Latency Reduction in Massively Multi-player Online Games by Partial Migration of Game State

Paul B. Beskow[1], Pål Halvorsen[1,2], Carsten Griwodz[1,2]
[1]IFI, University of Oslo, Norway [2]Simula Research Laboratory, Norway
Email: {paulbb, paalh, griff}@ifi.uio.no

Abstract

With the increasing popularity of massively multi-player online games (MMOGs), developers are continually forced to deal with the conflicting requirements of supporting a large number of concurrent users, while simultaneously providing low latency. As a result, a common way of distributing load is by dividing the virtual environment into logical regions. Geographically coupled users in such regions can be distinguished by analyzing IP addresses, RTTs or similar. This paper proposes the architecture for a decentralized middleware capable of utilizing such information, with the intent of decreasing the overall latency for the majority of users in that region. The latency reduction is accomplished by migrating a game region to a server closer in locality to the users, thereby lowering the response time of remote procedure calls. Due to the characteristics of MMOGs, the middleware implements a distributed name service, made possible by activating the system from a single node in the system.

1 Introduction

In recent years, massively multi-player online games (MMOGs) have become increasingly popular among consumers. According to the Entertainment Software Association [1], the number of gamers who play online games has increased from 31 percent in 2002 to 44 percent in 2006. In correspondence, the number of subscribers to MMOGs has steadily increased since 1997, exceeding 13 million in 2006 [2]. These applications enable a user to form and maintain social bonds, using the virtual environment as an interface. The persistent, continuous and interactive nature of this genre of games has largely contributed to this success. Incidentally, the success due to interactivity also raises the most challenging system requirement, i.e, *low latency* for all users.

MMOGs allow thousands of users to concurrently interact in a persistent virtual environment. A significant characteristic of this type of application is its lack of resilience towards network transmission delays [3–5]. For the users to have a consistent view of the world, the events of the game need to be distributed as fast as possible. With a large number of users, the quantity of events to distribute are considerable. This causes the capacity of the servers to deteriorate rapidly, which in turn lowers the quality of service. To support these virtual environments, with such considerable numbers of interacting entities, there is a need for an efficient architecture able to handle the load generated. As a result, MMOGs commonly deploy an exclusive instance of the virtual environment on a single, centralized, distributed system, such as a cluster, grid or mainframe. One such instance of the game world is commonly referred to as a shard, a concept inherited from Ultima Online [6].

There are varying strategies for how these game worlds are deployed. EVE Online [7], for instance, only utilizes a single shard, to which all subscribers connect, and this single cluster can handle 30,000 concurrent users. In contrast, games such as World of Warcraft (WoW) [8] use multiple shards, which can support approximately 3,000 concurrent users per shard. As the load increases, more shards are made available. Alternately, shards are merged if the amount of users becomes significantly lower. A main difference is that the users of EVE Online will have the ability to interact with all users in the same environment, while in WoW, the users will have to agree on a server to connect to in order to play together.

Irrespective of the number of users supported per shard, it is common to divide the game world into regions (also referred to as cells). This is to ensure an even distribution of the load and is accomplished by reducing the number of entities that updates need to be issued to. Most commonly, these regions are divided statically. Players then connect to the server managing the region of the game world where their character is situated. As the player moves between regions, the player's data is migrated to the server handling that region. However, as far as the user is concerned, the game world is a single entity.

The users interacting in MMOGs commonly are from widely different areas of the world. In many distributed systems, the effect of this is not necessarily prevalent, mainly because the architecture of the application can adapt to this by distributing users to servers accordingly. With MMOGs, users cannot necessarily be separated to accommodate this, because of the interaction that occurs with other users in the virtual environment. As a result, users cannot be placed in the virtual world according to their physical location. It becomes apparent that the static region based architecture on a centralized cluster, while efficient and relatively easy to maintain, does not cater to the varying geographical locations of the users. It can be argued that games, such as WoW, to a certain degree take this into account. They have shards located at multiple locations across the world, where users generally connect to one close in proximity. The distribution of users, however, occurs as a result of the availability of servers, not because of how the middleware is implemented. An ideal example of this is EVE, with its single shard structure, where all users are connected to the same centralized cluster. Users from all over the world interact in the same logical regions, with one region hosted by a single server. The result is an architecture which cannot adjust itself to the difference in latency among its users.

