Reducing game latency by migration, core-selection and TCP modifications

Paul B. Beskow, Andreas Petlund, Geir A. Erikstad, Carsten Griwodz, Pål Halvorsen

Simula Research Laboratory, N-1325 Lysaker, Norway
Department of Informatics, Univ. Oslo, N-0316 Oslo, Norway
Email: {paulbb, apetlund, geirerik, griff, paalh}@ifi.uio.no

Abstract: Massively multi-player online games (MMOGs) have stringent latency requirements and handle large numbers of concurrent players. To support these conflicting requirements, it is common to divide the virtual environment into virtual regions, and spawn multiple instances of isolated game areas; to serve multiple distinct groups of players. As MMOGs attract players across the world, dispersing these regions and instances on geographically distributed servers is plausible. With Ginnungagap, we present the prototype implementation and evaluation of functionality for migrating objects to support dynamic re-distribution of game state; with a distributed name server efficiently maintaining references to migrated objects. Core selection algorithms, using players latencies, can then locate an optimal placement for a given region or instance. The variance of the measured latencies (and core selection algorithm) are reduced by enabling our TCP modifications for latency reduction for non-greedy streams. Correspondingly, we anticipate a decrease in aggregate latency for the affected players.

Keywords: Latency reduction, massively multi-player online games, TCP enhancements, core selection algorithms, object migration, distributed name server

Biographical notes: Paul B. Beskow is a PhD student at the Department for Informatics, University of Oslo and Simula Research Laboratory. His research interests include optimization of resource utilization and distributed processing.
Andreas Petlund is a PostDoc at the Department of Informatics, University of Oslo and at Simula Research Laboratory. His research interests include network protocol optimization for time-dependant thin streams, operating systems optimisations and hardware offloading.
Geir A. Erikstad is a former master student at the Department for Informatics, University of Oslo. His main research interests include game latency optimisations.
Carsten Griwodz is a Professor at the Department of Informatics, University of Oslo and a researcher at Simula Research Laboratory. His research focuses mainly on distributed multimedia systems.
Pål Halvorsen is an Associate Professor at the Department of Informatics, University of Oslo and a researcher at Simula Research Laboratory. His research focuses mainly on distributed multimedia systems.

1 Introduction

In online games, some latency is tolerable (Claypool, 2005) as long as it does not exceed the threshold for playability that ranges from 100ms to 1000ms depending on the type of game (Claypool et al., 2006), but high pair-wise latency remains a primary obstacle for the quality of perceived game-play in a number of online games. For example, Massively Multi-Player Online Games (MMOGs) often rely on a client-server model for event distribution, where the events are collected at the server, and distributed to the interacting players. Players with high latency will inadvertently have a negative effect on the perceived quality of game play (Claypool, 2005). This occurs if one or more players connections are comparatively slow, because any added delay is not isolated to the player alone. Due to a centralized server, the delay will propagate to the interacting parties and potentially result in inconsistencies, from which the server must then recover. While having minor effects on the outcome of the game, it results in a perceived deterioration to the quality of interaction (Dick et al., 2005). As such, *low latency for all interacting players* is a prevalent goal.

This goal, however, contradicts other observations of online games, such that the game traffic varies strongly with time and the attractiveness of the individual game (Feng et al., 2002; Chambers et al., 2005) and that geographical dispersion of players in an online game depends heavily on the time of day (Feng et al., 2003). Thus, in large games like Anarchy Online and EVE Online, there are always players all around the world interacting in the same game instance, although the geographical location of the large bulk of users shifts dynamically.

In Beskow et al. (2008), we proposed the initial design of a middleware supporting migration of partial game-state to servers that are dynamically selected according to the majority of the players locations. The level of game-state granularity can range from entire virtual regions, to instances, or single objects in the virtual world. With GINNUNGAGAP (in Norse mythology GINNUNGAGAP refers to the vast, primordial void that existed prior to the creation of the universe) (Beskow et al., 2009a), we present an implementation of this middleware (the source code of the middleware, and logs from the PlanetLab experiments are available for download at http://simula.no/research/networks/software/). This middleware selects servers based on maximum latencies between all the players in a region and potential servers, and if the potential gain in terms of reduced latency exceeds a given threshold, the game state is migrated to this server. In this paper, we have focused on MMOGs that use a client-server model, though GINNUNGAGAP could easily be adapted for use in peer-to-peer scenarios. In which case, all interacting parties could potentially contribute as a server, greatly increasing the probability of locating an optimal node. It is nonetheless vital that the variance of the measured latencies, provided as input for the core selection algorithm, are as low as possible; one way to ensure this is to perform a number of probes, which can be costly and time-consuming (considering the number of potential clients). Another option is to increase the resilience of single measurements by applying our thin-stream TCP modifications (Petlund et al., 2008). Thus, we present results from experiments using this middleware running a simple game both in a lab environment and on PlanetLab. In summary, we show that the dynamic selection of servers is beneficial and can benefit from our TCP modifications, and that the overhead of migration and method invocation does not exceed the limits of a good perceived game play.

2 Background and Related Work

Migration techniques: Migration provides functionality to move entities of varying form and size (e.g., virtual machines, processes, data, code, etc.) between servers/proxies/clients in a distributed system. In Nelson et al. (2005), a virtual machine is transparently migrated with minimal impact on the user. Sprite and Mosix are *NIX based operating systems that allow for processes and their associated state to be migrated. Code migration makes it possible to defer the execution of code to an interacting party. Interpreted languages are typical for this, such as JavaScript in modern browsers and SQL in database management systems. Emerald and Chorus/COOL enable migration of objects (e.g., their code and state) during run-time execution of a program (per the object-oriented paradigm). Finally, we have eXternal Data Representation (XDR) and Boost::Serialization that provide functionality to migrate data. This is achieved by serialization, i.e., creating a binary representation of data in an application (e.g., ints, floats, chars, etc.), which is moved between instances of applications. Migration, at its various granularity levels, serves several purposes, such as dynamic load distribution, fault resilience, increasing resource locality or to facilitate system administration. We wish to achieve a combination of load distribution and increased resource locality by migrating game state to an optimal location (relative to the interacting users). Finding the correct level of granularity is important, and in online games there are two essential characteristics, 1) all code is shared, and 2) not all state related to an object needs to be migrated. Thus, we can eliminate some overhead by serializing only parts of an object. To this end, data migration accommodates both of these characteristics.

