Hydra: A Massively-Multiplayer Peer-to-Peer Architecture for the Game Developer

Luther Chan, James Yong, Jiaqiang Bai, Ben Leong, Raymond Tan
National University of Singapore

ABSTRACT

We present the design and implementation of *Hydra*, a peer-to-peer architecture for massively-multiplayer online games. By supporting a novel augmented server-client programming model with a protocol that guarantees consistency in the messages committed when nodes fail, existing game developers can realize the benefits of a peer-to-peer architecture without the burden of handling the complexities associated with network churn. Our key contribution is the development of a programming interface that is intuitive and easy to use, and that can be supported transparently at the network layer. We have implemented a prototype of Hydra and we demonstrate that our proposed architecture is practical by developing two games under the Hydra framework: a simple "capture the flag" tank game and a squad-based real-time strategy (RTS) game. Our experience in developing these games suggests that our proposed programming model is suitable for game development. Our preliminary experiments also show that Hydra imposes only a small message overhead and is thus scalable.

1. INTRODUCTION

Massively-multiplayer online games have been a huge commercial success in recent years. Existing deployments of such games have been built on a server-client architecture, even as some have claimed that such centralized architectures are inherently unscalable [20]. This claim has been shown to be untrue by Blizzard's World of Warcraft, which has some 8.5 million players globally as at March 2007, approximately 500,000 players online at any one time, and servers supporting several thousand players simultaneously [6].

Nevertheless, we believe that it is still worthwhile to develop a peer-to-peer architecture for such games because by exploiting the bandwidth and computational capabilities of the client hosts, their deployment costs can be significantly reduced and in some cases, their performance improved with reduced latencies. For this reason, there have been a large number of proposals of peer-to-peer architectures for networked games [18, 4, 16, 23, 15, 5].

Our key insight is that the server-client model is well-understood and works well. Therefore, instead of forcing game developers to have to think differently when developing their games for a peer-to-peer environment, our approach is to support the server-client model transparently, thereby insulating game developers from the complexities of network churn in such an environment.

In this light, we adopt a new approach in *Hydra*, our network architecture for peer-to-peer massively-multiplayer games. Instead of addressing issues of efficient event delivery and multicast overlays, Hydra seeks to provide a simple augmented server-client programming model to the game developer and implements a set of protocols to support the required interface. We hide the complexities associated with the recovery from node failures (i) by imposing some conditions on how the game application should process incoming messages, (ii) by having the game application provide an interface to the network layer for the checkpointing and restoration of application game state, and (iii) by providing basic guarantees on consistency in message delivery without using locks [1] or concurrency control [10].

Also, we understand that it is probably infeasible to deploy commercial games on a purely peer-to-peer architecture because for all practical purposes, such games will require support for billing and persistent storage. We believe however that this is not a concern because it is not difficult to implement such functionality in a separate centralized system and have the basic peer-to-peer game integrate with these functions into a hybrid architecture [20].

The key contribution of our work is in the development of a programming interface that is intuitive and easy to work with. We demonstrate that our proposed architecture is practical by implementing two games over Hydra: a simple "capture the flag" tank game and a squad-based real-time strategy (RTS) game. Our experience in developing these games suggests that our proposed programming model is suitable for the game development. Our preliminary experiments show that Hydra imposes only a small message overhead and is thus scalable.

The remainder of this paper is organized as follows: in Section 2, we provide a review of existing and related work. In Section 3, we describe the programming model and interface of the Hydra architecture and in Section 4, we describe how we support that interface. Our evaluation results are presented in Section 5. Finally, we discuss future work in Section 6 and conclude in Section 7.

