Exploiting player behavior in distributed architectures for online games

Sergey Legtchenko

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Sergey LEGTCHENKO

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Adaptation dynamique des architectures réparties pour jeux massivement multijoueurs

soutenue le 25 octobre 2012

M. Pierre SENS	Directeur de thèse
M. Sébastien MONNET	Encadrant de thèse
Mme. Anne-Marie KERMARREC	Rapporteur
M. Pascal FELBER	Rapporteur
M. Ernst BIERSACK	Examinateur
M. Antony ROWSTRON	Examinateur
Mme. Anne DOUCET	Examinateur

Acknowledgements

The three years of my thesis have been incredibly rich in estimable experience brought me by valuable people. I would like to thank those who spent some of their time and effort to help me.
In the first place, I am grateful to my advisors, Sebastien Monnet and Pierre Sens who managed to canalize my raw enthusiasm into well-organized and consistent research. Our numerous and passionate discussions made me realize that the best solutions arise as a result of an exchange of opinions.
I would also like to thank my thesis jury, especially Anne-Marie Kermarrec and Pascal Felber for devoting some of their precious time to evaluate my ability to do research.
Working in the Regal group made me meet many interesting and friendly people. I would particularily like to thank Gael Thomas for teaching me the very important art of organizing ideas and comprehensively relate them. I am grateful to Gilles Muller and Marc Shapiro for their good advices and recommendations that helped me to improve my research. They also encouraged me to apply for an internship, which resulted in a highly valuable experience, showing me the non-academic side of research.
I would like to thank Ant Rowstron, my internship advisor, for teaching me some of his rigorous, pragmatic methodology and for his trust.
My Ph.D. experience would not have been the same without the friendly support of many people which presence created a pleasant and creative working environment. For that, thanks to Thomas Preud'homme, Julien Sopena, Florian David, Lokesh Gidra and many others.
I would like to thank my family, who encouraged me to start a Ph.D. and has supported that decision ever since. Last, but not least, thank you to Elodie for bringing me the happiness I needed to achieve my work.
I am deeply grateful to all these people who participated, each in their own way, to something that is not necessarily a big step for Research, but is definitely a giant leap for me.

Table des matières

Part I

Thesis

Chapter 1

Introduction

Contents

"A dream you dream alone is only a dream. A dream you dream together is reality."

John Lennon

Immersive gameplay is a key requirement for a video game. Massively Multiplayer Online Games (MMOGs) succeed to offer a highly immersive gaming experience by enabling a collaborative gameplay. MMOGs provide a virtual world that can be simultaneously explored by many players. Each player is represented by an *avatar* and allowed to interact with objects and other players located inside the virtual world. Such virtual worlds are called Networked Virtual Environments (NVEs)[1].

Over the last decade, MMOGs have progressively become one of the most popular classes of distributed applications. Dozens of popular MMOGs are played by millions of users on a daily basis [12]. The cumulated MMOG population has rapidly grown since the late nineties and now represents more than 20 millions of active players around the world. The ten-year growth of the gamer population is presented in Figure 1.1.

Virtual environments are increasingly becoming more than just entertainment. They are used for advertisement [39], medicine [56], cybersport [160] or even diplomacy [129] and help people to build and maintain their social ties.

[1] In the rest of the thesis, we will use the terms *MMOG* and *NVE* interchangeably

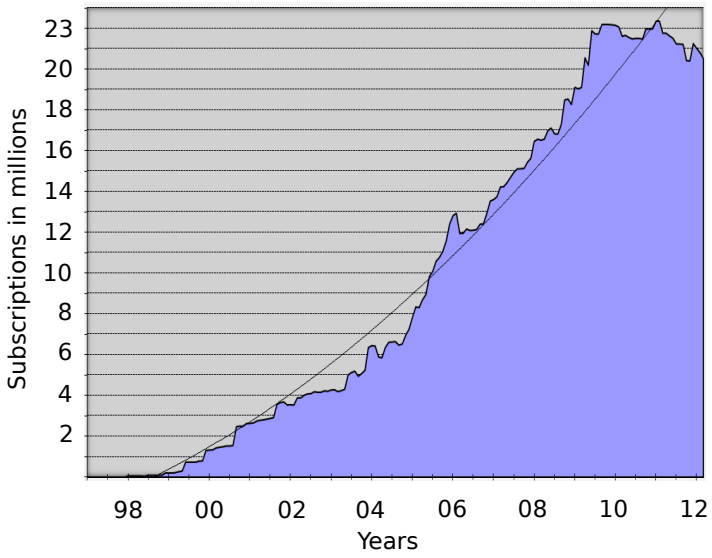


Figure 1.1: *Total MMORPG Subscriptions and Active Accounts [12] (MMORPGs are a popular sub-class of MMOGs).*

1.1 Problem Statement

Traditional online games rely on a unique server to manage the whole player population and store all the objects of the NVE. This simple client-server organisation is unsuitable for MMOGs because of the *massive* nature of their gameplay. For example, World of Warcraft, the most popular MMOG at this time, has more than 10 million subscribers [12], with an active population exceeding many tens of thousands of players [196]. Such scales are clearly beyond the capacities of a single server, and MMOG designers are forced to adopt large-scale distributed architectures to handle the applicative load.

MMOG workloads are complex and highly dynamic because they are shaped by player behavior. Players need to access objects located along their trajectory in the NVE, and the access frequency depends on the popularity of the object among the population. The distribution of the players across the NVE depends on their interests and the amount of information that needs to be exchanged between players is function of their degree of interaction. Player movement, interest and distribution patterns have been shown to be non-uniform and dynamic [205, 47, 165, 196] which greatly impacts content dissemination, data management and load balancing mechanisms of the MMOG architecture.

In addition to that, MMOG workloads have tight requirements in terms of:

- **Responsiveness:** the virtual environment should be rendered to the players with low latency. Studies show that updates of the NVE should be propagated to the players within a few hundreds of milliseconds to make the game playable [44].

- **Consistency:** players need to see a consistent view of the NVE. The player should not see: (i) outdated game state, (ii) Duplicate or missing objects, (iii) objects which state violates game rules. Moreover, the views of different players should be mutually

consistent: if players concurrently access the same object, critical updates on the object should happen in the same order for all the players.

- **Persistence:** the virtual world should evolve even when players are disconnected, giving the illusion of a true, living environment. The state of all the entities inside the NVE should therefore be stored and maintained despite failures and disconnections.
- **Availability:** players should be able to join the NVE at any time.
- **Elasticity:** the MMOG population dramatically varies over time because players tend to play at the same periods (*e.g.*, on week-ends) [67]. The architecture needs to ensure fair gaming experience even during population peaks.
- **Security:** cheating and other malicious behavior have to be prevented to ensure fair gameplay to honest players.

Meeting these requirements at large scale in presence of such complex workloads is extremely challenging. Currently, most of the commercially successful NVEs have a rigid architecture [2, 11, 19, 25]. The huge amount of players is divided into disjoint sets, called *shards*, each shard being handled by a set of servers [20]. Shards represent totally independent game instances with limited populations and no inter-shard communication is possible. Within a shard, each server is responsible for a well delimited zone of fixed size (this technique is called *static zoning*). Sharded architectures are able to scale, but have two major drawbacks that degrade the gaming experience. First, a sharded NVE is not seamless. Players have to join the same shard if they want to communicate in-game, and since the population size has a hard limit, the join success is not guaranteed. Second, if the population of a zone handled by a server exceeds its capacity, the responsiveness of the game degrades in that zone. Since the zoning is *static* and players are known to be non-uniformly distributed across the NVE, the responsiveness degradation is observed in commercial MMOGs [8].

Many research proposals have been made by the academic community to overcome the challenges of MMOG architectures. Many of them are based on the peer-to-peer (*p2p*) paradigm, so each arriving player's host shares its resources to maintain the NVE [121, 42, 49, 194]. Such architectures usually suffer from low responsiveness and lack of security. Although extensive research has been made in the field over the past few years, none of the proposed MMOG architectures succeeded to satisfy all the abovementioned requirements.

1.2 Research proposal and contributions

The MMOG workload is shaped by interactions occuring between the players and the NVE. These interactions are ruled by the gameplay properties of the MMOG and by the behavior of the player. Fortunately, gameplay properties are fixed by the application and are known in advance, and the behavior of the player can often be predicted. Therefore, as detailed in Chapter 3, there are strong assumptions on *how* players will impact the MMOG workload.

In this thesis, we claim that performance of existing NVE architectures can be greatly improved by monitoring and exploiting player behavior at runtime.

Our goal is to provide mechanisms to **dynamically**, and **automatically** adapt the distributed system to the characteristics of the workload. Such knowledge provides the ability

to better accommodate the workload by anticipating the applicative needs. The result is an increase of the distributed infrastructure efficiency with no substantial overhead.

This is not the first statement in favor of a better workload awareness for NVE architectures [47, 201, 75]. This thesis should be seen as a comprehensively evaluated confirmation of these ideas.

We first provide an analysis of game workloads issued from Second Life, a popular commercial MMOG [19]. We isolate characteristic trends of these workloads and propose a trace generator that produces arbitrary size traces exhibiting these trends. Finally, we propose novel techniques that will help existing MMOG architectures to better accomodate the NVE workload. The contributions of our work on MMOGs are thus[2]:

1. An analysis of traces obtained from Second Life highlighting their similarities with traces collected from other MMOGs.

2. A trace generator able to produce high resolution large scale traces exhibiting characteristic close to those observed in Second Life.

3. A mechanism that proactively modifies the topology of p2p NVEs to better handle avatar movements.

4. A technique to estimate player density distribution at very large scale in fully decentralized MMOGs.

5. An algorithm that improves the responsiveness of p2p NVEs by building density-distribution aware shortcuts in the NVE topology.

6. A novel server-based architecture that eases dynamic server placement to fit with player distributions. Our architecture decouples player management from data storage enabling the system to scale out without inducing unnecessary fragmentation of the NVE-data. This organization improves the overall performance and the adaptability of the system.

The thesis also incorporates two appendices that relate our work on key-value stores performed in parallel to our main research subject. The contribution detailed in the appendices are:

- Design and evaluation of Relax DHT, a churn-resilient replication strategy for DHTs that outperforms standard techniques in presence of churn. This work has been started during an internship at the LiP6 laboratory during the first year of Master's degree under the supervision of Sebastien Monnet.

- Design and evaluation of Jekyll and Hyde, a NoSQL [208] key-value store that efficiently handles mixed OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing) workloads. This work has been realized during an internship at Microsoft Research Cambridge under the supervision of Ant Rowstron.

[2]Security aspects of MMOG systems, while being crucial, were not adressed and are therefore out of the scope of the thesis.

The designed key-value stores are originally unrelated to MMOGs. However, the contributions of these works can possibly be used to improve MMOG architectures. DHTs are used to store NVE data at large scale in many p2p architectures [61, 89, 193], and Relax DHT can thus be implemented in these systems to improve churn resilience.

1.3 Outline of the thesis

This manuscript is organized as follows:

- **Chapter 2: Background.** This chapter describes the general background on distributed systems design. Through an overview of p2p overlays, we provide the reader with an insight of the architectures used in modern distributed systems. In the second part of the Chapter, we focus on techniques used for data management in a distributed context.

- **Chapter 3: State of the Art.** This chapter is composed of three sections that give an overview of the MMOG architecture design. First, it contains the description of the different types of architectures for NVEs. Then, it details the known characteristics of NVE player behavior. Finally, it enumerates state-of-the-art techniques that are known to use particularities of player behavior to improve global system performances.

- **Chapter 4: Trace analysis and generation.** Retrieving traces from real MMOGs is challenging. Most of the commercial MMOGs do not share their data and trace crawlers are often banned from servers when they are detected [196]. Moreover, shard populations rarely exceed a few hundreds of players, so retrieval of large-scale traces is impossible. Therefore, it is important to be able to generate synthetic traces that exhibit the characteristics observed in real MMOGs, but at a larger scale. In this chapter, we detail the methodology used to analyse Second Life traces and describe our realistic trace generator. We provide an comparison study between real Second Life traces and traces generated by us. *This work is a part of our work published in [DSN10].*

- **Chapter 5: Avatar-mobility resilient content dissemination in p2p MMOGs.** This chapter presents Blue Banana, a mechanism that helps p2p NVEs to handle player mobility. Blue Banana is a module implemented on each peer. It tracks avatar movements and detects predictable behavior. Once the prediction has been made, the module prefetches peer connections that will help to improve responsiveness during the movement. We perform a full evaluation of Blue Banana using traces produced by the generator described in the previous chapter. *These results are published in [DSN10].*

- **Chapter 6: Efficient routing in p2p MMOGs with non-uniform avatar distributions.** In this chapter, we introduce DONUT, a mechanism that reduces the latency of lookup queries in p2p overlays for range querying. Such overlays are well suited for MMOGs because range queries can be used to efficiently retrieve objects from the NVE [49], but their performance usually drops if the data distribution is non uniform. We thus demonstrate and evaluate a technique to estimate data density distribution and build efficient shortcuts in the overlay in order to reduce lookup latency. *We published these results in [SRDS11].*

- **Chapter 7: Scaling out virtual battlegrounds with in-memory databases.** This chapter presents a novel server-based architecture that improves horizontal scalability (*i.e.,* the ability to scale by adding additional hosts) of the game and eases dynamic player-management server placement. Our design is based on the observation that player management is much more resource consuming than data management. We decouple the two tasks by using a set of dedicated servers for the former, and an in-memory distributed database for the latter. As a result, the two components are able to scale out at their own rythm, avoiding unnecessary data-fragmentation, and player-management servers can be dynamically placed to fit to the load. We demonstrate the feasibility of our approach by adapting Quake III, a popular online game to our architecture. *This is an ongoing work planned to be submitted by the end of this year*.

- **Chapter 8: Conclusion.** To conclude our thesis, we provide an overview of the lessons learned from our work. We also detail the perspectives of our research advocating in favor of a better integration of MMOG workloads in the systems design.

Appendices

- **Chapter 9: Handling churn in DHTs.** This chapter presents Relax DHT, a churn resilient strategy for distributed hash tables. By relaxing placement of the data-replicas, our strategy induces less data transfers during churn phases, enabling the system to lose less data than other DHTs. The chapter presents a thourough evaluation of Relax DHT compared to state-of-the-art replication schemes. *This work has been published in [SSS09, TAAS11]*

- **Chapter 10: A key-value store for mixed workloads.** This chapter presents Jekyll and Hyde, an in-memory NoSQL database designed to efficiently handle mixed OLTP/OLAP workload. The key insight in Jekyll and Hyde is to use heterogeneous replication. A replica on a large server services random-access and transactional operations while a second replica partitioned across the smaller servers handles the scans.

Publications

International conferences

[SRDS11] Sergey Legtchenko, Sebastien Monnet, and Pierre Sens. DONUT: Building shortcuts in large-scale decentralized systems with heterogeneous peer distributions. In *30th Symposium on Reliable Distributed Systems (SRDS 2011)*, Madrid, Spain, October 2011. IEEE Computer Society.

[DSN10] Sergey Legtchenko, Sébastien Monnet, and Gael Thomas. Blue Banana: resilience to avatar mobility in distributed MMOGs. In *The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, Chicago, USA, July 2010.

[SSS09] Sergey Legtchenko, Sébastien Monnet, Pierre Sens, and Gilles Muller. Churn-resilient replication strategy for peer-to-peer distributed hash-tables. In *The 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*, volume 5873 of *Lecture Notes in Computer Science*, pages 485–499, Lyon, Fr, November 2009. Springer Verlag.

International journals

[TAAS10] Sergey Legtchenko, Sebastien Monnet, Pierre Sens, and Gilles Muller. RelaxDHT: a churn-resilient replication strategy for peer-to-peer distributed hash-tables. *TAAS*, 2011.

Chapter 2

Background

Contents

The design of a large-scale distributed system is a complex task. The two questions that a system designer needs to answer in the first place are (i) what architecture to choose? and (ii) how the data should be managed in the system?

The choice of the architecture determines the ability of the system to scale *horizontally*, *i.e.,* to be able to efficiently leverage the arrival of new hosts in the system. Driven by the popularity of peer-to-peer systems, a plethora of different topologies have been proposed over the last decade. Many of these architectural choices have proved their efficiency and are now reused in the design of new-generation distributed databases. The first section details the main proposals in the field peer-to-peer topologies.

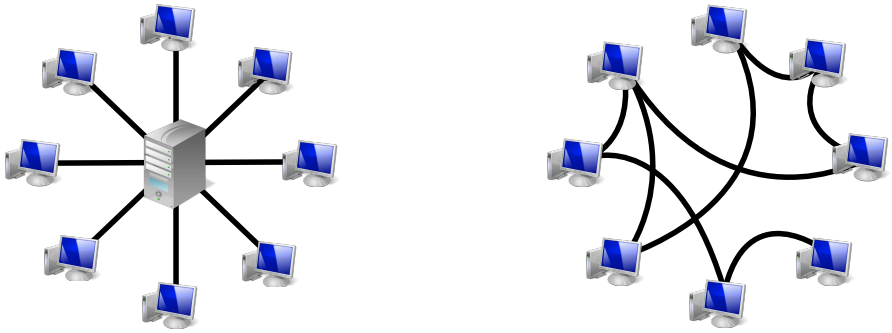
Described in the second section of this chapter, data management determines the ease at which the data will be processed in the system, and as a consequence, its ability to accomodate the workload imposed by the application. The designer should find the right trade-off between consistency guarantees, processing efficiency and fair load balancing.

The chapter is not a thorough survey on these two topics. The goal is rather to provide the non-expert reader with an adequate background to fully understand the rest of the thesis.

2.1 Peer-to-peer architectures

The traditional communication scheme in distributed systems is the client-server (c/s) model. In a basic c/s scheme, multiple client-hosts are connected to a single server-host. The server receives client queries and executes computational tasks on behalf of the clients (see figure 2.1.a). More elaborated c/s schemes include multi-tier infrastructures where servers can delegate tasks to other tiers, such as databases or backup servers.

In the peer-to-peer (p2p) paradigm, each host, called *peer*, is a client and a server at the same time. Services that were provided by the server in the c/s scheme are collectively provided by the peers of the system. A peer willing to access a remote service first needs to find the peer that provides that service. Peers are therefore logically interconnected, forming an *overlay* (see figure 2.1.b). At large-scale, each peer has only a partial knowledge of the overlay. The set of peers to which it is directly connected is called its *neighbor-set*.



(a) (b)

Figure 2.1: *(a) Client-server topology, (b) Example of peer-to-peer topology*

2.1.1 Motivation and challenges

Motivation of the peer-to-peer paradigm

The decentralized architecture of peer-to-peer overlays inherently demonstrates good scalability properties. Peers collectively support the applicative load, and each peer joining the overlay brings additional resources to the system. There is virtually no contention point, and unlimited scalability can be achieved if the resources are fairly spread across the overlay.

By removing servers, the overlay also greatly reduces the financial cost of the system at large scale. There is no need to invest in a large server infrastructure with expensive hardware to sustain the load of client queries. There is no need to maintain the infrastructure over time, because the overlay gracefully handles failures and elastically grows to meet applicative needs.

In other words, peer-to-peer overlays are cheap, robust, and are able to scale much better than c/s architectures. They open new perspectives to the design of large-scale data-intensive applications. Workloads that needed extremely expensive infrastructures with cutting-edge hardware can now be serviced by overlays created and maintained by small

companies, universities, non profit organizations or user communities. So far, peer-to-peer systems have been successfully used to process large amounts of data at a very low cost [9, 21, 34], geo-scale file sharing [79, 144, 139] or VoIP [41].

Challenges of peer-to-peer systems

The main challenges of peer-to-peer architectures are all inherent to the decentralized nature of these systems. We detail four of them:

- **Responsiveness:** to access a remote service, a peer first has to locate the peer responsible for that service. In other words, a query needs to be *routed* through the overlay. During the routing process, the query is forwarded to multiple peers through the network, which results in substantial latency overhead compared to a c/s scheme.
- **Churn:** In a p2p overlay, a peer typically belongs to a user. Therefore, when a user wants to access the shared resources, its peer joins the structure, and when a user has finished accessing the resources, its peer may leave the structure. At the system scale, this behavior results in continuous connections/disconnections of peers [186]. This phenomenon is called *churn* and is harmful to the overlay.
- **Heterogeneity:** The peers are usually spread across the globe. They have different operating systems and hardware characteristics. They are linked through a large panel of Internet connections with differents latencies and bandwidths [172]. Heterogeneity complicates the design and the maintenance of the overlays.
- **Consistency:** Many applications with consistency requirements need to store mutable data in the overlay. The overlay should guarantee that all the peers will see a consistent view of the data in presence of concurrent writes. Dealing with consistency in a fully decentralized context is a well-studied non-trivial problem and the mechanisms to address it may be costly in terms of latency [35, 92]. A widespread approach in peer-to-peer systems is to relax the consistency guarantees through best-effort techniques. The consistency requirements of distributed applications are detailed in the second part of this chapter.

A large number of p2p overlays have been proposed by the community over the past decade. In the next section, we provide a short overview of state-of-the-art overlays.

2.1.2 State-of-the-art overlays

We will classify peer-to-peer systems into three categories in function of the mechanisms they provide to lookup information from their overlay: unstructured, multi-tier, and key-based routing.

Unstructured overlays

In this category of overlays, the links between peers are chosen at random. The joining peer contacts a bootstrap peer randomly selected in a set of known overlay-peers. The bootstrap

peer initiates a search for overlay peers that have less in-links than a system fixed upper bound. The search is performed either with random walks or by flooding. At the end of the procedure, the joining peer receives a neighbor-set composed of totally decorrelated peers. The resulting overlay structure is close to a random graph [64].

The main representatives of this class of overlays are Gnutella [166] and Freenet [77]. The former was a very popular protocol in the early 2000's and was mostly used for file sharing. The latter was designed to provide anonymous large scale storage and is essentially focused on privacy issues.

Unstructured overlays are very well-suited to handle churn because random graphs have good connectivity properties. A disconnected neighbor can almost instantly be replaced by any other random peer of the graph, and a new arrival can be inserted almost anywhere in the overlay. On the other hand, unstructured overlays are not efficient in data retrieval, because there is no topological indication on where the resources are located. When a peer needs to retrieve information from the overlay, it has to flood the topology with limited-TTL search queries or use random walks [63]. In both cases, the process is expensive in terms of network usage and the result is not guaranteed. The query can be transmitted to hundreds of peers and the information may not be found even if it is actually stored somewhere in the overlay.

Multi-tier overlays

The second category is composed of p2p systems that do not have a completely symmetric architecture. Instead, they partially rely on a server for some operations, or elect super-peers to better manage the overlay. The first of this kind was Napster, an architecture for file-sharing that relied on a centralized index to map files to peers. Users first searched for the wanted file on the index server, and performed the data transfer directly from the storage peers. The index server was the main vulnerability of the system, and shutting it down disabled the whole network.

The FastTrack overlay (which included KaZaA, Grokster and Imesh) was a popular file-sharing platform with more than 3 million active users [139]. FastTrack is organized in a two-tier hierarchy consisting of super-peers in the upper tier and ordinary peers in the lower tier. The most powerful peers in terms of bandwidth and processing are elected as super-peers and are responsible for a cluster of ordinary peers. Each ordinary peer is linked to a unique super-peer, and super-peers form an unstructured overlay [139]. The file-to-peer index that was centralized in Napster is distributed across the super-peers in FastTrack. Ordinary peers searching for a file send the query to their super-peer, which returns matching rows in its index and forwards the query to several other super-peers[1]. By adding a super-peer layer to their topology, these overlays leverage peer heterogeneity to improve lookup performances compared to unstructured overlays. This concept has been also used in the Sun's JXTA framework, in which super-peers are called *Rendezvous peers*, ordinary peers being *Edge peers* [105].