Thus, it would be better to have a virtual world where the regions could be managed by nodes geographically located closer to the majority of the users. In this context, a recent study [9] of the MMOG Anarchy Online [10], analyzing the RTTs in traces from one of several hundred regions composing the virtual environment, three distinct groupings of users were revealed. Based on the location of the server, these were USA, Asia and Europe. It is safe to assume that one of these groups will be dominant, depending on the time of day. Thus, the assumption is that by analyzing the latency of users in a region from the virtual environment, one can determine where they are approximately located geographically. Similarly, one could use the IP addresses of the users, if available, to obtain this information. Though with a lot of research being done on integrating peer to peer based systems into MMOGs [11], there is no guarantee that such information is readily available. As far as we can determine, there has not been done a lot of research into the possibilities connected to load-balancing the regions of an MMOG based on the geographical location of users currently located within it. Most of the research is focused on effective load-balancing within a centralized cluster, by dynamically re-locating regions based on overall load. Therefore, this paper proposes the architecture of a middleware which will allow for the development of MMOGs which are aware of the physical locality of its users. The intent is to lower the response time of remote procedure calls for the majority of users connected to a given region, which in turn should lower the overall latency. This is accomplished by migrating the region to a server closer in physical locality.

Our middleware is based on a distributed model, where a single node acts as the point of initialization. That is to say, a single server in the system initializes all communication and object creation, which is necessary in order for objects in our system to be located after migration. This is a result of the way we implement our name service (see section 3). As it becomes necessary to migrate objects from one server to another, objects which share common characteristics, such as belonging to an application-defined region, are added to migration groups. These migration groups are formed on the basis of migration policies, which will select objects, managed by that node, based on a set of criteria. After the selection process, the objects in these groups are migrated to the server selected by the application. One possibility is to migrate based on geographical locations and the respective latencies. A goal of this project is to implement this middleware in order to see if placing objects in such a manner will benefit the majority of users in terms of latency. The architecture is designed in such a way that load balancing can be performed based on any number of requirements. If the load on the server becomes too large, another migration policy can be activated. As such, migration can occur implicit or explicit. In this scenario, an explicit migration is performed when a user moves from one region to another. An implicit migration can be of two types, reactive or preemptive. A reactive policy is issued in response to scenarios such as high load on the server, which left on its own can cause the game to become unplayable. In contrast, a preemptive policy is used in an attempt to improve performance, e.g., by moving users in a region to a server closer in locality.

2 Related Work

To handle the large number of concurrently interacting entities in a virtual environment, it is common practice to use a static, region based partitioning scheme. The virtual world is thus divided into smaller, more manageable parts, where each region is hosted on a single server in the cluster. Some implementations allow several regions to be hosted on a server, such as Anarchy Online [10], while others are more conservative and allow only for one region per server, such as Second Life [12]. A widely accepted problem with the static partitioning scheme is that it does not take into account the dynamic nature of MMOGs. Even if the static partitioning is based on population density trends, and arranged to accommodate this, it is still susceptible to imbalances due to unforeseen events. Thus, a lot of research has been done on how to improve the flexibility of these partitioning schemes, and consequently, algorithms for efficiently distributing entities and regions. This research, however, does not address how the locality of the users, in relation to a server, will affect latency.

Turck et al [13] have investigated the effects of dividing a game world into dynamic micro-cells. A study with a similar background is performed by Duong et al [14]. Such micro-cells can be reassigned to servers in a cluster if the load on the server they are currently residing on becomes too large. Three different load-balancing algorithms were used, none of which factored in locality of users, and the number of micro-cells supported per server varied. The test was done on a centralized cluster. The conclusion was that a dynamic approach is preferable, because it will decrease the chance for bottlenecks and lower the overall latency.

