# A Communication Architecture for Massive Multiplayer Games

[Position Statement]

Stefan Fiedler, Michael Wallner, Michael Weber
Dept. of Multimedia Computing
University of Ulm
89069 Ulm, Germany
{stefan.fiedler,michael.wallner,weber}@informatik.uni-ulm.de

## ABSTRACT

In this paper we present an approach for a communication architecture based on the publisher/subscriber model. The key issues considered here are scalability, ease of programming and dynamic system evolution. The division of the communication load into distinctive channels allows decoupling the domains of the game enabling the overall system to scale flexibly with the underlying network infrastructure, map size or number of players simultaneously online. Thus our approach is suitable to realize especially huge game-worlds as used in massive multiplayer online games. The underlying publisher/subscriber communication layer provides a high level of abstraction with a lean API therefore dramatically simplifying network programming in the game engine. Due to the fact that the communication is content-based, there is no fixed correlation between the game players and the game backend servers. This permits the dynamic evolution of the system by removing or adding not only players but also game servers and game content on the fly.

## Keywords

publisher/subscriber model, scalability, massive multiplayer games

## 1. INTRODUCTION

Over the last years multiplayer support in games of almost all genres has developed from an option to a feature essential for the success and the longevity of a product.

The first games providing multiplayer support where using one computer or a console with a split screen (e.g. Pitstop II), or two computers connected by a modem or a null-modem cable (e.g. Populous). With the introduction of local network support in games (e.g. DOOM) multiplayer support became increasingly popular. A low number of players and good connection quality within a LAN allowed a straight forward approach by maintaining good response time for user interaction.

With the success of multiplayer games and the growth of the internet the game industry realized the opportunity to combine them. But with multiplayer games using internet connections new problems arose. While LANs provide features like broadcast, low latency, high bandwidth and nearly no packet loss, quality of network connections in the internet are hardly predictable since it is not feasible to create point-to-point connections for many peers. As the internet comprises a wide variety of network configurations and connections usually involve hops of the magnitude of 10, negative side effects are introduced.

Of these, latency, jitter and low bandwidth feature prominently. Lack of a multicast mechanism leads to a splitting of bandwidth as each messages has to be sent to every players respectively. Although broadband internet connections become increasingly available, the majority of internet users is still using a modem line, forcing multiplayer games to cope with bandwidth limitations of at most 56k.

Interaction with other players is crucial, so low latency is essential so that players don't get alienated by the game's behaviour, the susceptibility in this regard differing by genre. While it's not too difficult to cope with latency, the impact of high jitter renders a game unplayable. To a certain degree this can be compensated by introducing a delay, the amount of which again is dependent on the genre of the game.

## 2. GAME COMMUNICATION ARCHITECTURES

Most online games can be separated into two categories: First-person and top-down view. Whereas first-person view games (e.g. first-person shooters, car racing games, etc.) are typically characterized by a low number of elements that need to be synchronized as well as the need of low latency, top-down view games (e.g. real-time strategy games) can often tolerate a moderate amount of delay if its variation remains small but employ a high number of units to be synchronized.

The two main architectures in use today are peer-to-peer and client/server. Peer-to-peer networks are predominantly used in top-down view games[3]. Ordinarily every participant runs a simulation on her own machine. To lower bandwidth consumption, the grouping of units is utilized. Further bandwidth is saved by transmitting input events instead of unit stats.

In most client/server architectures clients collect input events and render the scene output provided by the server[2]. The load remains with the server, the clients can employ techniques like dead reckoning to provide improved response time, by only updating the client if its reckoning deviates significantly from the server's simulation, the synchronization effort is minimized[1].

With a limited number of players all these problems are handled quite well in modern online games. New challenges arise when planning massive multiplayer online games where several thousand players that interact simultaneously in the same virtual world.

First massive multiplayer approaches with several thousand players like Ultima Online or EverQuest have already been working for several years and are typically based upon server clusters. Players connect to the cluster using their computers as "dumb" terminals. Employing peer-to-peer synchronization mechanisms prohibits itself due to the huge bandwidth requirements. Pure server solutions are possible, though costly to maintain, but considering that the game world's state has to be kept persistent as well as in a consistent state, the deployment of some kind of client independent server is inevitable.

In the next section we introduce the publisher/subscriber model, on which our approach is based.

## 3. THE PUBLISHER/SUBSCRIBER MODEL

The publisher/subscriber model of communication is based on the production and consumption of content. This is different from the commonly used client/server based communication paradigm, where communication typically takes place between exactly two nodes, the client and the server. Therefore, the client/server paradigm is perfectly suited for one-to-one communication relationships. Often however, communication is more complex, resulting in one-to-many or even many-to-many relationships. Modelling these in a pure client/server based approach results in difficult network code caused by the mapping of the m-to-n relation to many one-to-one relations. Taking a different view, consumer/producer based communication sets its focus not on the communication participants (clients and servers), but on the content that is to be exchanged. This is usually accompanied by the anonymity of communication, meaning that the content producer does not know (and not care) who receives his content. For example, the mouse cursor in a graphical window system produces mouse move events, and it knows nothing (and it doesn't care) about the windows that receive its events. In contrast, a client/server paradigm based approach would model a mouse movement server and the clients would have to poll the server whenever they were interested in mouse movement. Evidently, client/server based communication is very suitable for one-by-one communication where permanent services are offered to clients whereas consumer/producer based communication has its strengths in the producer based distribution of content-related information (e.g. events) to a larger number of consumers.

