Quiver: a Middleware for Distributed Gaming

Giuseppe Reina
Technicolor - Eurecom
France
giuseppe.reina@technicolor.com

Ernst Biersack
Eurecom
Sophia Antipolis, France
ernst.biersack@eurecom.fr

Christophe Diot
Technicolor
Paris, France
christophe.diot@technicolor.com

ABSTRACT

Massively multiplayer online games have become popular in the recent years. Scaling with the number of users is challenging due to the low latency requirements of these games. Peer-to-peer techniques naturally address the scalability issues at the expense of additional complexity to maintain consistency among players.

We design and implement Quiver, a middleware that allows an existing game to be played in peer-to-peer mode with minimal changes to the engine. Quiver focuses on achieving scalability by distributing the game state. It achieves consistency by keeping the state synchronized among all the players. We have built a working prototype of Quake II using Quiver. We analyze the changes necessary to Quake II and discuss how generic a software like Quiver can be.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

Keywords

Distributed Gaming, Peer-to-peer Systems, First-Person Shooter Games, Middleware Design, Distributed Gaming Challenges

1. INTRODUCTION

Over the last decade, the industry of multiplayer online games has seen a rapid growth in popularity and revenue. With the rise of Massively Multiplayer Online Games (MMOGs) large scale gaming infrastructures allow thousands of players to share a consistent experience over the Internet.

Most networked games rely on centralized architectures. The virtual world is managed by a game server that stores the game state and performs the game simulation, while coordinating remote players. The game server can be either located on dedicated hardware (*i.e.* a single dedicated machine or a cluster of servers) or on a player machine. The players remotely interact with the game using a client component that provides a graphical front-end to the remote simulation. The success of the centralized architecture comes from many advantages, namely ease of implementation, manageable game consistency (all players have the same view), cheating prevention, access control and billing.

Centralized architectures have drawbacks: (i) the cost of hardware is high, as game servers must be deployed on high-end servers that have to be operated and maintained; (ii) scalability is limited to the number of players that a server can support; (iii) finally, the presence of single points of failure makes the architecture not robust. Fast paced games such as First Person Shooter (FPS) games are more affected by the previous limitations. In FPS, the player controls the movements and actions of its avatar by looking at the world in a first-person perspective. The goal of such games is to compete in gun-based combats (*e.g.* kill the opposite team, steal the flag, etc.) as effectively as possible. The strict real time constraints limit the scale and the size of a FPS game session; only few players with limited network delays can interact.

Peer-to-peer (p2p) mechanisms naturally address these drawbacks by using players resources and offloading the servers from performing game management tasks. However, the absence of a central coordination entity leads to a number of new design challenges, including an efficient coordination of the distributed simulation, data storage and discovery, handling player churn, load balancing, consistency and latency related issues. These problems are extremely difficult to address, typically require low-level programming skills and a deep knowledge of the game architecture.

Our objective is to study the feasibility of a software that allows any existing single player game or centralized multiplayer game to be played in p2p mode (*i.e.* to be distributed among the players) with minimal changes to the game engine.

In this paper, we first study the basic *modus operandi* of a generic game engine (Section 2), we isolate the main functional components and we analyze how to distribute the game on multiple machines while maintaining an unaltered game experience (Section 3). We realize that design such a software to be as generic as possible is a challenge and we focus our analysis on understanding to which extent it is possible, and what the limitations are with current game engines.

We introduce Quiver (Section 4), a middleware for distributed multiplayer online games that can be applied to both, existing games and newly developed games. Our middleware implements p2p techniques to achieve system scalability and game state consistency. Quiver is designed to be "plugged" to any commercial game with minimal changes. It acts as a distributed entity storage that allows the game programmer to develop a p2p multiplayer game without the need to comprehend the low-level message exchange.

We show the feasibility of our approach by integrating Quiver with Quake II (Section 5). We show that the integration of Quiver

NOSSDAV'12, June 7–8, 2012, Toronto, Ontario, Canada.

in the Quake II game engine requires minimal code modifications and we discuss about the the limit of this approach.