Another approach to solving the problems with static partitioning is through maintaining consistency by limiting updates based on an area of interest. IBM has developed a middleware for distributed games called Matrix [15] using this approach. It is based on the observation that MMOGs are nearly decomposable systems, and as such, it is usually sufficient to update players with only those events that occur in their zone of visibility. Matrix thus provides pockets of locally-consistent state. Results show that Matrix outperforms static partitioning schemes when the workload exhibits unpredictable and dynamic skews. Matrix makes use of region based partitioning as an underlying foundation, but this is for the purpose of easily distributing the virtual world across multiple servers. Matrix is also intended for a centralized cluster of servers. Another middleware which implements this area of interest type partitioning is the Colyseus system [16], but this system is designed for first person shooter games and does not utilize the concept of regions.

In summary, the work on static and dynamic partitioning consider server load in a centralized cluster and not latency due to the geographical location of users. Most of the research tries to optimize the partitioning of the virtual environment into regions which can dynamically accommodate hot-spots. These regions can be user centric, in the area of interest approach, or area centric, in the micro-cell approach. The goal is nonetheless always to minimize the amount of events being distributed, be this through dynamically moving areas when a server becomes overloaded, or by limiting the scope of a user. Regardless, most of these partitioning schemes make the assumption that the system consists of a centralized cluster of servers, grid or similar. Thus, little or no efforts have been made to investigate the effects of a decentralized distributed system middleware, which would allow for regions of the game to be migrated based on the physical locality of the users in addition to their virtual locality. We have already seen that it is possible to determine the relative location of the users, based, for example, on an analysis of their RTT [9]. We also know that a common way of dividing virtual environments is by partitioning them into smaller cells or regions. There is also a number of algorithms for balancing the load. In this paper, we will now look at the system we intend to implement, which will permit for the accommodation of physical locality in addition to virtual locality. This migration of objects based on the locality of the majority of users would orthogonally further reduce the load and more importantly the latency. In the following section we will describe our proposed architecture for locating objects.

3 Name Service

In our distributed system, we wish to perform migration for the purpose of off-loading servers, and for minimizing the response time for reacting to events received from clients. When objects are distributed across multiple nodes in a system, access to objects which are not in local memory require special handling. Primarily, it is necessary to locate the node on which a particular object is currently residing. A name service provides an application with this type of functionality. Depending on the application being developed, there are different approaches to implement the name service. For our application, we have a set of six characteristics which impact the approach we decide to use:

1. The server must be able to handle thousands of concurrent users. For these users to have an optimal experience, the latency needs to be as low as possible, a requirement which is important for all interactive applications.

2. The users can be located anywhere in the physical world and virtual world.

3. Depending on the time of day, there is a high probability that a majority of the users will be represented by a specific time zone.

4. In order to handle the load generated by so many concurrent users, it is necessary to divide the world into regions which are spread out across a number of servers.

5. Clients and servers use the same libraries, that means that code is shared and that only data needs to be migrated. It also means that we can call any function of a remote object directly, without having to discover which function interface to use.

6. A large number of objects will be created, with greatly varying life spans. Consider the short life time of the bullets fired by a player's weapon, in contrast to the long life time of the player himself.

Znati and Molka [17] analyzed three approaches to implementing a name service; in form of centralized, distributed and hybrid versions. Prior to contacting the target object itself, the centralized version contacts a name server to obtain the objects location in the network. As such, the centralized naming scheme adds an extra level of indirection to the name resolution process. The distributed paradigm removes this level of indirection by placing the name of the object with the object itself. The hybrid approach is based on the design principle of keeping names together with the objects they are bound to on the local level, but resorts to multicasting when resolving names at a regional level. This study indicates that the choice of model for a name service will influence the performance of the service and the throughput of the network. The results showed that the centralized model could achieve acceptable performance only as long as the ratio of remote to local requests was kept reasonable. The performance of the hybrid model highly depended on the efficiency of the cache design. With all other network conditions set equal they found that, relative to the response times of the centralized simulation, the response time of the distributed simulation were smaller.