2. RELATED WORK

There have been a number of proposals for supporting peer-to-peer networked games and, in general, the focus of these proposals has been on the synchronization of game state across hosts, typically using a DHT substrate. Knutsson et al. proposed the use of the Scribe publish-subscribe system [7] (based on the Pastry DHT [21]) to support a simple model of a massively-multiplayer game [18]. Bharambe et al. likewise developed Mercury [4], a DHT-like routing protocol for multi-attribute range queries. They claim that because massively-multiplayer games have less stringent latency re-

NetGames '07 September 19–20, 2007, Melbourne, Australia

quirements than first-person shooters (FPS), the multi-hop latency arising from the underlying DHT lookups is acceptable and they validate their system through simulations. There has also been a number of other DHT-based proposals in the literature [16, 23, 15].

Bharambe et al. subsequently came to the conclusion that "lookups in DHTs can be too slow for finding required replicas" for interactive games like Quake II and thus developed Colyseus, a scheme to perform speculative fetching based on locality and predictability in data access patterns [5]. They evaluated Colyseus by running experiments with a modified Quake II FPS engine and demonstrated that the load per node is lower than that for a centralized server architecture.

In addition to peer-to-peer approaches, there are a number of other distributed architectures for peer-to-peer games. Cronin et al. proposed a *mirrored architecture*, which is a hybrid peer-to-peer and server-client architecture [8]. Rhalibi et al. proposed a similar architecture implemented on JXTA where a DHT is used as the underlying communication substrate and validated it by implementing a game called "Time Prisoners" [20]. Assiotis et al. also proposed an architecture with multiple servers each responsible for one region of the virtual game world [1]. The focus on their work however is in achieving efficient event delivery, replication consistency and liveness, and not on fault tolerance. Our work differs from these works because our focus is on developing a programming interface that can be supported transparently at the network layer, instead of designing a network architecture for efficiency or reliability. Optimal Grid [17], is a project that attempts to achieve a similar goal, but in a Grid environment.

There are several proposed architectures for *distributed virtual environments* (DVEs) and *collaborative virtual environments* (CVEs). DVEs, like DIVE [11] and MASSIVE [13], were designed mainly for local use and supported only a small number of participants. CVEs differ from DVEs in that they focus on the collaboration between avatars [3, 14]. None of these proposals address the specific needs of modern multiplayer games and they typically assume the availability of IP multicast, like the distributed multiplayer game MiMaze [19].

3. PROGRAMMING MODEL

Existing networked games tend to be locality-based, which means that the virtual game world can be divided naturally into regions. In fact, Hydra assumes that the game world is divided into disjoint regions that are each managed by a single server. The client will connect to the respective server that manages the region of the virtual world in which the player's avatar is currently residing. Clients can only interact with other clients that are connected to the same server. For games that require a smooth transition between two regions, it is the responsibility of the game application to manage the transition.

Hydra only delivers messages to the servers for the clients, it does not actually manage the connections; all connections are managed at the application layer by the game servers. The movement of a player's avatar from the region managed by one server to that managed by another requires either the client to transfer its connection from one server to the other, or to establish simultaneous connections to both servers.

In this section, we define the interfaces for the game clients and servers, and the programming model that a game server is expected to conform to in order to preserve correctness.

3.1 Game Client Interface

The network interface is relatively straightforward for the game client: it can send either reliable or unreliable messages over UDP to a game server, which is specified by a unique identifier and not an IP address. Unreliable messages are each sent once on a best-effort basis, while reliable messages are retransmitted until they are successfully delivered. Hydra provides no ordering guarantees for the reliable messages, but ensures that message ordering is preserved for unreliable messages. Unreliable messages that end up at the server after later messages have been executed are simply discarded. We are considering adding support for a blocking RPC-like interface for the client.

3.2 Game Server Interface

The game server is implemented on the assumption that it is solely responsible for a region of the virtual world. Messages are delivered by Hydra to the server in a priority queue that sorts incoming messages from the clients in a partial ordering. The messages are sorted in ascending order according to a discrete timestamp, called a *tick*, assigned by Hydra. The server will pop messages off the queue and process them, and the manner in which the messages are processed must adhere to three conditions in order for Hydra to guarantee that the game application is consistent following the recovery of a failed node.

