Eindhoven University of Technology
Computer Science and Engineering

Architecture of Distributed Systems

2IMN10

Architecture Comparison of Massive Multiplayer Online Games

Jasper Selman	0741516	j.w.m.selman@student.tue.nl
Ramon de Vaan	0758873	r.d.vaan@student.tue.nl

October 22, 2015
Eindhoven

Abstract

The gaming industry is currently one of the most rapidly expanding industries in the world, in particular the multiplayer online variant. Due to the vast success of games like World of Warcraft, the amount of producers for massive multiplayer online games has increased significantly. However, due to the number of players involved in these massive multiplayer online games, the requirements and concerns for data transfer between components has changed, compared to games before. This has sparked the research and development of many new architectures and datastructures to address these.
Therefore, it is rather interesting to know how the architecture of all these games are designed, and the problems they address. In this paper we will take a deeper look at some of these architectures. We will elaborate on the architectural and interaction styles used in the following papers: *Practical Middleware for Massively Multiplayer Online Games* [1], *Requirements of Peer-to-Peer-based Massively Multiplayer Online Gaming* [3] and *Towards Service Oriented Architecture for MMOG* [5]. We shall also describe how these design choices affect the view of the game state and compare how well the architectures work with respect to performance aspects.

Practical Middleware for Massively Multiplayer Online Games

The first architecture and it's design choices we are going to review, is the architecture described in *Practical Middleware for Massively Multiplayer Online Games* [1]. In this paper the authors elaborated on the middleware platform. The purpose of such a platform is, how the authors stated it, "Middleware helps programmers manage the complexity and heterogeneity of distributed computing environments". So for short it is a platform which makes it easier to develop and maintain distributed games. The authors have also developed their own platform and have made several architectural design decisions for that. The platform is called Distributed organized Information Terra (DoIT) middleware platform.

A lot of research has been done by researchers to define a framework for middleware platforms. The framework is a 4-tier architecture consisting of a client, a proxy/-gateway, a cell server and a database tier. The gamers have control over the client applications. The proxy is in charge of distribution of messages and security aspects. The cell servers maintain the virtual world, it also makes sure that no collisions appear between multiple game clients. The database stores the players states, so that all the changes people made are not wiped clean every time they log off. A graphical view of this architecture is shown in Figure [2].

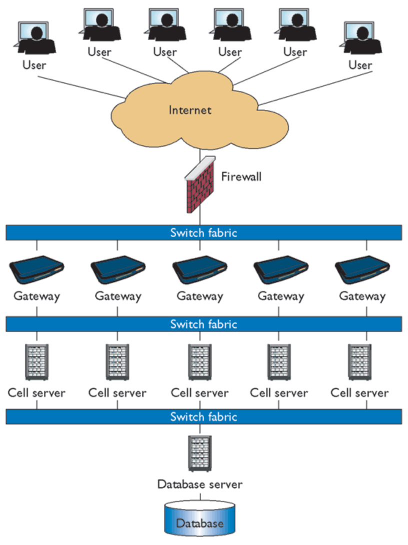

Figure 1: General framework for middleware platforms for MMOG

Next to this general framework for the architecture there is also a list of properties which a a middleware platform should adhere to. These properties are *Ease of Development*, *Ease of Deployment*, *Ease of Maintenance*, *Ease of Change* and *Performance and Load Balancing*. For an elaborated description of these properties, we refer to section "Practical Next-Generation MMOG Middleware" in [1].

Now we know the basics of a middleware platform, we shall discuss the architecture of the DoIT platform. This platform uses a message-oriented middleware instead of a RPC-based technology. The authors claim that this of great benefit due to the event-driven behavior of MMOG's. This seems logical. A lot of the game play of MMOG is indeed based on events made by other players. RCP does not really benefit from this, it would be better suited for single player games, because in that case all the procedure calls are done by the one client.

