State replication for multiplayer games

Carsten Griwodz

University of Oslo - Department of Informatics
Gaustadalléen 23 - 0371 Oslo, Norway
griff@ifi.uio.no

ABSTRACT

Massive multiplayer games have to cope with scalability problems that arise from the necessity of supporting network traffic for different game elements concurrently. In this paper, we show a simple means of separating these traffic styles in a proxy architecture by defining levels of urgency and relevance for each style. With this simple separation, low latency traffic styles can be preferred effectively over other traffic within the same game.

We continue with a proposal for a middleware approach to the use of this infrastructure. It is based on the assumption that game developers can specify static requirements for the distributed objects at design and development time. In this case, we propose a combination of code-generation and run-time support for the use of the network architecture.

KEYWORDS

Games, Distribution Infrastructure, Middleware

1. INTRODUCTION

More and more massive multiplayer games with action elements, strategy elements and a persistent game world are developed. Like most other networked games, they have the problem of latency due to their large geographical coverage, and of throughput due to the bandwidth restrictions of modem users. They have to cope with scalability problems that are worsened if server-controlled client-server architectures are applied that try to achieve an identical experience for all participants in the game.

In contrast to shoot-them-up games, massive multiplayer games require a communication subsystem that solves several competing requirements of the different game elements. These requirements must be taken into account in game design, but we realize that a perfectly identical experience is not necessary for all data. A persistent world requires that game servers are informed of all modifications to persistent objects of the game world, while much other information need not be kept. The necessary time in which the servers must be informed about such information may vary, but it must necessarily reach them reliably. Action elements involve direct player-to-player interaction with a need for low latency. Role-playing and strategic elements add requirements for communicating with other players or non-player characters, but the speed of human reflexes is not required. Player-to-player chatting comes in two variations, textual and audio chat. For textual chat, little bandwidth is required and considerable latency is acceptable, but reliable transfer is necessary. For audio chat, latency and jitter must be low but may still exceed the low latency required in action sequences that rely on reflexive behavior. This kind of chat may require considerable bandwidth but limited reliability may be acceptable, since players won't currently expect telephone quality in a game.

For the solution of these co-existing demands, we must take into account that the requirements may change within the course of the game. A massive multiplayer game tries to attract players with different interests and typically offers action-heavy and strategy-heavy, as well as social elements. This implies that it is not feasible to set network resources for one kind of game element aside for the duration of the game. Resources have to be re-assigned dynamically to suit the most important aspects of game play at any given time. An approach to exploit the different requirements to support such dynamics is proposed in section 2, including a protocol design and an evaluation of the approach. To make these functions available for game developers, we propose a middleware approach in section 3.

2. DISTRIBUTION INFRASTRUCTURE

2.1 Requirement mapping

The examples in section 1 present the variety of needs in a single massive multplayer game. In some of the examples, low latency is required from the network, in some reliable transfer is required, and in some both is essential. They demonstrate as well that different game elements have differently strong needs for low latency. Other examples may be found that demonstrate more than two intensities for reliability as well.

We assume that the lowest latency requirements of a game are typical for its action elements. Since most games have physical models for these elements, these requirements concerns typically very small groups of *participants* (players and servers), while other participants that are not actively involved in a given scene can accept higher delays. In a game with a limited number of participants, the selection of permissible latency may be made dynamically in the communication system for each message and each receiver. Potentially frequent changes of relative positions of participants and the potentially large number of receivers in a massive multplayer game make this infeasible. Similar examples may apply for reliability. By assigning a common limit for all receivers of a message, management complexity and computing resources can be reduced at the price of network resources. The specification of these limits can be expressed in two separate values:

- an *urgency* value: a higher urgency indicates a requirement for lower latency, and
- an *relevance* value: a higher relevance value indicates a requirement for higher reliability

The definition of a limited range of values along with the definition of receiver groups reduces the number of options that must be processed by the communication system. The specific mapping of these values must be defined by game design: each urgency value refers to a latency, each relevance value to a mechanism that increases reliability.