2. BACKGROUND

2.1 Game features

Games have specific requirements that depend on the gameplay. We identify some fundamental properties that are common to most networked games, namely consistency, responsiveness, data persistency, scalability, and security.

The first and most important requirement of networked games is *consistency*, it is achieved when the following two properties are satisfied (i) the virtual world state is equally perceived by all the players at any time and (ii) the outcome of a user action to the virtual world reflects the intentions of the user (*e.g.* the action of shooting a player should damage the targeted avatar). Due to a number of issues, such as delayed traffic on the Internet and unpredictability of the user demand, this task is considered to be one of the most challenging in networked games.

Reactiveness is a strict requirement for fast-paced games. In such games, the user must perceive quickly enough the changes in the virtual world state, *e.g.* neighbor avatar movements or missile explosions. This requires the game architecture to communicate the result of a user action, as well as it's consequences within a time period that preserve natural interaction (*i.e.* $\leq 100ms$ for FPS games [**?**]).

Scalability indicates the ability of the game to handle high number of participants without sacrificing the user-experience. Precisely, a networked game architecture is required to sustain a theoretically unbounded growth in the user demand with a sub-linear impact on the user experience. In FPS games, the real-time gameplay imposes strict reactiveness requirements that affect both consistency, due to increased latency, and scalability, due to the high workload of the simulation. In most centralized games, only few players can interact in order to preserve consistency and reactiveness.

Persistency is the capability of the networked game architecture to guarantee that information, *e.g.* objects or scores, are not lost during the evolution of the virtual world.

Finally, *security* is one of the most sensitive requirement of multiplayer games. MMOGs have created complex virtual economies as well as fierce competitions among users. Therefore, a networked game architecture needs to guarantee security in order to ensure fairness among its users (*i.e.* cheating prevention) as well as privacy of user information.

2.2 Game architecture

The software component behind a game is called the **game engine**. It is responsible of capturing and processing player intentions coming from input devices (*i.e.* mouse, keyboard, controller, etc.). It stores and simulates the virtual world and finally renders the most updated state to the output devices (*i.e.* screen, speakers, etc.). A game engine includes several subsystems such as a physics engine, graphics and audio rendering, artificial intelligence, networking, and others depending on the genre and complexity of the game.

Regardless of whether the game is played locally or over the Internet, few recurring abstractions are required to describe it, namely the *entity*, the *command* and the *game loop*.

The dynamic part of the game state, can be described as a collection of virtual objects that make up the virtual world. Each virtual object is stored in a data-structure called **entity**. Every part of the game that can change its state is described by an entity, this includes avatars, monster, bullets, explosions, moving doors etc.

Algorithm 1 Game loop

```
function tick():
 cmds ← fetchCommands(inputs_state)
 gameState_i ←executeCommands(cmds, gameState_{i−1})
 foreach entity in gameState_i
      gameState_i ← simulatePhysics(entity)
      gameState_i ← checkCollision(entity,gameState_i)
      gameState_i ← entity.think(gameState_i)
 end
 graphicRendering(gameState_i)
```

The **command** is a representation of the player intentions obtained by processing the user inputs. The command encapsulates an action that can be performed by the player on the subset of entities that he can control. The execution of a command by the game engine causes a change on the entities state.

Finally, at the very heart of the game engine there is the **game loop**, a single threaded execution process that periodically fetches and executes user commands, invokes all the subsystems to perform the simulation, and finally gives back the feedback to the player by rendering the player view (see *alg.* 1 above). At each step of the loop (often called *tick*), the simulation is performed by iteratively simulating each entity of the game state in terms of physics, collisions, and some internal behavior, often named *think function* that usually wraps time-triggered behaviors or calls to the artificial intelligence subsystem.

2.3 Multiplayer gaming

Multiplayer games give a single machine the authority to modify the game state. This architecture is called *client/server* or *centralized*.

The authoritative machine (*i.e.* the server) has the duty to communicate with the remote players (*i.e.* the game clients), to execute the player commands and to run the simulation that alters the virtual world. The client "packs" user inputs into commands to marshal to the game server and receives the view to render to the player. The game state can only be altered by the server, whereas the clients provide the players with the game interface to remotely interact with the state. The client also includes techniques to predict the future state of the game in order to compensate for the latency in the communication with the remote server [**?**, **?**, **?**].