Another great advantage of MOM over RCP is that MOM allows decoupling while RCP does not. This is also mentioned by the authors. Next to this difference the DoIT platform differs on two points from the framework mentioned before. The first is that DoIT is customizable and reduces the complexity for MMOG application programmers. The only proof for this, given by the authors, is that a code-generator model is introduced. This model generates message-factory and handler classes for a protocol. The developer still has to define the real detailed content for these handlers. We think that this seems not really enough evidence that this platform really reduces complexity for programmers. It only generates some code automatically (the handlers). The protocol still has to be defined by the programmer and the actual code in the handlers still has to be produced by the programmers. So in practical, the authors only stated that they made a module which generates the handlers for a protocol, but they still have to filled. The only real benefit from this is that you do not forget to create such a handler. So we think this claim is really questionable.

The authors also made a small notion of security of their platform. The protocols are modeled as a set of message fields and the generator engine randomly shuffles these fields. This makes hacking specific messages more complicated. The authors also claim with just that this is not really safe yet and that messaging protocols and encryption algorithms are needed to prevent attacks.

The last feature DoIT includes is a real-time virtual-world logic, called VWLogic. This logic has as a target to decrease the complexity for the programmers. The idea behind it is really nice. There is a VWLogic adapter which demultiplexes for each VWLogic component, as stated by the authors. When a message comes in, the VWLogic adapter finds the corresponding handler. After that handler is finished the VWLogic adapter sends an update request. Unfortunately the authors do not mention what architecture is used to accomplish this or give any proof of how this works. So the idea is really nice and the programmers certainly benefit from this, if it works.

Another question that comes to mind for these kind of platforms is, how does this architecture affects the game state of a client? This can all be derived from the general framework for middleware which DoIT follows. The MMOG has a central (or multiple, you can play with that a bit) database servers. In these servers the game state of a client is saved. This has as a great advantage that a client user does not depend on the availability of other users (like in a P2P network). The downside of this is that a lot of traffic has to go from/to that database server.

The framework tries to minimize this by letting several cell servers interact with this database server. Fortunately this database server does not have to be interacted very often, I assume once to load the clients game state when he logs on and once to store it when he logs of. The massive interaction happens between clients and cell servers. Now I assume that all one cell server handles the game server in which clients are playing or that when several cell servers handle such a game server that they communicate with each other. Otherwise it would not be possible to resolve conflicts between clients, like for instance user A picks up an item and user B picks up the same item at about the same time. This conflict has to be resolved by the client servers. Luckily this can be easily done by doing a request to the server whether this item can be picked up or not. The cell server(s) only allow one of those request. So this client server architecture in the 4-tiered framework ensures that client games are in a steady state.

There is now one problem that have to be addressed, that is what happens when one of the cell servers or database servers goes down? When one of the cell servers goes down, than it is possible to distribute the clients over other cell servers. This might affect the performance a bit, due to bigger loads on the other cell servers but it is still better than just dump the connections with the clients that were connected to the failed cell server. A bigger issue is what happens when the database server goes down? Do they have backup servers for that?

As shown in the architecture in [2] there is only one database server. So when this server is down, the game cannot be played. However we assume that there are (several) backup servers for this database server. In that way the game can still be played.

The last part left to discuss is performance of the architecture. We shall discuss some of the above mentioned properties, like maintenance and load distribution, but also properties like security, availability, usability and scalability. As mentioned before the general architecture of this particular system follows the architecture described in Figure , so we will refer to this figure regarding the properties. The first property we are going to check is maintenance. The authors themselves state "easy maintenance and monitoring are subsequent requirements" in section "Ease of Maintenance" in [1]. When we look at the architecture we see that the decoupling of the MOM system makes the maintenance easier for the developers, they do not have to recompile the whole program every time. Also the client server architecture that was provided in Figure shows us that it is possible to do maintenance on some of the client servers. They can do maintenance on some of the servers, while the game still runs on the other servers and when the maintenance is finished they can redistributed the games and maintain the other servers. This can also be done for the database server if they have backup servers. The last part which helps is the VWLogic. When some of the handlers are changed, the VWLogic itself makes sure that the right message is handled by the right handlers.

