Evaluating Ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction

Paul B. Beskow, Geir A. Erikstad, Pål Halvorsen, Carsten Griwodz
Simula Research Laboratory, Norway Department of Informatics, University of Oslo, Norway
{paulbb, geirerik, paalh, griff}@ifi.uio.no

***Abstract*—Massively multi-player online games (MMOGs) have stringent latency requirements and must support large numbers of concurrent players. To handle these conflicting requirements, it is common to divide the virtual environment into virtual regions, and spawn multiple instances of isolated game areas (known as dungeons or instances) to serve multiple distinct groups of players. As MMOGs attract players from all around the world, it is plausible to disperse these regions and instances on geographically distributed servers. Core selection algorithms can then be applied to locate an optimal server for placing a region or instance, based on player latencies. Functionality for migrating objects supports this objective, with a distributed name server ensuring that references to the moved objects are efficiently maintained. As a result, we anticipate a decrease in aggregate latency for the affected players. With Ginnungagap we present the implementation and evaluation of a middleware that combines these ideas and present its feasibility.**

I. Introduction

In online games, some latency is tolerable [11] as long as it does not exceed the threshold for playability from 100ms to 1000ms depending on the type of game [13], 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 [11]. This occurs if one or more player's connections are comparatively slower, 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, which the server must then recover from. While having minor effects on the outcome of the game, it results in a perceived deterioration to the quality of interaction [14]. As such, *low latency* for *all interacting players* is a prevalent goal.

This goal, however, contrasts other observations of online games, such that the game traffic varies strongly with time and the attractiveness of the individual game [8], [9] and that geographical dispersion of players in an online game depends heavily on the time of day [10]. 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 [4], we proposed the initial design of a middleware supporting migration of partial game-state (where the level of game-state granularity can range from entire virtual regions, to instances, or single objects) to servers which are dynamically selected according to players locations. With GINNUNGAGAP[1], we present an implementation of this middleware. 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. GINNUNGAGAP could also be applied in peer-to-peer scenarios, allowing all interacting parties to act as a server, which would mean a greater distribution of potential servers. We have not explored such a scenario in this paper, as we have focused on MMOGs that use a client-server approach, though we consider it a viable approach. 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 that the overhead of migration and method invocation does not exceed the limits of a good perceived game play.

[1]The source code of the middleware, and logs from the PlanetLab experiments are available for download at http://simula.no/research/networks/software/

II. 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 [17], 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. [7] have looked at how server selection can be

978-1-4244-5604-8/09/$26.00 ©2009 IEEE

optimized for a single client, when given a set of available servers. In a further study, Claypool [12] 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 [1], [2], 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 often limited. In [16], Lee et al. 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 [6] 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.

III. 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 were interacting 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 I and II); **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 **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 III-A).

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 objects state and its identifier. The state 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 [17], 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.

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

Table I
Migration message

Instrumental to the name service is an efficient way of maintaining references to the objects as they are migrated. To accomplish this we have implemented a distributed name service. A name service can be implemented in several ways. Discussed in the context of MMOGs, in [3]. 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, to which 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. Other advantages are 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, will depend 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 with a message that passes through such a chain, such that the reply is sent directly back to the calling node.

A. Core selection

In [18], Knut-Helge Vik 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.

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.

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)

B. Migration

Migration (see figure 1 for an overview) targets are found by core-selection **(1)**, though migration is only performed when latency increases above a predefined threshold. The migration is performed by the `MigrationService` **(2)**, which first sets the state of the migrating object to `MIGRATING` **(3)**. Thus, all incoming RMIs are

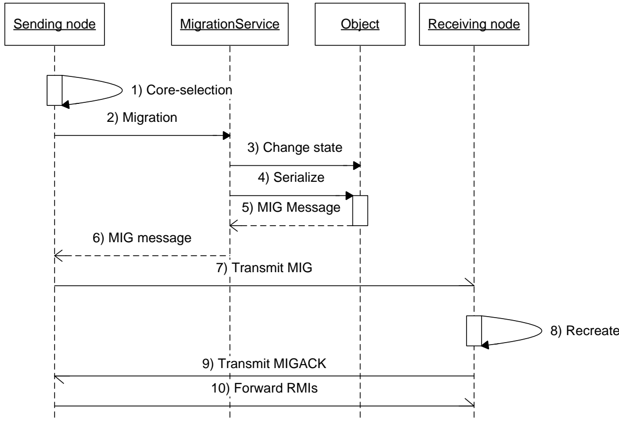

Figure 1. Sequence of a migration

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

Table II
RMI MESSAGE

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.