The publisher/subscriber model, which is consumer/producer oriented, provides in its most basic form a single medium for information exchange to which everybody can publish information. This information is being received by the other participants, who filter the information by analysing the content of the published message. While this form of pure content based addressing provides an immense amount of flexibility, the efficiency is very low. Every single publication has to be transported to every single participant, and this will often result in communication bandwidth and computational power requirements that are not available. Therefore, an optimization step is done in so far as the consumers only want to receive the content they are interested in. This is achieved by separating the single communication space into different areas where only a special, selected part of information is transported in. These areas will be called "channels", and the producers publish information only to the channels it belongs to. On the consumer side, information is only received from the channels the consumer is interested in. This interest manifests itself in the selection of certain channels, which is realized in the publisher/subscriber model by subscribing to them. After the consumer subscribed to a specific channel, he receives all content published to that channel. This evolution of the first publisher/subscriber model immediately reduces the load that is generated by distributing and filtering all content to what is needed to distribute the relevant content, where "relevant" means "relevant to the subscriber of a channel". This results in the possibility to dramatically increase the number of participants and the amount of information that is exchanged by the overall communication system. Also, the simplicity of this idea almost eliminates network programming for the participants, because there are only a few basic primitives for communication:

```
subscribe(channel_id)
unsubscribe(channel_id)
publish(channel_id, message)
receive(channel_id, &message)
```

The event channels themselves can have properties like message rates, QoS guarantees, priorized message transport or message deadlines[5][6]. Much work, of course, has to be done by the underlying communication system which has to provide this high-level abstraction.

## 4. COMMUNICATION ARCHITECTURE FOR MASSIVE MULTIPLAYER GAMES

### 4.1 Problems with scaling

As already stated above, most games use either a peer-to-peer or a pure server based approach.

Pure peer-to-peer approaches are limited to relatively small map sizes and limited player numbers due to the bandwidth requirements they pose. Pure server based solutions require powerful servers or server clusters and these server clusters require a high bandwidth connection to the players because they serve the aggregated single-client communication and computational load.

If one wants to implement a massive multiplayer game, a huge map with many players has to be implemented. To handle this, our idea was to split the overall communication bandwidth into smaller pieces and to find a way to distribute the computational load over many machines, not only a single server. These smaller pieces, however they are designed, should be handled in a more feasible way. In the following we work out their desing in a greater level of detail and how we can use the publisher/subscriber model.

Note that this paper does not deal with the network issues experienced in common multiplayer games as synchronization, latency, cheating, and so on. These issues are dealt with otherwise, while our focus is how to remove the problems experiencing when scaling up the game.

## 4.2 Divide and Conquer

Our first idea is, a single player needs not to know everything on the map as long as it does not affect him. Outside the scope of his visibility, or the visibility of one of his active units, the player needs not to have any notion of the map and other players. A possible supervising instance, e.g. a game server managing the map, instead needs to know everything on the map, but not everything that is happening on the map. For this supervising instance, only the permanent changes are relevant, not every chat message sent between two players or every bullet flying over the map. Therefore we propose a topological division of the map into smaller pieces and a semantical division of the content that is transmitted among the instances of the game.

## 4.3 Splitting the map

The first step was splitting the map into smaller pieces. Only the piece(s) inside the player's scope is (are) interesting. However, when approaching the border of his piece, a player also needs to know what happens in the adjacent pieces. Regarding the shape of this pieces, many options are possible. We evaluated rectangles and hexagons. Using rectangles, when approaching a corner, three rectangles are adjacent. Using hexagons, only two other hexagons are adjacent when approaching a border. However, hexagons have more corners than rectangles, which increases the probability of approaching a corner. As our evaluation showed, however, the overall number of event channels that must be subscribed, is smaller using hexagons. So from now on, hexagons will be used in the examples.

After a shape for the map pieces has been found, the size is still a question. To reduce the overall bandwidth needed for a client, the pieces should probably not be much larger than the client's scope of visibility. This would reduce the network traffic to a minimum, but it must be taken into account that, when moving, many subscribe und unsubscribe operations are required, generating unwanted overhead. Too large pieces, however, require more communication bandwidth, because the larger the pieces, the more players are in the same piece.

Indeed, no definite answer about the correct map piece size can be given at this point, without knowledge regarding the actual game. Different game scenarios will result in a different level of interaction; first person shooters use relatively small maps with many players where adventures have huge maps with sparse players. Our idea is, the map piece should be that large that it takes approximately one third of the communication load a specific game engine generates in "normal" multiplayer mode (not in "massive" multiplayer mode). As a single player is interested in up to three hexagons, the same level is reached as for a "normal" multiplayer game.