Now let's take a look at the load distribution. This is done quite well via the general middleware architecture. The client runs the graphics, the network communication and the interface, while the servers host the virtual world and handle collisions and the database handles the user state. This can be balanced by spreading the virtual world over multiple servers, or server farms to speed this up. This is also of direct influence on the scalability. If this is done correctly then it is easy to add multiple servers to a cluster or add multiple clusters, or multiple database servers to the entire network. This makes sure that more players can run the game. Note that the definition we use for scalability is that it has to be easy to scale the number of players who play the game, but also to scale the virtual world in which the game is played.

The security however was not satisfactory as mentioned earlier. There are very few security protocols integrated in the middleware and the authors stated that other security tools/protocols need to be implemented by the vendors themselves. So there is definitely room for improvement in this tool for this, but this can be hard since the correct protocol can depend on the game.

The next property is the aspect of availability. This was also briefly discusses before. As long as there are enough (clusters of) servers available for the cell servers, then people can play the game. When one of the servers goes down the load of that server can be redistributed over the other servers. The problem might be in the database server. When that serer is down, the availability of the architecture depends on whether or not (and how many) backup servers are available.

The last property we take a look at is the usability. We think that due to the customization of DoIT, vendors can establish the platform in such a way that they think is best for their game. It is however difficult to say something about the usability of the system. The DoIT system provides a framework, but not how people have to work within this framework. All we know is that protocols have to be defined within XML schemes from which handlers can be derived automatically. So it is difficult to say anything about the usability, except that it has to be useful on several platforms due to the fact that it is a framework. A lot of vendors should be able to use this framework to make their own games.

Requirements of Peer-to-Peer based Massively Multiplayer Online Gaming

In the second paper the authors [3] provide a whole different kind of view on an architecture for MMOG. The peer-to-peer view. Such kind of architecture in its ideal form, reduces the costs made by the game vendor significantly. For example think of the amount of servers which become redundant. The problem with this kind of architecture however is to ensure the consistency and availability of the game. In the next part we will see how well the authors handled these problems. A, not unimportant, note is that the architecture proposed is not yet finished. The authors have ideas how to implement their architecture but this did not happen yet. Therefore the architecture is not complete yet, but we try to reason about all the parts of the architecture for which the ideas are already there.

As mentioned before the traditional server-client architecture is very costly, that is why the authors chose for a peer-to-peer architecture. The underlying structure for this peer-to-peer network is going to be a *k-connected network graph.* Such a graph seems very suitable for a peer-to-peer network. When you establish a such a k-connected graph, the properties of it ensures that you are able to remove fewer than *k* vertices (players) from this graph and the graph is still fully connected. This full connectivity is a must have, because if the graph is not fully connected, people in one part of the graph are not able to receive events which are caused by other players in the other part of the graph. This has the result that these players seem to "freeze" for players in the other part of the network. A difficulty which is not solved by the properties of the k-connected graph is how to add new players to a graph or create a new graph. The authors themselves acknowledge that it is a problem when it is not guaranteed that players are added to an existing graph. Right now the authors are investigating how they can solve this problem. An already proposed solution is the IRC protocol, but we do not really see how this protocol should solve this problem.

In contrast to the number of architectural styles there are several interaction styles. The communication middleware used provides two kind of API's. The first supports a reactive world state propagation and the other a proactive world state propagation. The first API is used for activities like virtual crime investigation. The interaction style behind this API is a Request - Reply, or Remote Invocation, style. This style is well suited for this type of activities. The authors themselves give a good example for this: "*if a player is suspected of having acted inappropriately, a game master may actively query the network for all state information about the player's actions in a given time period.*". This also shows us why this interaction style is a terrible choice for the real game play. If the second, proactive, API was also based on this style then all the peers would have to make a request for every action that occurs, this would cause huge delays for the players and as we all know, in MMOG delays are one of the most irritating things that can happen.