Finally, Bittorrent [79], the most popular p2p file sharing protocol, is also based on an asymmetric architecture. Bittorrent is formed by many disjoint clustered components, called *swarms*, each one composed of peers interested in the same file. Each swarm is monitored

[1] A similar architecture was also proposed by late versions of Gnutella [162].

by a server, called *tracker*, that has a list of all the peers participating to the swarm. A user willing to download a file needs to download a metadata file (called *torrent*) containing the address of a tracker. Then, the user contacts the tracker and joins the swarm. The user can now download the file from the peers in the swarm.

Overlays for Key Based Routing

The overlays described in the two first categories are appropriate for file sharing. However, they are ill-suited for decentralized applications that require guarantees on data retrieval. Such applications want the overlay to *necessarily* locate the data if it is stored somewhere in the system. Moreover, since data retrieval is a frequent operation, the process should be scalable, *i.e.,* flooding the entire overlay is excluded.

To address these requirements, many peer-to-peer systems with non-random topologies have been proposed. Most of these systems rely on a mechanism called Key Based Routing (KBR [144, 163]) to efficiently lookup information. In KBR, resources and requests are assigned a *key* in an d-dimensional keyspace and each peer of the structured overlay is assigned an identifier in the same keyspace. Keys of resources and identifiers of peers are called their *coordinates* in the keyspace. The distance between any couple of keyspace-coordinates should be computable, *i.e.,* the keyspace must be a metric space. We name *keyspace distance* the euclidean distance between two coordinates of the keyspace. The peer which identifier is the closest to a resource key is called the *root* of the resource and is responsible for its management. Key Based Routing uses a decentralized greedy routing algorithm to guide lookup queries to their destination coordinate. The greedy routing algorithm always chooses as its next hop the peer that minimizes the distance to the target coordinate [54] and stops on the root of the target coordinate.

In order to implement KBR, the topology of the overlay should allow greedy routing to converge, *i.e.,* the final peer of the greedy hop-chain should always be the root of the target coordinate. This property is not verified for all the topologies and depends on the dimension of the keyspace used for routing [54, 43]. We call *KBR-overlays* decentralized systems which topology is suitable for KBR.

For single dimensional keyspaces, a ring overlay enables decentralized greedy routing. The ring topology is usually rewired with shortcuts for routing efficiency concerns. This topology, depicted in Figure 2.3 has been extensively used to design peer-to-peer systems with efficient routing [184, 169, 144, 130]. Kademlia and Overnet have been successfully used for filesharing by millions of people worldwide [130]. Pastry and Chord have been widely reused by the academic community.

Several topologies are known to provide greedy routing in multi-dimensional keyspaces. CAN [163] allows to build d-dimensional routable keyspaces. The CAN keyspace is divided amongst the peers currently in the system, each peer being responsible for a part of the keyspace. New peers join the overlay according to their coordinates in the keyspace: the zone that contains the coordinates of the joiner splits in two parts. The previous owner of the zone keeps one part, and the responsibility of the other part is given to the joiner. Each peer maintains links to its immediate neighbors in the coordinate space and, as shown is

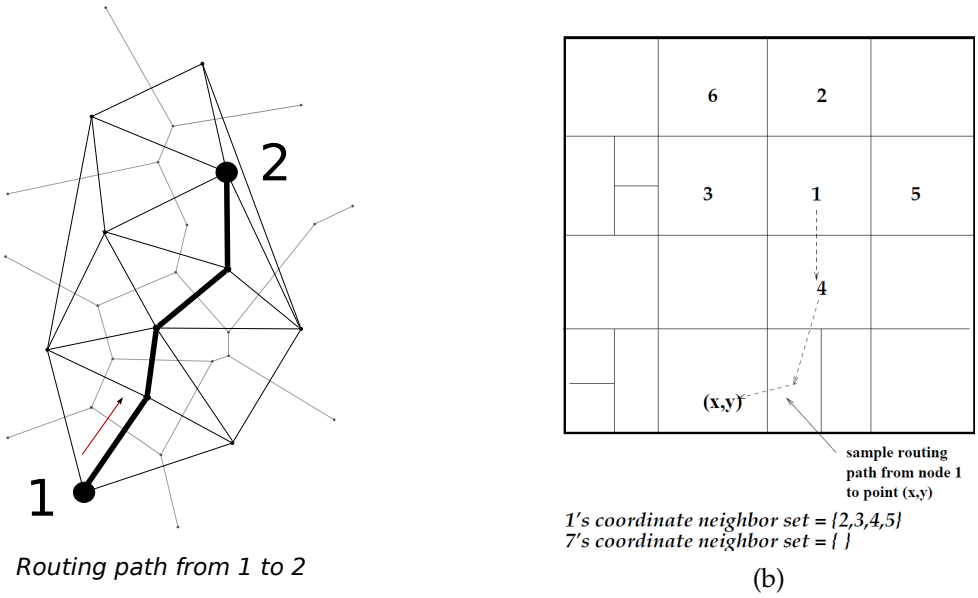


Figure 2.2: *(a) Delaunay/Voronoi topology, (b) CAN*

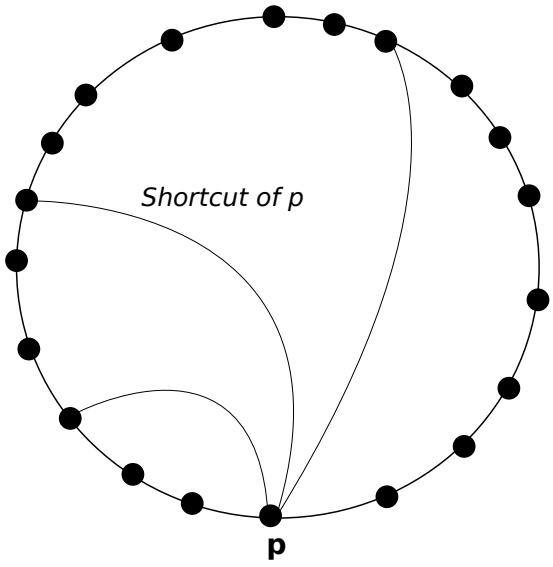


Figure 2.3: *Ring topology augmented with shortcut-links (only shortcuts of the peer* ***p*** *are represented).*

Figure 2.4.b[2], messages are forwarded to the immediate neighbor which is the closest to the destination cell.

Topologies based on Voronoi diagrams or Delaunay triangulations [114, 42, 140] are also able to support multi-dimensional greedy routing [54] (see figure 2.4.a). Over the past few

[2]*Data credit: S. Ratnasamy.*

years, two-dimensional voronoi-based overlays have progressively gained popularity in the MMOG community (as related in Chapter 3). Maintaining a Voronoi topology involves a high overhead if the dimension is larger than 2 [43], and a set of more lightweight topologies have been proposed [43, 53, 121].

2.1.3 Services on top of peer-to-peer topologies

Applications built on top of p2p systems access resources of the overlay by using system services. This section presents services that are used on top of p2p overlays.

- **Distributed Hash Table (DHT).** DHTs are storage systems that can be implemented on top of KBR-overlays. The storage information is split into blocks. Each data-block has a coordinate in the KBR keyspace and is stored on the root of that coordinate. DHTs provide a simple put/get interface to insert and access data. The data are usually made persistent by managing several replicas of each block. Depending on the design, the replicas are held on the neighbors of the root [170, 84] or on other peers of the system [163, 204]. During churn, blocks are lost, forcing the system to regenerate the information from the remaining replicas. Replication mechanisms are detailed in Chapter 9.

- **Range querying.** DHTs provide an exact-match interface to retrieve data: the user needs to know the coordinate of the block in the keyspace in order to retrieve it. Many applications need to specify a condition and retrieve a set of data-blocks that satisfy the given condition. For instance, the user of a file sharing system may need to retreive all the songs that a singer "foo" created between the years 2008 and 2010. The resulting query is:

  $get(\{file | file.type = song\ AND\ file.author = foo\ AND\ 2008 \leq file.year \leq 2010\})$

  The last condition in the expression targets a *range* of values, and the simple single-match interface of DHTs is not able to serve it. Such queries are called *range queries* and are supported by range-query overlays. Some of them are built on top of DHTs [161], others provide native support for range queries [48, 104].

- **Publish/subscribe (Pub/Sub)** is a communication pattern which allows to broadcast updates to users that share a common interest. Users subscribe to interest groups, and are notified of the updates that are "published" in these groups [65]. The publish/-subscribe communication pattern well matches interactions occuring within a social group, and is therefore widely used in distributed social networks and MMOGs.

- **Content dissemination and aggregation** allows all the peers to progressively gather/scatter some global information throughout the overlay. Usually, gossip algorithms are used for the propagation of information [198, 38]: each peer selects a set of peers throughout the overlay and transmits the disseminated information to them.

2.1.4 Leveraging topological properties to improve overlay performance

The ability of each system to offer efficient and robust services depends in the first place on the characteristics of its topology. This section details two main topological properties that improve performances of an overlay.

Small-world property for efficient routing

KBR guarantees that data lookup will succeed if at least one overlay-peer stores the data. However, the average number of hops necessary to reach the destination grows with the size of the overlay. To keep the system responsive even at geo-scale, the average routing complexity (in number of hops) should be less than $O(N)$, N being the number of peers in the overlay. For that purpose, most of the KBR-overlays need to maintain shortcuts in their graph.

The Small-World phenomenon was first described by Stanley Milgram in 1967 [145]. Milgram showed that there exists a short chain of acquaintances between any two people within the social graph of human societies. Small-world graphs have shortcut-edges that dramatically reduce the diameter of the graph. Watts and Strogatz suggested that the small-world property is very common among real-life graphs [199].

Yet, as pointed out by Kleinberg in [126], the fact that *there exist* topological shortcuts does not mean that a decentralized greedy routing algorithm will be able to exploit them. He proved that there exist a shortcut distribution that maximizes their usage by greedy routing. Formally, if the probability for a peer u to choose a peer v as a shortcut is $\frac{d(u,v)^{-r}}{\sum_{v \neq u} d(u,v)^{-r}}$, with $d(u,v)$ the graph distance (in number of hops) between u and v, and r the keyspace dimension, the obtained topology enables *polylogarithmic* greedy routing.

Many KBR-overlays use this property to enable scalable decentralized routing. Ring based systems, such as Pastry or Chord maintain a set of $log(n)$ long-range links that enable the small-world property, and thus provide logarithmic routing. Small-world rewiring techniques are also successfully used in multi-dimensional KBR overlays [42, 53].

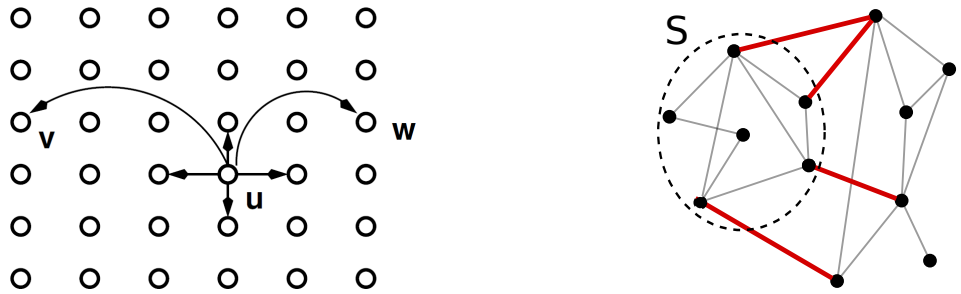


Figure 2.4: *(a) Kleinberg model: 2D lattice with shortcuts, (b) Expansion: Bold edges represent δS*

Enabling small-world property implies knowing the graph distance between peers. Small-world overlays, such as Pastry and Chord assign peer identifiers with a uniform hash function, making peers uniformly distributed through the keyspace. In that case the keyspace distance between peer identifiers is proportional to the graph distance between peers. However, such knowledge is not always straightforward to acquire. If peer identifiers are not uniformly distributed in the keyspace, the keyspace distance is no longer proportional to graph distance. In that case, peers needs to acquire information about the overlay

topology to compute graph distance. A detailed statement of this problem can be found in Chapter 6.

Relatively few overlays address the problem of routing in heterogeneous peer-distributions. VoroNet was one of the first proposals that described rewiring of nearest-neighbor graphs to enable the small-world property. VoroNet provides $O(log^2(N))$ routing, but does not use a dedicated mechanism to explicitly handle skewed distributions. Several systems are incorporating knowledge about the density distribution in their rewiring process. Mercury [48] and Oscar [104] probe the keyspace, making an approximation of its density.

Expansion property for sampling and churn resilience

A second topological property that impacts the performance of the overlay is the *graph expansion degree*. Intuitively, high expansion in a graph means that any two vertices are joined by a large number of disjoint paths. Formally, let $G = (V, E)$ be a graph with V the set of vertices and E the set of edges. Then, for all $S \subset V$, we define[3]:

1. the **Edge Boundary** of S, denoted δS, as $\delta S = E(S, \overline{S})$, with $E(S, \overline{S})$ the set of edges emanating from the set S to its compliment.
2. the edge **Expansion Ratio** of G, denoted $h(G)$, as: $h(G) = \min_{\{|S| \leq \frac{n}{2}\}} \frac{|\delta S|}{|S|}$

A graph with a low expansion ratio has clustered components that are loosely connected to the rest of the topology, which harms to the robustness of the distributed architecture. In case of churn or failures, the clustered components have higher probability to be disconnected from the topology.

Conversely, topologies with high expansion ratios better tolerate high churn rates, because there exist multiple disjoint paths between peers. For example, random graphs are known to have good expansion characteristics, which makes unstructured overlays very hard to partition. Tree topologies have very low expansion rates, and need to be rewired to tolerate churn.

2.2 Distributed data management

After the topological organization, the second important characteristic of distributed systems is how they manage the applicative data. Data management in a distributed context is a complex task. There are two fundamental questions that need to be answered to provide performant access to data.

1. What are the consistency guarantees that the system needs to provide?
2. How to distribute the data across the architecture to provide an efficient access for everyone?

Ideally, each entity of the system should have the illusion that all the needed data are stored locally and is only accessed sequentially.

[3] definition taken from [111].

2.2.1 Data integrity

Fundamentally, all data processing workloads are composed of two basic primitives: `read` and `write`. If the system allows only read access, *i.e.,* the data are not mutable, the consistency is guaranteed for everyone. The problem arises when write access is allowed. In that case, a host that reads data which is concurrently write-accessed may see the data in an inconsistent state, and two writes concurrently accessing the same data are conflicting. To address this problem, the distributed system should provide a *consistency model*. A consistency model is a set of rules on data processing operations that guarantees that the accessed data will remain in a predictable state. Depending on the guarantees provided by the chosen model, enforcing consistency has a cost for the system.

Linearizability [110], the strongest consistency model, is equivalent to non-concurrent execution and provides sequential access to data. However, linearizability has a high cost, because each operation needs to take a mutually exclusive lock on the data. Therefore, weaker consistency models were introduced to improve the concurrency of data processing for data-intensive workloads.

As an example, under *Snapshot Isolation* (SI), a read operation sees a consistent snapshot of the data as of the time the operation started, while concurrent writes modify the original data in a mutually exclusive fashion[4] [45]. SI allows to significantly speed up read operations, because they no longer conflict with writes, each read working on its own version of the data. A detailed description of existing consistency models can be found in [182].

Yet, even with a well defined consistency model for read and write operations, the integrity of the data can be violated. Indeed, for many applications, a data access is composed of several primitive operations. Such chains of reads and writes drive the data from one consistent state to another. If a chain breaks (*i.e.,* not all the operations succeed), the data may be left in an inconsistent state. The concept of *transaction* was introduced to address that issue. A transaction is a set of data processing operations that drive the data from one consistent state to another. Transactions should satisfy the so-called ACID properties, namely:

- Atomicity: Transactions are *indivisible*: either all the operations of the transaction succeed and the transaction *commits*, or the transaction fails. In case of a transaction failure, the data *rolls back* to the state in which it was before the beginning of the transaction.

- Consistency: The transaction leads the data from one valid (consistent) state to another. All the primitive operations of the transaction should follow the consistency model of the system.

- Isolation: Transactions should have disjoint executions. Two concurrent transactions with intersecting datasets should not harm data integrity. There are four isolation levels (from strongest to weakest), depending on how locks on data are managed (see [45] for details).

- Durability: The state of the data resulting from a committed transaction should remain available after the commit, regardless of system failures.

[4] SI is a type of Multiversion Concurrency Control (MVCC).

Transactional processing provides data integrity with concurrent read/write access. The different consistency models and isolation levels should be carefully chosen to provide good performance for data-intensive applications.

2.2.2 Data placement

The second challenge of distributed data management is the placement of the data. For efficient access, the data should be located close to where it is processed (we call it *preseving data locality*). Moreover, the system has to provide a fair load balancing to avoid bottlenecks in the architecture. It is a known fact that real-world data are most of the time non-uniformly distributed and has non-uniform popularity [137]. Because of that, preserving locality and load balancing in distributed systems often have opposite requirements, and efficiently handling both is a complex problem. This section describes different data placement techniques.

Partitioning through hashing

Uniform hashing, the most used data placement technique was introduced by DHTs. The data are organized in blocks and the identifier of each block is generated as a hash of its content. Upon join, peers receive an identifier generated by the same hash function[5]. As a result, peers and blocks are uniformly distributed across the hash keyspace, and each peer stores approximately the same amount of blocks. By definition, the hash-identifier of a block is totally decorrelated from its content, which ensures uniform distribution of new blocks, but totally breaks data locality: blocks with almost identical content are likely to have completely unrelated identifiers.

In practice, this technique is efficient for load balancing and is widely used in both large-scale peer-to-peer overlays [170, 204] and distributed databases [152, 197]. However, it provides no native support for complex queries, *e.g.*, range querying, because two semantically close blocks can be stored far from each other in the overlay. Range querying with this placement technique is not efficient because many hosts should be contacted to retrieve a semantic data-range, and an additional index structure has to be maintained [161].

Locality-aware partitioning

The locality-aware partitioning is used to provide efficient native support for range queries. Here, the data are placed according to some application-dependent coordinate. As an example, an MMOG could choose to place the objects of its NVE according to their (x,y,z) coordinates. In that case, objects that are close to each other inside the NVE will be stored on the same host and a range query will access only a limited number of hosts.

In practice, this technique allows very efficient data processing, because many applications access semantically related objects together. In our example, if two objects are located close to each other, a player that fetches the first object will also access the second with very high probability. Thanks to that, the technique is widely used in MMOGs. However, since the data distribution is skewed, hosts storing dense areas is likely to be overloaded.

[5] *e.g.*, SHA-1 [58], md5 [167]

Therefore, additional load-balancing mechanisms need to be provided with this partitioning scheme.

Replication

In order to provide data locality load balancing and fault tolerance, replication techniques can be used [175, 179, 200]. In that case, several copies of the data are stored in the system and users indifferently access any of the copies for reads. Updates need to be propagated to all the copies in order to guarantee consistent access to data. Therefore, replication is relevant for workloads dominated by reads, and is successfully used in large-scale overlays, such as PAST [170] and distributed databases [176, 156]. The important challenge is then to maintain all the replicas in a consistent state [200].

2.3 Conclusion

In this chapter, we described two main challenges of large-scale distributed systems: design of the distributed architecture and data management. Many of the solutions addressing these challenges were first proposed in the context of peer-to-peer overlays. Yet, with the growing need for data-intensive applications to scale out, these solutions are progressively being adapted to datacenter-scale architectures and are already demonstrating their benefits. As a conclusion to this chapter, we classified some of the state-of-the-art systems (both peer-to-peer and datacenter-scale) according to the solutions they bring to these challenges (see table 2.1). The choice to use one system over another should be dictated by the characteristics of the applicative workload it is intended to support.

		Data placement		
		Uniform Hashing	*Locality preserving*	*Random*
Overlay topology	*Unstructured*	-	MOve [149], Freenet [77]	Gnutella [166]
	Multi-tier	VoltDB [197], SILT [141]	Bittorrent [79]	FastTrack [139]
	1-dim KBR	Pastry [169], Chord [184], Cassandra [132], Dynamo [88]	Mercury [48], Oscar [104], P-Grid/Gridella [174]	-
	n-dim KBR	-	CAN [163], VON [114], RayNet [43], VoroNet [42], Solipsis [121], Oscar [104]	-

Table 2.1: *Existing distributed systems ordered according to their topology and data placement characteristics*

Chapter 3

State of the art

Contents

This chapter presents the state of the art in the field of MMOG design. In order to better understand MMOG requirements, we first describe the characteristics of MMOG workloads. Section 3.2 provides a thorough overview of existing MMOG architectures. The last part of this chapter is dedicated to systems that exploit properties of player behavior to improve their efficiency.

3.1 Analysis of MMOG workloads

3.1.1 Workload taxonomy

Modern online games form a vast ecosystem of applications designed to fulfill a large panel of gaming experiences. Some MMOGs are intended to give the illusion of life, providing

complex, highly interactive environments [19, 1, 22]. Others are only large arenas for fast-paced combat [24, 17, 16]. From the systems point of view, there are two main differences between all these applications:

1. **The amount of mutable content offered by the environment.** Handling read-only data is cheap and easy because it can be stored client-side. In contrast, access to mutable data is provided by the infrastructure, which should guarantee its availability and consistency in presence of concurrent updates. The amount of mutable data therefore has a direct impact on the bandwidth requirements of the workload.

2. **The responsiveness requirements of the MMOG.** The speed and the precision required by the gameplay depend on the actions the gamer is supposed to accomplish. Some actions, such as combat with precision weapons, require much more responsiveness than simple exploration of the environment [99, 78]. The latency that needs to be guaranteed by the architecture on update propagation thus directly depends on the type of gameplay.

In this thesis, we are considering three main classes of MMOGs each one having specific bandwidth and latency requirements:

- Massively Multiplayer Online First Person Shooters (*MMOFPS*) are large-scale virtual battlegrounds. The main activity of MMOFPS players is fast-paced action: players demonstrate their skills fighting each other[1]. Only the state of the players and a few other objects (*e.g., ammunition, medikits...*) are mutable. The rest of the environment is read-only and can be stored client-side. MMOFPSs have the lowest proportion of mutable data, but the tightest responsiveness requirements. Representative online games belonging to that class are PlanetSide 1 & 2 [16, 17] and World War II Online [24]. Popular deathmatch online games such as Counter-Strike [5] or Quake III Arena [18] are not strictly related to that class because of their small scale: they support at most 64 players per server. However, apart from the population size, they exhibit all the characteristics of MMOFPS. Therefore, we consider these games as MMOFPS in the rest of this thesis.

- The most popular class of MMOGs, called Massively Multiplayer Online Role Playing Games, or MMORPGs, offers a richer environment, with Non-player characters (NPCs) and complex avatar management. The player is able to interact with NPCs, manage its inventory and improve its avatar's characteristics. Still, the largest amount of data, such as object textures, cannot be modified and is available client-side. MMORPGs have a slightly higher amount of mutable objects that MMOFPSs, but have lower responsiveness requirements. Among many others, World of Warcraft [25], Aion [2] and EVE Online [7] belong to that class.