Server selection: Game server selection is an important facet of the playability for several types of online games. This is particularly true for games that are highly sensitive to latency, such as first person shooter (FPS) games. As such, the player's selection process is commonly guided by measuring dimensions that affect playability, such as latency and packet loss. This sensitivity to latency is commonly alleviated by distributing servers widely. With respect to this, Chambers et al. (2003) have looked at how server selection can be optimized for a single client, when given a set of available servers. In a further study, Claypool (2008) notes that we regularly find groups of players that wish to play together on a server, such as friends or clans (organized players). As such, he has investigated how server selection can be optimized from the perspective of a group of players. In two related studies by Armitage (2008b,a), efficient ways of ranking servers in the discovery process itself are examined. These papers have in common that they consider server selection from the perspective of the player(s), and additionally assume a certain availability of servers (it is common for geographically coupled players, such as real life friends, to play against each other). For a world spanning game, however, where all users interact in the same game instance, such as an MMOG, the number of available servers is often limited. Lee et al. (2005) present their heuristic for selecting a minimum number of servers satisfying given delay constraints (from the perspective of large scale interactive online games, such as MMOGs). Their aim, however, is to have well provisioned network paths in a centralized architecture. Thus, they do not consider the aspect of geographical dispersion of players. In a similar study, Brun et al. (2006) investigate how a server's location can influence the fairness of a game, and how selecting an appropriate server impacts this fairness. They use an objective function, which they call *critical response time* to rank the servers.

3 Implementation of Ginnungagap

The development of GINNUNGAGAP has been driven by the motivation of lowering the aggregate latency for groups of interacting players in interactive online games, such as MMOGs. To achieve this, three components were necessary, 1) a way of locating a server or proxy closer to the center of the players through core-selection, 2) a means of facilitating the migration of game-state, i.e., re-locating the players and the objects they interact with, and 3) allow for continuous interaction with the virtual environment, through remote method invocations (RMI), even during migration. GINNUNGAGAP supports all three components.

At its core GINNUNGAGAP runs three threads, **send**, which is responsible for transmitting middleware messages, labeled either RMI or MIG (method invocation and migration, respectively, see tables 1 and 2); **receive**, which accepts new connections and receives transmitted messages; and **process**, which processes the messages, and activates core-selection at given intervals in the servers and proxies. In addition, all clients run a (application level) **ping** thread, which is responsible for gathering and transmitting its latency information to the proxies and servers, which is used as input to the core-selection algorithm (explained in section 3.1).

All objects that support migration and RMI inherit from a base class that provides a minimal interface of methods and data that, correspondingly, must be implemented and present in all inheriting classes. The essential data in an object consists of the object's mode and its identifier. The mode of the object can be either `MIGRATING`, which implies that the object is in transit, and RMIs must be buffered until they can be forwarded and processed, or `NORMAL`, which implies that an RMI can be processed immediately. As an alternative to buffering messages it could be more beneficial to utilize related ideas, such as transparent migration of virtual machines (Nelson et al., 2005), where one variation utilizes a two-stage approach, with an initial push-phase, where data is pushed to the receiving node, data modified during this stage is marked as dirty and is retransmitted during the stop-and-copy phase, where, in our case, the object is again made available for interaction. The unique identifier of an object follows it throughout its life-time and throughout the distributed system, and is used to uniquely locate that object at any node. A name service is necessary, however, to maintain knowledge of which node the object is currently residing at.

Instrumental to the name service is an efficient way of maintaining references to the objects as they migrate. To accomplish this we have implemented a distributed name service. A name service can be implemented in several ways, which we discuss in the context of MMOGs in Beskow (2007). The conclusion is that a distributed name service is an efficient way of handling references as there is a minimal overhead in binding an object with the name service, because each node has its own name service, where the objects are bound initially. Communication between nodes (and thus the name services) only becomes necessary when an object is migrated. Given that servers in the system are geographically distributed, binding objects locally becomes quite beneficial. One other advantage is that there is no single point of failure, so large parts of the application can continue running if a server fails. Look-ups are also efficient, as we can directly query the node our name service has registered as current caretaker of the object. This access time, however, depends on the number of times an object has been migrated (after its point of creation) and at which point in this chain the invocation is performed. We partially alleviate this situation by embedding the calling nodes' information in a message that passes through such a chain, such that the reply is sent directly back to the calling node.

3.1 Core selection

Vik (2009) identified k-Median (see algorithm 1) as the heuristic algorithm for the graph-theoretical problem of locating an optimal proxy for a set of interacting clients, and this is the algorithm we use in the core-selection component. However, any of the node selection algorithms presented by Vik can be used instead of the k-Median algorithm in our middleware.

Algorithm 1 k-MEDIAN(G):

```
1: Input: An integer $k > 0$, a graph $G = (V, E, c)$. Sets $Z \subset V$ and $X \subset V$.
2: Output: A set $C \in X$ of core-nodes.
3: map<$x$-id, pair-wise> mapIdPairwise
4: for each $x \in X$ do
5:     pairwise = getPairwiseDistances($x$, $Z$)
6:     mapIdPairwise.insert($x$-id, pairwise)
7: end for
8: C = kLowestPairwise(mapIdPairwise)
```

The k-Median core-node selection algorithm finds k core-nodes that are the k nodes with the lowest average pair-wise distances to the nodes in the member-node set. The algorithm solves the k-minimum-pairwise problem, which when given a weighted graph $G = (V, E, c)$ and an integer $0 < k < |V|$, finds a set $C \subset V$ of size k, such that the sum of the distances from the vertices's $u \in C$ to all nodes $v \in V$ is minimal. The k-Median algorithm has a time-complexity of $O(n^2)$ on any graph.

3.2 Migration

Migration (see figure 1 for an overview of the process) targets are found by core-selection **(1)**, though migration is only performed when the latency gain increases above a predefined threshold. The migration is performed by the `MigrationService` **(2)**, which first sets the mode of the migrating object to `MIGRATING` **(3)**. Thus, all incoming RMIs are temporarily buffered locally, until they are forwarded and processed by the object at its new location. The buffered RMIs are marked as chained invocations, and as such, contain information about the node from which the call originated.