While clients are deployed on player machines, the server component can be either deployed on a dedicated machine or hosted by one of the player machines. Scalability is often affected by the reactiveness requirement of the game. In FPS, for example, due to the high pace of simulation (20-60 ticks per seconds) and the high amount of commands sent by the users, a single server can often support as little as 20 players in a match.

3. PEER-TO-PEER GAMING

3.1 Requirements

The implementation of a scalable p2p game is still a research topic, whose challenges have been identified by the community [**?**, **?**]. While the scalability issue can be intuitively solved by a p2p approach, it is challenging to perform the game simulation (see 2.2) in a distributed way while preserving the main features of the game consistency, reactiveness, scalability, persistency and security.

In addition, we identify churn and load balancing as two important features for a distributed gaming architecture.

Churn is a major problem in p2p games. The dynamics of player participation can change during the game: a player can join the ongoing game or can leave it with or without notifying the network, e.g. for a connection drop or a general failure. In a P2P game scenario, churn affects data availability, communication paths, and game state consistency, as the amount of available resources for computing and distributing the game state changes. *Load balancing* can be described as a set of methodologies for the distribution of computational, storage and bandwidth workload among the actual available resources, with the aim of maximizing the overall quality of experience. Thus, load balancing can be exploited for both handling churn and maximizing resource consumption.

In this work, our objective is to provide a system that scales with the number of users, while providing a level of consistency that ensures a satisfactory game experience.

Game distribution.

In a p2p environment, the tasks of game simulation and state storage, which for centralized architectures are performed by a single machine, need to be performed in a distributed manner. The distribution of a game can be divided in three subproblems: (i) state partitioning, (ii) state discovery, and (iii) distributed simulation.

State partitioning.

The state of the world, represented as a set of entities, must be stored and managed by multiple nodes, making sure that each node maintains in its local store only the relevant fraction of the virtual world. Scalability is affected by how the virtual world is distributed as smaller fractions of the virtual world require less resources in the player machines.

State discovery.

State discovery is a direct consequence of the state partitioning algorithm. It takes care to dynamically fetch the part of the game state that becomes relevant to the player. When new parts of the map are explored, the peer node must locate the remote peers responsible for the missing portions of the newly discovered state and fetch it from them.

Distributed simulation.

In the centralized architecture, the presence of a single authority (the server) guarantees to all the players a consistent view of the game. The p2p architecture inherently introduces a distributed parallel execution of the simulation that should allow the players to concurrently modify the game state. State replication and synchronization is needed to achieve consistency in the distributed environment.

A modification of one of the replicas should be propagated in a consistent manner to all the other instances in the remote nodes. The system should also provide collaborative algorithms for conflict resolution to address the problem of concurrent modifications of the same object.

4. QUIVER

4.1 Design

We present Quiver, a middleware for distributed multiplayer online games. We design and implement Quiver to be portable across different games (*i.e.* not based on any specific game engine[1]) and to be as little intrusive as possible for the game engine (*i.e.* few modifications are required to integrate Quiver into an existing game). Quiver can be applied to both existing games and newly developed games by working with the abstractions of commands, entity and game loop discussed in section 2.2.

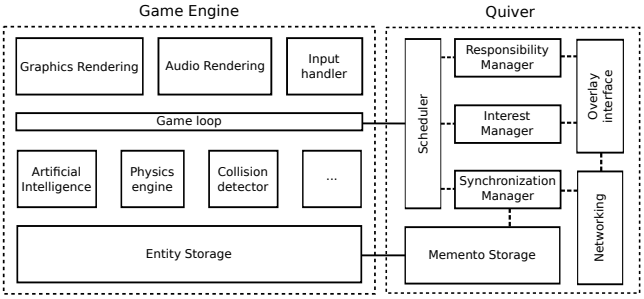

Figure 1: Quiver architecture and its interaction with the game engine

Quiver provides the game engine with object replication through a distributed storage facility that enforces game synchronization. Two high level predicates, the *responsibility* and the *interest*, are used as a means to respectively store and discover the game state. The distributed storage allows the game programmer to manipulate entities without the need to comprehend the low-level message exchange of the underlying distributed algorithms.