- Finally, Metaverses are the most complex virtual worlds since they offer a fully mutable environment to the players. The creation of new objects including new shapes and textures is allowed, resulting in a rich, highly interactive environment. The environment is populated with user-made objects, and the architecture needs to propagate full

[1] MMOFPS are notably used for cybersport [6]

description of the objects to the players and guarantee their consistency in presence of concurrent modifications. Fortunately, Metaverses are mostly used for social activities and therefore have the most relaxed responsiveness requirements. The most known Metaverse is Second Life [19]. Other representatives of this class are There [22], Active Worlds [1] and OpenSimulator [15].

The class of the MMOG defines the *bandwidth* and *latency* characteristics of its workload. A study of Counter-Strike, one of the most popular MMOFPS, reports that the downlink bandwidth requirements at the client are of about 34 kbit/s with latency required to be below 150 ms for correct gameplay [94]. In contrast, the Second Life metaverse requires up to 400 kbit/s but supports latencies higher than 500ms [96, 78]. MMORPGs seem to have the least tight requirements, supporting latencies between 500 ms and 1 s and using rarely more than 40 kbit/s of downlink bandwidth per client [189, 99].

3.1.2 Impact of player behavior on the workload

One essential property of MMOGs differentiates them from other applications. MMOGs do not provide a set of services to fill a particular customer need, but are designed to be *alternative realities*. The subtle difference lies in the fact that unlike ordinary customers, players are supposed to identify themselves with their avatar in the game. In many aspects, MMOGs are inspired by the real world. They offer a tridimensional environment and accurately simulate real-world physical behavior of objects (*e.g.,* gravity, ballistics, etc...). Players are therefore much more "linked" with the application, and their in-game behavior is similar to the one they have in the real world [131, 165]. For each player, interactions with other players and with the environment are dictated by its human nature. As a consequence, characteristics of human behavior greatly influence the MMOG workload. In the rest of this section, we will describe the behavioral trends of the players and their impact on the workload.

Impact of player perception on update propagation

The environment perception of a human being is constrainted by his physiological abilities and by the geometric properties of the environment. Distant objects in the real world are perceived with less details than closer objects, and their solid angle, *i.e., perceived* size appears to be smaller. Human beings can simultaneously focus on only a very small set of objects, typically seven [145]. This means that a player only needs to receive updates for a small set of NVE-objects, and only a few of them require *frequent* updates. The set of objects a player is interested in is called *Area of Interest* (AOI) [55, 150] of the player. The process of sending to the player only updates for objects in its AOI is called *AOI-filtering*. Efficient AOI-filtering techniques can dramatically reduce the load of the system. A detailed overview of AOI management techniques exploiting player behavior is given in section 3.3.

Impact of density distribution on load balancing

The distribution of objects across the MMOG is extremely non-uniform [196, 157, 158, 147]. For instance, an analysis of Second Life showed that nearly 30% of the monitored regions

contained almost no objects during the 4 weeks of the study. 65% of regions contained relatively few objects, whereas only 5% of the regions had high object densities [196]. The player distribution is even more skewed, with avatars forming highly clustered groups and leaving the rest of the NVE almost empty. The same study shows that 45% of the regions contained no avatars at all, whereas only 2% of the regions contained more than 20 avatars. The avatar groups were mostly formed by 2 to 10 avatars, with a negligible amount of very large groups. The presence of a few large groups is explained by specific points of interest, such as music concerts or art exhibitions, whereas small groups can form anywhere. The study also shows that the volatility of avatar clusters depends on their size: small groups tend to disappear more rapidly, while large groups are very persistent. Object distributions have been shown to be much more static, with 50% of the regions having almost completely static objects.

These observations are confirmed by the studies of other popular MMOGs. Monitoring of World of Warcraft realms showed that 75% of the regions had less than 10 players, while only 5% of them contained more than 40, with a maximum of more than 290 [157, 158]. The authors suggest that the distribution could be approximated by a power law. A study of Quake III, a popular first person shooter, shows a Zipf's law [205] distribution in region popularity [49]. Real-life population distribution is also known to be highly non-uniform [183]. While having small quantitative differences, all these studies suggest that the *qualitative* characteristics of the avatar distribution in MMOGs is not specific to one particular game, but is instead inherent to human behavior. These characteristics have a significant impact on the MMOG workload. The heterogeneity of avatar distribution implies that some regions will be extremely crowded, overloading the region infrastructure, whereas others will be completely empty. In addition to that, since almost half of the population forms small, volatile clusters, the load is likely to greatly vary over time. Finally, players entering a densely populated area will have more objects in their AOI, thus generating more incoming traffic. The heterogeneity of objects distribution means that different regions have different storage capacity requirements. Moreover, since the avatar distribution is skewed, the object popularity is highly non uniform, and some of them will be accessed orders of magnitude more than others.

Impact of player movement on neighbor discovery

Player mobility inside MMOGs has been shown to demonstrate similarities with real life movements of living creatures [131, 165]. Players tend to have slow, chaotic movements in crowded areas, and fast, straight trajectories in desert zones. Miller and Crowcroft show that the analysis of avatar mobility was unable to identify clear points in the space common to all the trajectories, making waypoint modeling is unsuitable to approximate avatar movements [147]. It is known that player mobility patterns can be modeled with Levy flights [131, 165]. Levy flights are particular sort of random walks in which the increments are distributed according to a "heavy-tailed" probability distribution [72] with short and chaotic movement and sometimes long and straight ones. Such mobility patterns have also been modelized by Markov chains [188, 158, 100] and Weibull distributions [95].

Here again, the abovementioned studies suggest only quantitative variations in avatar mobility between the existing MMOGs. The fundamental properties, such as exploration phases followed by fast movement phases, or the chaotic trajectory in crowded areas are however common to all the studies and similar to real-life behavior. These patterns can how-

ever be broadly impacted by the rules of gameplay imposed by the MMOG, such as group quests [28], or player versus player fights [29]. To accurately model such game dependent behavior, generators were proposed by the community [188].

Player mobility patterns have an impact on how the player accesses the data. Indeed, a player constantly changes its AOI as long as it moves, and the system needs to "feed" its AOI with new objects along its trajectory. The feed-rate depends on the speed of the avatar, and its trajectory determines what objects will be accessed. Some mobility patterns, such as crowd mobility, or very fast movement can produce extremely data-intensive workflows harming the infrastructure [37].

Impact of player volatility on churn tolerance

Several studies focused on the characterization of game usage for popular MMOGs by tracking connections and disconnections of the players. The results show clear daily churn patterns [95, 67, 135]: the game server population significantly varies over time, with a population peak at the end of the day and on weekends. The arrival/departure rates and population size varies by a factor of at least 4 during the day [95]. The measurements also show a strong correlation between the evolutions of different MMOGs over time (see figure 3.1): regardless of the observed MMOG, players all join at the same periods of the day.

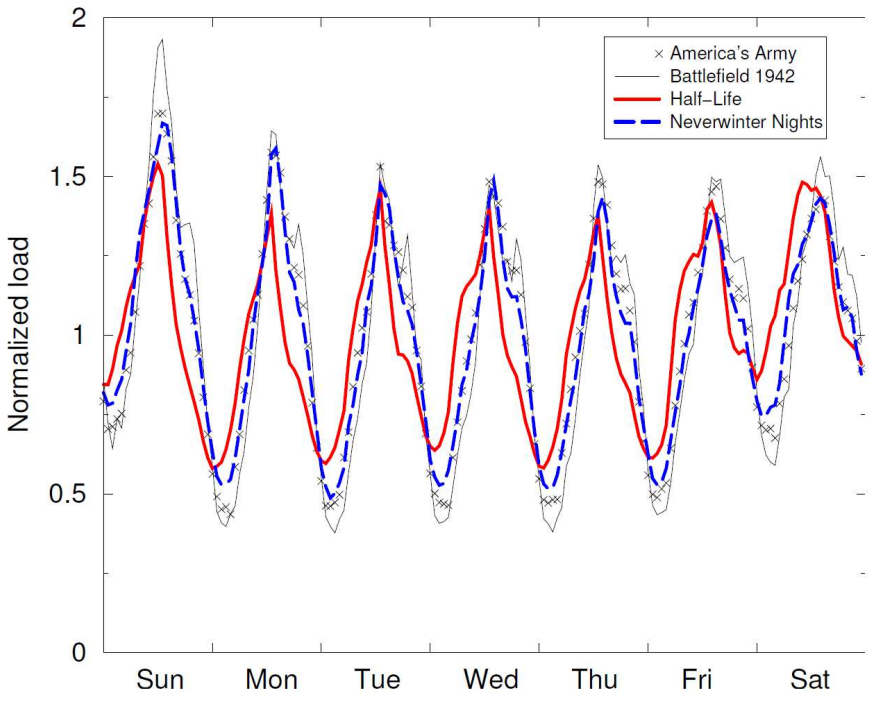


Figure 3.1: *Aggregate normalized load across four popular games for week of May 23, 2004 (source: Chambers et al. [67]). Neverwinter Nights is an MMORPG, while the others are FPSs.*

A study of World of Warcraft shows that 75% of the gamers play longer than 1.9 hours per day, and 25% played more than 4.9 hours per day, testifying of a strong addiction to the game [135]. Alternatively, in their study of popular MMOFPS, Chambers *et al.* describe

the gamers as impatient and not loyal [67]. They show that unlike for other applications, gamers hardly try to reconnect to a busy server, with 73% of them unwilling to retry even once. Gamers are also often change servers, with 42% returning only once, and 81% less than ten times to a server on which they played before. Finally, they show that the game popularity follows a power-law distribution.

The described game usage patterns are a consequence of human behavior: gamers tend to play *together* during their free time, and hardly endure delays induced by overload servers. Such characteristics point out a strong need for MMOG architectures to scale out on demand. A variation of more than 4 times in the game population over the day is clearly challenging to provision, and any delay due to lack of system adaptation potentially implies the loss of a customer.

3.1.3 On the predictability of player behavior

One important property of any workload is its *predictability*. Is it possible to foresee its evolution over time? The question is crucial, because if it is in fact predictable, it is possible to better adapt the architecture to fit with the load. Some of them, *e.g.,* OLTP, are known to be hardly predictable and are qualified of being random. In the case of MMOGs however, all the studies point out that their workloads are predictable at short to mid term.

For instance, the periodic variations in the player population happen in all the observed MMOGs and are thus clearly predictable [95, 67, 135]. Furthermore, Chambers *et al.* show that the interest of players for an online game is correlated to the evolution of their session times [67]. Therefore, it is possible to predict what players are about to abandon the game by looking at their play history. It implies that the overall population evolution can be rather accurately predicted at short (daily) to mid (global game popularity) term.

Regarding the distribution of the players in the MMOG, it is known that the durability of the hotspots depends on their size. The density hotspots also tend to form around *points of interest* in the game [196]. It is thus possible to approximately predict *where* large aggregations of load will arise, and *for how long* they will last.

Finally, the characteristics of individual players also exhibit some well predictable patterns. It is for instance known that there is a spatio-temporal locality in data-access history of a player, which means that a player has a good chance to access an object if it is close to him and already has been accessed by him in the past [68]. In addition to that, player's trajectory can also be predicted at short term [154, 201]. It means that the objects that are about to be accessed by player within a few seconds can be known in advance with a good probability.

Discussion

MMOG workloads can be qualified as extremely complex and system-unfriendly. All the measured workload aspects, such as player distribution, movements, interest or session times, were described as highly heterogeneous, often modelizable by a power-law.

The abovementioned studies tend to show that in-game player behavior has a significant impact on MMOG systems. Thanks to extensive research in the field of player behavior analysis, there is enough evidence to say that some of the evolutions of MMOG workloads can be anticipated. Although the precise behavior of each particular player is hardly predictable at

long term, global characteristics of player populations follow well identified patterns. This predictibility can be used to help MMOG architectures to accommodate their unfriendly workload.

3.2 Architectures for MMOGs

This section describes the state of the art of academic and commercial architectures for MMOGs. Three main categories are to be detailed:

- **Server based architectures:** the applicative load is handled by a set of servers (Fig. 3.2.a).
- **Peer-to-peer architectures:** the clients of the application collectively share the load in a fully decentralized fashion. We describe two families of overlays: multi-tier-based, with super-peers handling most of the load (Fig. 3.2.b), and nearest-neighbor overlays, where all the peers have the same role (Fig. 3.2.c).
- **Hybrid architectures:** servers are used for critical operations only, other events are directly propagated between the clients (Fig. 3.2.d).

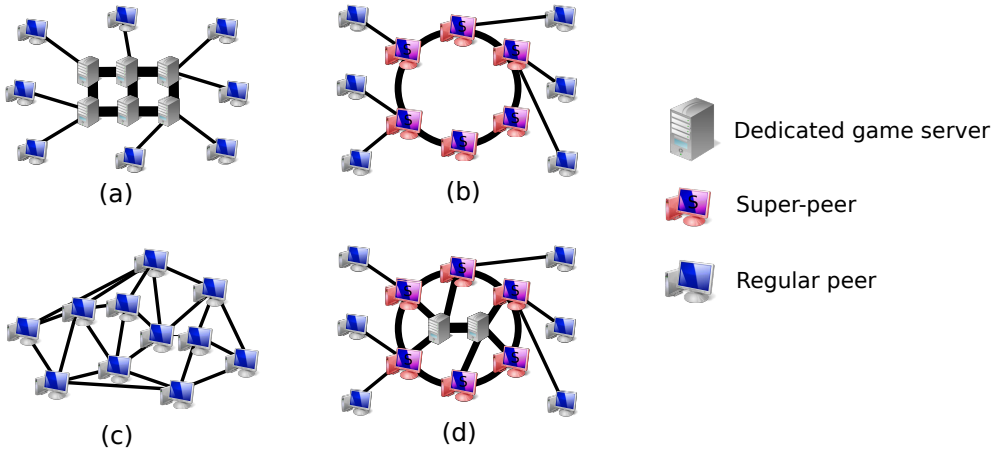


Figure 3.2: *Main topologies for NVEs: (a) pure c/s; (b) multi-tier p2p; (c) nearest-neighbor p2p; (d) hybrid.*

3.2.1 Server-based MMOGs

Server based MMOGs are a logical evolution of ordinary multiplayer games. A set of authoritative servers managed by the game provider hosts the game state, serializes updates and propagates consistent snapshots of the game to each player. Servers have good hardware characteristics and are hosted in a trusted environment interconnected by high bandwidth and low latency network.

The main advantage of these architectures is the simplicity of their design. Persistent storage and responsiveness in datacenter-scale environments are well-known and addressed problems [153, 141, 88]. The homogeneous, trusted environment guarantees a low failure rate and the absence of malicious behavior in the core architecture. Finally, the game provider keeps a hand on the game content, which is consistent with the business model of current MMOGs.

Commercial MMOGs

All the commercially successful MMOGs have a server based architecture [25, 2, 11, 7, 19]. They rely on a combination of sharding and zoning to partition the NVE across the server farm.

Second Life, one of the most popular NVEs, is composed of several shards (called *grids*). Each shard is split in square regions of size $256 \times 256\ m^2$, each one being hosted by a single process (called *sim*). Within a grid, the environment is claimed to be seamless, because avatars are able to transparently walk across regions. Each sim is connected to neighboring sims, and transparent handover is supported. However, the population of each sim is limited to 100 avatars and each one can easily get overloaded with as little as 20 concurrent avatars [195].

EVE online, a space opera environment offers an unsharded universe dubbed Tranquility, that support 40K concurrent users. The load of Tranquility is divided across a hundred servers interconnected by low-latency Infiniband network [8]. Here again, agglomerations of more than a few hundreds players within a region hosted on one server leads to an unplayable gamestate and potential server crash [8].

World of Warcraft, the most popular MMOG at this time with 62% of the total MMOG market share in 2010 [13], makes extensive use of sharding. Each shard, called *Realm* can handle several thousands of players, but fails to handle massive crowdings. In special zones where collaborative fights are allowed (*e.g,* dungeons), the architecture enables groups of tens of players to be separated from others and run a private instance of the game. This technique is called *instancing* and provides acceptable gameplay in crowded places, at the cost of splitting the population [27].

The main drawback of these architectures is their lack of flexibility. The resource allocation is static, and variations of population size and distribution lead to poor use of the available ressources. The academic research on server-based architectures therefore focuses on flexible load balancing techniques and efficient consistency mechanisms[2].

Dynamic load-balancing

The main load-balancing technique called *dynamic zoning*, consists of dynamically partitioning the NVE into regions in function of the load. The responsibility for each region is then dynamically affected to the servers. The optimal partitioning problem in NVEs is known to be NP-Complete [142]. Nevertheless, several dynamic zoning techniques were proposed to cope with non uniform distributions of players inside the environment. SimMUD splits the NVE into square regions [127] and attempts to load balance the regions across the servers

[2] AOI management, another important mechanism of NVE architectures, is described in section 3.3

while keeping contiguous regions on the same server. Depending on the load, a server is responsible for several regions, and the system provides two primitives, aggregation and shedding, to balance the load. SimMUD monitors the load inside each region and if the load exceeds a given threshold, shedding occurs and the region is migrated to another server. Inversely, if the system detects that there is excessive overhead due to communication between two regions of different servers, it triggers aggregation, and the two regions are migrated on the same host. Evaluations of SimMUD show that this dynamic scheme performs 8 times better than a static partitioning if the distribution of players is non-uniform. Similarily, De Vleeschauwer *et al.* propose to discretize the game space into microcells, each server hosting a set of contiguous microcells [87]. The paper investigates different microcell placement algorithms, including locality-oblivious load balancing and several locality-aware techniques. The microcell partitioning is found to be much less sensitive to the player distribution than static partitioning, and reduces the load by at least 30%.

Mammoth, an experimental NVE uses Delaunay triangulations to divide the game space with respect of the game landscape [177]. Each server is responsible for the management of polygons formed by contiguous triangles of the triangulation. Overloaded servers shed triangular areas, achieving fair load-balancing.

ALVIC-NG uses a quadtree to subdivide the environment into cells according to the load induced by players. The nodes of the quadtree represent square regions of the game space. A region can be split into four square subregions, resulting in the creation of four children on the corresponding quadtree node (see Figure 3.3). The system has a 3-layer architecture: a Client Proxy propagates updates to clients, Logic Servers handle NVE regions and a Region Management (RM) service associates regions to Logic Servers. For a Client Proxy, the RM service acts as a DNS: given the player's AOI, it finds Logic Servers that handle the regions intersecting with the AOI. Simulations show that ALVIC-NG has good scalability properties and low latency. A similar design has been adopted by Sirikata [75]. Instead of a quadtree, Sirikata uses a distributed kd-tree to map regions to servers. Lee and Chen propose to encapsulate each region management in a virtual machine and use virtual machine migration strategies to dynamically balance the load of the NVE [134].

Alternatively, other systems investigate *replication* techniques to share the applicative load. MiddleSIR uses replication to improve scalability and reliability of the game server. Optimistic and pessimistic replication techniques are compared. The results show that replication provides good fault tolerance, but has limited scalability, because all the game state updates should be synchroneously executed on all the server replicas. Server replication has been successfully applied to Quake III, a popular online game and improves overall scalability of the game without degrading responsiveness [159]. However, the scale of replicated Quake III is limited to a few hundred players at most. Cronin *et al.* use a *Mirrored server architecture* which consists of several replicated game servers [82]. Each server holds a copy of the game state, and the copies are synchronized by an algorithm called *TSS*. Client connections are equally spread among the replicated servers, thus ensuring fair load balancing.

Finally, Darkstar, the Sun game server technology [50] investigates the possibility of using one large database to store the game state. The NVE is run by multiple game servers, which write all the updates to the database. The updates are retrieved from the database and propagated to the clients. To avoid costly database accesses where possible, the servers act as a cache: they store frequently accessed data as long as it have not been updated by one of the servers. In case of a data modification, the cached data are invalidated on all the

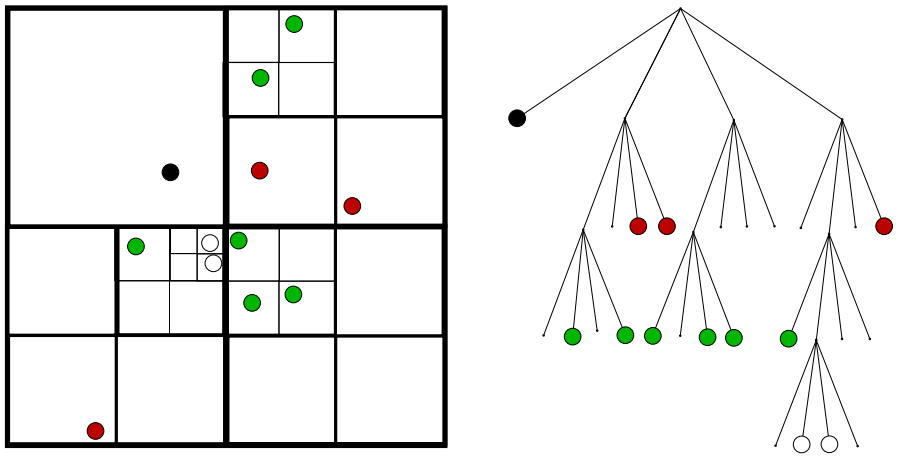


Figure 3.3: *Subdivision resulting from a non uniform peer-distribution and the corresponding quadtree structure.*

servers and a new database access is required to access it.

Dynamic resource management allows flexible load balancing, improving the system's ability to cope with non-uniform load distributions. Moreover, such systems have good responsiveness, because communication between infrastructure components is cheap.

Consistency concerns

All the servers of such NVE architectures collectively maintain a consistent game state. A consistency problem arises when inter-server consistency needs to be ensured. Systems relying on caching techniques, such as Darkstar need to provide an efficient cache invalidation protocol to cope with updates. Replicated architectures must implement a synchronization protocol that guarantees that updates will be executed consistently on each server. The cost of the consistency mechanisms is high and often limits the scale of these systems, because each update triggers the consistency mechanism.

In case of dynamic zoning, updates that concern data located within a region are cheap because they can be locally serialized by the server hosting the region. The distributed consistency mechanism only needs to be executed on updates that span multiple regions held by different hosts. Inspired by the database community, there are three popular classes of consistency protocols for NVEs [109]. The first one is formed by Lock Based Protocols. In order to read or update an object, a lock should be acquired for that object, typically by contacting a lock server. Assiotis and Tzanov extend the concept of object locks to regions [36]. *Region locks* are taken to execute distributed updates that span several regions of the NVE. The second family of consistency protocols is the Timestamp Based Protocols. This is a more

optimistic approach, in which updates on objects result in new, timestamped versions of the objects. The versions form a multiversion history for all the objects, and conflicts are resolved by the consistency server [178]. The third family of protocols is based on Object Ownership. In these protocols, each object is *owned* by a host, and updates on the object can only be applied by the owner. Other hosts are allowed to read the object state, but not modify it [109].

Finally, Gupta *et al.* propose a new family of protocols called *action based protocols* [109]. In these protocols, each host holds the complete game state. Actions performed by the hosts are optimistically sent to an authoritative server, which serializes the actions and broadcasts the serialized history. Hosts reconcile their optimistic actions with the received history, rolling back and re-applying failed operations.