Now we search a communication model that is able to associate every map tile with a virtual "network": In a normal multiplayer game you deal with one "network", and as you are normally playing on one map tile, this "network" should be associated with this map tile. Moving from one map tile to a different one, is changing the network. We decided to take a publisher/subscriber model, because it behaves relatively similar to standard network infrastructures, e. g. ethernet: typically, games are sending IPX or UDP datagrams over an ethernet network, and as they do not want to send packets to every single recipient, multicast or, more generally, broadcast is used. This results in the packet sender's network layer not knowing who receives his packets, the transmission (from the network layer's point of view) is anonymous. Looking at the publisher/subscriber model, the behaviour is similar: publishing a message datagram corresponds to sending an anonymous network packet.

The different virtual "networks" can now be modeled using the channel concept of our exdended publisher/subscriber model described above. Whenever the player (respectively the game engine controlling the player) is interested in what is happening in a specific "network" it subscribes to the corresponding channel and will therefore receive the datagrams sent over this "network" (i.e. the messages published into this channel).

With this approach the network traffic (and thereby the communication bandwidth) is reduced to the parts relevant for the player, and a behaviour very similar to ordinary ethernet network is maintained. Hence, the known techniques used to cope with network issued can be applied. However, this approach will not work out if the implementation of the abstract publisher/subscriber channel concept behaves too different from a raw physical network. Therefore, implementation issues concerning these channels will be discussed later.

## 4.4 Separating environment and interaction

In the next step, we investigate how the load on possible game servers managing the permanent parts of the game can be reduced. Therefore, for this consideration the aspect of network communication is reduced to the content transmitted.

Taking a closer look at this content, we were able to figure out at least two major categories:

1. The game map and items that are on fixed positions on the map. Usually these are game elements like resources, weapons, potions or other pick-ups.
2. The players that are moving around and interacting with each other. This includes chatting, killing, trading among them. However, these interactions do not change the environment.

Separating these two groups of information is the next step we propose. As illustrated above, with the map segmented into smaller pieces, this results in each piece having now two corresponding channels: one to subscribe for the environmental game data and one to subscribe for the intercation. While the game data channel contains relatively slow changing environmental information like map data, probably including player positions, the other channel transfers the events that are generated by the interaction produced during the active game play that takes place in that specific spatial partition of the map. This separation of environmental data and interaction results in the game data channel (the "environment channel") consuming only little bandwidth compared to the game play channel (the "interaction

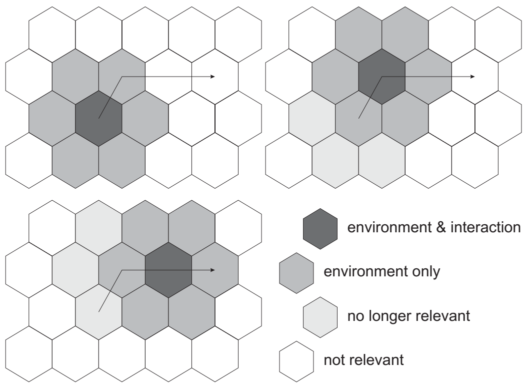

**Figure 1: Relevance of map pieces**

channel"). Given that a player is in a specific partition of the map, he only needs to subscribe to both channels for the specific partition, in which he wants to interact. Regarding the adjacent hexagons, it is sufficient to receive environmental updates as long as the player does not want to interact with the game play taking place there. In particular, the environmental data received through the environment channel should provide enough information to decide whether interaction with an adjacent hexagon might take place (e.g. reaching weapons range). In this case, the player must also subscribe to the interaction channel of this hexagon. As the player moves across the map it is now easy to request the data needed by simply subscribing the new environment and interaction channels and unsubscribe the ones now out of scope for the player (see fig. 1).

A game server managing a specific spatial partition provides only constant or nearly constant data such as terrain and objects within the map. This means that it is sufficient for the server to receive only the data exchanged through the environment channel. It is even sufficient for the game server to hold only approximated positions of all player and non-player characters, while the exact information can be received directly from the source of the elements. Given our model, where every hexagon is associated with its own two game channels, we now can postulate requirements for these game channels:

- The channel for the environment needs relatively small bandwidth. Between the players, it should meet at least soft real-time requirements, but the connection to the game server does not need to. Environmental updates can be delayed, and sometimes even aggregated (e.g. position updates).
- The channel for the interaction among the players needs more bandwidth and it should meet real-time requirements whenever possible as it takes the role of the classical LAN. Missing deadlines will result in delays to the interaction, and therefore the smoothness of game play will suffer.

## 4.5 Interaction aspects

These requirements can be fulfilled easily, if we assume that clans or guilds are often network topologically located in the same area, e. g. connected via the same backbone. It is highly presumable that the network quality is quite good between these groups. If all character interactions are published only between the participants, as stated above, a low latency can be provided. If changes occur to the environment, all users in the player's hexagon get the updates directly. These updates also have to be sent to the server, but since all participants who need this data immediately get them directly, the server may be updated with low priority, when there is free bandwidth. This especially will improve the reactivity in fights, when a high event rate exists between the users. The modelling of the publisher/subscriber channels, which will take place in the next section, will try to respect these aspects.