Quiver extends the basic functionalities of the game engine by scheduling its internal modules at each tick of the game loop and providing them with high level networking channels and an overlay network. We defined three abstract modules to address the challenges described in the previous section, namely the *Synchronization Manager* for distributed simulation, the *Responsibility Manager* for state partitioning, and the *Interest Manager* for state discovery (*fig.* 1). Each of these modules, together with the overlay network, can have multiple implementations depending on the requirements of the gameplay.

Object replication for distributed simulation.

We choose to implement the distributed simulation at entity level by providing a shared data-storage to the game engine. This way, the game state is distributed among the peers by replicating single entities ensuring that any entity can be instantiated in at least one peer. Quiver enforces consistency by handling the communication between the different replicas over the network.

Quiver treats the simulation phase at each tick of the game loop as a *black box*. Upon its invocation, Quiver enables the synchronization module to detect modifications applied to the game entity during the simulation phase by monitoring the evolution of the fields that describe the game entity.

The game synchronization module, defines the mechanisms that propagate and remotely apply local modifications to the game entity. Moreover, it specifies the conflict resolution policy upon concurrent write access to the same entity by two different peers in the network.

We believe that the shared data-storage approach best satisfies the *portability* and *non intrusiveness* requirements of Quiver. Precisely, as opposed to methodologies based on user commands or events exchange, such as the *Lockstep algorithm* [**?**], *Trailing State Synchronization* [**?**] and others [**?**, **?**, **?**, **?**], the shared data-storage

[1]Although FPS are the focus of this study we try to keep a broad view on other game genres, spanning from role-playing games to real-time strategy games

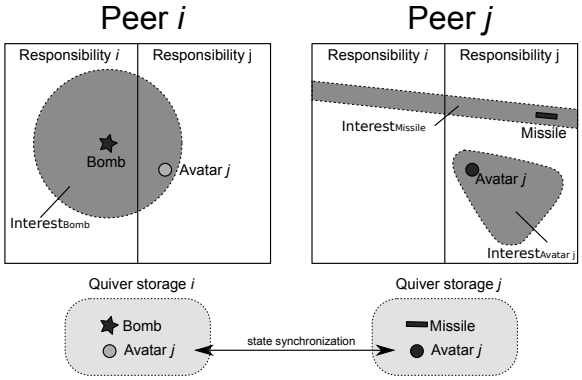

Figure 2: The three main modules in action: object replication and synchronization using spatial interest and spatial responsibility.

enforces a better decoupling from the logic of the game engine components that perform the game simulation. Moreover, the event-based approach would require the simulation to be performed only once all the events from the other peers have been received [**?**], thus violating the reactiveness requirement.

Responsibility Management and state partitioning.

We define the notion of **responsibility** to partition the game state and to assign replicas every peer.

A responsibility is a *predicate* assigned to each peer in the network, that applied to an entity returns true, if the object should be replicated on the peer, false, otherwise. The responsibility of a peer defines a subset of the game state that should be locally replicated. We use the responsibility as a means to cluster entities onto peers and ensure that for each entity there exists at least one peer responsible for storing the entity. Peers trade and negotiate responsibilities in the overlay network to achieve state persistency in an application specific manner.

State partitioning can be performed spatially by clustering entities according to their location inside the virtual world. This specific case of spatial responsibility, called *area of responsibility*, makes sure that the entities located inside the region are locally replicated (*fig.* 2).

Interest Management and state discovery.

While *Responsibility Management* defines how entities are disseminated among the peers, the state discovery defines how the peers communicate in order to fetch replicas of an entity located outside their responsibility. There are two main cases in which a peer should store a replica of an entity for which it is not responsible:

- The entity is "physiologically sensed" by the avatar of the player (*i.e.* eyesight, hearing, etc.)
- The entity is accessed by the simulation of a local replica during the execution of the local game loop.

We define the **interest** as a predicate of an entity that applied to another entity returns true if the latter should be locally available in the data-store, false otherwise. There are two kinds of interests, the *interest of an entity* and the *interest of the player*.