A fundamental problem with the centralized version is that all object resolution and registration is performed at a single point in the system. This means it easily can become a bottleneck in the system, particularly considering how high the object creation frequency can become in such systems. As such, it becomes apparent that a centralized version wont scale very well for systems which experience heavy traffic, which is the case for MMOGs. A centralized version also introduces a single point of failure in the case of a crash. It also raises an interesting question with regards to decentralized systems, such as our middleware, about where to place the name service relative to the servers in the system. The hybrid version of a name service solves a few of these issues, but introduces a few of its own. There is no longer a single point of failure, since the name service is distributed. Thus, only the objects managed by the crashing node will become unavailable. Though it still leaves the issue of partial failures to be handled. A possible weakness is in the way objects are located, the lookup method relies on multicasting, and there is a high probability the response time will be too high, particularly for a decentralized system. Since objects are bound locally, rapid object creation and destruction no longer creates the same problems as with the centralized version. Last there is the distributed approach, which raises an interesting issue; in that there is no clear way to show were an object is located without first contacting its name service and how that name service is located given only a high-level name. For our middleware, this is not an issue since we have a single point of initialization. A single node in the system initiates all communication and object creation, thus there is always a known path to an object. The distributed version also leaves the issue with partial failures unresolved, but apart from this it serves the purpose of our middleware well.

Given the characteristics described in the start of this section, and the choice of our name service model, we will now outline the architecture of our middleware, using terms from Mobile IP [18,19]. Mobile IP addresses the desire to have continuous network connectivity to the Internet irrespective of the physical location of a node. This coincides with our goals in that the objective is to make mobility transparent to the application. The analogy is suitable, because where Mobile IP is used to find a route to a mobile computer, moving from network to network, we need to find a route to a mobile object, moving from computer to computer. A prerequisite to accomplish this, is being able to uniquely name a computer or object in the context it is used. We find that the taxonomy used to describe these processes overlap. In the following discussion, we will primarily focus on the definitions of mobile node, home agent, foreign agent, care-of address and home address.

When a mobile node (object) has migrated to another node in the system, it registers its presence with the foreign agent (name service) at its new location. The foreign agent issues a message to the mobile node's home agent (name service), in the form of a care-of address (local object identifier), which the home agent can use to forward requests to the mobile node. Each node in the system has an active name service. This name service can take on the characteristics of a foreign agent and a home agent. There is, however, a logical difference, depending on whether the object is propagating to or from a node in the system. At this point, our implementation diverges from the approach of Mobile IP, where a mobile node has one home agent throughout its lifetime. As mentioned, a single server is used to initialize the system, objects composing the system are propagated to other servers based on necessity, and links to the propagated objects are maintained in the name service of the corresponding servers. If this were not the case it would be impossible to maintain links between objects, because we would have no way of locating them. This is a result of our distributed name service. For an object to function as a mobile node, it needs to be serializable. Serialization is the act of storing the state of an object with the intention of restoring it again at a later point in time. This functionality is commonly used to move objects between servers, and makes it possible to migrate objects. As such, these serializable objects can be added to migration groups. If and

HOSTNAME	PORT	LOCAL IDENTIFIER		
		OBJECT ID	TIMESTAMP	PSEUDORANDOM NUMBER

Figure 1: Format of the Home Address and Care-of Address

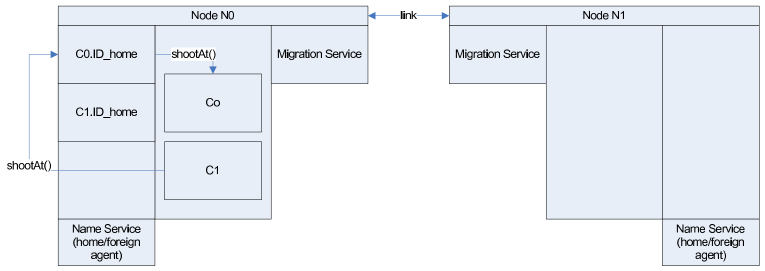


Figure 2: Overview of nodes before migration

when an object is added to a migration group depends on the currently activated migration policy. The migration policy can be preemptive as will be the case for a locality based load-balancing, or reactive if the load on the server becomes to great.

The care-of address and home address, which are used to identify objects in the system, have a format as seen in figure 1. The address uniquely identifies an object in the system throughout the lifetime of the application, since this type of application is designed to run indefinitely. We see that there are three main sections which compose the address. Each is required to identify an object because of our distributed name service. The hostname identifies the node where the object is located. The port provides an access point to the name service, which maintains local objects. The local identifier is specific to an object in the name service. The object-id is an index to more information about the object, such as the pseudo-random number and timestamp. The timestamp is required to identify the object temporally, but since this does not guarantee it to be unique over time, because of uncertainties related to computers and time keeping, we have a pseudo-random number in addition.