The interpretation of the two values depends on the features that are offered by the communication infrastructure. Some infrastructures may support service guarantees such as a limited end-to-end delay or a limited loss ratio, which could be exploited. In the current Internet, the meaning is weaker:

A high urgency value in a message requires that this message is transferred with a latency that is less or at most equal to that of messages with a lower urgency value. A low urgency value, on the other hand, indicates that forwarding of a message may be deferred.

A high relevance value makes the delivery of a relevant message safer than that of a less relevant one. One option for increasing safety is the use of retransmission. If these messages are also urgent, forward error correction may be chosen to prevent loss problems. The communication infrastructure may also drop less relevant messages, which can be used in several ways to reduce the bandwidth. They may be retransmitted only a fixed number of times before they are discarded. They may be kept by the communication infrastructure only for a limited amount of time, after which they are discarded. They may be dropped in an intermediate queue of the communication infrastructure when a threshold is reached.

2.2 Protocol use

To develop an architecture for the communication subsystems of a massive multplayer game with several urgency and relevance levels, we allocate several important details to the game design. One of the most important is concerned with the bandwidth that is consumed by the application. Just as with current game implementations, we require a game design that keeps the required bandwidth within the bounds that can be supported by the infrastructure - but these bounds will be slightly higher than before. We require the game design to handle issues of latency and jitter as before - but again, the bounds will be higher.

We call the data that is generated by the participants of a game *events*. Each event is created with an urgency, a relevance, and the id of a receiver group. Since these events must be delivered to groups of receivers, the use of multicast is an option. However, we refrain from using IP multicast for several reasons. One is that few ISPs support IP multicast, another that considerable effort is necessary to protect IP multicast groups from eavesdropping, a third that the assigned of one multicast addresses to each group would consume a large number of addresses. Finally, we assume that group membership is frequently changing, which does not work well with the time needed to join a multicast group [1]. Instead, we propose to deploy an overlay network of proxy servers that distribute the events to all participants that are members of the target group. Participants send all events that they generate to a statically assigned ‘closest’ proxy server for redistribution, which redistributes it all other proxy servers. Each event contains an identifier for its target group, which is evaluated by each proxy servers. The proxy servers identify all participants that are assigned to them and members of the group, and forward the event to them. We use the term *channel* to refer to such a group and its identifier.

Since event destinations are uniquely identified by their target channel and source IP address, we can multiplex events at the application level. A participant or proxy receives all events from another one on the same port. A pair of hosts that communicate directly are said to have a bi-directional *connection*, each with an outgoing and an incoming *interface*. The architecture exploits the urgency and relevance values of the events by introducing one priority queue for each supported urgency and for each outgoing interface. The queues are ordered by relevance and arrival time. Events from the queue with the highest urgency are always sent first. The relevance is considered in case of retransmissions and queue overruns.

Among proxy servers, we use UDP to apply prioritized retransmission and event dropping. The communication between proxy server and modem-connected clients is expected to be limited by the throughput rather than latency, so the TCP is also an option. On connections that use UDP, we implement retransmission. If the relevance of an event is so low that it will not be retransmitted, it is not kept after its transmission. Otherwise, it is re-inserted into its queue with a decreased retransmission count if it is not acknowledged. If a connection becomes a throughput bottleneck, the priority queue grows until the least relevant events are dropped. Since a proxy forward all events that it receives from a participant to all other proxies, the use of IP multicast among proxy is conceivable. Provided an appropriate retransmission approach, this would reduce bandwidth requirements in the backbone.

2.2.1 Reordering

The chosen approach deliver events out of order to the application. Even though events with identical urgency and relevance will not be reordered in the priority queues, packet loss may result in an out-of-order arrival. We assume that this is not a relevant problem. A distributed game must be able to handle out-of-order arrival of events from several participants, and out-of-order arrival of events from the same participant is only a minor additional problem. Additionally, events may be further de-multiplexed to unrelated game object inside the application, in which case out-of-order arrival may not have any significance at all.