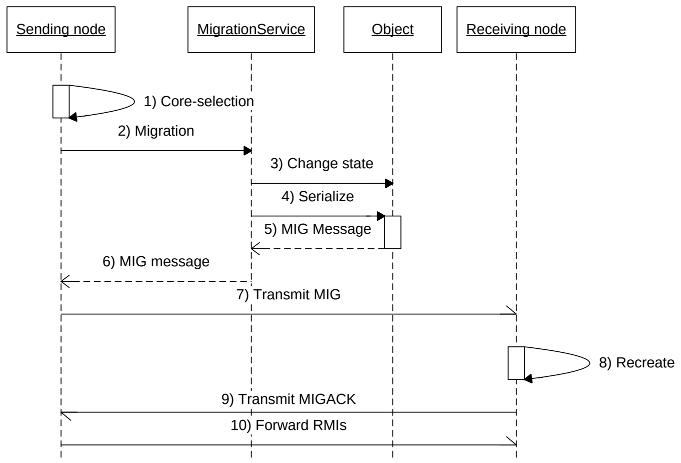


Figure 1 Sequence of a migration

After the mode update, the object is serialized **(4)**. The serialization results in the creation of a `MIG` message (see table 1) in the binary XDR format **(5)**. Not all attributes of an object are necessarily serialized; only those marked for serialization by the programmer. Once the message has been packed it is transmitted to the receiving node **(7)**.

Message type	Object id	Object type	Attribute	Attribute
MIG	7e2...31f	CLASS_PLAYER	Odin	98

Table 1 Migration message

At the receiving node, the message is processed **(8)**. The middleware reads the type of the message that it has received (`MIG`) and invokes the appropriate handler. It passes the remainder of the message to the MIG-handler, which reads the object type and requests that a new object instance of that type is created. The instance is created by calling the object constructor, which initializes the object with the attributes deserialized from the message.

Once the object has been created and initialized, the MIG-handler creates a MIGACK message with the id of the object and transmits this message to the sending node **(9)**. Upon receiving this message, the sending node knows that the migration has been completed and forwards any waiting RMI-messages in its buffer **(10)**.

This same process is used to migrate groups of objects, though with a slight modification. In the case of groups, all the objects are serialized into one large message, instead of being sent in small individual ones.

3.3 Remote method invocation

An RMI is the invocation of a method on an object that is not residing in local memory (see figure 2 for an overview). To accomplish this goal, a number of components need to be present in the middleware. In this example, we detail the interaction of these components, starting with a player logging into a remote server:

```
dist_ptr<Player> plyr = Startup::login(‘‘Odin’’);
```

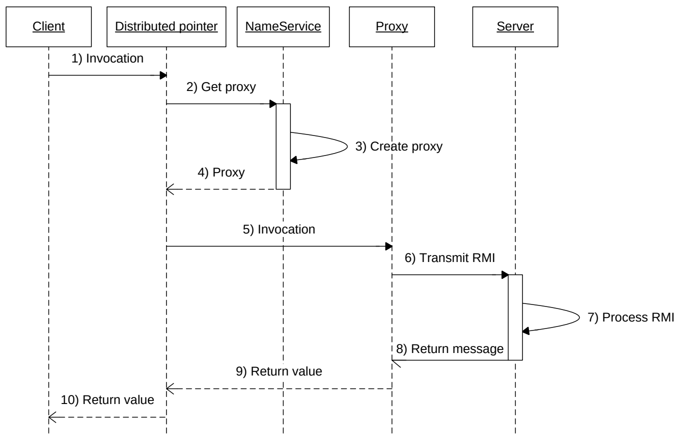


Figure 2 Sequence of a RMI

This `login(...)` invocation is in itself an RMI, but is possible because the server and `Startup` object have a well-known location and identifier. With this invocation, the server creates a player object and returns its corresponding universally unique identifier (UUID), which is guaranteed unique for this object throughout its lifetime. The UUID is entered into the name service of the local node. Together with network information about the sending node, this ensures that we have a way of contacting the remote node and identifying that instance of the object class. The UUID is also stored in the distributed pointer `plyr`. After input from the player, a request to move is issued:

```
plyr->move( Direction, Distance );
```

As `plyr` is a distributed pointer, the `move` method invocation is intercepted by the smart pointer **(1)**, which contains the UUID of the object it points to and is used to query the name service for a pointer to the object corresponding to that UUID. The name service looks it up in its records and performs one of two actions, 1) returns a pointer to the local object, or 2) returns a pointer to a proxy object if it is remote **(2)**. At the first request for a remote object, a proxy is created **(3)**, and is reused for subsequent requests.

As the object we have invoked is not local, the distributed pointer redirects the call to the move-method through the proxy object with the parameters passed to the method **(5)**. Methods in an object that support RMI are implemented as virtual methods and have their own implementation in the proxy object.

At this point the proxy object assumes control, and creates a message of type RMI (see table 2). This identifies the class type of the object, the specific object (UUID) and the function that is being invoked, and its corresponding parameters. All this information is serialized using XDR. With knowledge that the object is remote, the proxy object then transmits the message to its destination **(6)**.

Message type	Object id	Object type	Method	Parameter	Parameter
RMI	5c1...13c	CLASS_PLAYER	METH_MOVE	North	50

Table 2 RMI message

At the receiving node the message is processed by the middleware **(7)**. It determines the message type (`RMI`), and invokes its corresponding handler. This

RMI-handler determines if the object is local or remote, and in the latter case passes the message on (with the information of the emanating node embedded in the message). Thus, the reply, once the RMI is processed, can be sent directly to the original caller, effectively minimizing extraneous messages in the network. If the object is local, the corresponding skeleton method is invoked, which deserializes the parameters of the object, and invokes the method of the local object. Once the invoked method returns, the RMI has completed successfully at the remote node, and a potential return value from the function itself is serialized and returned. The middleware itself generates a reply to the calling node with information of the successful completion **(8)**. Finally, this return value is processed by the calling node **(10)**.

4 TCP enhancements for thin streams

TCP is used as the primary transport protocol in a number of existing online games (including MMOGs), because it inherently provides reliability and simplifies firewall traversal. Though using TCP in this scenario comes at a cost due to TCP's inherent retransmission mechanisms, which are optimized for TCP's targeted audience; bulk transfer scenarios. This is in contrast to the network traffic patterns exhibited by a number of online games, which send small packets with high inter-arrival times (*thin streams*). As a result, supporting MMOGs with reliable, low-rate, interactive streams using TCP on the Internet poses huge challenges due to packet loss. To reduce the loss-recovery delays, especially the infrequent but worst-case scenarios that have devastating impact on QoE, and to minimize the variance in the latency estimates for server selection, we have implemented enhancements to TCP's retransmission policies to reduce the latencies for thin streams (Petlund et al., 2008; Petlund, 2009). These modifications only need to be present at the sender (though the benefit is greater if both sides of the connection have them), and, as such, we support unmodified receivers running commodity operating systems (Linux, *BSD, OS X, Windows, etc.). These modifications are backwards compatible and only triggered when the system detects the described thin stream properties. In particular, we have changed the exponential back-off and fast retransmit algorithms, and added a bundling mechanism that uses available TCP segments to resend unacknowledged application data. It is important to acknowledge that these modifications are only enabled by the kernel for thin-streams, since enabling these mechanisms on greedy streams would violate TCP fairness principles.