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

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.

C. Remote Method Invocation

An RMI is the invocation of a method on an object that doesn't reside 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 in to a remote server:

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

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

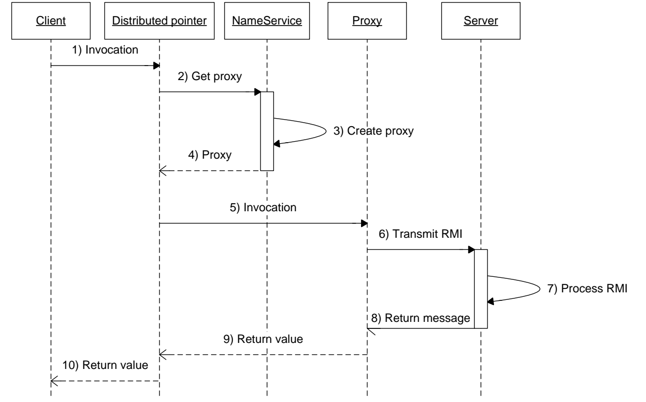

Figure 2. Sequence of a RMI

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 *dist_ptr* **(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 does a look 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 takes over the processing, and creates a message of type RMI (see table II). 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)**.

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. 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)**.

IV. 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 [15] for further results and details).

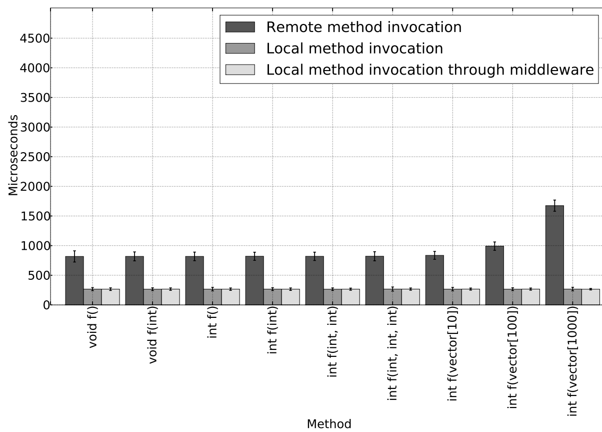

Figure 3. Average time of method invocation with stdev

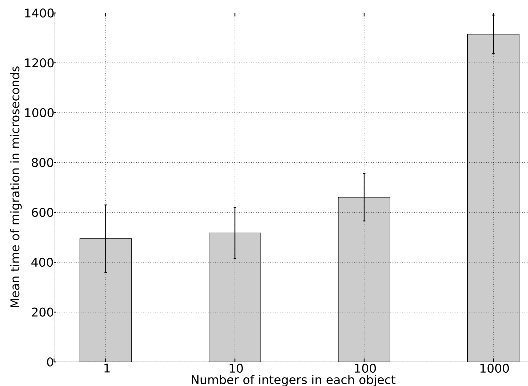

Figure 4. Ave. single object migration time with stdev

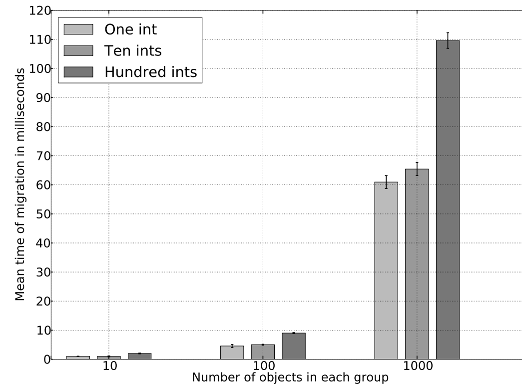

Figure 5. Ave. group migration time with stdev

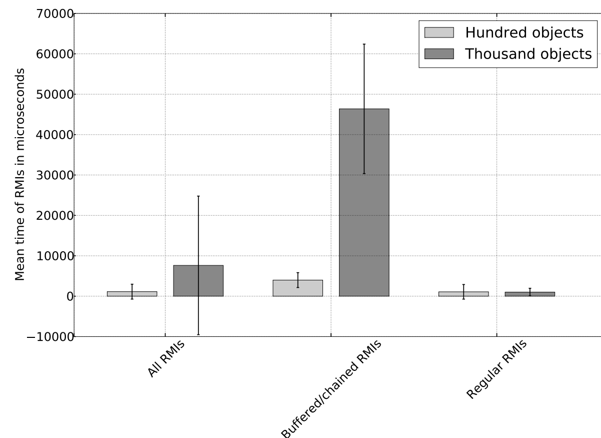

Figure 6. Average time of RMI during migration with stdev

A. Micro benchmarks

The micro benchmarks were performed using a local network. The test machines ran Ubuntu 9.04, and had identical hardware configurations, with an Intel Core 2 Duo E6750 2.66GHz CPU and 2 GB of RAM.