2.2.2 *Congested modem connections*

Games are currently designed to restrict the bandwidth needs so that dial-up connections are sufficient. Our model does not overcome bandwidth limitation; the same amount of data has to be exchanged among hosts. The goal of re-ordering is to reduce the impact of temporary congestion on the game experience.

The described approach has no means of protecting the game from congestion between a client and its assigned proxy. Since we expect that a variable number and size of events is generated by participants and distributed to all receivers of the event's channels, temporary congestion at a modem link is likely even if the game design reduces the average throughput of the channel. Jitter in the network may result in similar temporary congestion. Longer term congestion may be the result of reduced backbone bandwidth. If this overloads is transient, reordering gives more urgent packets the opportunity to pass less urgent ones. If it is not transient, the approach will still result in a preferred delivery of urgent events, while less urgent events will be lost. We consider this a game design issue. The game may allow the participant on the congested link to play the game with reduced quality.

2.3 Packing

Game participants produce events of different sizes. It is likely that many events are considerably smaller than a maximum MTU size, which implies that several events should be packed into a single packet. If we don't combine several events into a single IP packet, the traffic that is generated in the backbone may be considerable. For modem-connected clients, the reduced overhead that is achieved by packing will also increase the throughput of the connection. These advantages have to be weighed against a latency increase for some events on the other side.

In general, a participant in a game will generate independent events that should be made available to other participants with the lowest possible latency. This contradicts the necessity to delay the transmission of events in order to fill a packet.

In our approach, we apply the urgency criteria again. We introduce additional latency depending on the urgency of a specific event. The distribution system has two intuitive positions to perform packing. A packet may first be delayed at the client, and second in the proxy. The delays will result in more events to arrive in the priority queues, which can be combined into a single packet to exploit the maximum MTU size and reduce the networking overhead. Most urgent packets always trigger immediate transmission of a packet, even if it does not reach the MTU size.

The artificial delay will obviously result in a higher average end-to-end transfer time for events. This can only be turned into an advantage if more events reach their destination within a certain deadline than without packing and/or without the priority scheme. It is important to investigate whether the priority scheme by itself (i.e. by considering retransmission problems only) is more efficient than the priority scheme with packing.

2.3.1 *Operation*

It is a task of the game design to assign delay value to the urgency levels of the game. Whenever an event becomes available for transmission, its transmission deadline is determined from its urgency value, and it is inserted into a packing queue. The events is also marked with a maximum retransmission count (ala SCTP) depending on its relevance before it is inserted into the queue. The packing queue is a priority queue that is ordered by urgency as the first criterion and the latest possible transmission deadline for the event as the second criterion. The transmission deadline for the queue is the shortest deadline of all entries.

When a packet is filled, it is not necessary to pack events of the same urgency, relevance of channel. If events are in the same queue, they are relevant for all receivers at the other end of the queue.

2.3.2 *Performance issues*

Besides the latency problem, computing resources for the packing process must be considered. In clients, packing is a performance advantage because it reduces interrupts for arriving packets. The same is the case in servers. Proxies may serve a considerable number of participants, which increases the probability of good packing results. But the scalability of the proxy servers can be limited if disassembling packets and newly packing them consumes computing resources that impact the scalability.

The additional load must be compared with an applications that forwards individual events based on their channel. For the packing approach, a received packet must be disassembled, delayed, and packed. The disassembly does not necessarily involve copying of the data, but only parsing of the received packet. Similarly, sending a packet does not require a copy operation if a system call such as *writev* is available, which takes several memory blocks for combination into a single *write* operation. Considering the timing for delays, the number of interrupts can be reduced to maintaining a single timer per proxy, for the next deadline. Since the latency will always be dominated by network latency, specialized timers should not be necessary.

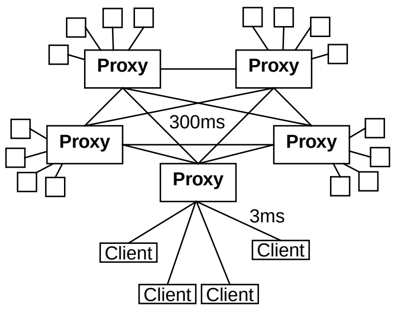

Figure 1: Topology

2.4 Evaluation

The evaluation of the approach is performed by simulation. The simulation consists of a fully meshed network of proxies, to which several clients are directly attached. Events are generated randomly at such a rate that their average combined bandwidth consumption does not exceed the link capacity of all receiving hosts.