I. Simulation Pause. The Hydra system maintains a current tick count that the game application may access with `getTick()`. The server should pause its simulation if the tick count of the messages at the top of the queue is larger than the current tick count, i.e. implement the following pseudocode in its simulation loop:

```
tick now = getTick();
while (now < currentTick) {
    yield();
    now = getTick();
}
currentTick = now;
```

Process messages from queue with tick $t \leq$ `now`.

In order words, the simulation must process incoming messages in the main queue no faster than the current tick. If the current tick does not change, the game simulation will be paused indefinitely.

II. Simulation Determinism. The network interface for the game server is similar to that for the game client: a server may also send either reliable or unreliable messages to its clients over UDP. However, the messages that a server sends to its clients should also be synchronized with the simulation at the server. The assumption is that the server will process incoming messages in batches according to their ticks and that messages will only be sent at boundary between ticks, i.e. if a message is sent by the server after all the messages with tick t have been popped off the queue and all the messages with tick $t + 1$ are still on the queue (and hence not yet processed), then the outgoing message is the message corresponding to the simulation state after all the incoming messages with tick t have been processed.

The simulation should also be deterministic, i.e. if the virtual game world is in a state $\mathcal{S}$ before processing a batch of the messages with tick t, the new state of the game world $\mathcal{S}'$ and all the messages that are sent after processing the batch of messages should be deterministic and dependent only on the contents of the processed messages with tick t. While most games will require some form of randomization, this requirement for simulation determinism can easily met with the use of pseudo-random number generators and pre-determined seeds.

III. Load/Save Interface. The game server must also support a method to checkpoint and save its internal state to an output stream and a corresponding method to initialize its state from an input stream. While this requirement may seem a little out of place for a

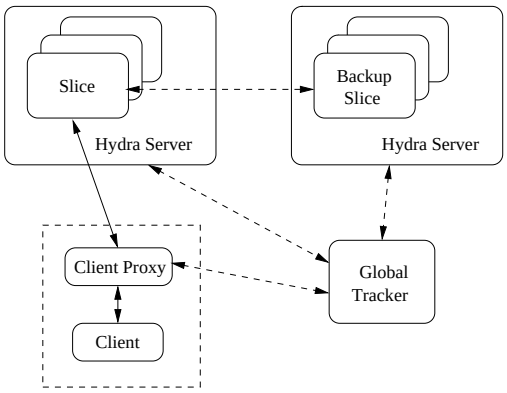

Figure 1: Hydra system architecture.

persistent game world, many existing single-player games do support some form of load/save functionality hence this requirement is not unreasonable. Like outgoing messages, a checkpoint is to be taken at boundary points where all the messages with tick t have been processed and before any with tick $t + 1$ are processed.

If the simulation application fails to adhere to these conventions, Hydra cannot guarantee that in the event of a failover, the state of the game will necessarily be consistent. Our experience in developing games under the Hydra framework has convinced us that the three conditions described above are unobtrusive and can easily be satisfied by a game developer.

4. SYSTEM ARCHITECTURE

Due to space constraints, we are only able to provide a brief overview of the Hydra system architecture in this section. While Hydra is a peer-to-peer distributed system and each Hydra node has both client and server functionality, the client and server modules are described separately for clarity. The various components of the Hydra system are shown in Figure 1.

The client module of the Hydra system consists of two components: the game client and the *client proxy*. The game client is a typical client that is oblivious to the Hydra system, except for the fact that instead of sending messages directly to the network, it sends messages through the proxy; similarly, instead of reading directly from the network, incoming messages for the game client are offered by the client proxy in a priority queue similar to that at the server.

The game servers, as described in Section 3.2, are implemented as components called *slices*, each managing the game state for a specific region of the virtual game world. Since it is possible for a single host to manage several regions of the virtual game world simultaneously when the number of clients is small, multiple slices may be contained in a component called the *Hydra server*. Each Hydra node consists of a client module and a Hydra server.