Remote Method Invocation: To test the RMI functionality and overhead we invoked methods with a number of different combinations of return values and parameters. The results of these tests are shown in figure 3, and each method was invoked 10000 times. As expected, there is an overhead introduced by a RMI, when compared to a local invocation, 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 RMI invocations of vectors of integers, 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 conclusion, 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 4. 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 5. 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 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, the results of this test are presented in figure 6. 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. 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 could then be ready to be processed as soon as the migration was completed at the receiving node, thus not having to wait for the sending node to receive the `MIGACK` message and forward the messages, effectively removing some of the overhead of the latency between the sender and receiver.

B. Planetlab tests

PlanetLab (PL) is a distributed research network consisting of nodes across the world. We ran several different tests, primarily to test the usability of the core node selection algorithm and middleware itself. Also, the PL nodes that we selected as a server and proxies were those with little or no load in the PL network.

To test the core node selection algorithm, we ran a test where clients, approximately 120 of them, were initially connected to a

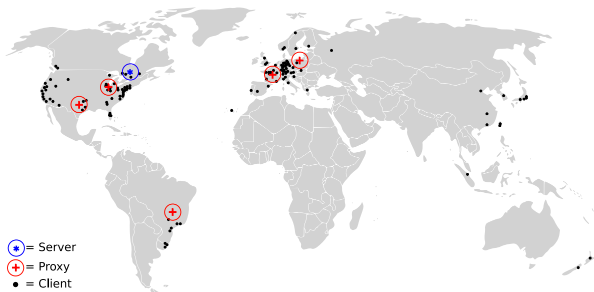

Figure 7. Geographical locations PL nodes

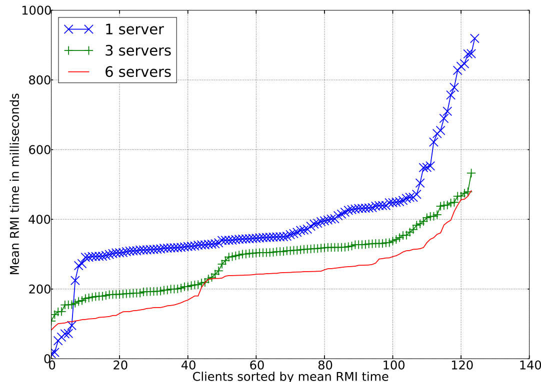

Figure 8. Aggregate mean of RMI

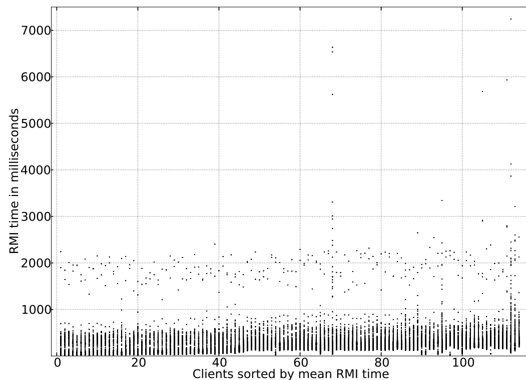

Figure 9. RMI time per client with varying numbers of proxies

main server. As more proxies were gradually added, the core node selection algorithm was activated (see figure 7 for an overview of the PL nodes included in the experiment). The results of this test are summarized in figure 8 and 9. As we can see in figure 8, adding additional proxies made it possible to reduce the aggregate mean latency of the interacting clients. Moving up to four 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.

In figure 9, 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 6.

In figure 10, we see the results of a test simulating a small

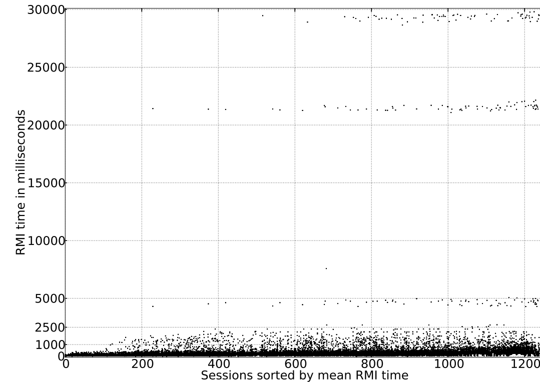

Figure 10. Mean RMI time of sessions

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 a 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 on 28868ms, and we can see the resulting spikes in the RMI times of the sessions in figure 10. PL uses virtualization to schedule slices for run-time on the PL nodes. We believe this spike is a result of such virtualization, where the slice running our application has been unscheduled. We did not see such spikes in our earlier PL tests, but these ran for shorter periods of time. This is one of the problems with running latency sensitive experiments in PL. A ping between the two proxies involved did not reveal any problems latency wise, but is not affected by such 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 either. 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.