We want to investigate the effect of the reordering in proxies and access networks on the average delay for events. To model the behavior in a wide-area network, we consider 300 ms delay in the backbone network, and 3 ms in access networks.

Figure 2 shows the results of a test that demonstrates the effects of the delay separation. Using the topology shown in Figure 1, the average delay of events is shown when urgent and non-urgent traffic is combined (1a and b), no mechanisms are applied (2), and packing is used (3).

In this experiment, we investigate the use of different loss rates. To see the development of end-to-end delays for the case of the various classes with increasing packet loss, we limit the access link speed to 56 KB/s but do not limited the capacity of the backbone links. In this experiment, 50% of all events are urgent, and 50% are non-urgent. The relevance is identical; events are retransmitted until they reach their goal. The average bandwidth consumed by the events is slightly below that of the clients' access links. As expected, the end-to-end delay of the system without the urgency mechanism (2) is quite accurately in the middle between that of graphs (1a) and (1b). The steep increase in the end-to-end delay with an increased loss is also expected because we require retransmissions by time-out until a successful acknowledgment has been received.

The clarity of the separation is more astounding, considering that retransmission time-outs are identical, the bandwidth of access links is not exceeded, and events are not artificially delayed at any host, but are combined into a packet if they are queued for transmission. This means that a latency differentiation of 50 ms is achieved by re-ordering the queue for jitter compensation at the downlink and retransmitted packets that enter the send queues between proxies.

3. MIDDLEWARE

Based on the networking infrastructure introduced in section 2, we propose a middleware that can help in the development and

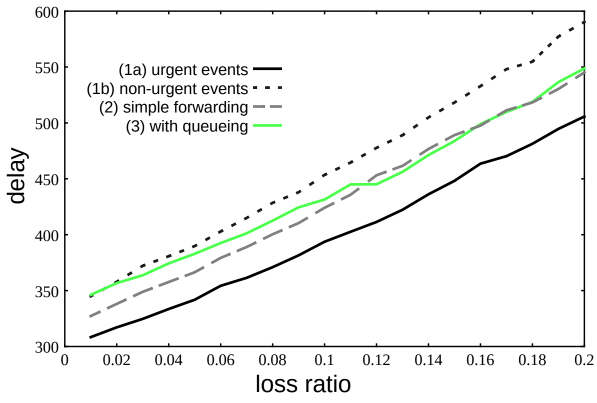

Figure 2: Delay separation

deployment of massive multplayer games and allows players to act in the presence of latency in the update transmission of events.

We have noted that some events which are generated by one participant in a massive multplayer game have high, and others have low relevance to other participants. We expect that developers will be able to determine high-level urgency and relevance requirements at game design time, in the same way as game developers make design time decisions for graphical quality or the exactness of physical models. Design time decisions allow distinctions between data that influences highly interactive scenes like fighting, slightly less time-critical data such as player-chats, and time-uncritical interaction like selling and buying of goods, which influences long-term game play but has little relevance for the immediate update of the display. At design time it is known, for example, in which way the requirements of data delivery from a client to a game server and the delivery of the same data to other clients differ.

3.1 Philosophy

We aim at support for games that are designed as distributed, partially replicated applications. Our middleware approach supports the development and operation of such replicated application. The middleware is applied in the development phase and in the operational phase. It is intended to reduce the effort of mapping liberties that are identified in the game design to working code. The middleware should allow the handling of an object in spite of its transparently changing state like local data in most situations. It must also enable the application to react to events that are received and need specific action.

The handling of dynamically changing objects like transparent atomic data is similar to the handling of streaming media objects in multimedia middleware. In addition to the consumption of their output, which is time-aware, they can frequently also be manipulated like static objects. In the same way that a state change in a video object may be triggered by the arrival of a frame, our objects change their state when an event arrives. By taking this view, it is intuitive to see the similarity to other streamed data such as audio, video and 3D scenes.