Like all peer-to-peer systems, there is a rendezvous node that is queried whenever a new node joins the system. This component is called the *global tracker*. In addition to acting as the point of contact for new nodes, the global tracker keeps track of the servers and slices in the system. Since slices are addressed by unique slice identifiers (*sliceIDs*), the global tracker is queried when a client proxy or server needs to determine the IP address and port corresponding to the server(s) hosting slices for a specific sliceID. We have currently implemented the global tracker as a simple server application. The global tracker can also be implemented as a distributed system with a DHT [21].

4.1 Hydra in Operation

To recover from node failures, each slice is replicated a number of times (determined by the degree of redundancy required). One copy (usually the original) is the *primary slice*, while the remaining replicas are called *backup slices*. The primary slice together with the set of backup slices is referred to as a *slice instance*.

Instead of sending messages directly to the network, a game client will forward its messages to a client proxy. The client proxy maintains a tick count ("message tick") that is incremented at regular intervals (currently $150\,ms$ in our implementation) and tags outgoing messages with the message tick. A tagged outgoing reliable message is multicast to all the servers hosting the primary and backup slices while an outgoing unreliable message is only sent to the primary slice. The client proxy resolves the IP addresses and ports of the hosting Hydra servers by querying the global tracker.

Each slice also maintains a tick count ("slice tick"). Primary slices will automatically increment their tick counts at a rate equal to that for the proxy ticks. Backup slices do not increment their tick counts automatically.

When a Hydra server receives a message from a client proxy, it will route the message to the appropriate slice. How a slice handles a message depends on (i) whether it is a primary or backup slice; (ii) the type of the incoming message (reliable/unreliable) and (iii) its current tick count compared to the message tick.

Primary Slice. If the receiving slice is a primary slice and the message tick is greater than or equal to the slice tick, the message is added to the priority queue for the game application. If the message tick is less than the slice tick and if it is unreliable, the message is discarded; if the message is reliable, its tick will be updated to the current slice tick and added to the queue. All messages (reliable or unreliable) that are added to the queue are also forwarded to the backup slices *in order and reliably* (i.e. retransmitted if necessary). These forwarded messages also contain the previous slice tick (i.e. current slice tick $-$ 1).

Backup Slice. If the receiving slice for a reliable message from a client is a backup slice, the message is put into a separate backup queue instead of the priority queue for the game application.

When a backup slice receives a forwarded message from the primary slice, the message is put into the game application priority queue. Since the forwarded messages contain information on the last slice tick of the primary slice, the current tick of the backup slice is updated accordingly. In effect, the backup slices do not need to increment their local slice ticks automatically because they are "clocked" by the primary slice. Reliable messages in the backup queue are flushed when a corresponding copy is received from the primary slice.

Game Server Messages. While Hydra differentiates between the primary and backup slices, all slices behave as if they are the sole server for their instance and they will generate outgoing messages for the clients accordingly. Since messages are forwarded through the Hydra server, the key difference is that outgoing messages from the primary slice will be forwarded to the clients while the outgoing messages from the backup slices will be discarded.

Like clients, the slices may also send either reliable or unreliable messages over UDP. Because of simulation determinism, the primary and backup slices will all generate the same outgoing messages with identical identifiers. The uniqueness of the identifier allows clients to determine if a received message is a duplicate during the recovery process for node failures (which occasionally generates duplicate server messages).

Synchronization. To synchronize messages between a client proxy and a primary slice, they separately maintain a tick count that increments at regular intervals (currently $150\,ms$). The client

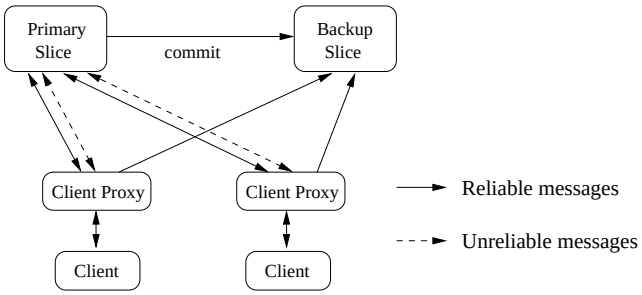

Figure 2: Flow of messages from client to primary and backup slices.

proxy synchronizes with the primary slice when it first joins. To synchronize with a primary slice, a client proxy sends a sync message containing the timestamp of the system clock. Upon receiving this sync message, the primary slice responds immediately with a message that contains the received timestamp and its current tick.