Luckily the authors had realized that themselves and use a different style for the proactive API. The style behind this API fits best to the event based interaction style, or Indirect Communication style. This seems to be a good choice because the greater part of the communication between peers is based on events in the game caused by other players. A clear problem now was how to distribute all the changes over all the peers. Simply notifying all the peers about every event that happened, even when the event happened miles away, is not a suitable option. So the authors came up with the following solution. The game is partitioned in to a number of sub-areas, called *zones*. In every zone there is one coordinator, which is just a normal peer. Now there are two scenario's. In the first scenario a zone population is sparse. In that case the coordinator observes the peers and determines which peers should exchange events. In the second scenario the zone is densely populated. In that case the coordinator instructs all the peers to stop sending direct messages but instead send everything to the coordinator. The coordinator on its turn sends a message to every other peer in the zone, consisting of a summary of these messages. The authors admit that this is not an ideal solution due to the delay for the update messages but it is an efficient one.

This style pops some question marks. For example what happens to the game state of a user who is elected as coordinator. The coordinator has substantially more work to do than other peers, does this mean the coordinator has a bigger delay? Also what happens when the coordinator leaves the zone, but it was densely populated and it has not yet send an update message? What happens when a peer is on the edge of a zone and can already see the other zone, does he still only get updates from the zone he is in or also already from the other zone? What happens when a lot of elections of coordinators have to take place due to the unfortunate fact that the newly elected coordinator leaves the game at that time? Does this cause a lot of delay? These are all questions that are very important and unfortunately these cannot be answered yet since the architecture is not yet finished.

A whole different problem in this peer-to-peer style, which did not occur in a simple client-server architecture, is the consistency of a game. This is one of the hardest issues in a peer-to-peer network. How do you ensure that the action you just performed is consistent with all the other events in the game? Also another problem is how (and where) do you store all the data when a player logs off? In a client - server you can just store it at a database server, but how do you do this now? The authors acknowledge this problem and think that they can overcome this by interpreting the changes in a game as a series of events. The problem is however how to determine the ordering of these events? A definition of a consistency model has been made, linearizability [4]. This should help the authors to achieve this ordering. Currently they are investigating how to use this in algorithms to achieve the game consistency.

The last part we were going to discuss about the architecture presented in this paper is the performance of it. Think of properties like , maintenance, load distribution, security, availability, usability and scalability.

First we take a look at the API's in the communication engine and to which properties these contribute. Likely is that they increase the usability of the engine, since multiple types of games fit in this middleware. The distribution framework gives support for distributed game management and computation, the usability also benefits from this.

Next we take a look at the k-connected network. If the authors succeed in implementing this, the availability, reliability and scalability would benefit greatly from it. With such a network the system is able to add people to a network or remove people from it. This represents people logging in and off. If this is possible the system is also able to ensure that the game state of players does not depend on how many other players are online and who goes offline while you are playing. For adding a person to a graph, the authors are currently busy how to add new people to a graph instead of creating a new graph. If and when this works the system is also very well scalable. People could just log in and they are connected to an existing graph.

At the moment the authors are also busy investigating how the game world changes have to be distributed over all the peers. If they do this well enough than the load distribution of the game is quite well, but they have to be careful. If the load per peer gets to high, the performance for that single peer might go down a lot. They have also a reactive world state propagation, this propagation is used to do research in virtual crime investigation in the game (which shows us they are thinking about the security of their architecture). Unfortunately we cannot say more about this part since there are no real decisions made yet how to handle this game state changes.

The last part the authors are currently busy with is the consistency of the game. For the game view of the user it is important the game is consistent in such a way that is player A picks up an item, player B cannot pick up this item any more. At the moment the authors developed a model for ordering of events in a system to be correct, but nothing of this is yet implemented.