4.1 Modified exponential back-off

TCP retransmissions are triggered by timeouts when a segment is not acknowledged in time and no fast retransmit occurs. To avoid congestion in a scenario with multiple consecutive losses, the retransmission timer is doubled each time (exponential back-off in figure 3). This leads to retransmission delays that increase exponentially when multiple consecutive packet losses occur. However, in the MMOG scenario, it is not necessary to use exponential back-off to prevent aggressive probing for bandwidth, because the stream is thin.

We therefore use linear timeouts for the first 6 timeouts, as shown in figure 3, before we switch to exponential back-off. For further discussions on the impact of these modifications on fairness and friendliness, we refer to Petlund (2009).

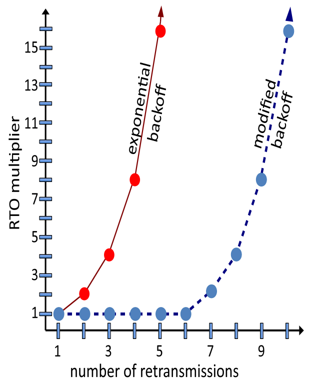
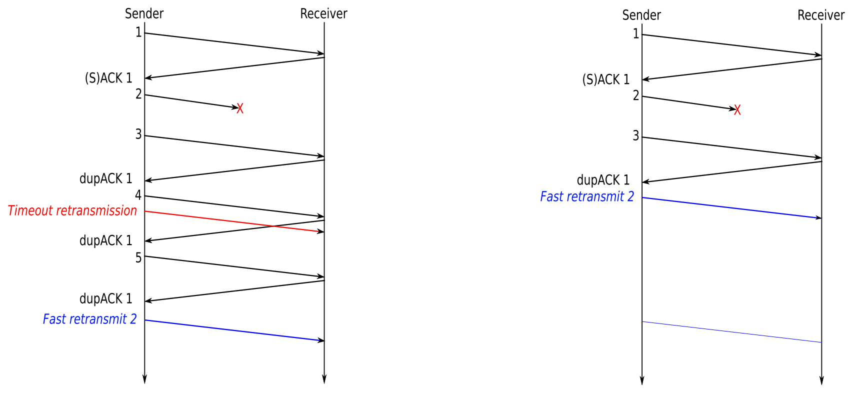


Figure 3 Modified vs. exponential back-off

4.2 *Modified fast retransmit*

Due to the high packet interarrival time in the game scenario, fast retransmissions occur seldom. In most cases, a retransmission by timeout is triggered before the three (S)ACKs required to trigger a fast retransmission are received (figure 4(a)). To deal with this problem, we allow a fast retransmission to be triggered by the first indication that a packet is lost (figure 4(b)). This is because retransmissions triggered by causes other than timeouts are usually preferable with respect to latency (Petlund, 2009).

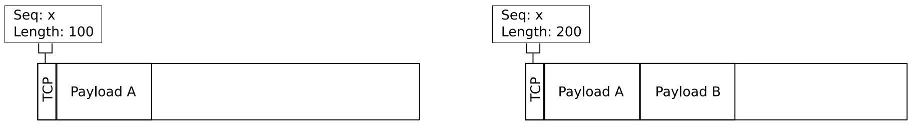


(a) Timeout and fast retransmission (b) Modified fast retransmission

Figure 4 Fast retransmissions

4.3 *Redundant data bundling*

To avoid retransmissions, we redundantly copy (bundle) data from unacknowledged messages in the send queue into the new packet as long as the combined packet size is less than the maximum segment size (MSS) (to avoid IP fragmentation). If a retransmission occurs, as many of the remaining unacknowledged packets as possible are bundled with the retransmission. This increases the probability that a lost segment is recovered faster. Figure 5 shows how a previously transmitted data segment is bundled with the next packet. The sequence number stays the same while the packet length is increased. If packet A is lost, the ACK from packet B acknowledges both segments, making a retransmission unnecessary. In contrast to the modified retransmission mechanisms of exponential back-off and fast retransmit, which are triggered when fewer than



(a) First packet (b) Second packet: Bundled data

Figure 5 Bundling unacknowledged data

four packets are in transit, redundant data bundling (RDB) is enabled by small packet sizes, and limited by the packet inter-arrival times (IATs) of the stream. If the segment is filled up to MTU size, either due to large writes from user space or due to sufficient data in the TCP send queue, which is typical for bulk data transfer, bundling is not performed.

5 Evaluation

To evaluate GINNUNGAGAP, we have performed several experiments and present the vital results of our testing, with micro benchmarks of the RMI and migration functionality and live tests on PlanetLab (see (Erikstad, 2009) for further results and details).

5.1 Micro benchmarks

The micro benchmarks were performed using a local 100 Mbps network and ran on two machines with identical hardware (Intel Core 2 Duo E6750, 2.66GHz CPU, 2 GB of RAM) with Ubuntu 9.04 installed.

Remote Method Invocation: To test the RMI functionality and overhead, we invoked methods with several different combinations of return values and parameters. The results of these tests are shown in figure 6. Each method was invoked 10000 times. As expected, an overhead compared to a local invocation is introduced by an RMI, partially due to the overhead of (de)serializing parameters (not necessary for local invocations), and also the latency introduced by the network. We see a gradual increase in the time to complete RMI invocations of vectors of integers (as the vectors grow in size), but this is mainly due to the increased transmission time of the data. Apart from this, the mean difference in a remote and local invocation remains quite stable. We can also see that the standard deviation is relatively low. In summary, there is some overhead associated with an RMI compared to a local invocation, but this is to be expected. The primary obstacle is the latency introduced by the network, and the size of data being transmitted.

Migration: In the migration tests, we have performed two sets of tests, one where we migrate single objects of varying size, and one set where we vary the size of the object groups and their size. The results of migrating a single object are shown in figure 7. As expected, the time to complete a migration of a single object gradually increases with the size of the object.