When the client proxy receives the response message, it is able to estimate the round trip time (RTT) to the primary slice. The time taken for a message from the client proxy to reach the primary slice is assumed to be half the RTT. Since the tick interval is known, the client proxy adjusts its message tick accordingly so that messages will arrive "just in time" at the primary slice (i.e. have a message tick that is one larger than the slice tick).

If the primary slice receives a message from a client proxy with a message tick that is smaller or significantly larger than its current tick, it will inform the client proxy to synchronize again.

4.2 Handling Node Failures

The details of the failover protocol are somewhat involved and due to space constraints, we provide only a brief overview in this section. The general principle is that each slice instance, comprising of a primary slice and its set of backup slices, is responsible for ensuring that it is replicated appropriately and handles the failure of nodes independently from other slice instances. When a primary slice starts up, it is configured to generate a pre-determined number of backup slices.

Creation of Backup Slice. Backup slices are created by the primary slice in a few steps. First, the primary slice queries the global tracker to obtain a list of available servers. Then it contacts an available server and requests for a backup slice of the appropriate type to be created. Once this is done, the associated client proxies are informed to forward packets to the newly created (empty) slice. It also updates all existing backup slices with the new list of backup slices. Concurrently, the primary slice obtains a checkpoint of the current game state via the load/save interface and sends the initialization data to the newly created backup slice. Once the transfer of the checkpoint data is completed, the backup slice synchronizes with the primary slice so that it will be able to determine the tick of the primary slice. The backup slice also executes the set of received messages from the checkpoint to bring its simulation state up to the current state.

Failure of Backup Slice. Since the primary slice communicates directly with the backup slices periodically via a reliable channel, the failure of a backup slice will soon be detected following a timeout. When a backup slice fails, a primary slice simply creates a new backup slice to replace the failed backup slice.

Failure of Primary Slice. A backup slice expects to receive committed messages from the primary slice periodically and has its tick clocked by the messages sent by the primary slice. Since the backup slice is synchronized with the primary slice, it is able to determine the tick of the primary slice. When the tick of the backup slice is too far behind the tick of the primary slice, it will check with the clients if the primary slice has timed out. If the majority of the clients respond that the primary slice has timed out, the primary slice is declared to have failed. The leader election protocol is started and one of the backup slices takes over as the primary slice and uses the messages in its backup queue to bring its simulation state up to date. The simulation resumes at the tick the failed primary slice would be if it had not failed. While the unreliable messages sent by the clients during the failover period will be lost, the reliable messages in the backup queue will still be committed.

Failures of the game clients are handled by the game application and not Hydra, though there is some state associated with the client proxies that is maintained at the servers and Hydra will perform some basic house keeping. The handling of client failures can be left to the game developers because it is an issue that they already have to deal with at present and it almost always requires some action at the application layer.

4.3 Load Balancing

We have only implemented the basic Hydra architecture and failover protocol. A key factor that affects Hydra's feasibility as a platform for massively-multiplayer games is scalability. To some extent, Hydra delegates the responsibility for scalability to the game developer. It is up to the game developer to divide the game world into separate regions so the expected load on each slice (i.e. number of clients connected) will not be excessive, or to implement some form of admission control. Scaling is achieved not by increasing the number of connections per slice, but by increasing the number of slices for the game world.

That said, it is our intention to explore some further optimizations for improving Hydra's scalability and we plan to implement the following two optimizations in the near future:

1. **Dynamic Broadcast Tree**. The primary slice often has to broadcast a message to all the clients connected to it and the number of messages scales linearly with the number of connected clients. A straightforward way to improve the scalability of this broadcast is to organize the connected clients into a dynamic broadcast tree [22].