3.2.2 Peer-to-peer MMOGs

Peer-to-peer MMOGs are an innovative alternative to server-based architectures. The advantages peer-to-peer based architectures are twofold. First, they have good scalability properties and built-in fault tolerance mechanisms. Second, a peer-to-peer design discards the cost of maintaining a server infrastructure, which is known to represent up to 80% of the MMOG revenue [123]. The adepts of the peer-to-peer approach advocate that it can be used to create low-cost geo-scale NVEs free from the influence of an infrastructure provider. Such NVEs would belong to the players, collaboratively managed by players, with player-created content.

However, the five fundamental requirements of MMOG systems related previously are hard to accommodate in a fully decentralized context. Three main problems are to be addressed:

1. The **neighbor discovery problem**. The architecture should provide an efficient mechanism to discover entities inside the AOI of a player.

2. The **connectivity problem**. There should always be a path between any two players of the NVE through the overlay graph. Failures and churn should not partition the NVE.

3. The **data availability problem**, which consists in guaranteeing a *persistent* and *consistent* access to data despite churn.

An extensive research has been performed in peer-to-peer MMOGs to address these problems. The design of state-of-the-art architectures follows two main directions. (i) *The multi-tier approach* that solves the three problems by forming an overlay of super-peers that handle the neighbor discovery and the data availiability. The NVE is partitioned into regions and a super-peer is selected to handle each region. The approach is an adaptation of the server based design to the peer-to-peer paradigm. (ii) The *nearest-neighbor approach*, that focuses on efficient neighbor discovery. Overlays using this approach often rely on Voronoi tesselations/Delaunay triangulations to provide natural connectivity with the peers located in the AOI, natively enabling efficient localized multicast.

The multi-tier approach

The idea of the multi-tier approach is to replace the NVE servers by super-peers selected among the player hosts. Similarily to the server-based approach, the NVE is split in regions, each one being managed by a super-peer, often called *coordinator* of the region. Coordinators handle storage, consistency and interest management of players inside their region, providing a solution to the neighbor discovery problem. However, unlike dedicated servers, the coordinators cannot be fully trusted, because they belong to users and are thus subject to churn or malicious behavior. The main challenge of this approach is to choose reliable coordinators in a decentralized context and provide persistent storage in presence of churn. The key component of such overlays is thus a decentralized mechanism that assigns NVE-regions to coordinators, and maintains the mapping between regions and coordinators. Players that join an NVE-region locate the corresponding coordinator through the distributed mapping structure and open a direct connection to the coordinator. After that, the coordinator propagates information about objects and avatars to the players inside its region. Neighboring players then interact through direct links, or ask the coordinator to arbitrate critical state modifications (*e.g.*, player's death).

The mapping mechanism is usually implemented on top of a DHT. Iimura *et al.* use Pastry to assign a region to its coordinator [116]. An identifier is computed for each region by hashing region metadata. The root of this identifier in the DHT becomes the coordinator of the region. Player peers access the metadata of the region they want to join, and a request for coordinator is performed by a Pastry-lookup with the hash identifier of the region. The coordinator uses direct links to propagate game updates to the players inside the region. Knutsson *et al.* adopt a similar scheme, but use Scribe[3] for state propagation: the coordinator is also the root of the Scribe multicast tree for the corresponding region. The two systems do not allow inter-region interactions, because game update multicast is implemented *per region*, resulting in an artificially partitioned environment. MOPAR improves the scheme by allowing inter-region communication [202]. In MOPAR, the NVE is divided into hexagonal cells, and coordinators are assigned to cells through the Pastry DHT. When a player joins an empty cell by querying the DHT, its peer becomes the *master* of the cell. It creates direct links to the masters of neighboring cells and retrieves information about all the neighboring players. The masters are used for neighborhood dissemination: they inform their players that someone entered their AOI, providing a continuous view of the NVE neighbors. All of these systems provide a *static* partitioning of the NVE, and therefore exhibit the same lack of flexibility that is characteristic to server-based games. The problem is aggravated by the fact that coordinators are untrusted peers with limited capacity.

Like for server-based architectures, one solution to address this problem is to provide dynamic partitioning. The dynamic creation of NVE regions according to the system load implies that their metadata, and therefore their hash-identifier is not known in advance. A more elaborate mapping mechanism is needed. To address this problem, Douglas *et al.* propose to implement a quadtree on top of the Chord DHT to dynamically partition the NVE [89]. They use the quadtree to compute *control points* in the game space. Control points are the middle points of the quadtree regions and uniquely identify the regions. The coordinates of the control points are hashed and stored in Chord. However, in presence of non-uniform distributions, resulting in deep unbalanced trees, a spatial search may result in many DHT

[3]Scribe is designed for publish/subscribe and based on Pastry [65].

accesses, and the evaluation of this mechanism is unfortunately not provided. Varvello *et al.* employ a dynamic partitioning technique inspired by CAN [163] and PHT [161] based on prefix tree indexing [193]. As in CAN, the NVE is recursively divided in regions: when the load of a region exceeds a threshold, the region is split in two sub-regions. The resulting partition can be represented by a prefix trie, with regions stored on the leaves. The nodes of the tree have hierarchically computed identifiers, so that the identifiers of the children are derived from the identifier of the parent. Nodes contain a flag indicating whether they are leaves or internal nodes of the tree and are stored in the Kademlia DHT according to their identifiers. Such an organization allows to navigate through the tree by performing DHT lookups at each level of the tree. In case of a well balanced tree, the cost of locating a cell is square logarithmic. The work explicitly focuses on object storage and does not handle interactions between avatars. An improvement of this scheme was proposed in Walkad [194]. The idea is to improve the performance of the index by minimizing the amount of DHT lookups. Links between parents and children are thus created to allow direct routing through the prefix tree. The obtained index is similar to P-Grid [30] and allows logarithmic routing. Walkad also provides constant-cost access to regions that are neighboring in the NVE by mapping them on neighboring Kademlia identifiers.

Another approach to ensure dynamic partitioning is to use overlays with locality preserving data placement [48, 104]. In that case, each peer has a coordinate in the NVE and stores objects that are the closest to is NVE-coordinate. A *region* is defined to be the set of objects stored on the same peer, and dynamic load balancing is performed by adjusting peers' coordinates in function of their load. Bharambe *et al.* demonstrate the feasibility of this approach by implementing Quake II on top of Mercury [48] in their Colyseus infrastructure.

A key concern of peer-to-peer NVEs is to guarantee a consistent and available access to data in presence of churn. In multi-tier architectures, data management is performed by the coordinators, so their robustness should be ensured. The Mediator framework focuses on the coordinator selection process [93]. It implements a reward scheme that helps choosing appropriate super-peers, called *mediators*. The reward scheme takes two parameters into account. First, it fetches hardware capabilities of the peers. Second, it tracks peers behavior history and rewards trustworthy contributors. Peers are rewarded by executing tasks published by the framework and strong, trustworthy peers are elected as mediators. Varvello *et al* replicate region coordinators in order to provide churn resilience [193], and in MOPAR, master nodes elect a new master when they leave a region [202]. Combining these techniques can allow peers to collectively build a robust decentralized infrastructure for NVEs.

The nearest-neighbor approach.

This approach aims to maintain a topology that inherently solves the neighbor discovery problem. Each peer maintains a direct connection to the peers which avatars are located inside its AOI. This enables efficient, one-hop communication with the peers that are the most interested in the updates of the peer. Since the AOI of a player changes with the movements of its avatar, the connectivity of its peer should evolve over time to match the AOI. This property, called *gradual self adaptation* is enabled by several topologies.

The most popular self-adaptive architectures are based on Voronoi tesselations/Delaunay Triangulations [42, 61, 114, 60, 195]. In such topologies, each peer is responsible for

a Voronoi-cell centered on the coordinate of its avatar in the NVE. The peer maintains direct links to the peers of the neighboring cells. The notion of neighborhood is defined by the AOI of the peer: all the peers which cell intersects with the AOI are considered to be neighbors of the peer. When the peer moves, it multicasts its coordinates to all its neighbors, which dynamically recompute the Voronoi tesselation. The outer-most neighbors act as watchmen for the peer, warning when new peers enter the AOI. This design, first proposed in VAST/VON [115, 114] has two main advantages. First, it enables efficient one-hop multicast to all the peers located in the AOI of a peer. This property dramatically increases the responsiveness of the NVE, since most of the updates propagated by a player are only limited to the peers located inside its AOI. Second, it enables n-dimensional KBR. Varvello *et al.* demonstrate the feasibility of this approach by implementing a Delaunay based peer-to-peer client for Second Life [195]. Their evaluation shows that the design improves the interactivity of the game compared to the regular client/server design of Second Life by a factor of 5.

However, the approach also has important drawbacks. First, maintaining a consistent Voronoi topology may be challenging and costly with highly skewed peer-distributions and high avatar speeds [37], especially for dimensions larger than 2 [43]. Second, since peers are connected only if they are close to each other inside the NVE, Voronoi topologies do not have the small-world property and have poor connectivity. In particular, Backhaus *et al.* show that the VAST overlay splits into separate partitions after a few hours of simulation because the neighbor discovery mechanism fails to discover fast moving peers [37]. And finally, the data availiability problem is hard to solve, because the responsibility region of each peer evolves over time. The overlay should either provide a dedicated ownership mechanism for NVE-objects, or constantly transfer the ownership to fit peer movements.

Several proposals address these problems. VoroNet builds shortcuts in the Voronoi topology to enable the small-world property [42]. At the condition that each peer has a rough approximation of the overlay size, the VoroNet's routing algorithm is proved to provide routing in $O(ln^2(N))$. In VoroNet, the ownership of an object is given to its creator, and the system builds a Voronoi graph of objects (opposed to VON's graph of avatars). The player's avatar is one of the peer's objects, and its movement does not involve data migration. However, the cost of maintaining the structure can be high: peers that create objects in many locations of the overlay end up maintaining a huge amount of Voronoi links. VoroGame, another system dedicated to MMOGs, uses the Voronoi topology for efficient content dissemination, while persistently storing objects in a DHT. This approach however incurs many DHT lookups while accessing objects, reducing the responsiveness of the system, and no evaluation of the latency overhead is provided.

The maintenance cost being one of the obvious limitations of Voronoi/Delaunay-based systems, some systems adopt less resource consuming overlay topologies. Solipsis, a massively multi-participant virtual world, is based on two fundamental rules [121]:

- *Local awareness rule.* If an avatar b is inside the AOI of a player a, the peers of a and b must be neighbors in the overlay. The size of a's AOI is adjusted to ensure that a has a number of neighbors contained between a minimum and a maximum bound.

- *Global connectivity rule.* Let N_a be the neighbor set of a peer a in the overlay. The avatar of a must be located inside the convex hull of the set formed by avatars of N_a. This

property aims that an avatar will not "turn its back" to a portion of the NVE, causing inconsistent views or possibly partitioning the Solipsis overlay graph.

To ensure these rules, Solipsis implements a mechanism called *spontaneous collaboration* similar to the "watchmen" behavior of VON. At each moment, thanks to periodic updates, a node is aware of the coordinates and the awareness area sizes of all nodes in its neighbor set. As it locally detects that one of its neighbors enters the awareness area of another of its neighbors, it sends a message to both entities to warn them that the local awareness rule is about to be broken. As they receive that message, the two entities become neighbors.

RayNet is an overlay that organizes its peers into an approximation of a Voronoi topology [43]. RayNet is based on the observation that only the neighbor-set calculation is important, while computing the exact shape of the Voronoi cells is not necessary. RayNet peers share their neighborhood knowledge with a gossip algorithm. At each cycle of the gossip algorithm, a peer receives from its neighbors a list of peers to add to its neighbor-set. Since the neighbor-set has a limited size, the peer only keeps the peers that minimize its Voronoi-cell volume. This algorithm converges to an approximation of a Voronoi-based topology. Concurrently, RayNet peers also run a second gossip-based algorithm to build small-world shortcuts in the topology. RayNet therefore enables poly-logarithmic greedy routing and efficient neighbor discovery. For that, it does not need to pay the computational cost of Voronoi-based architectures, which is high for dimensions higher than 2.

3.2.3 Hybrid architectures

Each of the two previously described approaches brings a well-identified set of drawbacks [90]. Server-based infrastructures have a high maintenance cost and usually suffer from a lack of flexibility. Pure p2p approaches lack of efficiency at the current state of developement of domestic connectivity [148] and have security problems because parts of the game state are managed client-side. As pointed out by several research papers, the two paradigms can be combined to bring their best in one hybrid design [69, 117, 91, 127, 73, 40]. The proposed hybrid architectures are based on mixing of the p2p multi-tier approach with the c/s paradigm, by adding an additional server-based tier to the multi-tier topology. A strict separation of tasks is usually applied: critical operations are handled by trusted servers while the other tasks, such as position updates, known to represent about 70% of the MMOG traffic [73], are performed by super-peers. One of the first proposals by Knutsson *et al.*[4] suggests that persistent information, such as player character experience and payment information should be handled by a server [127]. All the other tasks, including update propagation and access to data is left to the coordinator peers. A similar approach is adopted by El Rhalibi and Merabti [91]. A trusted server handles user account information and manages player login while the JXTA framework is used to build a super-peer overlay that handles the NVE load. The central server splits the NVE into regions and delegates management of each region to a JXTA super-peer. The mapping between regions and super-peers is performed by the central server and is used upon login, while in-game movements are handled through the overlay. Chan *et al.* describe Hydra, a similar hybrid architecture for MMOGs [69]. In Hydra, a NVE region is managed by a primary and a set of backup peers, with a central server maintaining a mapping between peers and regions.

[4] which uses Scribe and is described in section 3.2.2

Chen and Muntz propose a more advanced system design where servers and super-peers can cohabit on the same level of responsibility [73]. As for the first two hybrid proposals, critical login information is handled by a trusted server, which also maintains a list of region identifiers. Each region is handled by a set of super-peers called *peer cluster*. The peer cluster is composed of a primary region manager and several backup managers. The backup managers monitor the primary to detect malicious behavior, and can elect a new primary manager in case of failure. If a peer cluster becomes overloaded, it can request help from the server infrastructure. In that case, the responsibility for a region is dynamically transferred to a high capacity server. Peer clusters and high capacity servers form a heterogeneous overlay that collectively handles the NVE load. Jardine *et al.* describe a architecture in which super-peers are elected by a central server to handle player position updates [117]. The server is kept in the game loop and handles critical state changing updates such as object modifications or transactions between players. The server is also used to select the most capable peers to become super-peers based on their bandwidth and latency capacities. Finally, the server monitors super-peers, receives complaints from regular peers, and is able to discard a super-peer if its integrity becomes questionable.

Hybrid architectures can be a real step forward to a truly scalable NVE. They take the best out of the two paradigms: a secure and efficient c/s infrastructure helped by a scalable, elastic p2p topology. They are also compatible with the current business model of commercial MMOGs. Sign that game providers are becoming aware of the benefits offered by this design: World of Warcraft, the most popular MMOG, incorporates a Bittorrent client for propagation of new content in game [26].

3.2.4 Summary

In this section, we gave an overview of the state-of-the-art architectures for MMOGs. While commercial MMOGs all have a conservative server-based design, the research community is investigating a broad set of architectures. As of now, it is unclear which of the client/server, peer-to-peer or hybrid paradigms will prevail in future MMOGs. The proposed solutions all have their strengths and drawbacks, and in absence of a well-adopted evaluation benchmark the proposed solutions are hard to compare. To conclude this section, we attempt to make a classification of the most notable architectures proposed by the research community. For that, we isolate four important characteristics that impact the peformance of the MMOG systems:

1. *Type of architecture:* c/s, multi-tier, pure-p2p or hybrid.
2. *Data management:* Who stores what?
3. *Data access cost:* How does the lookup cost scale?
4. *Behavior-aware optimizations:* Is the system able to adapt to game evolutions?

The results of the classification are shown in Table 3.1.

	Architecture	*Data management*	*Data access cost*	*Behavior-aware optimizations*
Sirikata	c/s	region-based partitioning, with distributed kd-tree for indexing	intra-region: $O(1)$; extra region: < $O(log(N))$ for N servers (cost of kd-tree lookup)	Solid angles used for realistic AOI management
MiddleSIR	c/s	full-state replication	reads: $O(1)$; updates: $O(N)$ for N servers	None
Darkstar	c/s	Centralized database + data caching on servers	$O(1)$; updates: $O(N)$ for N servers	None
Donnybrook	p2p complete graph with dynamic multicast trees	object ownership spread among peers	$O(1)$ (cost of limited-size multicast tree propagation)	Accurate AOI Management: frequent updates for 5 objects only. Others managed with Dead Reckoning
Colyseus	p2p overlay for range-querying	locality-aware placement	O(log(N))	None
Mediator	multi-tier p2p	region-based	intra-region: $O(1)$, extra-region: $O(log(N))$	None
Walkad	p2p reverse binary trie	region-based	$O(1)$ for local, $O(log(N))$ for distant	None
Solipsis	Dedicated nearest-neighbor p2p	No objects, avatars only	O(1) for closest avatars, O(N) for others	None
VON/VAST	Voronoi-based p2p	No objects, avatars only	O(1) for closest avatars, O(N) for others	None
RayNet	Approximate Voronoi p2p	object stored on creator	$O(1)$ for nearest, $O(log^2(N))$ for distant objects	None

Table 3.1: *Summary of notable NVE architectures.*

3.3 Leveraging behavioral trends

The goal of this section is to provide an insight on the feasibility of leveraging player behavior to improve the performance of MMOG architectures. It describes MMOG architectures that use player behavioral and physiological trends in their design. The research in the area follows three main directions: (i) improvement of the player's interest management, (ii) accommodation of player distributions and (iii) avatar movement prediction.

3.3.1 Interest management

A player exploring an MMOG is mainly interested in the objects inside its AOI, which represent only a small fraction of the huge amount of information contained by the environment. The AOI can be seen as a filter that defines which updates are relevant for a player and should be propagated, and which are irrelevant and should not be sent. The AOI filtering should guarantee that all the needed updates are received by the player. At the same time, it should minimize the amount of irrelevant updates. Accurate interest management can dramatically decrease the load of the system without degrading gaming experience. The fundamental challenge of interest management is then to determine what objects the player is *really* interested in. This task is uneasy because studies show that player's interests are changing very often, *e.g.,* in Quake III, the average turnover is of 68% of the AOI objects per second [47]. The state-of-the-art approaches exploit geometric properties of tridimensional spaces and specificities of human perception to select objects that are the most likely to be interesting to the player.

The most popular approach is based on object proximity: close objects are more interesting than distant ones. Several proximity-based techniques were proposed in the last few years. The simpliest, most widespread approach is to define the AOI as a spheric area centered on the player's avatar. The technique has a low computational overhead, but can degrade the gaming experience by defining a too strict boundary between relevant and irrelevant objects. A player will thus not see objects located just outside the AOI while they would be perfectly visible in the real world. Moreover, if the AOI contains obstacles hiding some objects, the player will still receive updates about them while they are irrelevant. Another proximity based technique relies on the triangulation of the environment surrounding the player. The AOI is then defined as the set of triangles that are close to the player's position. This approach is more accurate, since it allows to take obstacles into account, but has a higher computational cost and can still miss some essential objects outside the AOI. More static, less efficient management were also proposed. For more details and a thorough comparison of proximity based approaches please refer to Boulanger *et al.* [55].

The proximity based approaches give a simple approximation of the objects a player is interested in. More accurate techniques based on specificities of player's perception have been proposed [75, 47]. Intuitively, they leverage the fact that the player is likely to be interested by all the objects that are visible from its position. A ray tracing algorithm determines all the visible objects and considers them being part of the AOI. However, ray tracing is costly: it can induce significant computational overhead [55] and the obtained AOI can be large. To solve this problem, Sirikata proposes to rely on *solid angles* [75, 112]. The solid angle of an object defines its apparent size as seen by the player. The angle size is function of the object's size and its distance to the player. The AOI managed by Sirikata is composed of the

objects which *appear* big to the player. To reduce the quantity of information contained in the AOI, Sirikata progressively reduces the amount of information transmitted about an object as its distance to the player increases. Consequently, an equal quantity of information about the AOI will produce a much more realistic view with the solid angle based approach than with the proximity based one (see figure 3.4).

(a) (b)

Figure 3.4: *Screenshot of AOI with 3000 objects: (a) Distance selection, (b) Solid-angle selection*

In order to further decrease the load of the system, Donnybrook, a peer-to-peer implementation of Quake III, exploits the incapacity of human beings to simultaneously focus on many different objects [47]. The idea is to identify the few (less than ten) objects in which the player is really interested. These objects will receive very frequent updates, whereas the other objects of the AOI will be lazily maintained. The most interesting objects are discriminated following three criteria: proximity, aim, and interaction recency. The most interesting objects are those which:

- are close to the player
- are aimed by the player (centered on the player's screen)
- have been recently accessed by the player

The separation between interesting and non-interesting objects helps Donnybrook to dramatically reduce the load of the system. Therefore, even if it has a complete graph topology, the overlay is able to scale to many hundreds of peers. The rate of non-critical updates is very low, allowing a peer to broadcast them to *all* the participants. Critical, highly frequent updates are efficiently propagated through limited-size, dynamically built multicast trees.

The evaluations of Donnybrook show that its interest management significantly reduces the load of the system without degrading gaming experience. Similarily, Chan *et al.* propose a MMOG-specific caching and prefetching policy called MRM [68]. MRM rates an object in function of its distance to the player and on its angular distance to the player's trajectory vector. MRM is shown to have a 40% to 50% improvement of the cache hit ratio compared to the traditional LRU policy [76]. Finally, Park *et al* point out that the interest for an object

also depends on its *collective* value, *i.e.*, a player is more likely to be interested in a popular object [155].

3.3.2 Dynamic load balancing

Nearest-neighbor overlays, such as Voronoi/Delaunay based architectures, naturally leverage player behavior to gracefully adapt to the density distribution of the NVE. However, it is known that the maintenance cost of Voronoi/Delaunay overlays is high in dense areas. Several efficient clustering algorithms have been proposed to lower the maintenance in highly clustered areas [181, 192]. The primary goal of these works is to identify clustered peers in a decentralized context. Steiner and Biersack propose an algorithm based on the evaluation of the link lengths [181]. On a peer, a link is considered to be inter-cluster if it is significantly longer than the average length of the peers's links. The algorithm allows dynamic cluster identification in presence of churn. Varvello *et al* describe an algorithm based on the estimation of the maintenance overhead induced by the clustering [192]. Each peer monitors its maintenance cost and proposes the creation of the cluster to its neighbors when the maintenance cost exceeds a given threshold. If the creation is collectively decided, a super-peer is elected to handle the cluster, dramatically reducing the maintenance cost of the peers outside the cluster. Clustered peers artificially expand intra-cluster distances to reduce their maintenance. Eventually, the overlay forms a multi-tier topology that dynamically adapts to the distribution. An evaluation of this mechanism with Second Life traces shows a significant decrease of the maintenance cost.