However, this assumption is not always true. A particular charm of massive multiplayer games is meeting somebody not sitting next to you, neither geographically nor network topologically. This is likely to result in large delays and low bandwidth among the players that interact with each other. Popular solutions to this problem are to terminate the interaction ("Hey, that player is not moving!"), which is quite boring, or to delay the game play until all participants are up-to-date with their internal state. Stopping the game, however, is not feasible in a large environment with many players, because the game will not leave its synchronization phase. A possible solution is, to delay only the game play in the specific spatial partition in which the delays occur. In our model this corresponds to implementing an ordinary synchronization protocol via the interaction channel of each spatial partition, thereby not interfering with other players in other partitions of the map and their interactions. The price, however, is the loss of a global (physical) time base as parts of the game run "slower" than others. But, after all, it is still possible to maintain a partial order by using a logical time, for example Lamport Time. If the scenario permits it, one might also use vector clocks.

## 4.6 Environmental aspects

In separating environment and interaction and letting the clients handle the interaction themselves, the load on the server cluster is dramatically reduced. This is the cause since nearly all interactions can be handled by the clients, and the servers just have to manage the terrain and fairly rough information on the players. This results in an increased number of spatial map pieces a specific server can handle. More important, the server does not have to scale with the number of the clients participating, but with the size of the hexagons or (equivalent to that) with the number of hexagons it has to maintain. As this parameter can be chosen by the game engineer, it is possible to flexibly add, remove or upgrade game servers as the game world evolves.

Since the exchange of information is done through information channels and not between two participants (one client, one server), the clients never have contact a specific server during game play. Using the publisher/subscriber model, the addressing is based on selecting the interesting channels, and not based upon a hardware dependent address, like an IP or an IPX network address. Therefore it is even possible to have complex hexagons handled by more than one server, replace a game server on-the-fly or to transfer hexagons from one server to a different one without the clients even noticing it. Needless to say, this poses immense challenges to the game server software, but from the communication layer's point of view, it is no problem.

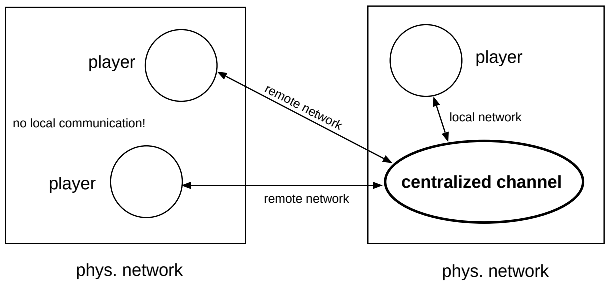

**Figure 2: Centralized channel approach**

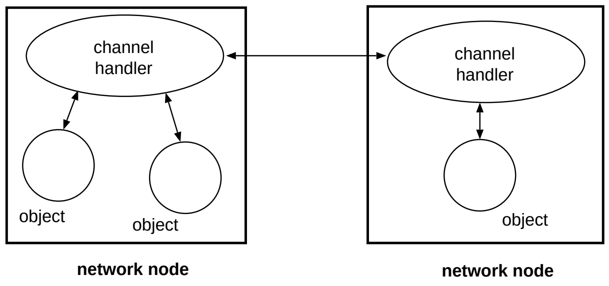

**Figure 3: De-centralized channel approach**

## 5. IMPLEMENTATION ASPECTS REGARDING THE PUBLISHER/SUBSCRIBER MODEL

The implementation of the publisher/subscriber model that is used in our approach is a revised version of the master thesis of one of the authors [7]. Therefore it will not be discussed in every detail. However, regarding the fact that it is vital to make our approach work as predicted, the most important aspects will be discussed briefly.

One of the most central points is the modelling and the implementation of the abstract publisher/subscriber "channel" concept. A first idea would be to generate a central instance handling all messages published into this channel[4], see fig. 2. This, however, is not suitable for our approach. Not only that two publishers (players) in the same physical network would not communicate directly with each other, the centralized channel would also reintroduce a centralized "game server" handling all object communication, both interaction and environment updates. It was an explicit goal to avoid this.

The next step ([4] calls this "federated event channel") is to distribute the channel over the participating network nodes. As this distribution must be transparent to the participants (otherwise you lose the lean programming interface), a new component is created that will handle the distribution of the messages to the other parts of the channel. This component is, in our implementation, named "channel handler" [5]. Figure 3 illustrates this model. Now, with the channels distributed among the nodes, no central instance is needed for communication. Every object publishing a message will publish to its local channel handler. The channel handler, in turn, will transfer the message to other local objects subscribed to this specific channel and to other channel handlers having objects subscribed to this channel.