In order to solidify the architecture presented here, consider the lifetime of a fictive object in our distributed environment. In this scenario, we have two nodes in our system, the initializing node, N_0, and the secondary node, N_1. At startup, all collaborating services are initiated, such as the name service, migration service, host service and similar. After initialization, a player connects to the application, and an object C_0 representing his character is instantiated at N_0. Since C_0 is a serializable object, an identifier is generated called $C_0.\text{id}_{home}$ which is logically equivalent to this object's home address. The format is as detailed in figure 1. N_0 has in effect become the home agent of C_0. C_0 now exhibits all the properties of a mobile node: it is possible to migrate it, and it has a home agent. A short while after, a second player connects to the server, and the object C_1 is created. Figure 2 shows what the node would look like at this point in time. If these two players are fighting each other, and C_1 decides to shoot at C_0, we can see that the *shootAt()* function call goes via the name service, which determines that C_0 is a local object and directs the shot accordingly. As time goes by, more users connect to N_0, and eventually a migration policy is activated. Based on the requirements of the migration policy, C_0 is added to a migration group. Based on an analysis of available nodes in the system, a node is selected as the recipient of the objects. Depending on the the requirements of the migration policy, we might want a node as physically close as possible to the majority of users. Based on the requirements issued by the migration policy, N_1 is selected by the host service as the destination node. At this point, C_0 is migrated to N_1 (see figure 3). The name service at N_1, which is logically equivalent to a foreign agent, accepts the mobile node C_0 and generates an identifier, $C_0.\text{id}_{care-of}$, which is logically equivalent to its care-of address. This identifier is sent in return to the home agent and replaces the home address $C_0.\text{id}_{home}$. If the second character C_1 now fires a shot at C_0, the *shootAt()* function call will go via the name service, but

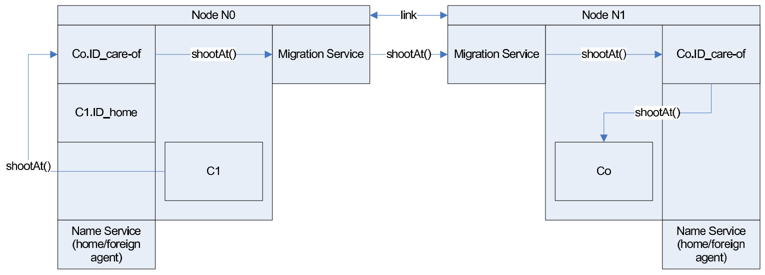


Figure 3: Overview of nodes after migration

unlike last time, C_0 is no longer a local object. Instead of accessing the object locally, the care-of address is used to issue a remote procedure call to the object at its new location.

Following this example a few things become apparent. Any serializable object is identified through the name service by querying on its identifier, i.e., an indirection which is necessary for accessing objects at remote nodes. It also becomes apparent that we are highly vulnerable to partial failures. This happens when a node in a distributed system becomes unavailable, effectively rendering the objects managed by it inaccessible.