3.3.3 Movement prediction

Player behavior prediction can also be used to decrease the impact of high latency on the gameplay. The technique, called *Dead reckoning* allows to predict the player's future position based the current state of the game. Traditionnaly, the system computes a short-term prediction based on the path history of the player [154], or on the history of user generated events (*e.g*, mouse events) [68]. Yahyavi *et al* have recently shown that taking player's interest into account could greately increase the accuracy of dead reckoning [201]. In their approach, the NVE is split in square cells and each cell is assigned an attractiveness rate depending on the entities it contains. While making a trajectory prediction, the system takes cell popularity into account. This enhancement is shown to improve the accuracy of the prediction by 30% [201]. Dead reckoning is used on player hosts to reduce the lack of responsiveness perceived by the player in case of high latency or packet loss. If the update of an avatar inside the player's AOI is not received on time, the dead reckoning mechanism of the player's host optimistically makes a prediction of the new avatar's position. Predicted and actual positions are reconciled when the update is finally received. As an example, Donnybrook uses dead reckoning to relax the update rate of the AOI avatars on which the player is not directly focused [47].

3.4 Conclusion

An extensive set of studies show that modern MMOGs have complex workloads that are shaped by player behavior. Players like to play together, periodically generating population peaks and dramatically increasing the amount of applicative load. They tend to cluster in high density zones, skewing the load distribution. Their movements are erratic, with possible high speeds and group phenomena. These characteristics make the MMOG workloads extremely hard to accommodate. The underlying system must dynamically adapt to absorb load peaks, have efficient load balancing mechanisms, and handle player mobility.

Architectures of commercial MMOGs are too rigid to exhibit these characteristics. Overprovisioned to absorb the load, they have a huge maintenance cost, but still fail to correctly accomplish the task because of the player distribution heterogeneity. Despite the phenomenal amount of consumed resources, servers of popular areas are overloaded, while the rest of them remain hardly used. The research community proposed an extensive set of solutions to address the problem. However, this effort suffers from a lack of availability of realistic workloads, which makes many contributions unclear. As for the works that propose a thorough evaluation, many of them are based on the p2p paradigm and suffer from a lack of robustness and reactivity. The problem of these research and commercial proposals is their incapacity to accurately take the complex MMOG workload into account.

Yet, the main characteristics of player-population evolution, distribution and mobility are now well identified. These characteristics can serve as a base to create models describing the MMOG workload. Such models can be used for realistic evaluation testbeds and be integrated directly into the architectures for a better workload accommodation. Therefore, we believe that they are necessary to increase the impact of the research community and to offer robust, flexible architectures to future MMOGs.

Chapter 4

Workload generator

Contents

4.1 Introduction

Building distributed systems that efficiently support MMOG workloads is a challenging task. Over the last decade, the community offered many solutions to problems posed by the emerging MMOG applications (see Chapter 3). Yet, all this research work still has very little impact on commercial MMOG architectures. All of the most popular online games on the market have a static architecture and are unable to support even the simpliest requirements of MMOGs. Their incapacity to elastically adapt to the workload leads to severe gameplay limitations and significant financial loss for the game provider.

Miller *et al.* and Pittman *et al.* state that one of the key reasons of this lack of visibility is the absence of commonly adopted realistic test workloads [146, 157]. There is no agreement on a unified benchmark within the community, and without the authorization of the game providers, good-quality real MMOG traces are hard to gather. Consequently, evaluations are often performed with unrealistic generated workloads[1] making the benefits of the proposed solutions unclear and hard to compare.

[1] *e.g.*, many systems assume a uniform random access pattern [157]

We believe that realistic workloads are the key to further research and development on the next generation popular MMOGs. The first goal of our thesis is thus to provide a realistic player behavior model. While previous approaches focused on a thourough analysis of the MMOG workloads (please refer to Section 3.1.1 for details), this is, at the best of our knowledge, the first realistic-workload *generator*[2]. While our work is a first, simple approach to synthetically recreate avatar behavior, we believe that it can be reused as a base of a more complex, standardized benchmark platform.

This section presents our first contribution: a model of mobility that allows to generate realistic MMOG traces. We show that the traces generated from this model are similar to the real traces collected by La and Michiardi [131] from Second Life. These real traces have been collected by crawling two Second Life islands called "Dance" and "Isle Of View". These traces are composed of the coordinates of avatars wandering on the island indexed by timestamps. These two islands were chosen by La and Michiardi to be representative of the Second Life players' behavior.

4.2 Mobility model

The analysis of existing MMOGs shows that in all of them, the majority of the avatar population is gathered inside a few density hotspots. These hotspots can be points of interest of the game, such as towns or social events, or casual interaction groups. Figure 4.1.a shows a snapshot of the avatar distribution on Second Life's "Isle of View" with 50 avatars and three clearly visible clusters. This kind of distribution with hotspots is also characteristic of real world populations: people are attracted by points of interest, they cluster into cities and cities cluster into megalopolises, such as the european *Blue Banana*, the north-american *Northeast megalopolis* or the japanese *Tokaido corridor*.

As presented in section 3.1.2, the movement of avatars is usually chaotic in hotspots and straight between hotspots. Regarding the player mobility in NVEs, studies have shown that it is quite similar to human mobility in the real world [131, 165]. This mobility pattern is most of the time modeled with Levy flights [165]. Levy flights naturally differentiate the two observed behaviors of avatars: periods of travel and periods of exploration with chaotic movement. However, Levy flights do no help to model hotspots because they do not ensure that avatars stay grouped around hotspots and that density around hotspots remains the same despite avatar mobility. Because of that, we choose to model the density and the movement of avatars in NVEs with a markov chain to discriminate the periods of exploration from the periods of travel. This choice is consistent with other studies that point out that avatar movement cannot be modelized by waypoint navigation, but rather with markov chains [188, 158, 100].

At bootstrap, the generator first defines two opposite types of regions in the gamespace: **hotspots** (*i.e,* high density zones), and **desert zones**. Then, all the avatars are placed on their initial positions on the game map (an example of the obtained distribution is shown in Figure 4.1.b). The generator ensures that most of the avatars are grouped in the hotspots. After that, the generator computes new maps step-by-step from the previous ones by moving avatars. The model of movement ensures that avatars remain principally grouped in

[2] A similar approach has been very recently proposed by Suznjevic *et al.* based on World of Warcraft traces [188].

hotspots during time. To generate movement, at each step, each avatar is in one of the following states:

- **(H)***alted*: the avatar does not move at this step.
- **(E)***xploring*: the avatar is exploring the map.
- **(T)***raveling*: the avatar is moving to a new location on the map.

We consider that an avatar has a constant application-defined acceleration during a move (states *E* and *T*). The acceleration value is a parameter of the model generator. As an avatar moves from a position to another, at each step, its speed increases. When an avatar reaches its final position, it suddenly stops and its state machine enters the H state. The trace generator is configurable and takes the following parameters: 1) the number of avatars; 2) the size of the map; 3) the number of hotspots and the radius of each hotspot; 4) the proportion of avatars inside hotspots; 5) the acceleration; 6) the probabilities of transitions between the different states. A resulting initial placement with 100 avatars, 3 hotspots, 80% of the avatars in hotspots and the transition probabilities given in Figure 4.2 is presented in Figure 4.1.b.

4.2.1 Generation of the initial map

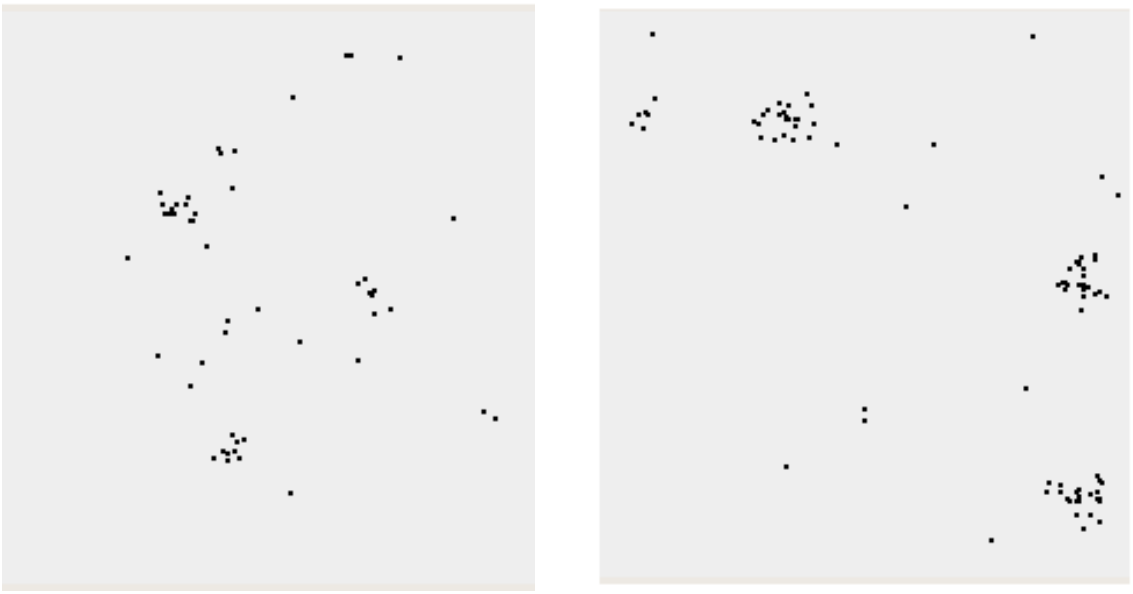
During the first phase, the trace generator randomly chooses the positions of the hotspots. Then, for each avatar, the generator decides if it should be placed in a hotspot accordingly to the proportion of avatars inside hotspots. If this is not the case, the avatar is placed randomly on the map using a uniform probability law (it can therefore be placed in a hotspot or in the desert). Otherwise, the generator randomly chooses one of the hotspots using a uniform law and computes the polar coordinates of the avatar from the center of the hotspot: the angle is chosen using a uniform law and the distance to the hotspot center with a Zipf's law [205]. The Zipf's law ensures a very high density in the center of the hotspot, comparable to the ones observed in both NVEs and real life. Initially, all avatars are in the state *H*.

4.2.2 Movement generation

During the second phase, the trace generator moves the avatars step by step. Figure 4.2 presents the markov chain used for state transition with the associated state transition probabilities used to generate Figure 4.1.b. At each step, the generator reevaluates the state of all the avatars according to the state transition probabilities:

State H: If an avatar enters or stays in the state H (transitions *TtoH*, *EtoH* and *HtoH*), the avatar does not move at this step.

State E: If the avatar takes one of the transitions *HtoE*, *TtoE* or *EtoE2*, the avatar picks a new position on the map. To ensure that the density remains globally the same during the trace, if the avatar is in a hotspot its new position is chosen inside the same hotspot with the Zipf's law, otherwise, its position is chosen randomly on the map. If the avatar is in the state *E* and takes the transition *EtoE1* it continues its movement to its new position. We



(a) (b)

Figure 4.1: *Avatar distributions, each dot represents an avatar: (a) Traces of Second Life (50 avatars), (b) Generated traces (100 avatars).*

differentiate the two transitions *EtoE1* and *EtoE2* to ensure that an avatar regularly changes its direction and therefore has a chaotic movement.

State T: If the avatar enters in state *T* (*HtoT* or *EtoT*), the avatar picks a new position on the map by using the initial placement function to ensure that the density remains roughly the same. It also begins its movement to its new position. If it takes the transition *TtoT* , it continues its movement to its new position.

4.3 Evaluation of generated traces

The generated traces are compared with the real ones crawled by La and Michiardi in Second Life. However, while the Second Life traces are limited in size and in number of avatars, the generated traces do not suffer from this limitation. The evaluation relies on the metrics proposed by La and Michiardi [131]:

- **Avatar travel length.** The travel length is the distance an avatar walks without changing its direction or making a pause. The distribution of avatars walk lengths are compared.
- **Spatial distribution of avatars.** The map of the NVE is divided in cells. The cell density distribution is computed by counting the number of avatars in each cell.
- **Clustering coefficient of avatars.** Circular areas are defined around each avatar (radius=80m), then all avatars located inside an avatar's circular area are added in its neighbor set. The local clustering coefficient of the graph is then built to give an idea

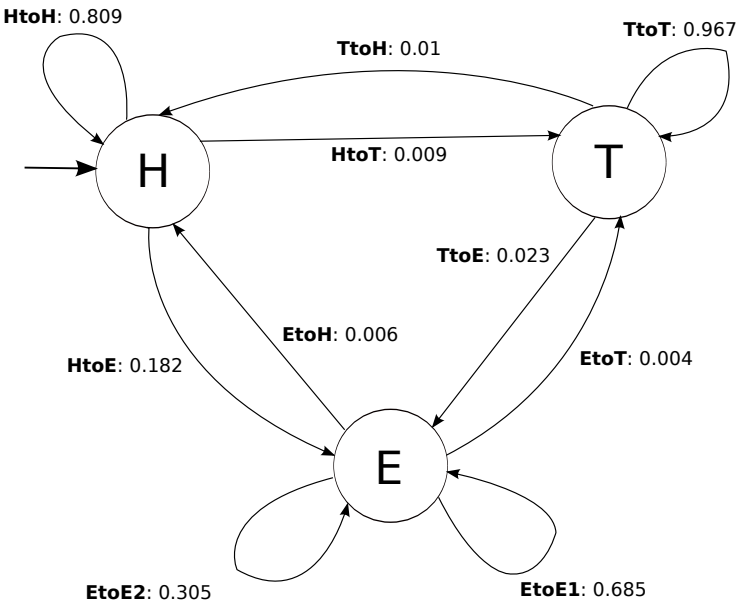


Figure 4.2: *The markov chain modeling Second Life avatar behavior.*

of the density. Intuitively, the clustering coefficient of an avatar is the the proportion of connections among its neighbors which are actually realized compared with the number of all possible connections [199].

The primary goal of our generated traces is to simulate a realistic MMOG workload. Our concern is thus to create *large scale* traces exhibiting the same properties as the Second Life traces so we generate traces with 1000 players. Since Second Life islands are designed to host approximately 100 avatars, the generated map used is nine times bigger than the Second Life maps. Because maps have different sizes, the number of hotspots is adjusted so that average distance between hotspots remains comparable between the two kind of traces. We tune the generator to obtain three hotspots containing approximately 80% of the population to obtain similar results between the real traces and generated traces. The number of steps for each trace are 1945 for the "Isle of View" island, 4808 for the "Dance" island and 2000 for the generated trace. Finally, the probability of transitions, empirically chosen to fit with Second Life mobility, are given in Figure 4.2. The evaluation of each metric is performed as follows: for each step of the trace, we compute the value of the metric for that step. We then perform an average of all the step-values. Figures 4.3 and 4.5 depict the average values of the traces for the three metrics.

The evaluation of the traces with the avatar travel length metric is presented in Figure 4.3. We plot the CCDF (Complementary Cumulative Distribution Function) of the walk lengths, which shows the proportion of the overall population which current trajectory is longer than the given length. The first observation is that regardless of the trace, avatars have majoritarily very short walk lengths: about 40% of the trajectories exceed 1 meter, but only

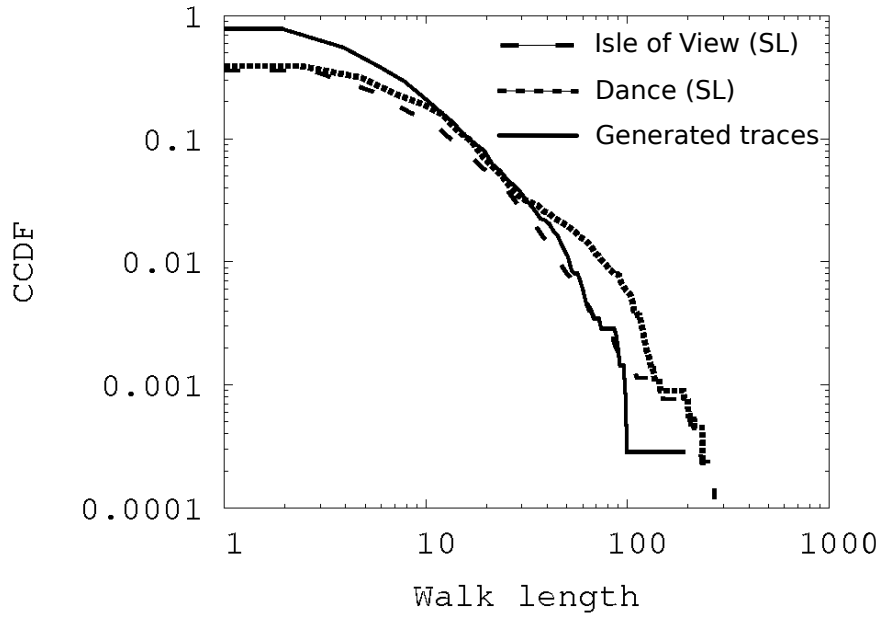


Figure 4.3: *Evaluation of the generated avatar movements.*

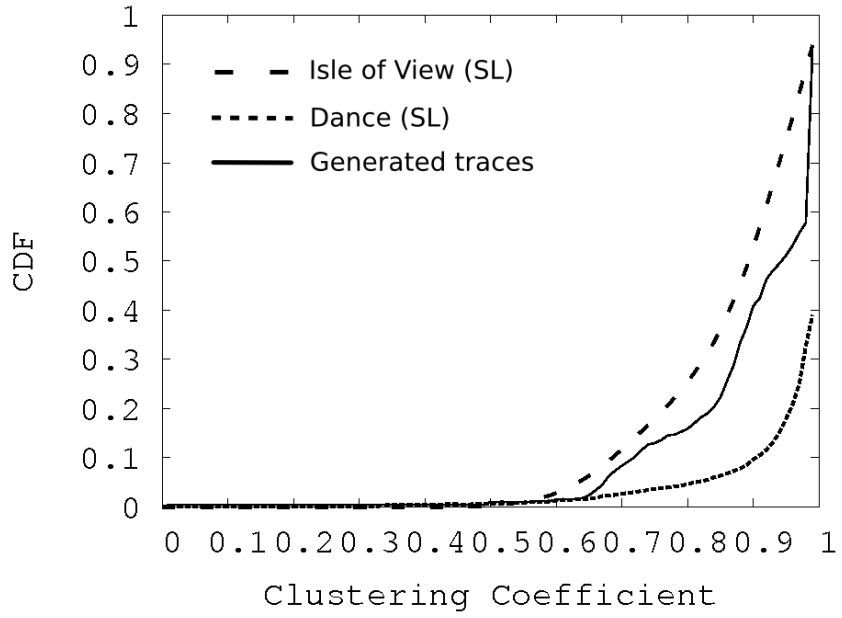


Figure 4.4: *Evaluation of the generated avatar distribution with clustering coefficient metric.*

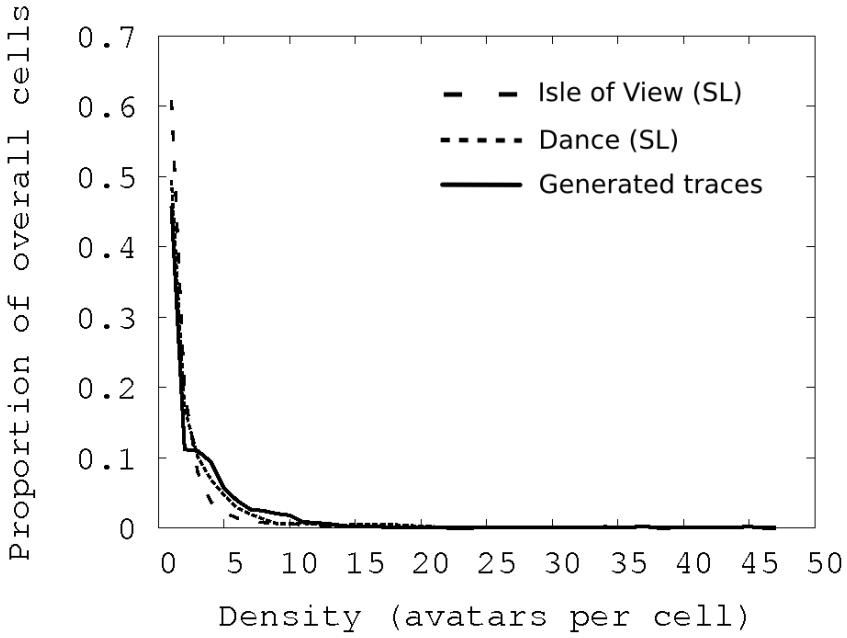


Figure 4.5: *Evaluation of the generated avatar distribution with spatial distribution metric.*

6% exceed 20 meters, and none of them exceeds 300 meters. Second, we can see that the walk-length distribution of the synthetic trace is quite similar to the real trace distributions. The main difference happens to be for very short trajectories: 80% of the trajectories exceed 1 meter in the synthetic trace, versus only 40% in the real traces. We believe this comes from the fact that many Second Life avatars log-in for social activities, such as chat, and therefore do not move *at all*. This kind of behavior is specific to games with a strong social dimension. Our model could be tuned to take this type of behavior into account: the *HtoH* transition probability can be increased, or a new "long halt" state could be added.

The comparison of the density distributions is summarized in Figure 4.5. As expected, Second Life avatars are non-uniformly distributed across the map. The plot 4.4 shows the CDF (Cumulative Distribution Function) of the clustering coefficient of the avatars. It represents the proportion of avatars which clustering coefficient is below a given value. We can see that the neighborhood graph of Second Life avatars is highly clustered: *all* of them have a clustering coefficient above 0.6. However, the clustering coefficient significantly varies between the two Second Life traces: while more than 60% of the avatars have a clustering coefficient below 0.9 for Isle of View, only 10% have the same characteristic for Dance. This suggests that the avatars tend to be much more clustered in Dance than in Isle of View. Similarily, Figure 4.5 shows that 60% of the cells contain less than 3 avatars, while a small number of regions can contain up to 40 avatars. Both density metrics show that our synthetic traces are similar to the Second Life traces regarding the avatar distribution.

4.4 Discussion

The evaluation of any system is only valuable if it is performed under realistic load conditions. Currently, the main challenge of the MMOG research community is to evaluate proposals with realistic workloads. Game providers are reluctant to share their logs, and collecting good quality traces without their consent is a complex task. Trace generators offer a simple and flexible alternative to create workloads exhibiting characteristic patterns of existing MMOGs. They provide the system designer with high quality, large-scale traces that would have been extremely hard, if ever possible, to collect and can be tuned to cover a set of different test cases. However, trace generators are only legitimate if they create traces that reproduce the characteristics of real workloads.

Our model is a first approach to generate realistic avatar movement. Only two basic behaviors are supported: exploration and travel. This model is very simple and may be unsuitable for complex, gameplay-dependent behaviors, such as raiding, questing or trading [187]. However, we believe its simplicity is a key to its genericity. The mobility it models is gameplay-independent, characteristic of human behavior and is consistent with observations made by studies of many online games. Its parameters can be tuned to fit with observations of some particular MMOG. Our generator can therefore be used to give a first rough evaluation of any MMOG system. The contributions described in the next two chapters use traces created by our generator.

Chapter 5

Avatar-mobility resilient content dissemination in p2p MMOGs

Contents

This chapter presents Blue Banana, our second contribution. Blue Banana is a mechanism that improves the resilience of nearest-neighbor p2p overlays to avatar movements.

5.1 Introduction

5.1.1 Problem statement

Efficient content dissemination is a key requirement for MMOG systems. Each player needs to propagate updates of its avatar state to the other participants in a real-time fashion. The

players mostly interact with their surroundings in the NVE, so the major part of these updates are only relevant to the neighbors of the player. The content dissemination mechanism should ensure that the updates of each player are propagated to its neighbors at a high rate and with low latency.