2. **Server Migration**. Networks are often heterogeneous. Hence, if we can identify the higher-bandwidth nodes and have such nodes host the primary slices, we will be able to support more client connections. The same approach can also be adopted to migrate a primary slice to a node that has the best latencies to the set of connected clients.

5. EVALUATION

To demonstrate the practicality of the proposed architecture, we implemented *Tankie*, a simple "capture the flag" game where players control tanks with the goal of collecting a flag and bringing it back to a home base. In addition, there are some obstacles in the game world that can be destroyed and tanks can also destroy other tanks.

In addition to a playable client, which allows a human player to control a tank and play the game, we also have a number of bot agents available that are able to play the game relatively well[1].

[1]The code base for Tankie was used in a class assignment for the introductory AI class at the National University of Singapore. In the assignment, the students were tasked to implement an agent to play the game. The two best-performing agents out of the 19 submissions for the class were used in our experiments.

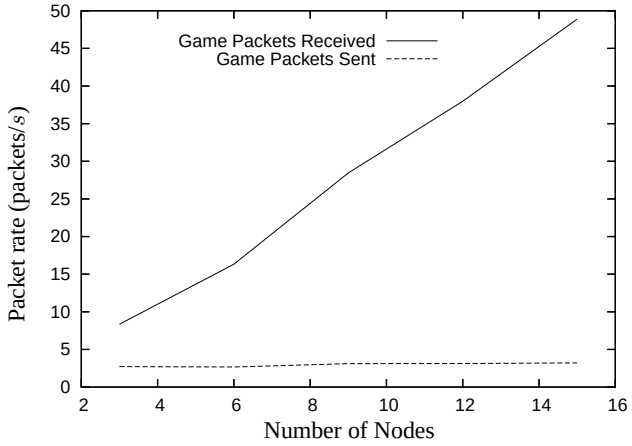

Figure 3: Plot of game packets received and sent by a client.

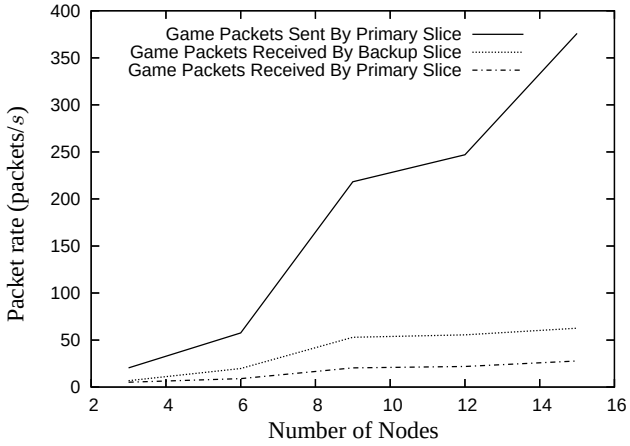

Figure 4: Plot of game packets received and sent by a Hydra server.

Tankie was used for the preliminary performance evaluation of Hydra.

5.1 Experiment Setup

We ran our experiments on six Pentium 4 PCs running Linux kernel version 2.6.20 and connected with a switch. We ran between 3 and 15 separate instances of the Tankie client with one node randomly chosen to host the primary slice and the client instances distributed over the six machines.

Each experiment was conducted as follows: clients are started incrementally at 5 second intervals. At 2 minutes, a backup slice is created on one randomly chosen node by the primary slice. At 7 minutes, the client instance hosting the primary slice is killed. Once this failure is detected, the backup slice will initiate the recovery protocol and take over as the primary slice. Two minutes later, at approximately 9 minutes, the new primary slice will in turn create another backup slice on another randomly chosen node. We end the experiment at 14 minutes.

In our experiments, the primary slice failure detection protocol detects the failure of the primary slice in around 5 to 8 seconds. Since we only created one backup slice, leader election among the backup slices is unnecessary.