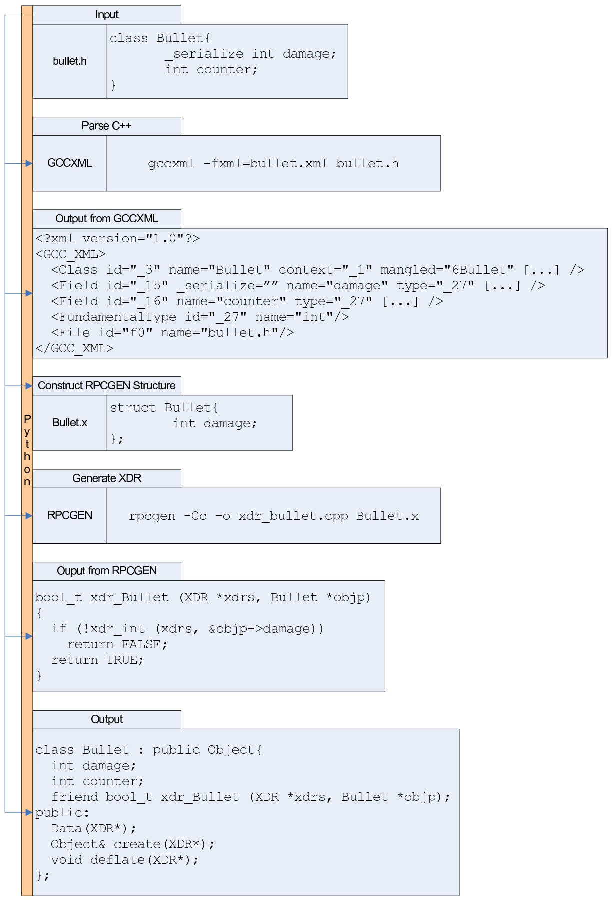
While the current architecture does not accommodate for partial failures, there are ways to minimize the repercussions of these incidents. One possibility is to utilize a central registry, where all object migration is recorded. In the case of a miss, when attempting to access a remote object, the central register can be queried instead. Once the node has recovered, normal object access resumes. A flaw with this approach, is that the central register becomes a single point of failure in the case of a crash. In addition, it must be capable of handling the traffic generated by all migrations, including any misses. This is a poor sign, since migration most likely is activated as a result of heavy load. A different approach has its roots in peer to peer based filesystems, where copies of an object will be distributed to several nodes in the system. PAST [20] and OceanStore [21] have, for example, implemented such systems with success. PAST copies objects to random nodes, in an attempt to distribute the objects evenly. OceanStore uses a more deterministic approach, and places the objects close to nodes which access them. Lookup of objects in the system can then be implemented in a fashion similar to that of Chord [22] or Tapestry [23]. These implementations are based on the principle of incrementally forwarding messages from point to point, until they reach their destination. Each node in the system keeps a small routing map, which is used to determine which nodes to forward the message to. A problem with this type of lookup is that the response time might be too high for interactive applications. Both the centralized and distributed fail safe techniques offer their own set of advantages and disadvantages. As such, we intend to investigate viable methods for recovering from partial failures in our middleware in future work. In the following section we will look at the process used for automating the creation of serializable objects.

4 Code Generation

Most computer games are developed using the object oriented programming language C++ [24]. Consequently, our middleware is also implemented using this language. Since C++ has no built-in mechanisms for the serialization of objects, and manually writing code for this is a tedious and error-prone process, we provide the application developer with a tool for automatically generating this functionality. The result of the generation process is a skeleton, which can easily be integrated with the middleware. We implement the serialization mechanisms using inheritance, polymorphism, run-time type identification (RTTI) and virtual functions.

In order for an object to be serializable, it must inherit the abstract class called *Object*. This class defines a set of pure virtual functions, which all derived classes need to provide an implementation for. These functions perform the serialization and deserialization of an object. When objects are passed to the migration service and migrated, they are up-cast to look like an instance of an *Object*. Since the serialization functions are implemented as virtual functions, the derived object's implementations of the functions are called. The type name of the object always precedes the object itself on the stream. When deserializing the object at the receiving end, the correct deserialization function is called for the object by looking up the type name of the class in a type register. The type register consists of mappings between the type name and an instance of the corresponding class for all serializable classes. The type name of a class is determined by a python script during generation and is a combination of the filename and class name, this is done because RTTI type names are not portable. Once the object is deserialized it can be downcast accordingly, using RTTI, to an instance of its type, so the functionality associated with such an object becomes available. The reason it needs to be downcast is because the deserialization process returns a reference to an instance of *Object*.