The *interest of an entity* is used to fetch all the the entities that can be modified during the future simulation of the local replica. Every moving entity has a default spatial interest of a sphere centered around the object so that all the other objects close to it can be

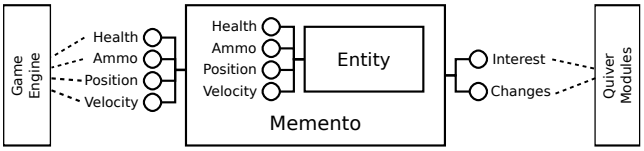

Figure 3: Entity/Memento wrapping and its interaction with the two layers

processed by the physics and collision engine. In addition, when a moving entity implements a think function, the interest is used to pre-fetch all the other entities that the think function can modify.

The *interest of the player* is a special kind of interest, and recalls the *area of interest* used in centralized games [**?**]. Every player perceives the virtual world by looking at the view rendered by its client. Such a view depends on the specific game genre, FPS, for example, renders the view as the player is seeing the world through the eyes of his avatar, while real-time strategy games renders a bird-eye view that can be controlled by the player. The peer, therefore, must specify an interest in order to fetch all the objects that are required to render to the view of the player, regardless of the local responsibility.

Figure 2 summarizes the described modules through a simple scenario.

4.2 Implementation

Quiver has been implemented in Java and it has been designed to be modular and extensible. The main scheduler (*fig.* 1) manages the execution of three main modules that perform *Synchronization Management* to achieve per-entity eventual consistency, *Interest Management* for state discovery and *Responsibility Management* for state partitioning. Quiver provides the developer with a library of different implementations of such modules that can be used by the game engine or extended to meet specific game requirements. The low-level description and the implementation details of these modules is out of the scope of this paper.

Quiver defines its own data-structure for the entity, called *Memento*. The Memento exposes functionalities to both the game engine and the middleware providing a shared data-structure for the two communicating layers (*fig.* 3). The Memento keeps track of the modifications applied to the entity from the game engine allowing the underlying layer to detect when the entity state must be negotiated with the remote peers. In addition, the memento exposes the *Interest* of the entity at each step of the game simulation. Finally, the memento data-structure supports a transparent per-field serialization used by the Synchronization module to marshal entity updates across the network.

The *Networking* module provides the other Quiver modules with high-level networked channels to communicate using message exchanges that can be either reliable or best effort.

Game Integration.

Integration of Quiver with a game engine is straightforward and consists of three steps.

- Invoke the Quiver scheduler at the end of each tick of the game loop
- Adopt the Quiver storage to store the game objects (*i.e.* insertion, deletion and simulation must be notified to the middleware).
- Link or wrap each entity using a Memento.

In order to provide a single data-structure shared by the game engine and middleware, some functionalities must be defined in the Memento data-structure. In particular, a Memento must specify, at each tick of the simulation, the Interest of the game entity used for state discovery by the Interest Management module. The Interest is used in the overlay network to *subscribe* for the portion of the virtual world that is relevant to the entity associated. For example, an avatar Memento is "interested" in the portion of the world it can perceive, while a bomb is "interested" in the portion of the world that can damage upon its explosion.

Quiver requires the entity to expose a bitmask that indicates which fields or parts of the entity have been changed during the last tick. The bitmask is used to reduce conflicts caused by simultaneous modification of the game entity by acting at field-level rather than object-level. Moreover, the bitmask is used to produce delta-encoded updates to reduce load of the synchronization messages on the network.

Figure 4: Screenshot of Quake II in p2p using Quiver

5. ANALYSIS

5.1 Quake case study

We evaluated Quiver with Quake II, the popular FPS developed by idSoft (*fig.* 4). Quake II is the first FPS game that offered excellent game-play in online matches. Moreover, Quake II has been the object of several mods by the open source community that spawned some of the most played free FPS such as WarSow and CodeRED: Alien Arena.