Nearest-neighbor overlays provide an optimal solution to the content dissemination problem in p2p MMOGs because peers that are close to each other in the NVE are directly connected in the overlay. Thanks to their topology, nearest-neighbor overlays enable efficient, one-hop propagation of players updates. The main challenge of nearest-neighbor overlays is to maintain their topology up-to-date in presence of avatar movements. The virtual neighborhood of a player changes while she moves and the overlay must dynamically link new neighbors. There is evidence that some mobility patterns can degrade the structure of nearest-neighbor overlays [37].

In particular, if the relative speed of two avatars is too high and the distance between them is too low, the neighbor discovery mechanism of the overlay fails to update the topology on time. Such failures produce temporary update inconsistencies, that we call *transient failures*, and can eventually result into permanent partitioning of the overlay [37]. This happens because the neighbor discovery mechanism does not *anticipate* the movement and therefore has only an extremely limited amount of time to react and modify the topology.

5.1.2 Contribution

To solve this problem, we propose a mechanism called Blue Banana that *anticipates* avatar movements and proactively adapts its topology to fit with the prediction [DSN10]. The algorithm makes a prediction on the avatar trajectory, and if the prediction confidence is estimated to be good enough, the peer creates overlay links with peers located in the direction of its movement with respect to the avatar's speed. The prediction is based on the analysis of player behavior. It relies on the simple observation that fast movement most of the time has straight, predictable trajectory, whereas slow movement is often chaotic. The estimation of the prediction confidence is a key challenge of our algorithm. It should only prefetch peers when the confidence is high because a high rate of irrelevant prefetching is resource consuming. Our algorithm decreases the number of transient failures occuring in the system due to avatar mobility. When two avatars with a high relative speed happen to meet in the NVE, there is a high probability that their peers have already known each other for a long time, and can therefore immediately adapt their topology to the movement.

We implemented our anticipation mechanism in PeerSim, a widespread discrete event simulator [118]. The mechanism was used to improve the resilience of Solipsis, a nearest-neighbor overlay. The evaluation performed with realistic traces from our trace generator shows that Blue Banana allows to avoid 20% of the transient failures with a network bandwidth overhead of only 2%. Moreover, our mechanism does not decrease the robustness of the original protocol: when the movement is erratic, transient failures do not increase. Finally, thanks to Blue Banana, a peer knows 7.5 times more of its future neighbors in advance, allowing it to prefetch relevant information about the future AOI of its player.

5.2 Design overview

Each peer of the nearest-neighbor overlay incorporates a Blue Banana module in addition to the regular maintenance mechanisms of the overlay. The main component of the module is a finite state automaton. The role of the automaton is to discriminate chaotic from straight avatar movement.

5.2.1 The state machine

The automaton is a simple model that describes mobility of the local avatar. According to the mobility pattern related in the studies of MMOG workloads 3.1.2, we define an avatar to have two main states: ($\tilde{T}$)ravelling and ($\tilde{E}$)xploring. A travelling avatar is rapidly moving on the map and its trajectory is straight, while an avatar in the exploration phase have a slow and chaotic movement.

While the player is interacting with the NVE, its peer locally analyzes the state of the avatar. If it detects a behavioral modification, it switches the state machine to the appropriate state. The behavior of an avatar is defined by its *speed*. If the speed of an avatar increases and reaches a threshold, the state machine is switched to the state $\tilde{T}$, otherwise, it is switched to the state $\tilde{E}$. This simple model is a first attempt to describe avatar movement and can be refined: it could take into account the acceleration of the avatar, or try to predict player behavior by analyzing its movement history. However, this simple model already provides a sufficient prediction accuracy to decrease the number of transient failures.

If the state machine is in the state $\tilde{T}$ (the avatar is traveling), its trajectory is highly predictable. Therefore, our algorithm tries to prefetch the forthcoming peers, *i.e.,* peers that are located along its predicted trajectory.

If the avatar is exploring a zone (state $\tilde{E}$) its trajectory is chaotic and its speed is low. In this case, its path is difficult to predict, therefore the Blue Banana module does not anticipate the loading of the forthcoming peers. Notice that because of the slow speed, the native algorithm of an NVE is likely to adapt itself on time anyway.

5.2.2 Movement anticipation

To maximize the prediction accuracy, we make two assumptions: (i) only short term prediction is accurate, (ii) the faster an avatar is moving, the more it is likely to continue on its current trajectory. The first assumption implies that future probable positions calculated from the avatar's present location and movement vector form a *cone*, as shown in Figure 5.1. Indeed, the more a position prediction is far in the future, the more it is likely to diverge from the real path. The second metric implies that the prediction accuracy increases with the avatar speed: the sharpness *i.e.,* the apex of the cone is proportional to the speed.

If all the peers located inside the cone are prefetched on time and if the avatar stays in the cone, the peer of the moving avatar will then very quickly adapt to the mobility thanks to the local information provided by Blue Banana.

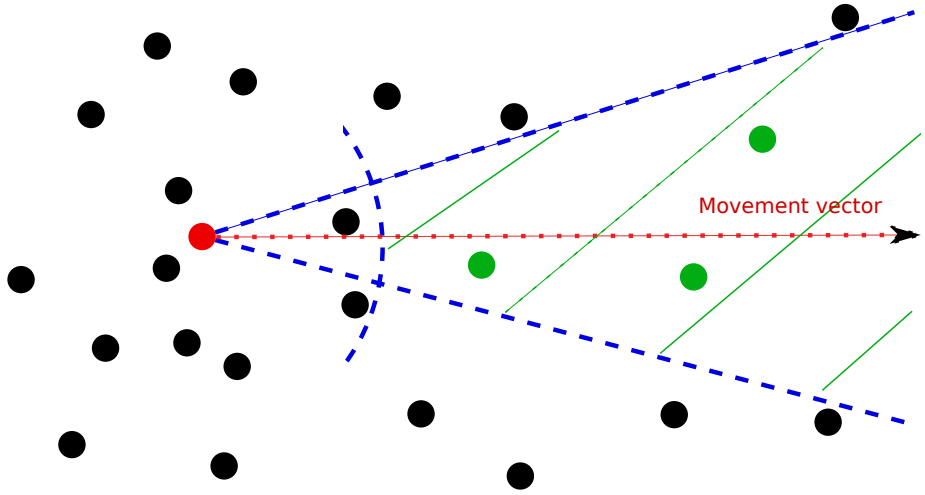


Figure 5.1: *Blue Banana prefetches peers inside the cone centered on the trajectory, except peers that are too close to be contacted on time.*

5.3 Implementation of Blue Banana on top of Solipsis

We have implemented our Blue Banana prefetching algorithm on top of Solipsis [121], which we believe to be one of the most lightweight representatives of nearest-neighbor overlays.

5.3.1 Details of the Solipsis protocol

Solipsis is an overlay designed to enable locality-aware content dissemination. Each Solipsis peer is responsible for one avatar, and no object management is provided. As described in section 3.2.2, its protocol is based on two fundamental rules:

1. *Local awareness rule.* If an avatar b is inside the AOI of a player a, the peers of a and b must be neighbors in the overlay. The size of a's AOI is adjusted to ensure that a has a number of neighbors contained between a minimum and a maximum bound.

2. *Global connectivity rule.* Let N_a be the neighbor set of a peer a in the overlay. The avatar of a must be located inside the convex hull of the set formed by avatars of N_a. This property aims that an avatar will not "turn its back" to a portion of the NVE, causing inconsistent views or possibly partitioning the Solipsis overlay graph.

To ensure these rules, Solipsis implements a mechanism called *spontaneous collaboration*. At each moment, thanks to periodic updates, a peer is aware of the coordinates and the AOI sizes of all peers in its neighbor set. As it locally detects that one of its neighbors enters the AOI of another of its neighbors, it sends a message to both entities to warn them that the local awareness rule is about to be broken. As they receive that message, the two entities become neighbors. Our simulations showed that this technique is very efficient: most of the time, a peer receives a warning message and does not have to initiate a costly new-neighbor query. The global connectivity rule ensures that a peer is always surrounded by its neighbor set, making spontaneous collaboration more efficient.

To sum up, if the local awareness rule is violated for a peer n, it means that an avatar has arrived into the AOI of n and is not yet included to the local knowledge of n, causing a transient failure. If the global connectivity rule is violated for a peer n, it means that n is not surrounded by its neighbor set. It will then not receive spontaneous data updates for a part of its neighbors, which will mandatorily lead to transient failures.

An avatar keeps breaking fundamental rules as long as it moves because the spontaneous collaboration mechanism is not always able to react on time. For that reason, a more efficient anticipation mechanism is required.

5.3.2 Implementation of the anticipation mechanism

Blue Banana is a generic module and could be implemented on top of any nearest-neighbor overlay. Blue Banana's main aim is to provide each peer with a prefetched peer-set (the size of the set is user defined). For this purpose, it finds peers in the direction of the avatar's movement. This is performed by propagating a prefetching message *ahead* of the peer's movement. This message contains the characteristics of the prefetching cone and peers receiving this message parse their neighbor-set searching for peers located inside the prefetching cone. As a result, the emitter of the message receives a list of the peers located in its cone. Once the moving avatar approaches a prefetched peer, the prefetched peer is added in the regular neighbor set managed by Solipsis. Hence, Blue Banana substantially helps Solipsis native algorithms to restore the fundamental rules, minimizing resulting transient failures.

Important properties of the algorithm

The first important quality of the algorithm is the consideration of avatar movement during message transfer time. Indeed, during a message transfer, the NVE changes, and so do interesting prefetched neighbors. For example, if A and B are 2 meters apart and if A runs toward B, B is probably an interesting prefetched neighbor. But if the network latency is around $200ms$ and if A runs at $36km/h$ ($10m/s$), the time to transfer a message from A to B is exactly the time to reach B for A in the NVE: the communication time between A and B makes B an uninteresting prefetched neighbor.

To take into account message transfer time, each peer estimates a low and a high bound of the network latency by using the last observed round trip times with its neighbors. When a prefetching message arrives, the algorithm uses these latency bounds to roughly estimate the new avatar-position of the peer that emitted the message. Even if this estimation is clearly rough, it permits to send more accurate responses.

The second important quality of the algorithm is the number of messages generated to prefetch the neighbors: a peer receiving a prefetching request answers for all its neighbors whose avatars are in the probability cone (see Section 5.2.2). As a consequence, each candidate does not have to answer to the request.

Algorithm description

Technically, if the algorithm observes that the avatar of a peer B (for Blue Banana) is in the state $\tilde{T}$ (*i.e.*, it reaches the speed threshold) and if the prefetched neighbor set is not full, B

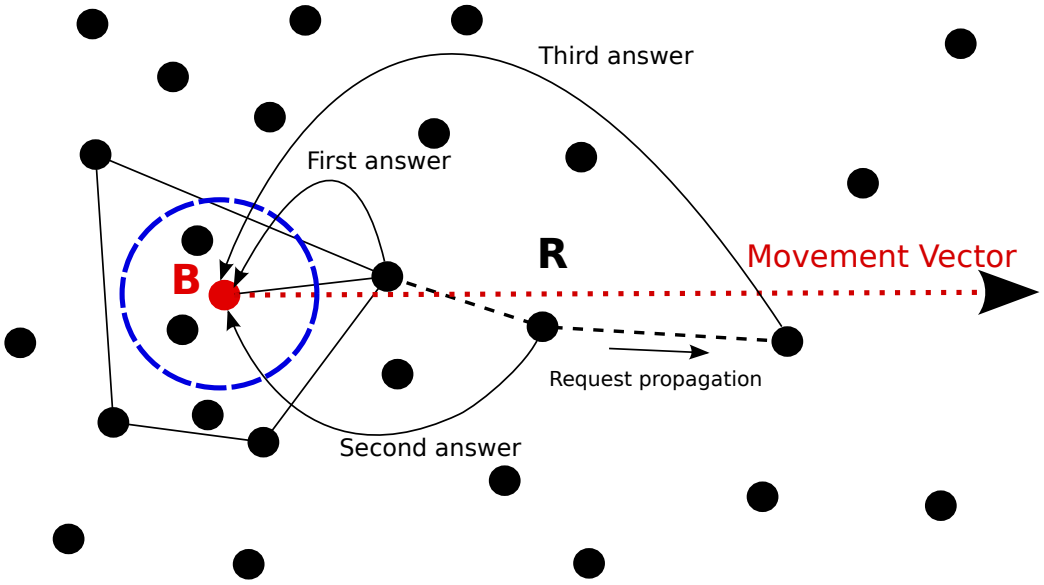


Figure 5.2: *Propagation algorithm: the request is transmitted to peers ahead of the movement.*

starts searching for new prefetched neighbors: it sends a message to its neighbor which is closest to its *movement vector* as illustrated by Figure 11.2. The message contains the number of prefetched neighbors that B is willing to retrieve (called the TTL) and the description of the probability cone (the apex of the cone, the direction of the movement and the speed).

Algorithm 1: Upon reception of a prefetching request

Result: gathering of prefetching candidates and prefetching request propagation.

```
1  emitterPosition = estimateCurrentEmitterPosition (msg);
2  ttl = msg.getTTL ();
3  if (emitterPosition, myPosition) ≥ minDist then
4      trajectoryClosestNodes = chooseClosest (neighborSet, msg);
5      size = trajectoryClosestNodes.getSize ()−1;
6      if size > ttl then
7          size = ttl ;
8          trajectoryClosestNodes = trajectoryClosestNodes [1 .. size ];
9      if size > 0 then
10         ttl = ttl − size + 1;
11         response.addSet (trajectoryClosestNodes);
12         send (response, msg.emitter ());
13 if ttl > 0 then
14     msg.setTTL (ttl −1);
15     send (msg, findNextNodeInTrajectory (msg));
```

Upon the reception of a prefetching request from B, a peer R executes Algorithm 1. R first estimates the current position of B by using the estimated network latency, the initial position and the speed of the avatar (line 1). Then, R checks if it is not too close to B (line 3): if B overpasses R during the message exchange, R is located behind of B when the response is received by B, making the prefetched information useless. Then, if R is located far enough (lines 3 to 12), R analyzes its neighbor set and selects peers located inside the new estimated prefetching cone of B (line 4) to send them to B (lines 10 to 12). If the size of this set of

candidates exceeds the TTL, only the first TTLs are selected (line 5 to 6) and if R does not have interesting neighbors, R does not send its response to B (line 9). While the TTL has not expired, R forwards the request to its neighbor that is closest to the *movement vector* of B (lines 13 to 13). At the end, if no message have been lost and if messages arrive on time, B retrieves TTL prefetched neighbors located inside its probability cone.

Network overhead

Blue Banana does not interfere with the maintenance protocol of Solipsis: the prefetched neighbors are not placed in the regular Solipsis neighbor set, but in a separated one. Therefore, Solipsis does not use network resources to maintain links with prefetched neighbors. We prefer not to spend network resources to maintain a link with a peer which is useless *at the moment* since it is not yet in the AOI.

As a consequence, once inserted in the prefetched neighbor set, the position of a peer's avatar is not updated, while it can move outside the probability cone. Blue Banana automatically removes useless prefetched neighbors (i) when they have been overtaken by the moving avatar, (ii) when the avatar changes its direction or (iii) when it changes its state. It is possible to consider another policy by periodically updating the state of the prefetched neighbors. However, the risk is to spend network resources to update possibly useless peers. The comparison of these two policies is part of a future work.

In order to compensate the small network overhead, Blue Banana peers take advantage of the high predictability of the avatar movement in desert zones. In Solipsis, a peer periodically propagates the coordinates of its avatar to all the members of its neighbor set, so the neighbor-peers can update their view of the NVE. Blue Banana doubles the period of such updates for peers when the state machine is in state $\tilde{T}$. The neighbors of that peer simply predict the position of the avatar between two updates by using its initial position and its speed. This technique is a very simple form of dead reckoning, but it could easily be enhanced with more sophisticated mechanisms such as those described in section 3.3.3.

5.4 Evaluation

This section presents a detailed evaluation of Blue Banana. The evaluation compares Solipsis *with* and *without* Blue Banana to measure the performance of the anticipation mechanism. Both Solipsis and Blue Banana are implemented in PeerSim. Realistic traces produced by our trace generator described in chapter 4 are injected in the simulator to recreate MMOG activity.

5.4.1 Description of the simulations

The PeerSim simulator is a widespread platform for testing distributed applications [80, 32]. It has been designed for scalability and is simple to use. It is composed of two simulation engines, a simplified (cycle-based) one and an event driven one. The simulation is realized with the event driven engine which provides the ability to perform more accurate simulations.

At the beginning of the simulation, the initial map of the trace is injected in the simulator. The simulator, based on this map, initializes the Solipsis overlay and then waits until every peer respects the two Solipsis rules (see Section 5.3.1). After the convergence, mobility of avatars is simulated by injecting the rest of the trace. Evaluating Blue Banana with the real traces is irrelevant because the benefits are not significant with such small-scale NVEs.

The parameters of the simulations are: 1) 1000 avatars, 2) A surface equivalent to 9 Second Life maps, 3) 3 high density hotspots, 4) Hotspot density: 9549 avatars per square kilometer (24720 avatars per sq. mile), which is, for example, just below the density of New York City, 5) The constant acceleration of avatars during movement is 5 $m.s^{-2}$, 6) Peers have an ADSL connection with a 10Mbit download and 1Mbit upload bandwidth, 7) The network latency between peers is randomly set between 80 and 120 ms with an uniform distribution. Notice that with the constant acceleration, the *maximum* speed of avatars *between* hotspots can reach the speed of a helicopter (100 $m.s^{-1}$). This speed may seem exaggerated, but MMOG participants need to be provided with a fast mean of transportation[1]. Moreover, the constant acceleration is 5 $m.s^{-2}$, so the avatars do not instantly reach the maximal speed. In fact this speed is only reached in the worst case: when an avatar moves from a hotspot in a corner of the map to a hotspot in the opposite corner. The actual speed of most avatars is much lower: Figure 4.3 (Chapter 4) shows that 99% of the Second Life avatar movements have a length inferior to 40m, which means that the speed of 99% of the avatars does not exceed 20 $m.s^{-1}$.

5.4.2 Evaluation metrics

To highlight the qualities and the drawbacks of Blue Banana, each experiment depends on the *mobility rate* of the NVE: the proportion of avatars that have a straight and high speed movement, *i.e,* that are in the travelling state. Indeed, these avatars are the ones that need to quickly adapt their topology to the rapidly changing neighborhood. The higher the NVE mobility rate is, the faster the underlying overlay has to adapt, which means that high mobility rates are likely to cause a lot of transient failures. The mobility model, tweaked to be close to Second Life traces (see chapter 4), has a mobility rate of approximatively 5.5% (which means that the average number of avatars simultaneously in the state T is around 5.5 % at each moment of the simulation). Therefore, we vary the mobility rate between 0.5% and 11% during the evaluation. To achieve that, we vary the probabilities of the transitions that lead to the T state of the trace generator.

The following metrics are used to evaluate mobility resilience of Blue Banana:

- **Violation of Solipsis fundamental rules.** The failure of the global connectivity rule or the local awareness rule leads to transient failures (see the description of Solipsis in Section 5.3.1).
- **Knowledge of peers ahead of the movement.** In a fully decentralized context, a moving peer constantly needs to download the game state from peers ahead of its movement. Knowing peers of its future AOI for a longer time is beneficial to the moving peer, because it allows to download more information about the AOI on time. This

[1]For example, Second Life players are able to fly.

metric measures, for fast-moving avatars (in state $\tilde{T}$ of the Blue Banana state machine) for how long time, in average, a peer knows another peer ahead of its movement.

- **Exchanged messages count.** This metric measures the impact of Blue Banana on the network. The measures only count the number of messages because Blue Banana messages and Solipsis maintainance messages are small: they only contain the coordinates of the prefetched/maintained peers (a Solipsis identifier, geographic coordinates of its avatar and the IP address of the peer).

The evaluation of the second metric only takes into account the subset of avatars in state $\tilde{T}$ of the Blue Banana state machine. This specificity is due to the fact that Blue Banana sends prefetching requests only when an avatar is in this state. Yet, the proportion of that subset of avatars is extremely small: at maximal mobility rate, there are simultaneously only about 110 avatars in the state $\tilde{T}$ for 1000 avatars. The avatars that are not moving know most of their neighbors for a very long time. If they were all considered for that metric, the mean values would have been skewed, and the benefits of Blue Banana would have been difficult to evaluate.