The results of migrating groups of objects are shown in figure 8. Here we see much of the same pattern as migration of single objects, with the gradual increase in completion time being mainly affected by the size of the objects and the number

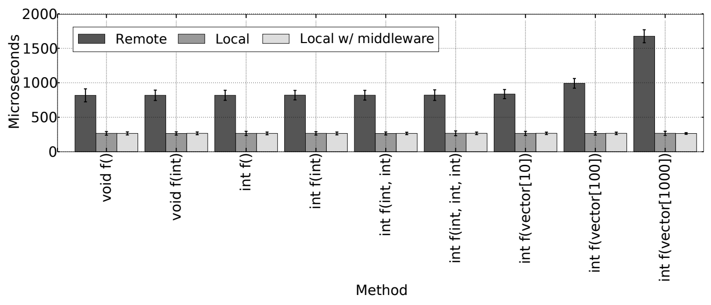


Figure 6 Average time of method invocation with stdev

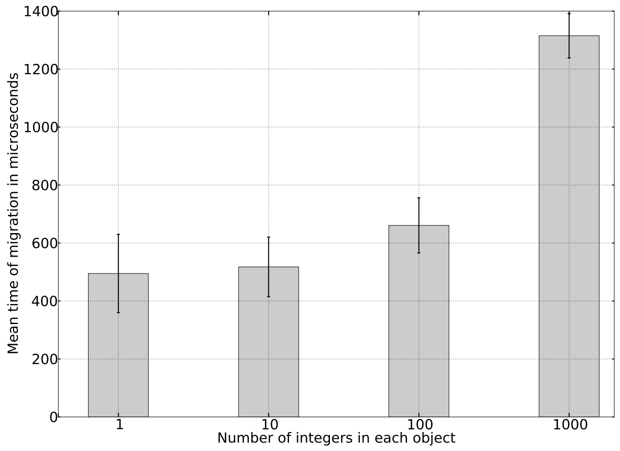


Figure 7 Average single object migration time with stdev

of groups. There is, however, more overhead introduced by the migration of groups, as the middleware has to perform additional work; it combines the objects into a single large message.

RMI during migration: We have also looked at the overhead introduced while performing RMI on an object while it is being migrated, and the results of this test are presented in figure 9. We continuously performed RMI to a group of objects that were being migrated back and forth between two servers at regular intervals. As we can see, it is clear that performing an RMI during migration adds overhead. This overhead is introduced because the name service has not been updated yet, and as such, RMIs are buffered until they can be forwarded. This buffering effect is witnessed in the greater variance displayed in the completion time for buffered/chained RMIs. Currently, we do not forward messages as they arrive, instead we wait for a `MIGACK` before forwarding the messages. It is reasonable to assume that this overhead could be reduced by changing this to an on-the-fly forwarding of the incoming messages. The messages would then be ready for processing as soon as the migration was finalized, effectively removing

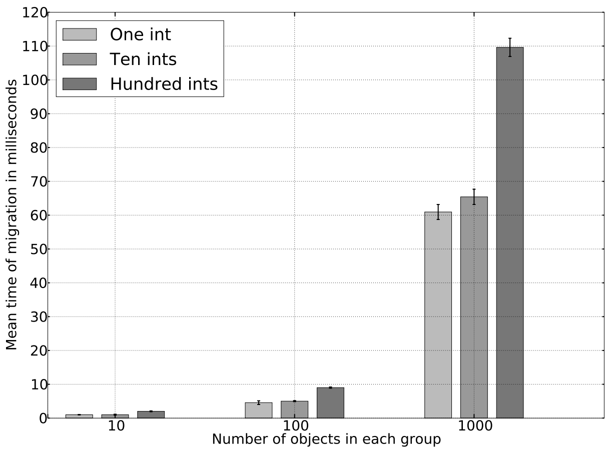


Figure 8 Average group migration time with stdev

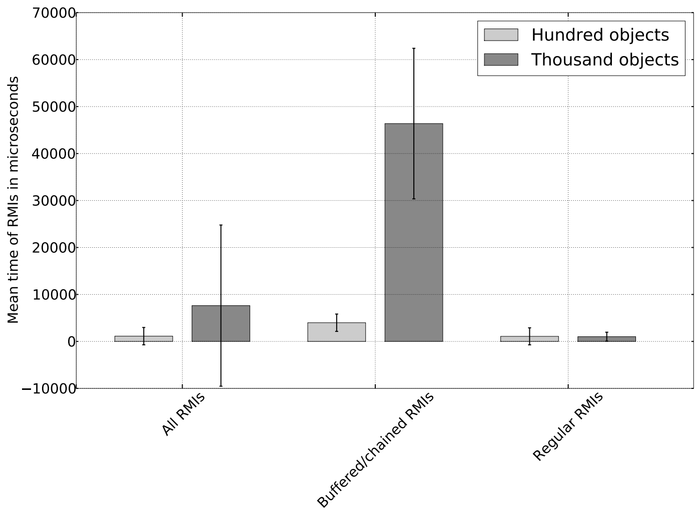


Figure 9 Average time of RMI during migration with stdev

some of the overhead (latency) introduced by having to wait for the forwarding of messages, thus reducing the variance in completion time for an RMI.

5.2 PlanetLab tests

We ran several different tests on the open world-wide research platform PlanetLab, primarily to test the usability of the core node selection algorithm and middleware itself. Also, the PlanetLab nodes that we selected as a server and proxies were those with little or no load in the PlanetLab network.

To test the core node selection algorithm, we ran a test where clients, approximately 120 of them, were initially connected to a main server. As more proxies were gradually added, the core node selection algorithm was activated (see figure 10 for an overview of the PlanetLab nodes included in the experiment).

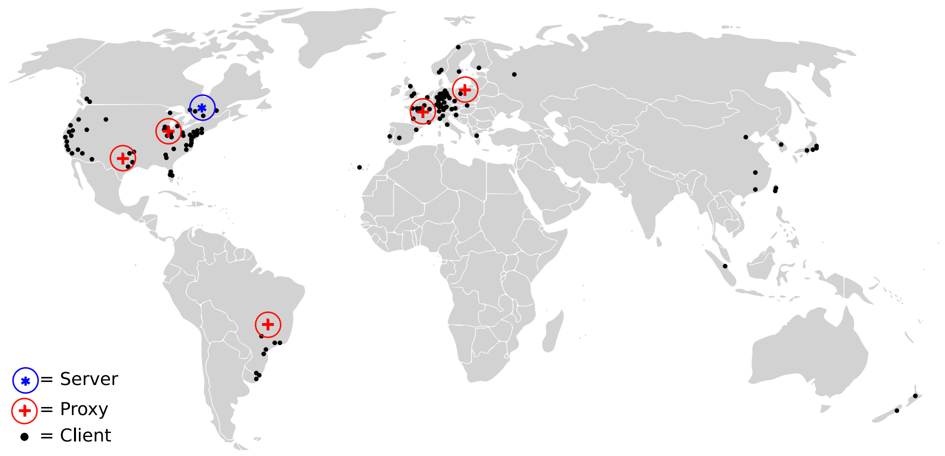


Figure 10 Geographical locations of PlanetLab nodes

The results of this test are summarized in figure 11 and 12. As we can see in figure 11, adding two additional proxies made it possible to reduce the aggregate mean latency of the interacting clients. Adding an additional three proxies further decreased the aggregate mean latency, but the gain was not as great. It is to be expected that the reduction does not necessarily improve greatly with a large number of added proxies, as there is always variance in the latencies of the connected clients, but it is clear that the gaming experience can be improved by the introduction of a small number of proxies. The crossover at the far left of figure 11 shows how a small number of connections benefit (when only a single server is available), while many suffer. By introducing a small number of proxies, performing a core-selection and migrating the state and interacting players to this proxy (if it is able to improve the aggregate latency of the players) the number of connections that benefit is reduced, but correspondingly the number of connections that suffer is also reduced.