Overall the system has great potential regarding these performance properties, but it was still in progress so we had to wait for the result how well it is implemented in practice. Since the paper was quite old we tried to find how this peer@play project ended and if it was a success or not, but unfortunately we were not able to find any information on this. So this might insinuate that the architecture in practice was not so good.

Towards Service Oriented Architecture for MMOG

The final architecture we will discuss is the one described in the paper *Towards Service Oriented Architecture for MMOG* [5]. The main focus of the paper is to propose a service oriented architecture for massive multiplayer online games, that ensure a certain quality of service and provide scalability. Quality of service seems mostly seems to be expressed in terms of real-time interaction. The authors have different measures for real-time interaction, ranging from 'non real-time' to 'extreme real-time'. In user communication, the writes attempt to achieve 'hard real-time' interaction. However, some of the other requirements of the system seem to collide with the quality of service requirements.

As is apparent from the title, the architecture proposed in the article is aimed at games. In this case, apart from only focussing on game state, the authors have considered other services in a game as well, for example, the account login service, or the account billing service. These services are modeled by the service-oriented architecture (SOA) concept. The SOA ensures functionality like scalability, re-usability, monitoring and tracking of information, and others. Based on this concept, the architecture provides processes for user authentication, user checks, and credit card checks. These processes are implemented using Business Process Execution Language (BPEL). BPEL is very well suited to the processes above, while also having a very low deployment cost. It is even capable of real time services. However, this is where it clashes with the quality of service requirements.
The language can only provide 'soft real-time' interaction, one step below the hard real-time interaction we need. Hence, it does not have the capabilities for peer to peer interaction as far as games go.

To solve this, the proposed architecture employs a form of middleware to facilitate peer to peer interactions, separate from the BPEL implementation. The article proposes the use of an open source 'Data Distribution Service (DDS) Standard'. This standard provides an information hub where users can dynamically connect in order to publish and subscribe to each other's data as needed, in an efficient, high-performance manner, with minimal overhead. The flexibility of this communication exchange ensures the quality of service requirement. Additionally, having users publish data to subscribed users themselves saves a lot in terms of needed communication hardware. As a result, the proposed system scores very well in the scalability field, since the producer only has to provide the middleware where users can connect to each other.

The architecture seems very well thought out in terms of scalability and quality of service. Splitting the architecture in two parts with slightly different requirements ensures that the system as a whole is more efficient. The split makes sense, as well. The SOA concept ensures that the authentication, user check, and credit card check processes remain scalable, with the added benefit that BPEL has a very low deployment cost. DDS also seems to be very suitable for having users send data to each other.

Using DDS says something about how the architecture handles the game state. Users subscribe to other users for their data, and users publish their data to every user subscribed to them. However, the article does not state how users are to determine to which users they should subscribe, or what/when they should publish data. Supposedly, this should be handled by the game engine. In that manner, each user has it's own portion of the game state, only being subscribed to the users that have data important to that particular user. This does give rise to some questions regarding security. Users may be able to influence the data they send, or just not send data at all. All of these might result in inconsistencies in the intersection of game states between users, which might put some users at a disadvantage.
However, inconsistencies in the game states do not even have to be deliberate. Users with a bad internet connection may also affect the game state of other users. As per the example given in the article, in a race game, multiple users may think they're at the number one position, due to delayed messages from other players regarding their position on the track. This may be addressed by game developers, but it does seem like placing a burden upon them, which should actually have been solved in the architecture itself.