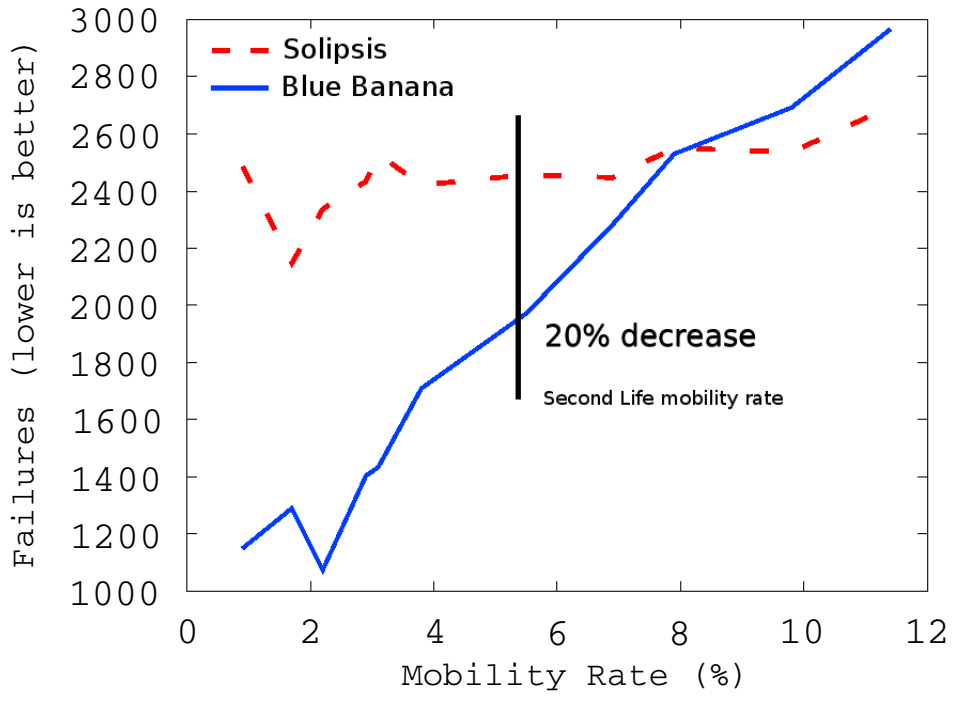


Figure 5.3: *Average number of overlay transient failures per second (lower is better).*

5.4.3 Result analysis

Figures 5.3, 5.4 and 5.5 present the evaluation for the three metrics for Blue Banana (solid lines) compared with Solipsis (dashed lines). The most interesting results for a realistic mobility rate of 5.5% show that Blue Banana (i) decreases the number of transient failures by

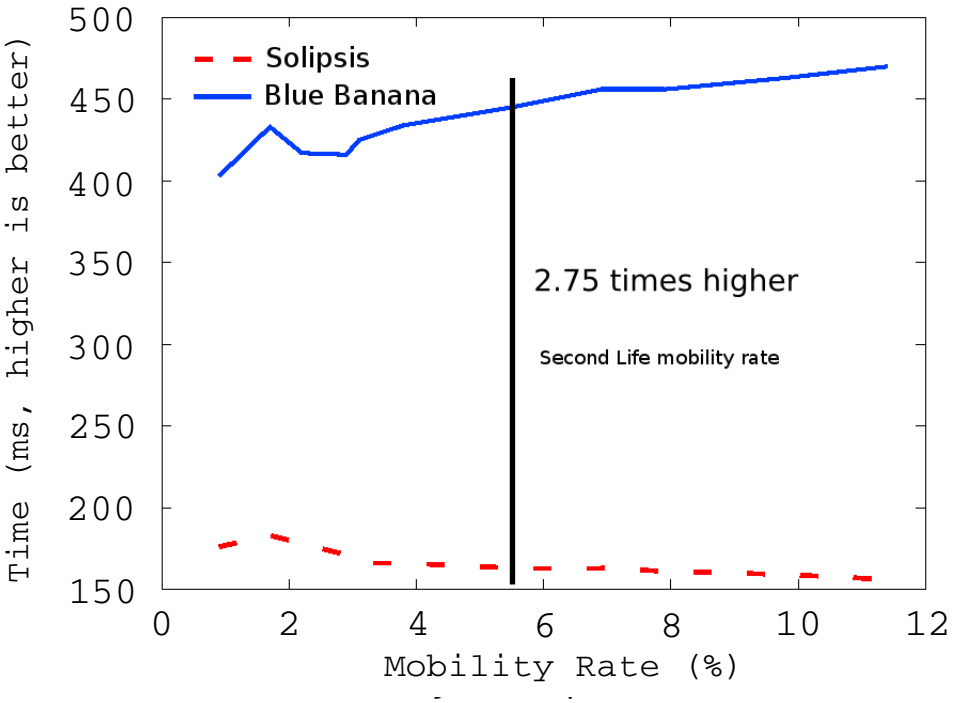


Figure 5.4: *Average knowledge time of peers ahead of movement (higher is better).*

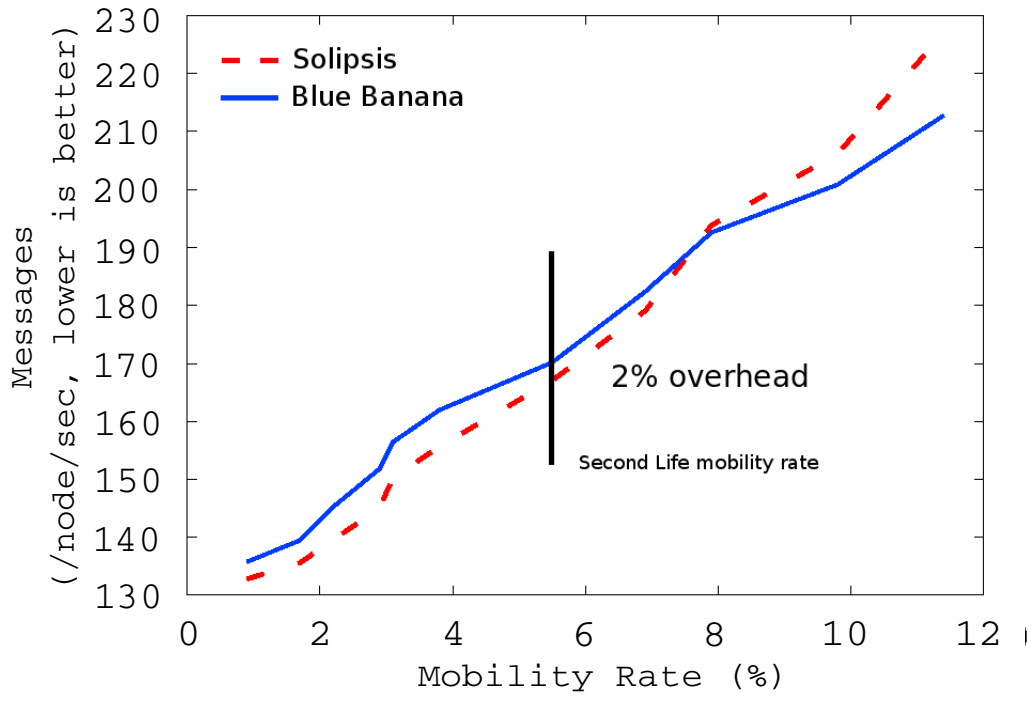


Figure 5.5: *Average number of messages sent per peer per second (lower is better).*

20%, (ii) increases the average knowledge time of forthcoming peers by 270% and (iii) generates a network overhead of only 2%. This positive results are analyzed in detail in the rest of this section.

Violation of Solipsis rules

The first metric evaluation presented in Figure 5.3 shows that the Blue Banana prefetching technique helps the Solipsis overlay to adapt itself on time, significantly reducing the number of violations of the Solipsis fundamental rules. With a mobility rate lower than 8%, Blue Banana decreases the number of transient failures. For the mobility rate observed in real traces (5.5%), the Blue Banana algorithm decreases the number of transient failures by 20%. For low mobility rates, Blue Banana helps avoiding approximatively half of the rule-violations.

As the mobility rate increases, the efficiency of Blue Banana decreases. This is due to the fact that when the mobility rate increases, the avatars of the prefetched peers are more likely to move fast and thus to become useless when the avatar reaches their supposed position. For very dynamic NVEs (mobility rate greater than 8%), Blue Banana stops helping the overlay. Most of the prefetched neighbors are also moving and when they are injected in the regular neighbor set of Solipsis they are useless, forcing Solipsis to find new interesting neighbors by itself. However, this kind of dynamicity is far above the mobility rates observed in real Second Life traces (mobility rate around 5.5%). To summarize, this first experiment shows that Blue Banana decreases the number of transient failures and suggests that the mobility of prefetched peers should be taken into account when responding to a prefetching request.

Knowledge time of forthcoming peers

The second metric evaluation presented in Figure 5.4 shows that, for fast moving avatars, the knowledge of peers ahead of the movement is far greater with Blue Banana than with the basic Solipsis overlay: a peer knows every neighbor ahead of its movement between 2 and 3 times longer than with Solipsis (2.7 times longer for a real-trace-like mobility rate of 5.5%). Moreover, subsidiary measures show that with Blue Banana, the peer of an avatar in state T is in average aware of 7.5 peers located ahead of its movement. On the other hand, a basic Solipsis peer is in average only aware of one peer ahead of its movement. These two results permit the evaluation of the average quantity of information that a fast-moving avatar can download ahead of its movement. By using Blue Banana with the Second-Life-like mobility rate of 5.5%, a peer has time to download up to about 430 KBytes of information (with a 10 down / 1 up ADSL connection) about its new AOI, versus only 20 KBytes without prefetching. This means that the NVE application can display substantially more information (about 20 times more) about the forthcoming avatars *on time*, thus clearly limiting applicative transient failures.

Network overhead

The last important result of the experimental evaluation is the low network overhead induced by Blue Banana. Figure 5.5 shows that this overhead is around five messages per peer

per second, which is almost negligible compared to the number of messages generated by the Solipsis overlay. Indeed, a basic Solipsis peer sends, depending on the mobility of its neighbors, between 130 and 230 maintenance messages per second, thus the network overhead of Blue Banana is approximatively between 1% and 3%. Moreover, these are maintenance messages, with a small, constant size (see the *Exchanged messages count* metric description). This overhead is low thanks to the fact that the prediction technique of Blue Banana is sufficiently accurate. In most of the cases, the prefetching requests provide information about peers that will be requested in the near future: as these peers are actually needed, the overlay simply takes them in the prefetched set, without emitting additional messages (see Section 5.3.2). The little overhead comes then from the wrongly prefetched peers that are not reused by the overlay. In addition to that, the update interval for rapidly moving avatars is doubled (see also Section 5.3.2), which also lowers the overhead. This optimization explains that beyond a mobility rate of 8%, the basic Solipsis overlay generates more messages than Blue Banana: as the mobility rate grows, the proportion of rapidly moving avatars increases. Therefore, the number of economized messages due to the relaxed updating proportionally grows.

5.5 Conclusion

In this chapter, we described the design and the evaluation of Blue Banana, a mechanism that improves the resilience of nearest-neighbor overlays to avatar mobility. The mechanism leverages characteristics of avatar behavior by integrating a simple mobility model that enables the peer to predict avatar movements. Having an insight of the future trajectory of the avatars helps the overlay to proactively adapt its topology to follow the movements. Our technique has only 2% of network overhead and has three main benefits:

- It decreases transient failures by 20% and helps preserving the integrity of the overlay in presence of fast movements.
- It improves the content dissemination of p2p MMOGs: The peer knows its future neighbors and can start to retrieve information about them far in advance compared to usual techniques.
- Blue Banana is generic and can be used on top of any nearest-neighbor overlay.

Beyond this context, we claim that the obtained results advocate for a better integration of player behavior in MMOG architectures. The behavior model of Blue Banana is trivially simple and only describes two possible avatar-states. Yet, exploiting that model at the architecture level brings a non-negligible increase of the overall system efficiency at a negligible cost. We argue that the integration of more sophisticated behavior models is a key to the next generation of MMOG architectures.

Chapter 6

Efficient routing in p2p MMOGs with non-uniform avatar distributions

Contents

This chapter presents DONUT[1], our third contribution. DONUT is a mechanism that monitors the avatar distribution of the NVE and dynamically adapts the topology of the underlying nearest-neigbor overlay to preserve efficient routing.

[1]DONUT: Density-aware Overlay for Non-Uniform Topologies.

6.1 Introduction

Problem statement

Nearest-neighbor overlays are a key component of many p2p MMOGs because they provide efficient content dissemination. The placement of a peer in the nearest-neighbor topology is determined by the position of its avatar in the NVE, as described in Section 3.2.2. Consequently, when an avatar joins the NVE or teleports to a distant region, its peer has to be routed to its new location in the topology. Existing MMOGs experience high churn rates and many of them provide a teleportation service for efficient transportation. Scalable routing is thus a key requirement of current MMOG applications.

For that, the topology of nearest-neighbor overlays enables decentralized greedy routing. To make the process efficient at large scale, shortcuts, or *long range links* are added to the nearest-neighbor graph. Kleinberg showed that the optimal routing process is achieved in a grid if the probability for a peer p to choose a peer q as its long-range link depends on the graph distance[2] between p and q. Kleinberg's distribution of the shortcuts is called *d-harmonic* [126]. To reach this distribution, a peer needs to find peers that are at the appropriate graph distance in the overlay (see section 2.1.4).

Implicitly, it requires for a peer to know, or at least have an approximation of the overlay topology. When the distribution of the avatars inside the NVE is uniform, this knowledge is straightforward: the graph distance *linearly* increases with NVE distance. However, all the studies show that avatar distributions of existing MMOGs are highly skewed. In that case, estimating graph distances requires the knowledge of the NVE density distribution. Existing solutions, such as Oscar [104] and Mercury [48] are using random sampling techniques to probe the distribution during the rewiring process. However, MMOG workload distributions are highly dynamic, and peer-clusters can form and vanish at any time, making small-world shortcuts obselete. Without density monitoring, existing solution do not notice the modification and can only observe and react to the routing performance degradation.

Contribution

We introduce DONUT, a decentralized mechanism that monitors the density distribution of the NVE. DONUT allows each peer to acquire and dynamically maintain an approximation of the global avatar density distribution. The peer can thus detect changes in the distribution and adapt its shortcuts before the routing performance degrades. At any time, a peer can locally estimate the graph distance to any coordinate of the keyspace with the density map. The peers use this estimation to build density-aware small-world shortcuts in the overlay graph. The rewiring process is very lightweight, because the localization of appropriate peers is made *locally* by using the map.

The emergence of a global density map on each peer is the result of a collective effort. Each peer monitors the NVE density that surrounds its avatar and propagates an approximation of its local knowledge through the overlay by piggibacking existing lookup messages and by using a dedicated gossip algorithm. The evaluations of DONUT show that the global density map can help to improve the routing process by 20% compared to state-of-the-art

[2] we call graph distance the shortest path, in number of hops between the peers.

techniques. Our mechanism requires only a few bytes per second per peer, and is thus affordable even for low-capacity peers.

The knowledge of the global system state is highly valuable. Apart from efficient routing, it can also be useful to other distributed system mechanisms, such as efficient global load balancing, system monitoring, or network size estimation. The contributions of this work are thus: 1) a distributed algorithm provide each peer with an approximation of the NVE density distribution and 2) an technique that uses this density map to build efficient long-range links, illustrating its practical value.

6.2 Theoretical background

The theoretical background of this work mainly relies on Kleinberg's small-world model [126] and its generalization to non-uniform keyspaces by Girdzijauskas *et al.* [103].

To create the long range links, we use the *log-partition algorithm*, successfully employed in Oscar [104] (see Chapter 2.1.4). A brief description of the algorithm is therefore necessary to the understanding of this Chapter. Let p be the peer executing the algorithm. All peers are locally sorted according to their graph distance to p. Peers which distance to p is bigger than the median form the first set of the partition (the set 1 in Figure 6.1). Then, the set of peers which distance to p is lower than the median is halved the same way. The process is repeated until the subset contains only the close range neighbors of p (the set 4 of Figure 6.1 is the last step of the algorithm). Finally, p chooses uniformly at random a long-range link in each subset. The formed set of links follows the Kleinberg's distribution [104].

Since it is impossible for p to retrieve information about the graph distance to all peers, Oscar estimates the subset's population by using random walks. At each step of the partition algorithm, it performs a constant number c of bounded random walks inside the current interval. The result after $c \times log(n)$ bounded random walks is a partition of the keyspace built by taking into account the distribution of peers. However, no global peer distribution is actually built. Evaluations of Oscar described in [104] show that it outperforms other

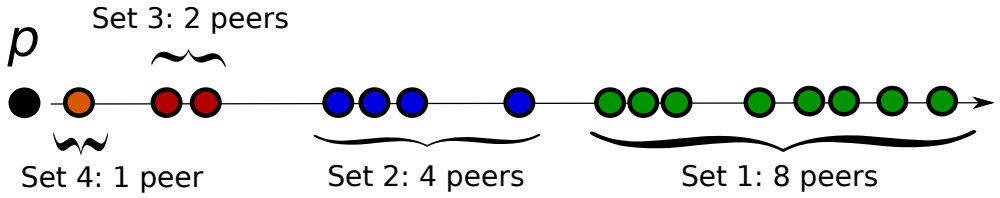


Figure 6.1: *Population partition formed by the log-partition algorithm.*

systems over heterogeneous-keyspace overlays, proving the efficiency of the log-partition algorithm.

On a broader level, our work is also related to overlays that exploit peer clustering to build multi-tier nearest-neighbor topologies [181, 192]. Similarily to these proposals, our peers monitor the avatar density distribution, and adapt their logical links to better accommodate the load. However, the main objective of these systems is to reduce the maintenance overhead induced by skewed distributions, and no mechanism ensures efficient routing.

6.3 Building a global keyspace density map

Each DONUT-peer maintains a view of the keyspace density distribution. This section describes the algorithms used to build such a density map.

6.3.1 Overview

On peer, the map is implemented by using a tree that locally indexes each region of the map. Since we choose to implement a bi-dimensional keyspace to illustrate our contribution, we need to use a quadtree to correctly index all the regions of the map (see Fig 6.2).

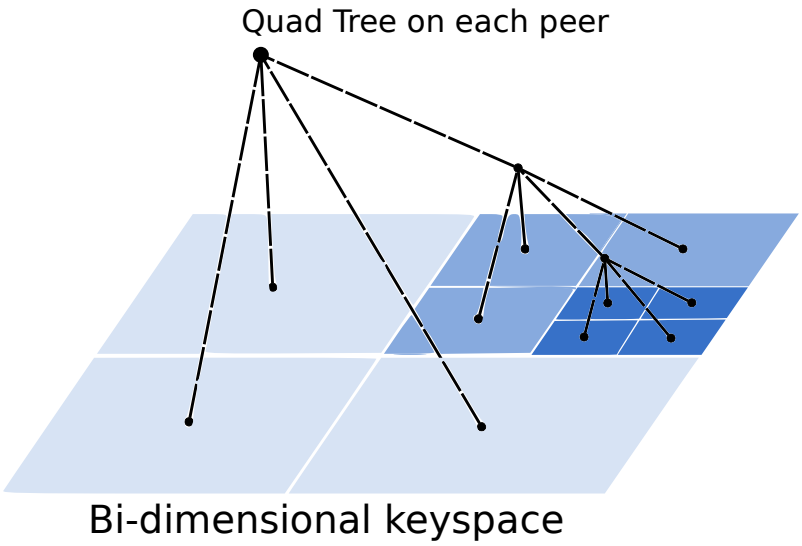


Figure 6.2: *Local quadtree that stores density information on each peer.*

Each node of the quadtree is responsible for a square region of the map. A leaf of the tree may engender four children representing the four cardinal directions. In that case, its region is split in four equal subsquares, and the responsibility for each subsquare is given to the corresponding child (*e.g.*, the upper left square is given to the North Western child). Thus, the union of the subsquares forms a partition of the keyspace (see Figure 6.3). Each leaf of the tree also contains information about the peer density inside the region it is responsible for. When a leaf is split, its children inherit its density value. Therefore, the set of the quadtree's leaves forms an approximation of the keyspace density distribution function.

At the bootstrap, a peer holds no information about the keyspace distribution: the map has to be filled with density information. There are two possible ways for a peer p to refine its map: 1) use local information and 2) receive information from other peers.

At the beginning, the only available information is local information. As the peer joins the overlay, it is assigned coordinates in the keyspace by the distributed system and is routed to its location in the overlay according to these coordinates. During the process, it obtains the coordinates of its overlay neighbors. Thanks to that, it is able to determine the density of the keyspace that contains its coordinates and its neighbors' ones. The density is obtained

by computing the surface of a circular portion of the keyspace centered on the peer's coordinates $coord$, with a radius rad equal to its distance to the farthest neighbor's location. This area is then divided by the number of neighbors of the peer to obtain a density d. The triplet ($coord$,rad,d) forms the local information about the keyspace density.

After the insertion of the local information, every peer locally stores a map of the keyspace with an estimation of its surrounding density (see Figure 6.3.left). The union of all the peer maps forms an accurate density mapping of the whole keyspace (see Figure 6.3.right). Therefore, to fill their maps, peers need to exchange their local information.

The first step of the map completion is performed by Algorithm 2, and the insertion of the information received from other peers is done by Algorithm 3. It is important to highlight that once the density information has been received, no operation is performed on the network. All the steps of Algorithms 2 and 3 are executed *locally*. It is important to notice that *nodes* composing each quadtree should be distinguished from *peers* forming the overlay in the rest of the Chapter.

6.3.2 Adding new information to local map

Initially, the map contains no density information and the root of its quadtree has no children. Thus, the local responsibility of the whole keyspace is given to the root, and a default density value[3] is assigned to the root. The triplet $(coord, rad, d)$ is then locally inserted in the quadtree following Algorithm 2.

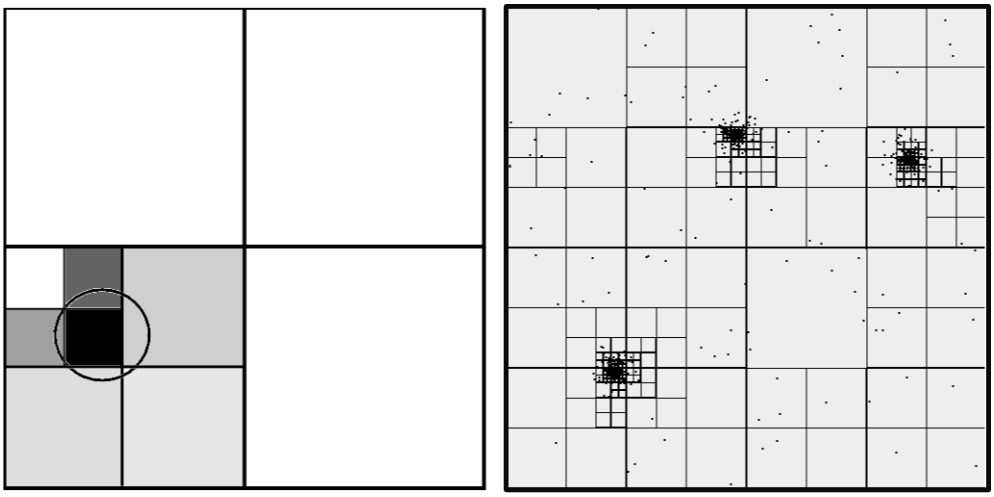


Figure 6.3: ***Left:*** *After local execution of Algorithm 2 on a void map. (the greyscale represents the density, black being the highest).* ***Right:*** *Approximation of a keyspace with three density zones.*

The goal of the algorithm is to approximate the circular density area by a set of squares. The size of the squares may be variable, but in order to achieve a reasonable approximation accuracy, their size must be smaller than the circle. Therefore, the algorithm needs to find suitable squares in the quadtree and assign correct density values to each square. It explores

[3]This value is set to zero in our evaluations, but an approximation of the mean overlay density may help.

the quadtree in search of leaves which squares 1) intersect the circle and 2) are smaller than the circle.

The algorithm progressively "zooms" on the square that contains *coord*, the origin of the circle (loop from line 2 to 15). If the square that contains the origin is bigger than the circle, it is split (lines 3 to 4). Then, the value $coef$, which is the proportion of the surface occupied by the intersection with the circle, is computed for each of the current node's subsquares except the next subsquare on the path of the zooming process (lines 8 to 14). The influence of the circle's density on the updated density of the subsquare is proportional to $coef$ (line 10). If the child responsible for the subsquare is a leaf, its density value is simply updated (line 12). Otherwise, the value is propagated to its children (line 14). The propagation to the children is also performed by calculating the intersection with the circle for each of the descendants. The zooming process stops when the side of the square that contains the origin of the circle is smaller than the diameter of the circle.

Once the center of the approximation has been found, the algorithm ends by computing its new density (lines 16 to 22). This is done by calculating its intersection with the circle (lines 16 to 18), and propagating the value if the central square is not a leaf (line 22).

Algorithm 2: Upon the insertion of $(coord, rad, d)$ in the quadtree.

```
   Result: Information about the local density is incorporated to the map.
 1 currentSq = rootOfQuadTree ;
 2 while currentSq.sideSize > 2 × rad do
 3     if isLeaf (currentSq) then
 4         currentSq.splitNode ();
 5     next = currentSq.subSquareContaining (coord);
 6     foreach child in currentSq.children do
 7         if child is not next then
 8             intersection = overlapCircleSquare (child,coord,rad);
 9             coef = intersection / child.surface ;
10             newDensity = coef× d +(1 − coef) × child.density;
11             if isLeaf (child) then
12                 child.density = newDensity ;
13             else
14                 child.propagateToChildren (newDensity);
15     currentSq = next ;
16 intersection = overlapCircleSquare (currentSq,coord,rad);
17 coef = intersection / currentSq.surface ;
18 newDensity = coef× d +(1 − coef) × currentSq.density;
19 if isLeaf (currentSq) then
20     currentSq.density = newDensity ;
21 else
22     currentSq.propagateToChildren (newDensity);
```

6.3.3 Exchanging information between peers

Merging trees

On each peer, information about the density of a map region r is held by the subtree which root is responsible for the area that contains r. Thus, a peer a willing to share information about r with a peer b sends to b a message containing the corresponding subtree. Upon

the reception of the update message from a, b executes Algorithm 3 in order to merge the received subtree with its local quadtree.