Obviously, this introduces new problems. For the approach to massive multiplayer games outlined in this paper, two of them are extremely important: first, an explicit group membership is introduced, as one channel is now represented by a group of channel handlers. Second, how is the communication among the channel handlers realized, is it or shall it be reliable, unicast, multicast, broadcast, real-time, and so on.

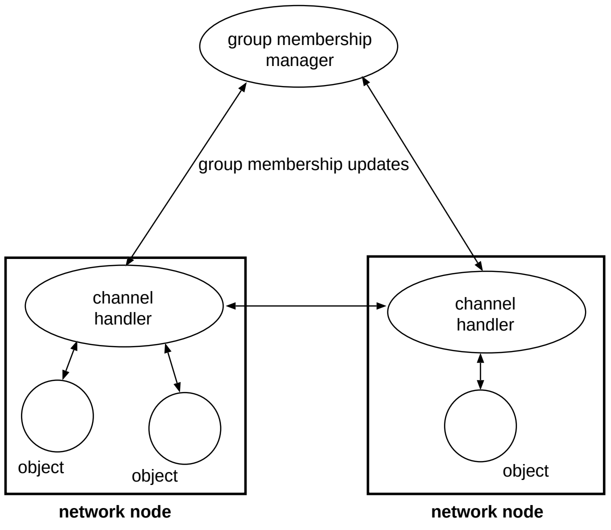

**Figure 4: Managing channel memberships**

Concerning the group membership, a channel is represented as a list of objects subscribed to it. Given our approach with the channel handlers, every channel handler can maintain a list of its local objects subscribed to a channel, so from a meta-view the channel is a list of channel handlers. A channel handler is part of that list, i. e. member of the channel group, if it has at least one subscriber. Still the problem is left, who maintains these lists. In our implementation we use a single instance, a group membership manager (see figure 4). While using single centralized facilities (=single points of failure) normally is not a good idea due to reliability and scalability issues, it proved itself feasible for our approach: The load generated by group membership management is small, and therefore we ran our tests perfectly using the centralized group management implementation. However, this is clearly a point for future improvements.

Concerning the communication among the channel handlers, this is difficult, because not every channel has the same requirements. For some channels it might be important that every message is delivered within a given deadline, for other channels it might be more important that no single message is lost, no matter how long it takes to deliver it. Presuming that the implementation of the publisher/subscriber protocol we use will connect nodes over the Internet and that we want to maintain a behaviour similar to a raw network, the network protocols we may use are restricted to the possibilities of IP:

- *UDP unicast*: The UDP unicast is an unreliable datagram service. As it adresses only one recipient per datagram, a publishing channel handler has to transmit a single datagram to every event channel handler that has an object subscribed to the channel.
- *TCP unicast*: The TCP unicast is a reliable data stream

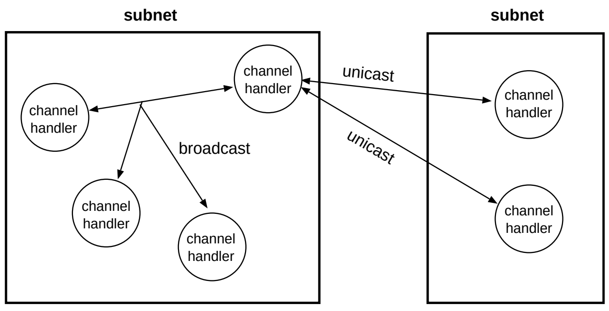

**Figure 5: Subnet clustering is needed**

connection. It also adresses only one recipient for the stream, so a publishing channel handler has to generate a new stream to every event channel handler that has an object subscribed to the channel.

- *UDP broadcast*: The UDP broadcast is an unreliable datagram service. It adresses every network node in the sender's subnet, and therefore a publishing channel handler has to transmit only one datagram to reach every event channel handler in its subnet. Normally, UDP broadcasts are not routed through the Internet.

- *UDP multicast and own protocols:* The UDP multicast and own protocols are probably what could be most useful for the needs of publisher/subscriber. But, unfortunately, they are not guaranteed to be routed in the Internet[1], thus they are not used in our implementation. A game is worth nothing if nobody can play it.

Taking into account the characteristics of the two types of channels (interactive, environmental) our approach lined out earlier, the following transport protocols have been chosen:

The environment channel produces (compared to the interactive channel) less network traffic, hence consumes less bandwidth, messages can be delayed but they should not be lost. Therefore all connections made to deliver messages in the environment channel are TCP connections.

The interactive channel should, as stated above, behave as similar as possible to a raw network (e. g. ethernet). Due to this, in the local network (a local network is identified by a network route indicating a direct route, not a gateway route), UDP broadcast is used. Concerning the connection between the channel handlers not in one subnet, we decided to use TCP: channel handlers not in the same subnet are probably spread far over the Internet, where packet losses are not unlikely. While a possible game engine still has to cope with the increased latency that will be induced by a TCP retransmit, it does not have to handle the retransmit itself.