Now we are going to analyze the architecture proposed in [5] with regarding to performance properties like maintenance, load distribution, security, availability, usability and scalability. First we would like to discuss the load distribution. The architecture proposed in the paper showed us a that the overall system consists of many different decoupled modules. There are servers for accounts to login, to handle the billing process (which is of course to great importance for the vendor), auxiliary servers for services like chatting and of course the game servers which run the real game. So you might argue that the load is well distributed, but the real work is done on the game server. In the previously discussed paper the game was split into client services like the graphics and server services like the game state. This is now done all on this single server. So this load is not well distributed. It is also questionable how well this works when there are multiple game servers since the authors state in the section *simulation of SOA Concept for Servers* that all servers are implemented as standalone servers. This implies there is on real communication between servers.

Now lets take a closer look at the usability. The decoupling discussed above together with the BPEL language proposed in section *Simulation flow in BPEL* of [5] seem to give a great boost for usability. The BPEL has great functionality when the processes involved do not have a hard real time constraint. It is also stated that this language is a standard for defining processes. So this insinuates that there the tool is very usable (usable in the fact that it is easy to use and can model a lot of different processes). When there are however a lot of processes involved which do require real time constraints, this solution does not work. In that case the authors state that they have to use the DDS standard. This standard is open source and dynamic. This implies that the system is usable as a middleware system.

When we take a quick look at the availability we see that it is easy to keep the system running when parts of the game go down (like the chat), because of the different types of servers, since they are all separated. But it is difficult keep the game running when the game server goes down. Since there is no communication between other game servers it is hard to redistribute the load of a game server when one goes down. On top of that nothing is stated about backups or something, so if a server does go down we think the users lose everything (since their last login).

However since all the services are placed on separate severs it might be easier to do maintenance on these parts. Also since the BPEL language is widely supported it might be easy to do maintenance on parts which are created with this language. The real question in this part his how easy is it to do maintenance systems following the DDS standard, however nothing is really stated about this standard we do not really know how well this adheres to the maintenance property.

The security provided by this architecture is, that important information, like account information and payment (credit card) information is stored on separate servers. The DDS server also does not use the TCP protocol, which is well known and quite safe, but use their own protocol for information exchange. For this protocol we know nothing about how we have to authenticate ourselves or what we have to do to get data from this protocol. In the rest of the paper nothing can be found on what security measures are taken to prevent malicious users on running their own code in the server or how the servers which store the information are protected. So overall it is hard to say how secure this architecture is, because this part is not really handled in the paper.

The last property we are going to discuss is the scalability. The architecture described with the standalone servers does not really seem scalable. When the number of users grows, does the number of servers grow or does the server get bigger? It seems unlikely that still one game server can be used to maintain all the information. The BPEL language does scale well with protocols, it is widely used and as long as the protocol does not have a real time constraint it can be modeled within this language. The DDS standard needs the proposed architecture to be scalable and the authors promise that their architecture is scalable to multiple users, but nowhere any evidence can be found which supports this claim. So it is really questionable how well this architecture scales, because it seems to not scale very well.

Conclusion

gerghr

References

[1] Tsun-Yu Hsiao& Shyan-Ming Yuan. 2005. Practical Middleware for Massively Multiplayer Online Games. *IEEE Internet Computing*, September - October 2005, 47-54.

[2] Tsun-Yu Hsiao& Shyan-Ming Yuan. 2005. Practical Middleware for Massively Multiplayer Online Games. *IEEE Internet Computing*, September - October 2005, 49, figure 1.

[3] Gregor Schiele, Richard Süselbeck, Arno Wacker, Jörg Hähner, Christian Becker & Torben Weis. 2007. Requiremetns of Peer-to-Peer-based Massively Multiplayer Online Gaming *Seventh IEEE International Symposium on Cluster Computing and the Grid.*

[4] J. Hähner, K. Rothermel & C. Becker. 2004. Update-Linearizability: A consistency concept for chronological ordering of events in MANETs. *Proceedings of the 1st IEEE conference on Mobile Ad Hoc and Sensor Systems.*

[5] Farrukh Arslan. 2012. Towards Serice Oriented Architecture (SOA) for Massive Multiplayer Online Games (MMOG). *UKSim*, 82, 538-543.