The objects can thus be interpreted by the applications as event streams. A distributed data item in this design is not a passive data unit for the application that includes it, but it can be considered as a time-dependent stream of events.

For want of better terms, we use terms *data type* for the generated code of these objects, and *object* for their instances. It would be inappropriate to use the term component because we don't intend to re-use the results of the code generation.

3.2 Task List

To exploit the advantages of limited exactness, it must be made possible to implement software that formulates its requirements. This requires several levels of support to developers. At the top level, it must be possible to design applications with a finely granular definition of the requirements. These definitions can be statically verified and exploited in code generation, which should result in data types and functions that simplify the asynchronous communication at run-time.

For this reason, we have identified issues that are performed in each of the phases. We use compile time decisions for information that depends only on game design decisions:

- Assignment of urgency and relevance level
- Evaluation of relations between objects
- Selection of resolution models
- Selection of a prediction mechanism
- Intermediate data types for delayed evaluation support

Run-time decisions are related to object handling based on these decisions, such as:

- Dereferencing of object vs. remote procedure call
- Replication and de-replication, including garbage collection of the final copy
- Output from generated objects to local objects, including prediction
- Creation of related objects with the same channel, and dynamic assignment of participants to channels as they dereference the object

3.3 Code and Data

In our assumption about the distribution of code and data, we follow the design that is derived from the idea of state machines that communicate by exchanging events. Code and static data is distributed off-line, and distributed copies of shared dynamic data are updated by exchange of events. This approach is similar to remote function calls but application level processing results are not handled implicitly by the middleware. An implementation of function calls on top of it would be straight-forward. It is also similar to distributed shared memory but the data units that are transferred are typed at the middleware level.

Code that influences the locally available data is locally available to all participants. Code that manipulates data that is not locally available, such as statistics or writable copies of the user profile, exists only on some hosts; typically all of the game servers.

Objects are dynamically created and named by a host. The objects will be made available indirectly available to other hosts, by updating objects that include pointers to the newly created object. Two critical problems of this scheme are race conditions in the creation of new objects and conditions for their deletion.

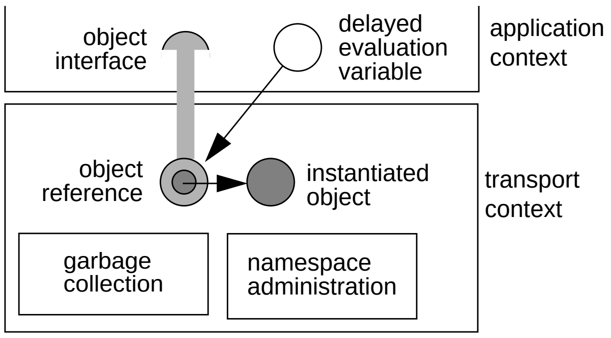

Figure 3: Separation of contexts

3.3.1 *Context Separation*

To enable the use of a distributed object like a local object by the application, we need to perform the distribution of events transparently for the application. We distinguish a *transport context*, that handles communication matters of objects on behalf of the application, and an *application context,* in which the local copy of the game executes with limited awareness of the distribution of its objects.

Figure 3 depicts the separation. In the transport context the middleware communicates with its peers asynchronously to the application that uses the objects. Updates that are received from peers modify the local copies of objects transparently unless conditions are met that require notification of the application context. In the application context the middleware provides access to the objects. The available mechanisms for loss or latency hiding are applied on-demand when the application reads data from the objects. As seen in Figure 4, a write operation in the application context results in the distribution of events in the transport context.

3.3.2 *Data Types*

As usual, classes in our approach are specified during the development phase. In contrast to a purely data-driven design of the data types, we allow the specification of urgency and relevance level for a class. The mapping of these urgency levels to maximum transmission delays, and relevance levels to retransmission counts is done for the entire application. In the code generation phase, strong resolution between data types are detected, and the use of different urgency and relevance levels for strongly related data types is refused.