The final aspect concerning the implementation of the publisher/subscriber model that will be discussed here, is clustering. Figure 5 illustrates the problem: While a group of event channel handlers in the same subnet as the publisher of a message will be reached by only one UDP datagram, a group in a remote subnet will be reached by a corresponding number of unicasts producing avoidable network traffic. To avoid this traffic, local clusters are introduced, wich is technically done by instantiating the whole publisher/subscriber model as it is described in this chapter into itself. Every subnet cluster is represented to the outside as a single channel handler, just that it has not "simple" objects as subscribers but channel handlers. The one event channel handler that is visible to the outside, call it cluster gateway, manages its own group membership lists for the machines inside the cluster and it subscribes to a channel as soon as one machine inside the cluster subscribes to the channel (at the cluster gateway). Similarly, every message published inside the cluster is bridged to the outside and vice versa. In figure 6 a clustered scenario is displayed. You can see how a message that has been published by an object connected to channel handler 'AB' travels through the system. No redundant network connection is made. Also is not visible wether channel handler 'C' is a single channel handler or wether it is hiding a cluster behind it. Please note that the programming interface for the client applications is still publish, subscribe, unsubscribe, receive.

[1]Many ISP's block multicasts and protocols other than TCP/UDP in fear of hackers that are trying to expoit IP implementations.

## 6. EVALUATION

### 6.1 Goal

As the implementation of the publisher/subscriber network layer was adapted to meet the requirements described in this paper, we needed to verify our assumptions. For this reason we implemented a demonstrator to generate network traffic similar to the traffic arising in real games. The demonstrator is solely focused on the network behaviour. We did not program any mechanisms to ensure fairness or bandwidth optimisations like dead reckoning. A simple synchronization scheme is used, in which the objects publish their updates in the next simulation step whenever necessary. And since all people using the demonstrator are well known, there was no need to prevent cheating. Our implementation of a game client simulator is quite simple, but should cover most game situations from the view of the network layer. It realizes two different types of objects: Objects that will represent static or mostly static environment objects, and active objects for characters under control by a player or by a computer. According to their properties these objects may interact as described in the following paragraph.

### 6.2 The Map

As described in the previous chapters we tessellate the map into pieces of the magnitude of normal multiplayer maps. We implemented two map types in the demonstrator, one with rectangular and one with hexagonal tiles.

On the screenshot shown in fig. 7 you can see three active objects, with two sharing environment channels. All tiles that are black are not involved in network communication on this node. Tiles drawn in dark grey are subscribed only to their environment channels, while the light grey ones communicate on both, the environment and the interaction channel. When characters from the same node stay in the same tile, the overall number of channels involved is reduced. This constellation is quite probable since in games using many characters these are typically grouped so that the player can handle them easily.

Handling rectangular tiles is very simple compared to hexagonal tiles. But from the communication view a hexagonal