We chose a Java version of Quake II called Jake2[2] created by faithfully translating the original C code of Quake II into Java code[3]. We chose the Java implementation since we need a rich environment of tools for testing and debugging, moreover Java makes the game easily portable to different hardware architectures and operating systems. Versions of Jake 2 have been ported to multiple architectures spanning from embedded systems such as Android to browser integrated HTML5 WebGL

We integrate Quiver with the server component of Jake2 maintaining the decoupling with the client component. This integration can be used for both, a pure p2p mode by simultaneously activating both the client, the server and the middleware component in each player machine, and multiple server architecture, by activating only the server and the middleware on dedicated machines, while players connect to them by using ordinary clients.

We use Aspect Oriented Programming (AOP) and AspectJ [**?**], to integrate the middleware into Jake2. AOP enabled us to inject calls of the middleware without modifying the original game code. Using AOP, we were able to decorate the entity state into a Quiver's Memento intercepting single reads and writes to the fields of the entity and injecting control code used to monitor the evolution of the entity state.

The resulting number of lines of code of the bridge between Quiver and Quake is 8k, versus the 150k for Quiver. Due to the verbosity of the Quake entity and the triviality of the wrapping operations, the code related to the entity-Memento integration has been automatically generated using a compiler to Java code.

Game distribution modules.

In order to evaluate the middleware we implement some strategies for the game distribution modules.

We adopt the replication strategy described by Bharambe *et al.* [**?**] for game synchronization. Entities are replicated among the peers either as primary copy or as a secondary copy. For each entity in the virtual world, there exists a unique primary copy and the peer that owns the primary copy is the authoritative peer for the entity. The peers that can interact with the entity locally replicate the remote entity by holding a secondary copy. The primary copy owner acts as a central coordinator for all the secondary copies and taking care of forwarding entity updates and managing conflicts upon simultaneous modification of the same entity.

The game state is partitioned into *disjoint* spatial responsibilities, or area of responsibility (AoR), such that each primary copy is assigned to one and one peer only. We use the BSP tree nodes of the Quake II map both to partition the game map into AoR's and to define Interests of the avatars and monsters. Negotiation of responsibilities and interests are done over an DHT network based on Kademlia that has been adapted to better work with the BSP tree nodes of Quake.

We establish that in order to reduce message exchanges between the peers, and reduce conflicts caused by simultaneous modifications, the primary copy is the only copy that can be simulated[4].

We handle parallel write attempts of an entity at the peer who owns the primary copy. The primary copy decides which of the concurrent modifications should win the race condition. The decision on how to resolve the conflict comes from the entity itself and its properties. By default, changes are applied in a first-come, first-served basis in order to boost reactivity. Therefore, conflicts between a primary and a secondary are always won by the owner of the primary copy due to the direct modification by the game engine.

There are, however, entities that need a different resolution mechanism. For example, the effect of rollback on an avatar position negatively affects the game experience of the player controlling it. Rollback of an avatar to an old position causes a change of the whole view of the player and this operation must be avoided as much as possible. In this context, we prefer to give priority to the update coming from the machine that handles the player, regardless of whether it stores a primary or a secondary copy.

Play experience.

The resulting prototype provides a good play experience. Operation of handing-over of an entity due to its migration from a responsibility to an other is lightweight enough to not be noticed by the players. The effects of rollback due to modification conflicts of an

[2]Jake2. `http://www.bytonic.de/html/jake2.html`
[3]Unofficial Jake2 Resource. `https://wiki.in-chemnitz.de/bin/view/RST/Jake2`

[4]In FPS games, the high pace of simulation causes frequent changes in the state of an entity.

entity are significantly reduced by the adoption of the Memento bitmask that allows multiple peers to simultaneously modify different disjoint subsets of the entity fields. However, due to its early state of development the prototype suffers from the classic inconsistencies introduced by latency (jerky movements of the other players outside the local responsibility).

5.2 Limits and future development

During our study we found two main limitations with our approach. First, the animations of entities that are synchronized by remote peers appear jerky. This is a common problem in networked games and is caused by the latency in the p2p communication. Our choice of decoupling the game engine from the middleware, makes the adoption of well known centralized techniques for latency compensation [**?**] challenging. We are currently working on a new prediction system able to work on our distributed environment without sacrificing the modularity of our current design.