In addition to these levels, we introduce a choice of conflict resolution models into the data type specification phase. Resolution models determine the choice of synchronization mechanism that is instantiated for each object of the class. Examples are *merge* or *rewind* operations. Each class has two resolution models, one model if chosen for conflict resolution if the urgency deadline is met, one for conflict resolution if it is not met. Resolution functions that need insight into the semantics of object can only be generated for simple data types. For complex data types, developers have to provide the specific resolution functions. This means that a 'rewind' resolution can be generated, but a 'merge' resolution will usually require developer code.

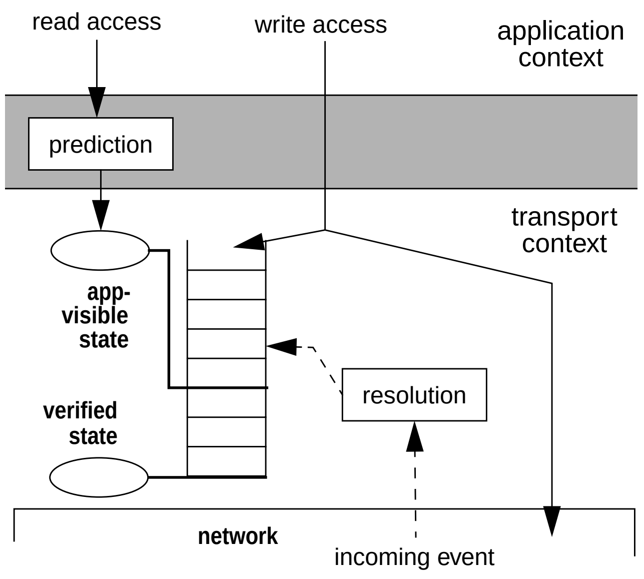

Figure 4: Interaction of contexts

While the conflict resolution is performed in the transport context, access to data is performed in the application context. To retrieve a valid value, two mechanisms are applied: a prediction mechanism for delay compensation and delayed evaluation for improved performance.

Like resolution models, prediction mechanisms can only be generated automatically for simple data types and simple prediction mechanisms. More complex data types need support functions for the computation of member of the data type that are not transferred. More complex prediction mechanisms, such a prediction based on a physical game model, requires developer code.

Delayed evaluation functions and intermediate data types are generated to allow abstract processing in the code. In the application, the generated data types are used like regular object, but unless output to a non-generated data type is produced, the evaluation of the current state of the object is not performed. This has several advantages:

- Costly prediction is not performed as frequently as if evaluation would be performed immediately.
- When the evaluation has to be performed, further updates from peer objects may have arrived, yielding a more accurate result.
- If the procedure involves a dereference operation on an object that has been replicated yet, the evaluation delay hides part of the time before the object becomes available

3.3.3 Grouping of Objects

Grouping of objects is an appropriate means of reducing the management overhead of the transport context. It can help in predicting whether objects will be used by a participant in the near future. If this information is know, a group defines common receivers for events and can be mapped to the channels that we defined as a multicast replacement in section 2. Groups of objects may also require a certain degree of synchronity, resulting in similar urgency and relevance values. An automatic determination of groups would thus be helpful. However, we find that it requires considerable work by the game developer. Two approach to grouping can be taken A group can identify objects that are related by implementation, i.e. if one of the objects changes, the others will be changed as well. And it can identify objects that are related by presentations, i.e. that will likely be important to a participant at the same time, without allowing an abundance of information that could be used for cheating.