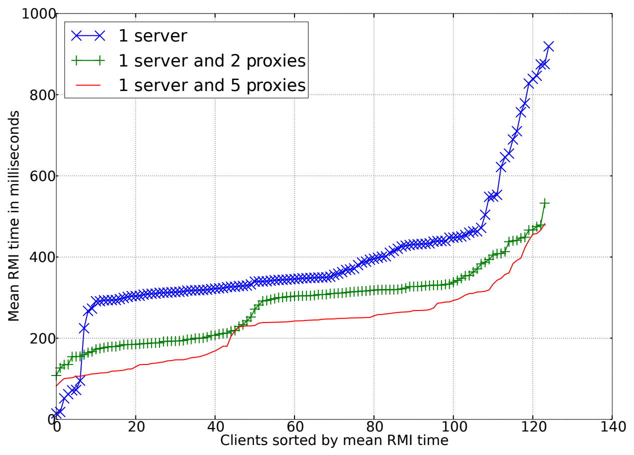


Figure 11 Aggregate mean of RMI

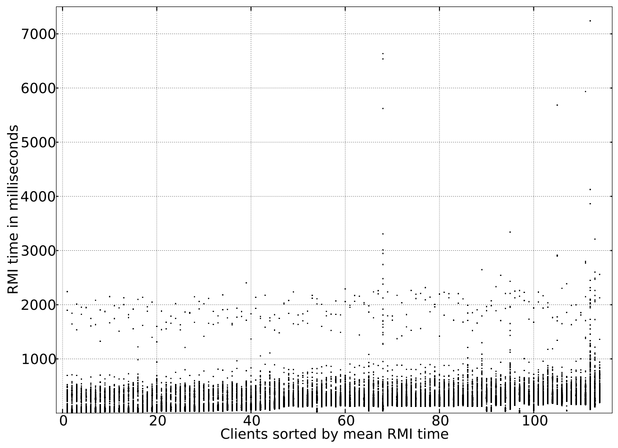


Figure 12 RMI time per client with varying numbers of proxies

In figure 12, we see the completion time of RMIs issued, as proxies are being added. We can see that most of the invocations are well below the 1000ms mark, but that there are at least a couple of invocations that take longer time. These are caused by migrations, as we noted during our micro benchmarks of RMI during migration, as seen in figure 9.

In figure 13, we see the results of a test simulating a small region in a virtual environment with players entering and exiting the region continuously. The setup consisted of 1 server, 5 proxies and approximately 120 clients. The session time of the clients follows an exponential distribution with an average scale of 60 seconds. The core-selection algorithm was activated every 30 seconds, with a migration occurring only if the minimum gain of each client was 2ms. This resulted in a total of 23 migrations, where all, except the first (from the server to a proxy) were between two of the proxies in the setup. The mean migration time was 813ms, excluding one exception, which was 28868ms, and we can see the resulting spikes in the RMI times of the sessions in figure 13. PlanetLab uses virtualization to schedule slices for run-time on the PlanetLab nodes. We believe this spike is a result of such virtualization, where the slice running our application was scheduled out. We did not see such spikes in our earlier PlanetLab tests, but these ran for shorter periods of time. This is one of the problems with running latency sensitive experiments in PlanetLab. An ICMP ping between the two proxies involved did not reveal any latency-related problems, but ICMP ping is not affected by the virtualization (because we use application layer ping, we are affected by virtualization). We ran the test several times with different configurations, but experienced similar spikes each time. We did not experience these in our micro benchmarks of the migration functionality. Apart from these spikes, most of the invocations during this test were well within the 1000ms mark. As such, it should be possible to perform migrations on regions with heavy traffic quite frequently, without adverse side-effects.

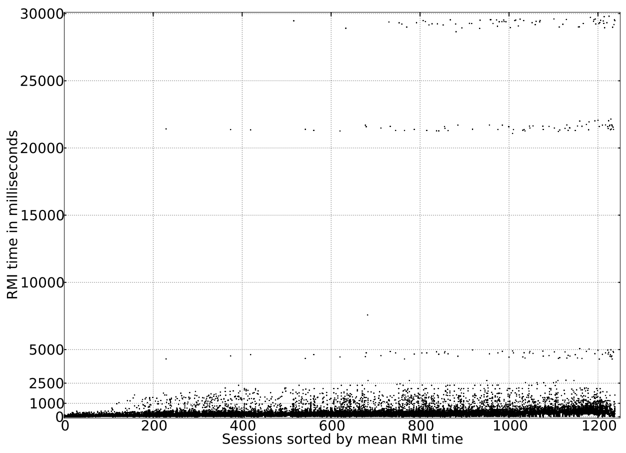


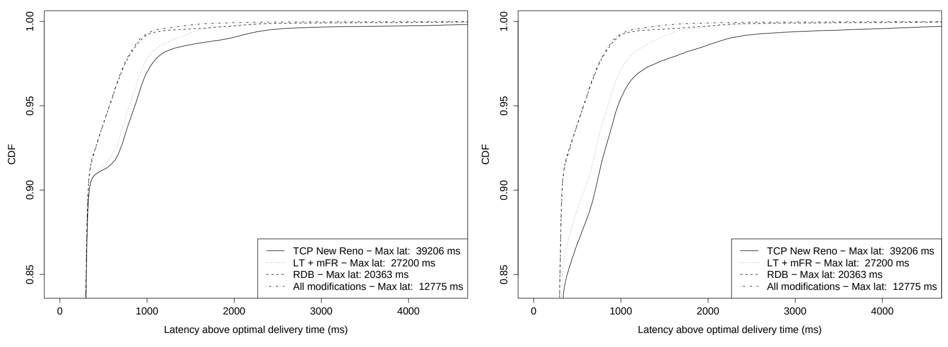
Figure 13 Mean RMI time of sessions

5.3 Latency effect of the TCP thin stream modifications

The modifications to TCP described in section 4 influence both the network monitoring, used for server selection, and the application data delivery. As our TCP modifications are implemented in the Linux kernel, we cannot use PlanetLab as a testbed (as we did in our previous experiments), because we are not permitted to modify the kernel on machines hosted in this network. As such, we had to use machines controlled by us instead.