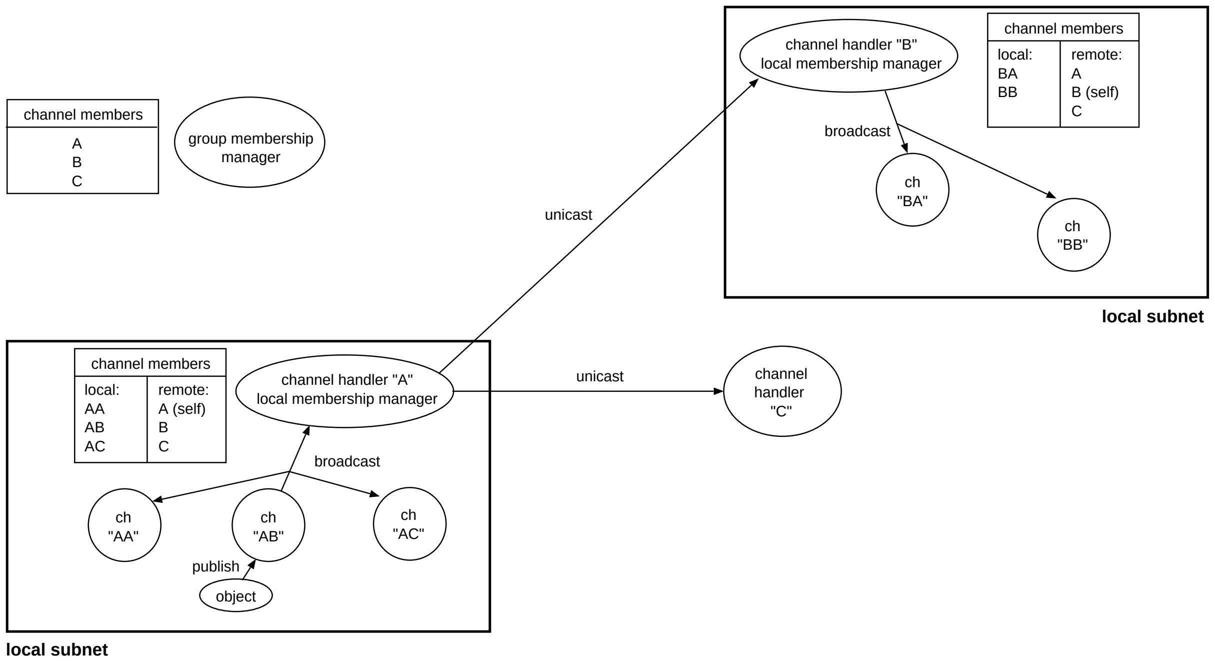

**Figure 6: A clustered publisher/subscriber scenario**

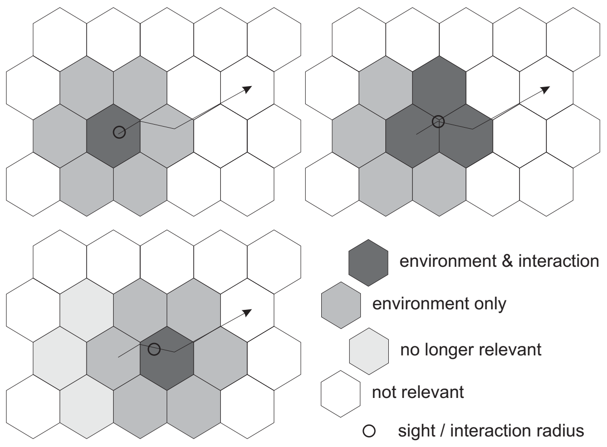

**Figure 8: Walking over tile borders**

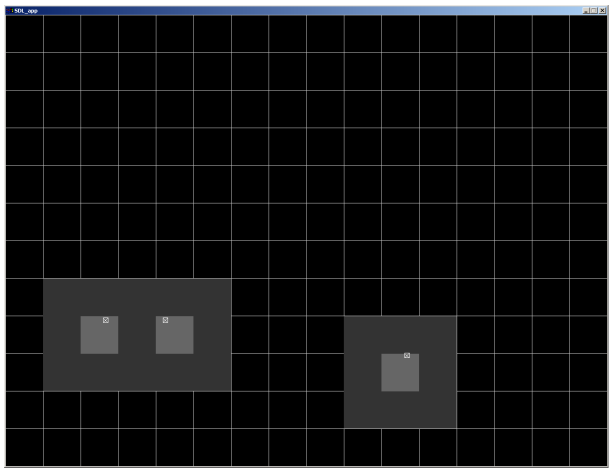

**Figure 7: The demonstrator with rectangular tiles**

tile has less adjacent tiles and thus less environmental channels that need to be subscribed and unsubscribed. This is especially the case when a player is near the border of a tile so that he may see into and interact with objects of adjacent tiles (see figure 8). When using a hexagon pattern a maximum of three interactive channels are subscribed simultaneously for one object compared to four in a rectangular map model. One of this additional interactive channel will most likely become the new current channel. Thus the ratio of additional channel subscribes regarding tiles useful only for a very short period of time, is one for the hexagon model compared to two for rectangular tiles in worst case. Considering this we estimated better results for the hexagonal model regarding both, interactive and environment channel subscribes.

## 6.3 Objects and Interaction

The first category is comprised of static objects that will communicate via the environment channel. The term static object includes all objects other than players or NPCs or their representation on other machines. Players may not interact with all objects of this category directly. Some objects represent map data and static scene objects that are used for rendering purposes only. This comprises environment data that may consist of terrain information, buildings, light sources and similar. Although the player cannot directly interact with these objects, they interact with him indirectly, e.g. by restricting his movement abilities when he tries to walk through undergrowth or walls. Other objects of this category support direct interaction with the player and may be used and collected by him. Some of these alter the character's properties like weapons and upgrades (e.g. a health pack or an armor), others may be simulated shops where the player can trade items, etc. All static items generate network traffic for themselves and for the character interacting with them. Upgrades disappear when they are taken, new weapons are picked up, item prices have to be negotiated, switches may be pressed and so on. Most of these updates are of low priority, and are thus handled by the environment channel. The second type of objects implemented are active objects. All characters controlled by the players are modelled as active objects. These interact with the static objects as describes above, but they also interact with other active objects, e.g. talk, trade or shoot.

## 6.4 Communication

Although the demonstrator has only very limited graphics, it is able to simulate a game situation with static and active objects that are distributed and updated between the participants. All objects are dynamically distributed through the mechanisms described before. All objects from remote clients are represented by proxy objects that will encapsulate the network layer and provide interfaces for interaction. On one client all interactions are done between local objects that propagate the updates themselves. These objects also provide their own rendering method, so drawing them is easy. If an interaction between two object occurs, the updates for all involved objects are calculated locally. The local instances of all objects affected publish their updates through the appropriate channels to all other proxy objects and of course the real instance of the object. Since the instances and all its replicates are located in the same tile (if they are not out of sync), it is sufficient to send updates only through the channel of the current tile.