Serialization in our system is necessary so we can transmit objects across the network. As a side effect of this, we need an architecture and operating system independent encoding. The rpcgen [25] utility was developed for generating client and server stubs for remote procedure calls, we use rpcgen to automatically generate routines for serialization using C-like data structures. Furthermore, rpcgen uses the eXternal Data Representation (XDR) [26, 27] format for the serialization of parameters, which is a standard for the description and encoding of data. It is useful for transferring data across networks between different computer architectures. XDR is based upon implicit typing. The sender and receiver must agree on the order and type of all data. Moreover, XDR makes use of symmetric data conversion. Both the client and server convert from and to a standard representation. XDR routines are direction independent, the same routines are called to serialize and deserialize data. XDR supports all C data types, such as int, double, char, and arrays of these types. Filters are provided for the serializing and deserializing to and from their local representation. These basic filters can be combined to allow for more complex data types, such as structures, to be serialized.



```
class Bullet{
        _serialize int damage;
        int counter;
}
```

```
gccxml -fxml=bullet.xml bullet.h
```

```
<?xml version="1.0"?>
<GCC_XML>
  <Class id="_3" name="Bullet" context="_1" mangled="6Bullet" [...] />
  <Field id="_15" _serialize=”” name="damage" type="_27" [...] />
  <Field id="_16" name="counter" type="_27" [...] />
  <FundamentalType id="_27" name="int"/>
  <File id="f0" name="bullet.h"/>
</GCC_XML>
```

```
struct Bullet{
        int damage;
};
```

```
rpcgen -Cc -o xdr_bullet.cpp Bullet.x
```

```
bool_t xdr_Bullet (XDR *xdrs, Bullet *objp)
{
  if (!xdr_int (xdrs, &objp->damage))
    return FALSE;
  return TRUE;
}
```

```
class Bullet : public Object{
  int damage;
  int counter;
  friend bool_t xdr_Bullet (XDR *xdrs, Bullet *objp);
public:
  Data(XDR*);
  Object& create(XDR*);
  void deflate(XDR*);
};
```

Figure 4: Code generation process

Figure 4 depicts the process used to generate the skeleton. It is implemented as a python script. To parse the C++ header files, we make use of the GCC-XML [28] parser, which is a tool that extends the open source GCC compiler, using its internal representation to produce XML output. Based on information obtained by processing the XML-file, we derive rpcgen language structure definitions. These definitions are in turn processed by the rpcgen tool which generates functions for serialization. The output of the process is a skeleton we have created, which combines the components of the structure described earlier in this section. The skeleton generator expects a C++ class declaration as its input. In addition to normal C++ class syntax, GCC-XML allows for defining additional attributes. We make use of this ability to extend the C++ syntax with our own keywords.

When an object is migrated, one does not necessarily want all the data to be serialized. We therefore provide a special keyword (*_serialize*, see figure 4) to specify what data is to be serialized. Notice how the *counter* variable is not serialized, in contrast to the *damage* variable. Any number of these keywords can be added to aid in the parsing process. Other suitable keywords will be introduced to support remote method invocation, mark the classes that can be migrated and so forth.

5 Conclusion

MMOGs are distributed applications that have a range of unique characteristics, e.g., the stringent requirement to latency, which means that to cope with the server load the virtual environment needs to be logically divided across a number of servers. Another factor is the diversity of locations among the users connected to the application, and the fact that users located in the same virtual region, will be connected to the same physical server, and as such cannot necessarily be separated.

The assumption is, that based on an analysis of the RTTs for users in a given virtual locality on a server or by looking at IP addresses if available, we can identify if the majority of those users are located at the same geographical location in the world. Our assumption is based on the fact that people from certain time zones will be more active depending on the time of day. Given this, we believe that the overall latency of that area can be lowered by migrating the region to a location as close as possible to the majority of the users. It is our understanding that little or no work has been done on how latency can be improved based on this type of migration policy. It is not to say that this is the only type of migration policy that needs to be present on the server. Migration can also be triggered because the server is becoming overloaded.

In this paper, we have therefore presented a work in progress for a middleware which accommodates the physical locality of the users in addition to their virtual locality. This is accomplished through our object model, which takes advantage of polymorphism and virtual functions, to make C++ objects serializable, and thus transferable. In order to distribute and locate objects across multiple servers, in an efficient manner, we use a distributed name service. Additionally, our middleware provides the developer with a tool for automatically generating skeletons, in order to minimize the number of coding errors and ease the development of the application. We intend to implement and test the feasibility of this architecture for use with a real MMOG, and also to see if there are any real benefits of migrating based on locality. In order to accomplish these goals, the middleware will eventually need to be extended with functionality for handling concurrency control. Susceptibility for partial failures and the possibilities for recovering from such incidents also needs to be examined.