5.3.1 Application data delivery

To observe the effects of latency at the application layer in a real-world Internet scenario, we replayed packet traces from Anarchy Online (approximately 1 hour of game play from Funcom's servers) between a machine located in Worcester, Massachusetts (USA) and a machine in Oslo (Norway). Both machines were connected to commercial access networks. In figure 14, the results are summarized. The observed path RTT was 116 ms, and the observed loss rate during this test was surprisingly high, 8.86% on average for the TCP New Reno stream. For the transport layer delivery latency (figure 14(a)), 9% of the traffic shows improved latency (0.91 on the y-axis), which is a reflection of the observed 8.86% loss rate, which comes as a result of the aggressive retransmission mechanisms. We also see that 99% of the data at the transport layer level is delivered within 1000 ms (the latency threshold where MMOG users start to get annoyed (Claypool et al., 2006)), while TCP New Reno delivers 97% within 1000 ms. On the application layer (figure 14(b)), where TCP's in-order delivery is accounted for, about 17% of the data is delivered faster using the modifications. With the modifications, about 99% was delivered within 1000 ms, while for TCP New Reno the corresponding percentage was 95%. Finally, the worst-case latencies were also greatly reduced, to about 1/3 of the New Reno results (39.2s versus 12.8s). The plots show that all mechanisms improve data delivery time when loss occurs, but RDB, being more



(a) CDF: transport-layer latency. (b) CDF: application-layer latency.

Figure 14 Anarchy Online packet trace replayed (Massachusetts - Oslo).

aggressive, is most efficient. Thus, our TCP modifications reduce the loss recovery experienced at the application level greatly, i.e., directly influencing the users' game experiences.

5.3.2 Server selection

Server selection is an important facet of Ginnungagap. To make the best possible decision of whether to migrate game state (and the corresponding players) or not, it is important to have minimal variance in the gathered latencies. As such, we ran a test to determine the effect our TCP thin stream modifications would have on the variance of the gathered latencies, and subsequently the accuracy of the core selection algorithm. The results of this test are summarized in figure 15 (the source code of the test application is available for download at http://simula.no/research/networks/software/).

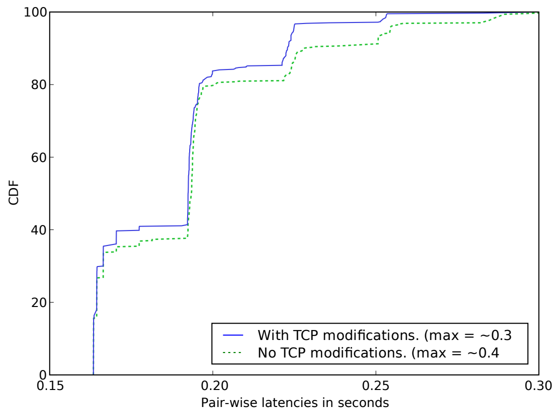


Figure 15 CDF of pair-wise latencies after core selection

For this test, we connected 20 clients to a server, where the clients measured their latencies using an application layer ping. For realistic results, we used *netem* to induce additional packet loss and latency on the client and server connections.

As reflected in section 5.3.1, we can experience peak loss rates of 9%. Thus, we used 1 and 4 percent (average) packet loss respectively for the server and client, and a latency of 70-120ms (randomized per connection and test). In our tests, the clients reported the average from 10 application layer measurements (see figure 15). The server then performed core selection using these measurements as input, and as such, the figure reflects the lowest pair-wise latency of the clients to the server (see section 3.1 for further details). The test ran in two iterations, one with TCP thin stream modifications enabled, and one where they were disabled.

From figure 15, we observe that the thin-stream modifications improve the decision of the core selection algorithm, because a greater percentage of the measurements are affected less by the presence of packet loss. This is so, because inaccuracies can only take the form of overestimated latency, which occurs when the retransmission mechanisms of unmodified TCP fail to recover lost packets quickly because packets rates are too low. The resilience to packet loss observed when the TCP modifications are enabled is thus a result of the aggressive retransmission strategies of the TCP thin stream modifications.

Basing a core selection decision on measurements gathered over time is preferential to single measurements, as multiple measurements converge towards the player's "real" latency. However, the scalability of such an approach is uncertain. One possibility is to have the server read the RTT value from TCP, based on in-game communication (state updates etc.), though in cases where no communication channel is open to (potential) proxies this problem has to be resolved in another way, such as active application layer probing.

6 Discussion

Due to the distributed name service resolving references to remote objects, migration can become a time-consuming task. An object that is frequently migrated creates trails of object references at the nodes it visits. We handle this scenario partially by embedding information about the emanating node in a message that is forwarded, so the reply can be sent directly to the calling node.

Another concern is related to the side-effects of migrating game-state. If migration is not performed transparently, we might adversely affect interacting players. In Nelson et al. (2005), it is demonstrated how an active application is moved from one virtual machine to another, with minimal perceived impact to the user's interaction with the application.

Partial failures occur when a node in a distributed system becomes unavailable, effectively rendering the objects managed by it inaccessible. The current framework does not accommodate for partial failures, but there are ways to minimize the repercussions of these incidents. One possibility is to distribute multiple copies of an object in the system, and utilize a peer-to-peer based look-up system such as Chord or Tapestry, to locate the object. However, the drawback of utilizing such an approach is the overhead introduced by having to keep the multiple instances of an object synchronized.

Core selection and migration require resources in the form of processing power and network messages when activated, as such, policies should govern their activation. Core-selection can be performed on the basis of churn (in the form of players joining/leaving), time of day, player arrival rate, history (a preemptive approach, where known situations, such as battles, trigger the activation) and server-load. Migration can be performed on the basis of threshold of aggregate latency, number of players, packet loss, and server load. In addition to activation policies, it might also be useful to define cool-down periods for core-selection and migration, to avoid thrashing in the system.

To this effect, accurate measurements of the player latencies to the available servers and proxies are instrumental for making correct decisions. These measurements can, for instance, be retrieved from packets containing game state, but typically a client interacts with only one server at a time. And even in this case, the player may be vulnerable to the packet loss present in the Internet, and the inherent weaknesses of TCP as a transport protocol for thin streams. One possibility is to send a train of packets, and measure the mean of these. This, however, is costly, and the scalability of such a solution for a large number of players, such as we commonly witness in MMOGs, is uncertain. As such, there is much to be gained by utilizing thin stream modifications to TCP, which are able to accommodate for the packet loss in the Internet for thin streams.