V. 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 will create trails of object references at the nodes it visits. We partially handle this scenario by embedding information about the emanating node in a message that is forward, 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 [17], Nelson et al demonstrate 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. Though this introduces the overhead of having to keep multiple objects up-to-date.

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 based (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.

Core selection is dependent on full knowledge of the network, but obtaining latency measurements through active probing is not a scalable solution. Latency estimation techniques provide an alternative, and in [5], [19], we examined the feasibility of using estimated latencies for situations when real-time measurements are unable to scale to the number of interacting players. We have not applied these techniques in this paper, as we were able to gather live measurements during our tests.

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.

VI. 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.

In [4], we proposed a middleware providing dynamic server selection and migration of game state to the new server. Here, we have described 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.

References

[1] G. Armitage. Client-Side Adaptive Search Optimisation for Online Game Server Discovery. *Lecture notes in computer science*, 4982:494, 2008.

[2] G. Armitage. Optimising online fps game server discovery through clustering servers by origin autonomous system. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, May 2008.

[3] P. Beskow. Migration of objects in a middleware for distributed real-time interatctive applications. Master's thesis, Department of Informatics, University of Oslo, Norway, May 2007.

[4] P. Beskow, K.-H. Vik, C. Griwodz, and P. Halvorsen. Latency reduction by dynamic core selection and partial migration of game state. *Proceedings of NetGames'08, Worcester, MA, USA*, oct 2008.

[5] P. Beskow, K.-H. Vik, P. Halvorsen, and C. Griwodz. The partial migration of game state and dynamic server selection to reduce latency. *Springer's Multimedia Tools and Applications Specia l Issue on Massively Multiuser Online Gaming Systems and Applications*, 2009.

[6] J. Brun, F. Safaei, and P. Boustead. Server topology considerations in online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 26, New York, NY, USA, 2006. ACM.

[7] C. Chambers, W. chang Feng, W. chi Feng, and D. Saha. A geographic redirection service for on-line games. *In Proceedings of the eleventh ACM international conference on Multimedia, Berkeley, CA, USA*, pages 227–230, November 2003.

[8] C. Chambers, W. chang Feng, S. Sahu, and D. Saha. Measurement-based characterization of a collection of on-line games. *In the Proceedings of the 5th ACM SIGCOMM Workshop on Internet measurement, Berkeley, CA, USA*, pages 1–14, October 2005.

[9] W. chang Feng, F. Chang, W. chi Feng, and J. Walpole. Provisioning on-line games: a traffic analysis of a busy Counter-strike server. *In the Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement, Marseille, France*, pages 151–156, November 2002.

[10] W. chang Feng and W. chi Feng. On the geographic distribution of on-line game servers and players. *In the Proceedings of NetGames'03, Redwood City, California, USA*, pages 173–179, May 2003.

[11] M. Claypool. The effect of latency on user performance in real-time strategy games. *Elsevier Computer Networks*, 49(1):52–70, Sept. 2005.

[12] M. Claypool. Network characteristics for server selection in online games. *In Proceedings of the fifteenth Annual Multimedia Computing and Networking (MMCN'08), San Jose, CA, USA*, 6818:681808, January 2008.

[13] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, Nov. 2005.

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

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

[16] K. Lee, B. Ko, and S. Calo. Adaptive server selection for large scale interactive online games. *Computer Networks*, 49(1):84–102, 2005.

[17] M. Nelson, B. Lim, and G. Hutchins. Fast transparent migration for virtual machines. *Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference table of contents*, pages 25–25, 2005.

[18] K.-H. Vik. *Group Communication Techniques in Overlay Networks (submitted)*. PhD thesis, Department of Informatics, University of Oslo, Norway, Dec. 2008.

[19] K.-H. Vik, C. Griwodz, and P. Halvorsen. On the influence of latency estimation on dynamic group communication using overlays. In *Multimedia Computing and Networking (MMCN), San Jose, CA, USA*, January 2009.