Relations between objects by implementation should be prevented by the game developers. Objects should be as independent from each other as possible. Data types should be specified as unrelated as possible. Unrelated objects have no semantic relation with each other and can be handled separately. This is the ideal situation because no synchrony between the events that are distributed for each of the objects must be taken into account for the other. Each of the objects can even be re-synchronized with each copies according to differ virtual clocks. Lightly related objects are such that do not actively influence each other but that should not be manipulated independently. Such a relation must be specified by the game developer because it may mean a relation due to user perception (two items located close together in the game) or due to a relation that is introduced by a mutual influence through non-generated code. If such a light relation exists, the objects will share an urgency level and a virtual clock. Closely related objects can result in dependent updates, i.e. when an update to data arrives for one object, computations on the related object lead to an update that should be made available to all other participants in the communication as well. For this reason, relations have to be specified explicitly to our code generator. To prevent broadcast storms, we update the dependent objects in all copies of the game where it has been dereferenced, without generating updates for these changes. The code generator will enforce that in the presence of transitive operations (A is updated, A changes B, B changes C), a dereference-operation on A and C will automatically result in a dereference operation on B.

Relations between objects by presentation can be mapped more easily to a common channel. It would seem feasible to group related objects automatically based on the relation between them that can be identified at run-time. In the case of games this is unfortunately not appropriate. It is conceivable that data should not be available to a participant before certain requirements are met. For example, enemy positions should not be given away before they get into the participants line of sight. If the opponents are in the same room, it is likely that they interact with the same object without seeing each other. This means that although a relation between the mutually available object and each player is available, the closure of relations must not be used to determine which objects to make available to each participant. The solution that we consider feasible is that the game developer provides a segmentation algorithm to allow for the dynamic creation of groups; since the semantics are unknown to the middleware, it can not make reasonable decisions about presentation groups of objects without support by the specific application.

4. CONCLUSION

We have presented a network architecture to support the variety of networking requirements that massive multplayer games have because of their integration of several game elements. The approach relies on proxy servers but not on network QoS support. We propose to separate the traffic of the competing game elements be specifying their precedence in term of an urgency level and a relevance level. A high urgency level gives events forwarding precedence and reduces the average end-to-end delay. A high relevance level gives events loss protection. We have shown by simulation that these goals can be met, and that the preferred traffic achieves a relevant performance gain over a situation without classification.

To make these mechanisms available to game developers, we propose a middleware that provides of compile-time and run-time support. It separates transport context and application context to reduce the visibility of the network support in the application code to the necessary level.

4.1 Related work

The interface to the application reminds of SCTP with partial reliability extensions and in, this would be one means of implementing the protocol between the proxy servers [2]. Extended SCTP allows the definition of a transmission time-out as well as a retransmission count. It would reduce the necessity of the proxy infrastructure because of its additional operation as a filter.

A publisher/subscriber model for games has been proposed [3]. In our model, the channel assignment is defined by algorithms that are chosen by the game developers at design time. The operation is not as granular as the definition in the publisher/subscriber and may lead to overhead, we consider it the better choice because the assignment of channels to clients can be made by game servers. This prevents clients from cheating assign subscribing to events illegitimately.

An application of the relevance levels to fully exploit the bandwidth of clients is to integrate elements into the gameplay that are optional and can increase the players experience if present. This behavior is untypically for data distribution in games, but it is frequently applied in the graphics aspects of games [4].

4.2 Future work

In the near future, we want to complete an implementation of the distribution infrastructure, and verify the observations that were made by simulation. We plan to experiment with an integration into Quake and another game. A first version of the code generator will be built to support this integration. We will extend the simulation to validate new mechanisms. While the evaluation in section 2 is using simple priority queues, other schemes may be more feasible, such as priority promotion for retransmissions. We will also determine the applicability of the technique in other multimedia environment, for example interactive scenes in MPEG-4 sessions.

5. REFERENCES

[1] A. Garg, S. K. Kasera, R. Kumar and D. Towsley, “Measurement of Join Latency on the Mbone”, UM-CS-1999-047, August, 1999

[2] R. Stewart et al., “Stream Control Transmission Protocol”, RFC 2960, IETF, October 2000

[3] Ashwin R. Bharambe, Sanjay Rao, Srinivasan Seshan, “Mercury: A Scalable Publish-Subscribe System for Internet Games”, Proc. of NetGames 2002, April 16-17, 2002

[4] P. Astheimer, M.-L. Pöche, “Level-Of-Detail Generation and its Application in Virtual Reality”, Proc. of VRST’94 conference, August 23-26, 1994