Core selection depends on full knowledge of the network, which we can obtain by observing TCP latency. Latency estimation techniques provide an alternative, and in (Beskow et al., 2009b; Vik et al., 2009), we examined the feasibility of using estimated latencies for situations when real-time measurements are unable to scale to the number of interacting players. These active probing techniques generate an amount of traffic that has roughly the same intensity as a modern game's traffic itself. Since the TCP thinstream modifications minimize the variance in the RTT measurements available from the TCP connection structs we have considered using such a passive probing approach, based on the TCP connections.

The proposed framework is tested in a client-server setting, but in general, the framework should also be useful in a peer-to-peer scenario trying to dynamically select a server among the peers. Thus, as the peers join and leave the game, the selection of peers controlling the game could be performed using our middleware.

The implemented thin stream TCP modifications may increase the number of spurious transmissions (Petlund, 2009). However, the reduction in latency, both for application data delivery and latency monitoring for core selection, justifies the need to suppress occasional spurious retransmissions.

7 Conclusion

High latency may be devastating for the game play experience. An important factor for world-spanning games, such as MMOGs, lies in the diversity of its user base where there will, at any point in time, be a number of players connected from different physical locations. The large challenge is that the group of players in the game or in the game region is dynamic, and using a statically selected server may therefore give good results in one moment but poor in the next.

Beskow et al. (2009a) describe the implementation of this middleware and presented experimental results showing the performance gains from in-house lab experiments and running a simple game on PlanetLab. Here, we have expanded on these ideas and tested the effects of our TCP thin stream modifications for improving the resilience of latency measurements and for improving the throughput of game state in the network. As a result, players receive data sooner, and the core selection algorithm is able to make a better decision with regard to the optimal placement of game state.

As future work, we would like to review the possibility of optimizing the placement for smaller, highly coupled groups of interacting players within a region. This can be achieved by applying graph partitioning algorithms or similar techniques for isolating these groups.

Bibliography

ARMITAGE, G. (2008a) Client-Side adaptive search optimisation for online game server discovery. In *Lecture notes in computer science 4982*, pp. 494-505.

ARMITAGE, G. (2008b) Optimising online fps game server discovery through clustering servers by origin autonomous system. In *Proceedings of International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'08), Braunschweig, Germany*, pp. 3-8.

BESKOW, P. (2007) Migration of objects in a middleware for distributed real-time interatctive applications. Master's thesis, Department of Informatics, University of Oslo, Norway.

BESKOW, P., VIK, K.-H., GRIWODZ, C., AND HALVORSEN, P. (2008) Latency reduction by dynamic core selection and partial migration of game state. In *Proceedings of NetGames'08, Worcester, MA, USA*, pp. 79-84.

BESKOW, P., ERIKSTAD, G., GRIWODZ, C., AND HALVORSEN, P. (2009a) Evaluating Ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction. In *Proceedings of NetGames'09, Paris, France*, pp. 1-6.

BESKOW, P., VIK, K.-H., HALVORSEN, P., AND GRIWODZ, C. (2009b) The partial migration of game state and dynamic server selection to reduce latency. In *Springer's Multimedia Tools and Applications Special Issue on Massively Multiuser Online Gaming Systems and Applications 45*, 1-3, pp. 83-107.

BRUN, J., SAFAEI, F., AND BOUSTEAD, P. (2006) Server topology considerations in online games. In *Proceedings of NetGames'06, New York, NY, USA*, p. 26.

CHAMBERS, C., CHANG FENG, W., CHI FENG, W., AND SAHA, D. (2003) A geographic redirection service for on-line games. In *Proceedings of ACM International Conference on Multimedia, Berkeley, CA, USA*, pp. 227-230.

CHAMBERS, C., CHANG FENG, W., SAHU, S., AND SAHA, D. (2005) Measurement-based characterization of a collection of on-line games. In *Proceedings of ACM SIGCOMM Workshop on Internet measurement, Berkeley, CA, USA*, pp. 1-14.

chang Feng, W., Chang, F., chi Feng, W., and Walpole, J. (2002) Provisioning on-line games: a traffic analysis of a busy Counter-strike server. In *Proceedings of ACM SIGCOMM Workshop on Internet measurement, Marseille, France*, pp. 151-156.

chang Feng, W., and chi Feng, W. (2003) On the geographic distribution of on-line game servers and players. In *Proceedings of NetGames'03, Redwood City, CA, USA*, pp. 173-179.

Claypool, M. (2005) The effect of latency on user performance in real-time strategy games. In *Elsevier Computer Networks 49*, 1, pp. 52-70.

Claypool, M. (2008) Network characteristics for server selection in online games. In *Proceedings of Multimedia Computing and Networking (MMCN'08), San Jose, CA, USA*, pp. 681808.

Claypool, M., and Claypool, K. (2006) Latency and player actions in online games. In *Communications of the ACM 49*, 11, pp. 40-45.

Dick, M., Wellnitz, O., and Wolf, L. (2005) Analysis of factors affecting players' performance and perception in multiplayer games. In *Proceedings of NetGames'05, Hawthorne, NY, USA*, pp. 1-7.

Erikstad, G. A. (2009) Latency reduction in distributed interactive applications by core node selection and migration. Master's thesis, Department of Informatics, University of Oslo, Norway.

Lee, K., Ko, B., and Calo, S. (2005) Adaptive server selection for large scale interactive online games. In *Computer Networks 49*, 1, pp. 84-102.

Nelson, M., Lim, B., and Hutchins, G. (2005) Fast transparent migration for virtual machines. In *Proceedings of USENIX Annual Technical Conference*, pp. 25-25.

Petlund, A. (2009) *Improving latency for interactive, thin-stream applications over reliable transport.* PhD thesis, Department of Informatics, University of Oslo, Norway.

Petlund, A., Evensen, K., Griwodz, C., and Halvorsen, P. (2008) Improving application layer latency for reliable thin-stream game traffic. In *Proceedings of NetGames'08, Worcester, MA, USA*, pp. 91-96.

Vik, K.-H. (2009) *Group Communication Techniques in Overlay Networks.* PhD thesis, Department of Informatics, University of Oslo, Norway.

Vik, K.-H., Griwodz, C., and Halvorsen, P. (2009) On the influence of latency estimation on dynamic group communication using overlays. In *Proceeding of Multimedia Computing and Networking (MMCN'09), San Jose, CA, USA*, 7253, pp. 725307.