The second limitation regards inconsistencies inherently caused by the object replication methodology. In particular, when executing the game loop it may happen that a think function (*e.g.* a bomb exploding near a group of avatars) or a player command (*e.g.* a spell is casted upon a team), can modify multiple entities at the same time. Since each entity follows an isolated process of update propagation, it may happen that the changes made locally on a subset of objects are reverted back to an old state due to a remote conflict. The result is that entities are not modified by the command or simulation (*i.e.* an avatars survives the bomb explosion, or a player is not affected by the spell) thus violating the consistency requirement.

Finally, we plan to support *churn* in the next version of Quiver to allow each player in the game to freely join and leave the game at any moment. The resources allocated by the leaving peer will be redistributed to the remaining peers in a timely manner without affecting the gameplay of other players.

6. CONCLUSIONS AND OUTLOOK

A scalable solution for p2p gaming requires to properly address the following questions: (i) How to distribute the game state (ii) How to discover the game state and (iii) How to perform the game simulation on the game state in a distributed way.

We designed a middleware, called Quiver, that addresses the three previous questions. Quiver encapsulates each aspect in one of its modules. Precisely, the game state is distributed by defining a *Responsibility* for each peer. The game state is discovered by subscribing to the *Interest* exposed by each entity. Finally, entities are replicated on the machines according to their Responsibility and the Interest. We proved the feasibility of our approach by creating a working prototype with a popular FPS game (*i.e.* Quake II).

We showed that our approach allowed us to abstract from the internal design of the game and can be adopted with minor modification to an existing engine: few entry points are required to Quiver to interact with Quake II. Using Aspect Oriented Programming as a methodology, we were able to integrate Quiver with Quake II by simply injecting code into the game engine.

As future work, we plan to evaluate the performance of Quake II comparing the centralized version against the Quiver-integrated version. The results obtained will allow us to define the architectural direction for the next generation of Quiver. Finally, we will integrate Quiver in more recent and diverse games in order to conduct performance comparisons between different game genres.

7. REFERENCES

[1] Aspectj framework, cross-cutting objects for better modularity. `http://eclipse.org/aspectj/`.

[2] Y. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. `http://bit.ly/wGNhMO`, 2001.

[3] A. Bharambe, J. Pang, and S. Seshan. Colyseus: a distributed architecture for online multiplayer games. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation*, NSDI'06, Berkeley, CA, USA, May 2006.

[4] M. Claypool and K. Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11), November 2006.

[5] E. Cronin, A. Kurc, B. Filstrup, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. pages 7–30, Hingham, MA, USA, May 2004. Kluwer Academic Publishers.

[6] L. Fan, P. Trinder, and H. Taylor. Design issues for peer-to-peer massively multiplayer online games. *Int. J. Adv. Media Commun.*, pages 108–125, March 2010.

[7] G. Fiedler. What every programmer needs to know about game networking. `http://bit.ly/7jSZl5`, January 2011.

[8] L. Gautier and C. Diot. Design and evaluation of mimaze, a multi-player game on the internet. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, ICMCS '98, pages 233–, Washington, DC, USA, 1998. IEEE Computer Society.

[9] S. Hu, S. Chang, and J. Jiang. Voronoi State Management for Peer-to-Peer Massively Multiplayer Online Games. In *Consumer Communications and Networking Conference*, CCNC '08, pages 1134–1138, 2008.

[10] C. Neumann, N. Prigent, M. Varvello, and K. Suh. Challenges in peer-to-peer gaming. *SIGCOMM Comput. Commun. Rev.*, 37(1):79–82, October 2007.

[11] A. Steed and M. Oliveira. *Networked Graphics: Building Networked Games and Virtual Environments*. Morgan Kaufmann publishers, 2009.

[12] J. Steinman. Breathing time warp. In *Proceedings of the seventh workshop on Parallel and distributed simulation*, PADS '93, pages 109–118, New York, NY, USA, 1993. ACM.

[13] J. Steinman and J. Wong. The speedes persistence framework and the standard simulation architecture. In *Proceedings of the seventeenth workshop on Parallel and distributed simulation*, PADS '03, pages 11–, Washington, DC, USA, 2003. IEEE Computer Society.