5.2 Results and Discussion

In Figure 3, we plot the number of game packets sent and received by the client. We recorded the packets sent by the game application and the control packets sent by Hydra separately. Similarly, we plot the corresponding data for the Hydra server in Figure 4. The number of Hydra control packets in both cases is not plotted in Figures 3 and 4 because they are significantly smaller than the number of game application packets. The extra message overhead imposed is typically less than 1%. While the number of messages sent by the primary slice seems to grow in $O(n^2)$ as expected since it broadcasts update messages to all the clients, we can reduce the number of outgoing messages by using a dynamic broadcast tree [22].

Since response time is critical for networked games, we also measured the duration between when a game client sends a message and when it receives a response to that message. We call this the *command response time*. In Figure 5, we plot the command response time for a client node in a 15-node experiment. The steady state command response is approximately $300\,ms$, which is sufficient to support realtime strategy and role-playing games.

At 7 minutes, the failover introduces a period of around 5 to 8 seconds of lag when the unreliable messages sent by the clients are lost. We believe that this is acceptable since existing games occasionally suffer from lags that last between 30 seconds to a minute.

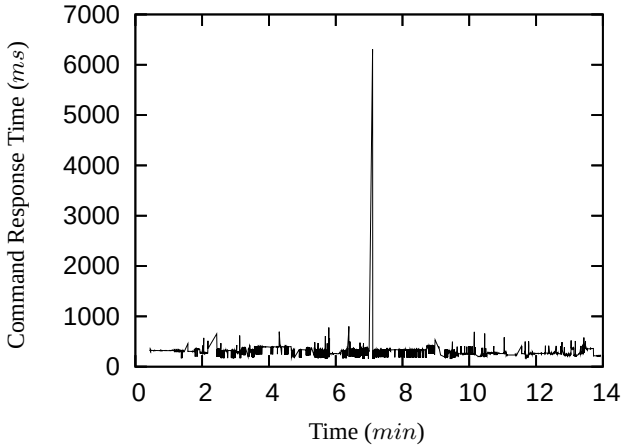

Figure 5: Plot of command response time for a client node in a 15-node experiment.

6. FUTURE WORK

Hydra is still work in progress. One of the key limitations we faced in the current evaluation was access to only small number of machines for running our experiments and we were thus limited to 15 nodes. We found that this was the maximum number of node instances that could be run on the available hosts without causing excessive performance degradation due to CPU overload. Our plan is to procure more machines for experimentation in order to evaluate the performance of Hydra with more than 15 clients connected to the same slice. We also plan to evaluate Hydra over a wide-area network and evaluate the feasibility of introducing congestion control.

We are currently implementing *Squad 101*, a procedurally-generated multiplayer real-time strategy (RTS) game, where each player controls a small squad of soldiers, in a virtual landscape, that fights other factions for territory and resources. The game was implemented in Java using the Irrlicht 3D Engine [12]. In Squad 101, the game world is divided into sectors. Each faction in the game may build structures in a sector and use it as a launching area for attacks on other enemy sectors. Under the Hydra architecture, each sector is implemented as a slice that may be hosted on different nodes.

We have several reasons for implementing Squad 101. First, we hope to demonstrate that the programming model defined by Hydra is sufficiently flexible for supporting a RTS game. Second, in Squad 101, the squads are able move from sector to sector. This effectively translates to the migration of connections between slices and thus we can use Squad 101 to test and develop the connection migration interfaces. Third, with Squad 101, the size of the landscape is unbounded. Sectors are procedurally generated and so in theory, there can be infinitely many sectors and therefore be able to support a large number of players. Last but not least, we would like to attract real players to play our game so that we can validate the Hydra architecture in a "live" setting. At the time of writing, Squad 101 is still under development.

Finally, while cheat prevention is of significant interest for networked games [2, 9], cheat prevention for the Hydra architecture remains as future work.

7. CONCLUSION

There have been many proposals for implementing networked games on peer-to-peer architectures [18, 4, 16, 23, 15, 5]. Game developers are however not experts in distributed systems and they should not be required to be experts on peer-to-peer algorithms. Hydra seeks to bridge the gap between the research community and the game developers. Our key contribution is the development of a programming interface that is intuitive and easy to use, and that can be supported transparently at the network layer. In our work, we have demonstrated that it is practical to provide necessary infrastructural support to implement a massively-multiplayer game in a distributed peer-to-peer architecture by adopting an augmented server-client programming model.

