent the game from running smoothly, and might even render it entirely unplayable [1]. Therefore, some preliminary tests were performed to assess the JMS communication mechanism, which handles an important part of the inter-component communication, and as such might have an important influence on the performance of the platform.

The goal of the test was to have an indication of the performance impact of using JMS for the communication between MicrocellControllers. The test consisted of a client and a stateless session bean. The client makes a call to the session bean which results in a JMS message being sent back.

For this test, the client was deployed on an AMD athlon xp 1400 with 256 MB RAM. The server machine (with the session bean) was a dual opteron 1.6 GHz with 4 GB RAM, a Java virtual machine with maxmem set to 1 GB, and a JBoss 4.03SP1 application server, tuned for low latency applications. A 100 Mb network connected both machines.

The round-trip time (RTT) for this entire scenario was measured for a number of different message sizes. Each test was performed 1000 times, discarding the first 250 results to allow for the steady-state performance to be measured. The results of the tests are presented in table 1. Note that these tests were not run on the kind of hardware an MMOG platform should consist of. This means that the RTTs in the final system would be smaller than those presented in the table.

Table 1: Measured round-trip times for JMS.

message size	average RTT
750 B	3.22 ms
7500 B	4.66 ms
75000 B	29.83 ms

The test shows that, while JMS does incur some overhead, it is still a viable means of communication in the platform. The latency caused by using JMS inside the platform is not a prohibiting factor, especially for small messages, up to a few KB. Most messages however are small, e.g. player actions, or messages telling that a player has successfully picked up a rock and that it has been added to his inventory. Large messages are only used when large amounts of data need to be transferred. This happens for example during microcells migrations, which occur rarely compared to player actions.

6. CONCLUSIONS AND FUTURE WORK

In this paper, an MMOG platform was presented that adapts to varying load distributions by dynamically redeploying part of the game world. The platform uses well-known application platforms and mechanisms, such as J2EE and JMS. A description has also been provided of an algorithm for the migration of microcells during play, a key feature of the platform. Preliminary performance tests indicate that JMS is a viable alternative for inter-component communication.

One area that needs more research is the design of algorithms to determine the "optimal" microcell deployment. A number of algorithms were already proposed and evaluated in [4]. However, these algorithms do not take the current deployment into account. Therefore, in order to completely take advantage of the dynamic nature of the microcell deployment, a new set of algorithms is needed that try to find a balance between the performance of the new deployment and the effort needed to redeploy the microcells (e.g. the number of microcells to be moved).

We are currently implementing the outlined architecture. This implementation will be used to test its performance under varying loads and to improve the microcell deployment algorithms.

7. ACKNOWLEDGMENTS

T. Verdickt is supported by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

The research in this paper is partially funded by the Interdisciplinary institute for Broadband Technology (IBBT) T-Case project.

F. De Turck is a postdoctoral Fellow of the Fund for Scientific Research - Flanders (F.W.O.-Vlaanderen).

8. REFERENCES

[1] G. Armitage. An experimental estimation of latency sensitivity in multiplayer Quake 3. In *Proc. of the 11th IEEE International Conference on Networks (ICON2003).*

[2] BigWorld Pty Ltd. [online]. http://www.bigworldtech.com.

[3] CCP. [online]. http://www.eve-online.com/.

[4] B. De Vleeschauwer, B. Van Den Bossche, T. Verdickt, F. De Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for massively multiplayer online gaming. In *Proceedings of Netgames 2005: 4th Workshop on Network and Systems Support for Games*, New York, USA, 2005.

[5] S. Fiedler, M. Wallner, and M. Weber. A communication architecture for massive multiplayer games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 14–22, New York, NY, USA, 2002. ACM Press.

[6] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Haase. *Java Message Service API Tutorial and Reference: Messaging for the J2EE Platform.* Addison-Wesley Professional, 2002.

[7] T.-Y. Hsiao and Y. Shyan-Ming. Practical middleware for massively multiplayer online games. *IEEE Internet Computing*, 9(5):47–54, September-October 2005.

[8] F. Marinescu. *EJB Design Patterns: Advanced Patterns, Processes and Idioms.* Wiley Computer Publishing, 2002.