## 6.5 Results

In order to generate comparable situations for statistical purposes, all actions taken by active objects in the demonstrator are simulated using a pseudo random algorithm. The results were obtained with one single client moving over the map, while the other objects were provided by other clients. First we evaluated the average number of subscribed channels in our favoured hexagon model compared to simple rectangles. The results are shown in table 1. Column "A" is the average number of interactive channels subscribed with hexagonal tile shape, column "B" corresponds to rectangular tile shape. Columns "C" and "D" show the average number of environment channels subscribed with hexagonal (C) and rectangular (D) tile shapes. According to the average number of subscribed interaction channels the hexagon model nearly equals the results of the rectangular model. The average number of subscribed environment channels is approximately 2 channels less in the hexagon model. This is as expected: A hexagon has sixn adjacent hexagons, a rectangle eight adjacent rectangles.

**Table 1: Comparing hexagons and rectangles**

| #tiles | tile size | A | B | C | D |
|---|---|---|---|---|---|
| 5x5 | 160x160 | 1,13 | 1,13 | 7,29 | 9,29 |
| 10x10 | 80x80 | 1,42 | 1,45 | 7,58 | 9,61 |
| 20x20 | 40x40 | 1,91 | 1,89 | 8,04 | 10,01 |
| 40x40 | 20x20 | 2,78 | 2,78 | 8,79 | 10,78 |

**Table 2: Varying number of tiles and tile size**

| #tiles | tile size | #cells | A | B |
|---|---|---|---|---|
| 5x5 | 160x160 | 25 | 4,54% | 29,14% |
| 10x10 | 80x80 | 100 | 1,42% | 7,58% |
| 20x20 | 40x40 | 400 | 0,48% | 2,01% |
| 40x40 | 20x20 | 1600 | 0,17% | 0,55% |

Table 2 shows the relationship of the tile size and the percentage of the map subscribed. Column "A" shows the percentage of interactive channels while "B" shows the percentage of environment channels subscribed when moving one player over the map. It is visible that with an increasing number of tiles the percentage of the map which must be subscribed is reduced dramatically.

Tables 3 and 4 evaluate the number of events received when varying the number of objects on the map. Table 3 shows that, when the tiles are very large, the number of events received (and thereby the communication bandwidth needed) grows dramatically. However, table 4 shows that, if the tiles are chosen of a middle size, the number of events received increases very slowly, even with an increasing number of objects on the map. This is because the amount of objects in interaction range is reduced by the cell concept. While in the scenario shown in table 3 every increase in object density is refelected in an increased number of received events, choosing smaller cells (as in the scenario outlined in table 4) will keep the number of received events in the same order of magnitude, even when the object density increases. This allows the number of players to be increased without increasing the communication bandwidth needed for every single player too much.

## 7. CONCLUSION

In order to cope with the challenges that rise with massive multiplayer online games, an architecture based on the publisher/subscriber model has been presented. Several steps have been proposed to handle network- and communication-

**Table 3: Map with 25 cells, tile size 160x160, 5x5 tiles**

| static obj. | dynamic obj. | events received |
|---|---|---|
| 5 | 5 | 1758 |
| 10 | 10 | 2753 |
| 20 | 20 | 5255 |
| 40 | 40 | 13067 |

**Table 4: Map with 400 cells, tile size 40x40, 20x20 tiles**

| static obj. | dynamic obj. | events received |
|---|---|---|
| 5 | 5 | 786 |
| 10 | 10 | 836 |
| 20 | 20 | 846 |
| 40 | 40 | 830 |

related topics: The game map has been split into smaller pieces, as not every participant needs to know every move on the complete map. The publisher/subscriber model has been found suitable to fulfil the communicational needs for inter-player and player-environment interaction. By separating the interaction among the players and the environment, a possibility was found to reduce the demands posed to game servers regarding both computational power and processing bandwidth to levels known from existing multiplayer games.

On the downside, the appproach presented deals only with the scalability issues introduced in massive multiplayer games. No proposal is made how other network issues or cheating can be dealt with. A detailed evaluation regarding the feasibility of the integration of existing solutions for these issues in our concept is missing. Finally, as a real massive multiplayer online game cannot be realized easiliy for a quick test of concepts, we were restricted to simulating certain aspects. The final proof, however, is missing.

## 8. REFERENCES

[1] J. Aronson. Dead reckoning: Latency hiding for networked games. *http://www.gamasutra.com/features/19970919/aronson_01.htm*, September 19 1997.

[2] Y. W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization.

[3] P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming an age of empires and beyond. *GDC 2001*, March 22 2001.

[4] T. H. Harrison, C. O'Ryan, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time corba event service. *Proceedings of OOPSLA 1997, ACM*, October 1997.

[5] J. Kaiser and M. Mock. Implementing the real-time publisher/subscriber model on the controller area network (can). *ISORC99 proceedings*, Saint Malo 1999.

[6] L. R. Rajkumar, M.Gagliardi. The real-time publisher/subscriber inter-process communication model for distributed real-time systems: Design and implementation. *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, 1995.

[7] M. Wallner. Ein publisher/subscriber Protokoll fuer heterogene Kommunikationsnetze. *University of Ulm, Germany*, March 2001.