In addition, while Hydra is an architecture that was specifically developed for massively-multiplayer online games, we believe that the techniques described are more broadly applicable to a number of peer-to-peer applications. In our description of Hydra above, each slice instance in the Hydra system is described as a game server object that manages one part of the game world. In practice, we can also implement a chat service, an in-game email system and an in-game auction system each as a slice instance under the Hydra framework. In fact, we believe that further work on the Hydra system will provide us with insights on enabling server migration and load balancing for a large class of peer-to-peer distributed applications.

8. REFERENCES

[1] M. Assiotis and V. Tzanov. A distributed architecture for MMORPG. In *Proceedings of NetGames '06*, page 4, October 2006.

[2] N. E. Baughman and B. N. Levine. Cheat-proof playout for centralized and distributed online games. In *INFOCOM*, pages 104–113, 2001.

[3] S. Benford, C. Greenhalgh, and D. Lloyd. Crowded collaborative virtual environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 59–66. ACM Press, 1997.

[4] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of SIGCOMM 2004*, August 2004.

[5] A. Bharambe, J. Pang, and S. Seshan. Colyseus: A distributed architecture for multiplayer games. In *Proceedings of NSDI 2006*, 2006.

[6] Blizzard Inc. World of Warcraft. http://www.warofwarcraft.com.

[7] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002.

[8] E. Cronin, B. Filstrup, A. R. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. In *Proceedings of NetGames '02*, pages 67–73, New York, NY, USA, 2002. ACM Press.

[9] M. DeLap, B. Knutsson, H. Lu, O. Sokolsky, U. Sammapun, I. Lee, and C. Tsarouchis. Is runtime verification applicable to cheat detection? In *Proceedings of the NetGames '04*, August 2004.

[10] S. Ferretti and M. Roccetti. Fast delivery of game events with an optimistic synchronization mechanism in massive multiplayer online games. In *Proceedings of ACE '05*, pages 405–412, New York, NY, USA, 2005. ACM Press.

[11] E. Frécon and M. Stenius. Dive: a scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering*, 5(3), November 1998.

[12] N. Gebhardt, T. Alten, C. Stehno, G. Davidson, A. F. Celis, and J. Goewert. Irrlicht engine.

[13] C. Greenhalgh. Awareness-based communication management in the MASSIVE systems. *Distributed Systems Engineering*, 5(3), November 1998.

[14] C. Greenhalgh and S. Benford. Supporting rich and dynamic communication in large scale collaborative virtual environments. *Presence: Teleoperators and Virtual Environments*, 8:14–35, February 1999.

[15] T. Hampel, T. Bopp, and R. Hinn. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of NetGames '06*, page 48, October 2006.

[16] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *Proceedings of NetGames '04*, August 2004.

[17] J. Kaufman, T. Lehman, G. Deen, and J. Thomas. OptimalGrid – autonomic computing on the grid, June 2003.

[18] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *Proceedings of IEEE INFOCOM'04*, March 2004.

[19] E. Lety, L. Gautier, and C. Diot. Mimaze, a 3D multi-player game on the internet. In *Proceedings of the 4th International Conference on Virtual System and Multimedia*, volume 1, pages 84–89, November 1998.

[20] A. E. Rhalibi and M. Merabti. Agents-based modeling for a peer-to-peer MMOG architecture. *Computers in Entertainment*, 3(2):3, 2005.

[21] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[22] S. Yamamoto, Y. Murata, K. Yasumoto, and M. Ito. A distributed event delivery method with load balancing for MMORPG. In *Proceedings of NetGames '05*, pages 1–8, October 2005.

[23] A. P. Yu and S. T. Vuong. MOPAR: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In *Proceedings of NOSSDAV 2005*, pages 99–104, June 2005.