Finally, we must add functionality for selecting which objects to migrate, e.g., migrating all members maintained in a group for group communication. This is ongoing work [29], and the intention is to eventually combine these components, and analyze the results.

References

[1] The Entertainment Software Association. ESAs 2006 essential facts about the computer and video game industry. *http://www.theesa.com/*, jan 2007.

[2] B. S. Woodcock. An analysis of mmog subscription growth. *http://www.mmogchart.com*, jan 2007.

[3] M. Claypool. The effect of latency on user performance in Real-Time Strategy games. *Computer Networks*, 49(1):52–70, 2005.

[4] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The effects of loss and latency on user performance in unreal tournament 2003. *Proceedings NetGames' 04, Portland, Oregon, USA*, pages 144–151, aug 2004.

[5] M. Dick, O. Wellnitz, and L. Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. *Proceedings of NetGames' 05, Hawthorne, NY, USA*, pages 1–7, oct 2005.

[6] Electronic Arts. Ultima Online. *http://www.uo. com/, January*, 2007.

[7] CCP. EVE Online. *http://www.eve-online.com/, January*, 2007.

[8] Blizzard. World of Warcraft. *http://www.worldofwarcraft. com/, January*, 2007.

[9] Carsten Griwodz and Pål Halvorsen. The fun of using TCP for an MMORPG. *Proceedings of NOSS-DAV' 06, Newport, RI, USA*, pages 1–7, may 2006.

[10] Funcom. Anarchy Online. *http://www.anarchy-online.com/, January*, 2007.

[11] Abdennour El Rhalibi and Madjid Merabti. Agents-based modeling for a peer-to-peer mmog architecture. *Comput. Entertain.*, 3(2):3–3, 2005.

[12] Linden Lab. Second Life. *http://secondlife.com/, January*, 2007.

[13] 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. *Proceedings of NetGames' 05, Hawthorne, NY, USA*, pages 1–7, October 2005.

[14] T.N.B. Duong and S. Zhou. A dynamic load sharing algorithm for massively multiplayer online games. *ICON' 03, Sydney, Australia*, pages 131–136, October 2003.

[15] Rajesh Krishna Balan, Maria Ebling, Paul Castro, and Archan Misra. Matrix: Adaptive middleware for distributed multiplayer games. In Gustavo Alonso, editor, *Middleware*, volume 3790 of *Lecture Notes in Computer Science*, pages 390–400. Springer, 2005.

[16] A. Bharambe, J. Pang, and S. Seshan. Colyseus: A Distributed Architecture for Online Multiplayer Games. *Proceedings of NSDI' 06, San Jose, USA*, pages 155–168, May 2006.

[17] T. B. Znati and J. Molka. A simulation based analysis of naming schemes for distributed systems. In *Proceedings of the 25th Annual Simulation Symposium*, pages 42–53, Los Alamitos, CA, USA, April 1992.

[18] C. E. Perkins. Mobile IP. *Communications Magazine, IEEE*, 35(5):84–99, 1997.

[19] C. Perkins. RFC 2002: IP mobility support, October 1996.

[20] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *Proceedings of SOSP'01, Lake Louise, Alberta, Canada*, pages 188–201, oct 2001.

[21] Dennis Geels. Data Replication in OceanStore. Technical Report UCB//CSD-02-1217, Computer Science Division, U. C. Berkeley, nov 2002.

[22] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of SIGCOMM' 01, San Diego, CA, USA*, pages 149–160, aug 2001.

[23] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, apr 2001.

[24] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[25] Sun Microsystems. *rpcgen Programming Guide*. Sun Microsystems Inc., Mountain View, CA, 1987.

[26] Raj Srinivasan. XDR: External data representation standard. RFC 1832, August 1995.

[27] Sun Microsystems, Inc. XDR: External data representation standard. RFC 1014, June 1987.

[28] B. King. GCC-XML the xml output extension to gcc. *Undated. Online: http://www. gccxml. org/HTML/Index. html.*

[29] K. Vik, C. Griwodz, and P. Halvorsen. Applicability of group communication for increased scalability in mmogs. *Proceedings of NetGames'06, Singapore*, oct 2006.