First of all, b needs to locate the region of its map that is concerned with the update. Each node in the quadtree is in charge of its own region of the keyspace map, and two nodes cannot be responsible for the same region. Thus, a region referenced by a node of the tree may be identified by the location of its node in the tree. As b receives an update for a region of the map, it is able to locate the corresponding node in the local quadtree (lines 4 to 7). However, the node responsible for that region may not exist in the local quadtree. In that case, the merge algorithm splits the quadtree until the node is created (lines 5 to 6).

Then, the merging process may begin. Let u be the root of the subtree received in the update message, and l the local node in charge of that region on the map. There are four possibilities at this stage of the algorithm: 1) u and l are both leaves. In that case, the density value of u is simply assigned to l because u is more recent than l (line 10); 2) l is a leaf and u is not. Then the subtree of u is assigned to l (line 12); 3) u is a leaf and l is not. Then, the subtree of l is deleted (line 15) and the value of u, more up-to-date, is assigned to l; 4) none of the nodes are leaves: the merge algorithm is recursively called on each child of the nodes (line 18). At the end of the algorithm, the local map of b holds new information about the density in the region r of the keyspace. Algorithm 3 is designed to erase obsolete information (line 15), which is important since the density of the keyspace is likely to evolve over time.

Algorithm 3: Upon the receipt of a quadtree update from a distant peer.

```
   Result: The distant update has been merged with the local tree.
 1 currentSq = rootOfQuadTree ;
 2 receivedRoot = root of the received update-subtree;
 3 mergeSubtree
 4     while not representSameRegion (currentSq,receivedRoot) do
 5         if isLeaf (currentSq) then
 6             currentSq.splitNode ();
 7         currentSq = currentSq.subSquareContaining (coord);
 8     if isLeaf (currentSq) then
 9         if isLeaf (receivedRoot) then
10             currentSq.density = receivedRoot.density ;
11         else
12             currentSq.subtree = receivedRoot.subtree ;
13     else
14         if isLeaf (receivedRoot) then
15             currentSq.deleteSubtree ;
16         else
17             for childId in NW, NE, SW, SE do
18                 mergeSubtree (currentSq.child [childId ], receivedRoot.child [childId ]);
19 end mergeSubtree
```

Propagation of information

We implemented two distributed mechanisms to propagate the subtree-updates among peers. First, we used a slightly modified gossip algorithm. An important property of our mechanism is the fact that peers that are semantically close to each other have extremely correlated maps. Thus, the updates between them would be nearly useless. For this reason, in our gossip protocol, a peer p locally assigns priorities to its overlay neighbors. The

priority of a neighbor is inversely proportional to its graph distance to p. It means that a peer is likely to propagate updates only through its long-range links. Second, taking into account that the keyspace distribution follows data popularity, and is therefore likely to evolve over time, our mechanism should avoid the propagation of outdated information about the distribution. Therefore, the data propagated to the neighbors during the gossip process is selected by novelty: recent information is propagated first.

Another possibility is to use join messages of arriving peers. As a peer j joins the overlay, it first connects to an entry point peer e that may be semantically located anywhere in the overlay. Semantic coordinates are then assigned to j according to some attribution mechanism. The coordinates of j are usually semantically far from e, and j has to reach its semantic neighbors through the overlay. A join request is routed by e to the semantic coordinates of j. At each step, it is possible to add some information about the density surrounding the current-hop peer. Each time the request reaches another peer on its route, the latter can benefit from the knowledge accumulated inside the request during the previous hops. It is rather an efficient way to propagate density knowledge. However, our evaluation shows that the amount of exchanged data increases with churn, while the increase is very moderate for the gossip propagation.

Using one of the mechanisms described above is necessary to assimilate changes in the key distribution that occur over time. On the other hand, a peer that just joined a fully bootstrapped overlay has no need to rebuild a full map from scratch. The map is relatively lightweight (see Section 6.5) and can be recovered from an overlay neighbor during the join process.

6.4 Drawing density-aware shortcuts

Thanks to the collective use of the algorithms described in the previous section, peers progressively acquire an approximate map of the keyspace density distribution. This section describes how this knowledge may be used to build efficient long-range links in order to decrease the latency of the message routing.

6.4.1 Graph distance estimation

Thanks to the algorithms described in the previous section, each peer owns a density map of the bi-dimensional keyspace. This information allows it to locally *estimate* its graph distance to any coordinates of the keyspace. The difference between keyspace distance and graph distance is exposed in Figure 6.4. Namely, having the density distribution and the semantic distance, a peer is locally able to approximate the graph distance to the coordinates.

Let src be the coordinates of the peer, $dest$ the coordinates of the keyspace on which the estimation is performed, $d_{[src,dest]}$ the semantic distance and $GD_{[src,dest]}$ the graph distance between them. If the key distribution is uniform all the way from src to $dest$, $GD_{[src,dest]}$ is roughly proportional to $d_{[src,dest]}$. More precisely, the semantic distance is equal to the number of hops to reach $dest$ in the graph multiplied by the mean euclidean distance of one hop multiplied by a shrinking constant k, *i.e.*, $d_{[src,dest]} = GD_{[src,dest]} \times meanHopDist \times k$.

The constant k is added because geometrically, the euclidean distance between src and $dist$ is shorter than the sum of the lengths of all the hops in the graph. The exact value of k

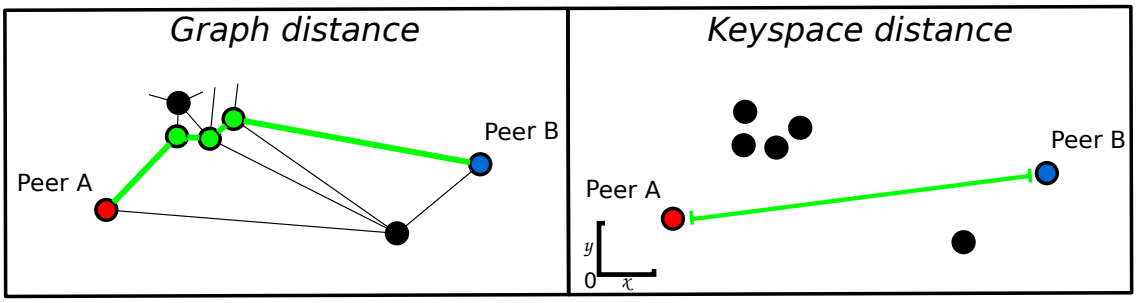


Figure 6.4: *Graph distance in number of hops vs Keyspace distance (euclidean distance) between peers A and B.*

depends on the topology of the graph, but can be empirically approximated. For instance, for the Delaunay graphs used in our simulations (see Section 6.5), we have $k = 0.5$. We now need to calculate $meanHopDist$, the mean euclidean distance of one hop.

Assume n points uniformly distributed inside a square SQ of size $S = side \times side$. It is possible to approximate the mean euclidean distance between two points that are neighbors in the square. Let $meanDist_x$ and $meanDist_y$ be the projections of that mean distance on the x-axis and the y-axis. Since the nodes are uniformly distributed across SQ, we have $meanDist_x = meanDist_y = \frac{side}{n}$. That allows us to compute the mean distance between two neighbors in the square which is $meanHopDist_{SQ} = \sqrt{meanDist_x^2 + meanDist_y^2}$.

In case of heterogeneous distributions, regions crossed by the line $[src, dest]$ have different densities, so that the graph distance is no more proportional to euclidean distance. Therefore, the peer has to retrieve from the map information about the density distribution on the way from src to $dest$. The peer first determines the set $S = \{sq_1, sq_2, ..., sq_n\}$ of map square-zones that intersect the segment $[src, dest]$. Let $L = \{l_1, l_2, ..., l_n\}$ be the set of segments of $[src, dest]$ formed by its intersections with S. It is important to notice that:

- L forms a partition of $[src, dest]$. Therefore we may assume $GD_{[src,dest]} = \sum_{i=1}^{n} GD_{l_i}$.
- The quadtree contains one density-value per square, so the density inside each square is considered to be uniform. Therefore, $\forall i$, GD_{l_i} can be computed by the peer.

Thanks to the two previous assertions, the peer is able to approximate the graph distance *locally*.

6.4.2 The rewiring process

We use the log-partition algorithm described in Section 6.2 to achieve a d-harmonic distribution of long-range links. However, no random walks are performed. Instead, at each step of the algorithm, our system locally *estimates* the subset's population using the technique described above. To bootstrap the rewiring process, a peer p needs to find coordinates k in

the keyspace that are estimated to be the farthest in terms of graph distance. This is done in order to reach as many peers as possible with the future long-range distribution[4].

The localization of the farthest coordinates in terms of graph distance is not straightforward. Indeed, with heterogeneous distributions, the coordinates with the highest semantic distance are not necessarily the farthest in terms of graph distance if the keyspace has more than one dimension. As random uniform sampling of the map in search of the maximal distance is inefficient, another technique has to be employed. We propose to use the following property: if a node Y is the farthest from a node X in terms of graph distance, X and Y are responsible for opposite semantic values in at least one dimension of the keyspace.

Formally, let U be a n-dimensional borderless semantic keyspace, min_{dim} and max_{dim} the minimal and the maximal value of U for the dimension dim. Consider $x = (x_1, x_2, ..., x_n)$ and $y = (y_1, y_2, ..., y_n)$ the coordinates in U of two nodes X and Y of the topology supporting U.

Property 1 *If Y is the farthest node from X in terms of* graph distance, *then Y is responsible for coordinates $c = (c_1, c_2, ..., c_n)$ of the keyspace such that: $\exists dim \in [1..n] : |x_{dim} - c_{dim}| = \frac{(max_{dim}+min_{dim})}{2}$*

Proof:

1. Each dimension dim of U is borderless (*i.e.*, min_{dim} is semantically close to max_{dim}). Due to the modulo, the maximal semantic distance achievable in dim is $dmax_{dim} = \frac{(max_{dim}+min_{dim})}{2}$.
2. Let X and Y be two nodes and d_{dim} the value which is at a distance $dmax_{dim}$ from X in $dim \in [1..n]$ with $|d_{dim} - x_{dim}| = dmax_{dim}$. Let M be the farthest point from X belonging to (XY) in the direction of Y. Then, $\exists i$ such that $M = (m_1, ..., d_i, ..., m_n)$. Assume that Y is not responsible for M. Therefore, M is located farther from X than any coordinates of (XY) owned by Y. Thus, whether a) M is not owned by any node or b) M is owned by another node Z. The supposition a) is impossible because all the coordinates have a root. The supposition b) implies that thanks to the greedy routing, the graph distance to reach Z from X is higher than the graph distance to reach Y *i.e.*, Y is not the farthest from X in terms of graph distance. By 1) and the contrapositive of 2), the property is correct. □

In our case, the two dimensions have the same size, therefore $min_1 = min_2 = min$, $max_1 = max_2 = max$ and $dmax = \frac{(max+min)}{2}$. Let P be a peer of the overlay with coordinates $p = (p_1, p_2)$, and $S = \{x \in U : max(|p_1 - x_1|, |p_2 - x_2|) = dmax\}$. The property implies that $\exists M \in S$ such that M is owned by the peer that is the farthest from P in the graph. The coordinates of S form a square centered on P with a side of size $dmax$.

The peer P uses a Monte Carlo method [151] on its density-map to uniformly sample a set of coordinates that belong to the square formed by S. Then, the samples are ordered by their estimated graph distances to the peer. The sample M for which the estimated graph distance is the highest is supposed to be owned by the farthest peer. A shortcut-request is routed to these coordinates in the overlay, and the peer responsible for it becomes the longest link of the routing table.

[4]This procedure also gives an estimation of the graph diameter, and thus of the overlay size, which may be useful.

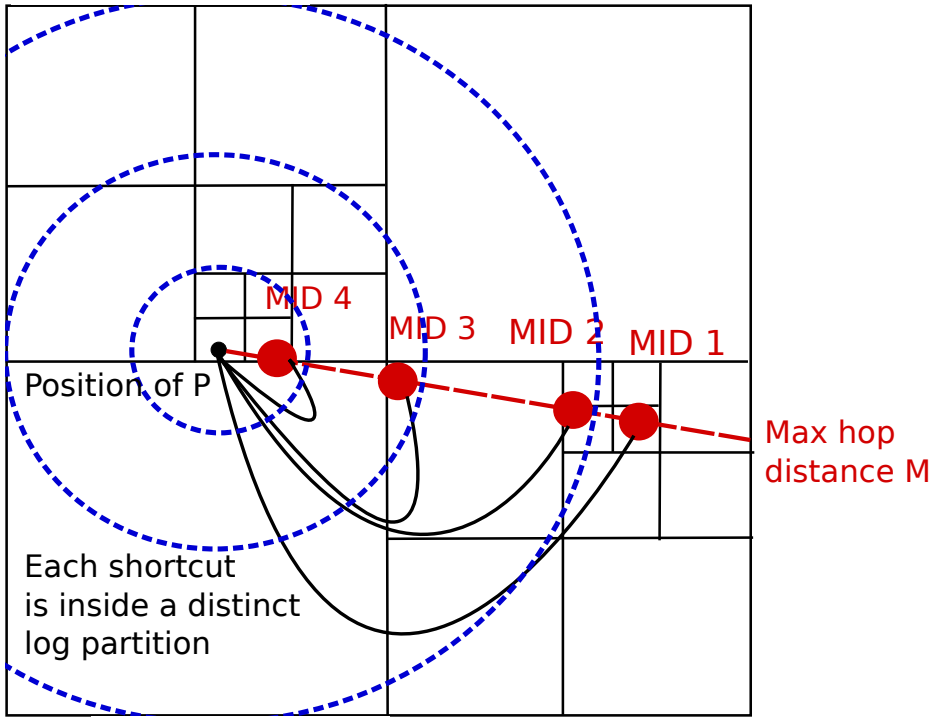


Figure 6.5: *Execution of the log-partition algorithm on the local map of the peer P.*

Then, as depicted in Figure 6.5, P determines by dichotomy the coordinates MID that belong to the segment $[PM]$ for which the estimated graph distance is the half of the estimated graph distance to M[5]. A long-range request is routed to MID and the process is repeated with the segment $[P, MID]$. Similarly to Oscar, the process ends when one of the close neighbors of P is reached (or when the wanted number of links has been created).

If the process has ended without achieving the wanted number of long-range links, a random member of S is selected, and the process is repeated until the wanted number of shortcuts is reached. At the end of the algorithm, the peer has the wanted number of long-range links that are distributed following an approximation of the d-harmonic distribution.

Putting aside the cost of the density map maintenance protocols, the rewiring process is extremely lightweight: all estimations are done locally. Therefore it requires only one routing process per long-range link. Since the number of links to add follows $log(n)$ and the routing process is still efficient during the rewiring of a peer, the overall cost in number of messages is expected to grow following $log^2(n)$.

[5] *I.e.*, the segment $[PM]$ is divided in two semantically equal parts. If the graph distance of the semantic middle is less than half of the graph distance to M, the second interval is taken, and so on.

6.5 Evaluation

This section presents a detailed evaluation of DONUT. We first describe the evaluation environment. Then, we compare the long-range rewiring process of DONUT to state-of-the-art rewiring techniques. Finally, the behavior of the map maintenance algorithms is studied. All the systems have been evaluated using PeerSim [118].

6.5.1 Evaluation environment

To perform our evaluations we simulate a bi-dimensional keyspace. This choice was made because such keyspaces are used in many architectures for MMOGs [49, 121]. The keyspace forms a square, each side of which is bound to the opposite side to ensure the modulo. We choose to use a Delaunay triangulation topology to support the keyspace. This choice is believed to be relevant because 1) the greedy routing algorithm is proven to work for all Delaunay based topologies [54]; 2) such topologies are already used by overlay designers [114]; 3) the triangulation is generalizable to higher dimensions.

The environment of our simulation is dynamic: peers join and leave the overlay. We use both synthetic and real traces. Our synthetic churn model follows an exponential law for disconnections and a Poisson law for joins. This model is widely used in the literature, *e.g.,* to simulate the availability of electronic components. The parameters of the two laws are bind to ensure a roughly constant overlay size. In our evaluations with that model, we vary the mean session time of peers from 10 minutes to 6 hours. The mean session times we observed in the peer-to-peer churn traces is bound by these values.

Our semantic keyspace, produced by our trace generator described in Chapter 4, is formed by three high density zones that contain 90% of the overall peer population. The distribution of peers inside each density zone follows a Zipf-law. The rest of the population is uniformly distributed between the hotspots. Such distributions seem to be rather common for existing keyspaces (see Section 3.1.2). For instance, this density distribution is shown to be comparable to the population distribution of Second Life, a popular MMOG [136].

In real life, the dynamicity of the environment impacts the characteristics of the keyspace. When a peer joins the overlay, it is inserted in the keyspace with new coordinates. The coordinates may be defined by the overlay *e.g.,* the peer joins the most overloaded region, helping to support the load. It is also possible that the coordinates are user-defined (in a distributed game, the player chooses to join a particular region). As the application load varies in time, and popularity zones may evolve, the density of the semantic keyspace changes over time. In order to study the adaptive capabilities of DONUT, the coordinates of all the density hotspots periodically change. The mean session time also impacts the keyspace: once the hotspot coordinates have changed, it determines the new hotspot growth-speed.

Finally, to make our simulations more realistic, we use:

- A matrix containing real-latency traces,
- Real-system churn traces.

We use real latencies that were measured by Madhyastha et al. between 2500 hosts spread over the Internet [143]. We use churn traces collected from several existing dis-

tributed systems, such as Overnet [46] or Skype [107]. Traces of desktop personal computer usage were also injected [51][6].

Since the latency matrix has 2500 entries, except for Figure 6.6, all synthetic evaluations were performed with an overlay size of 2500. We simulate *one week* of activity. The gossip map-maintenance protocol period is set to ten minutes, and is allowed to propagate 60 Kbytes to three direct neighbors per period. For the propagation by lookups, each lookup message may contain 10 Kbytes of map data. The rewiring protocol occurs once per hour, and the coordinates of all hotspots change once per 24 hours.

6.5.2 The rewiring process evaluation

We compare DONUT to several techniques of long-range link construction, namely:

- **Random approach:** n coordinates are selected uniformly at random in the keyspace.
- **Uniform Kleinberg approach:** the log-partition algorithm described in Section 6.2 is used, and the keyspace is assumed to be uniform. Therefore, the graph distances are supposed to be proportional to semantic distances.
- **Near optimal approach:** the peer locally has a real-time copy of the current topology. The estimated graph distance equals the real graph-distance.
- **Oscar:** algorithm described in [104] (see Section 6.2). However, the original Oscar algorithm is designed for one-dimensional keyspaces. Therefore, we have extended Oscar to our bi-dimensional keyspace.

Each time a peer joins the overlay, the join message is routed to the semantic position the peer has chosen. We measure the performance of that routing process[7]. The evaluation shows that DONUT outperforms all the strategies (except the Near Optimal one) by at least 20%.

A first important result concerns the Oscar strategy, which exhibits poor results in our bi-dimensional keyspace. For each measurement, Oscar had only slightly better results than the random strategy. In particular, Figure 6.6 shows that the number of hops during the greedy routing linearly grows with the overlay size. The result is a bit surprising, because the system performed well in one-dimensional keyspaces [104]. We believe that the problem comes from random walks used by Oscar to sample the keyspace. This sampling technique is successful only on graphs that have good expansion properties [48]. Flaxman showed that the graph used by Kleinberg (a bi-dimensional lattice with d-harmonic shortcuts) is not an expander [98]. Figure 6.6 shows that except the random strategy and Oscar, all the rewiring processes scale well. The increase of the path-length seems to be logarithmic for DONUT as well as for the Uniform Kleinberg and the Near Optimal approaches. Like in the other evaluation results, DONUT is the closest to the Near Optimal approach, showing that most of the graph distance estimations performed by DONUT are accurate. The bad scalability of the random approach comes from the fact that the coordinates of future long-range links are chosen uniformly across the keyspace. The links are thus likely to "miss"

[6]These traces are publicly available: `http://fta.inria.fr/apache2-default/pmwiki/index.php`.
[7]Both latency and number of hops.

most of the density hotspots, increasing the inefficiency of the approach. We believe that Delaunay graphs with d-harmonic shortcuts are not expanders. This results in non uniform sampling of the keyspace, skewing the estimation. Indeed, when the random walks are replaced by a cheat mechanism that uniformly picks random nodes from the node-set of the simulator, Oscar shows much better performance.

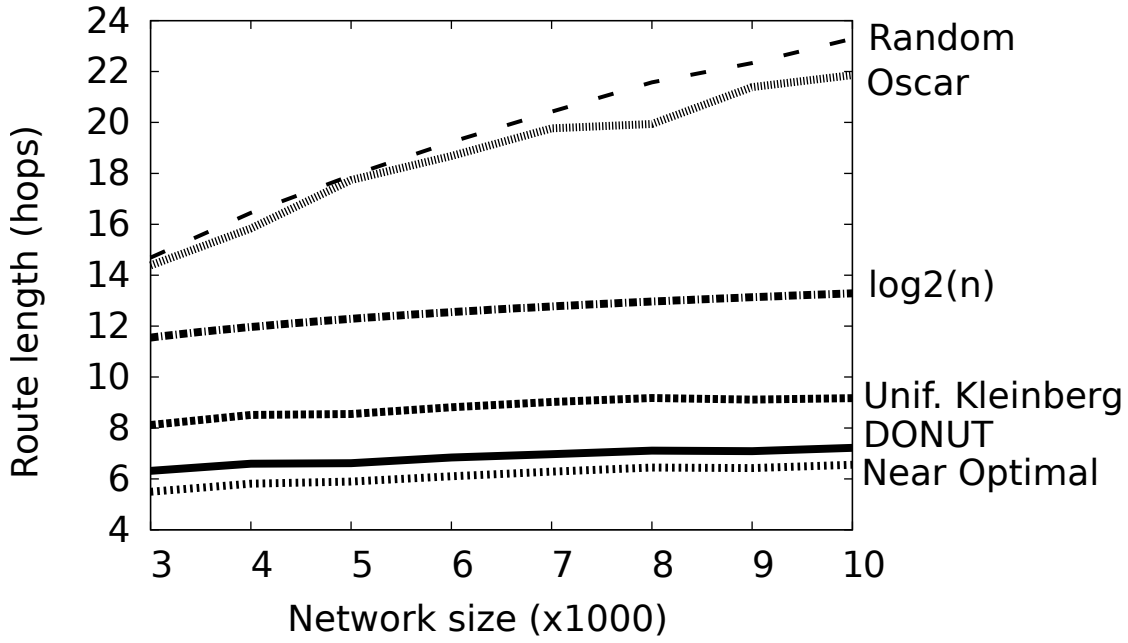


Figure 6.6: *Scalability test: Mean route length in function of the overlay size.*

Figure 6.7 describes the evolution of the route length while varying the mean session time. We can see that the churn rate has no significant influence on the Near Optimal, Uniform Kleinberg and DONUT approaches. For the Near Optimal technique, it happens because each peer locally has a real-time graph of the topology and is therefore perfectly aware of the topology for any churn rate. The Uniform Kleinberg approach assumes the uniformity of the keyspace and thus behaves the same way despite the churn rate increases. On the other hand, the fact that the efficiency of DONUT does not decrease shows that the map adapts on time and is again accurate when the rewiring is performed. The positive impact of the density map on the rewiring process is confirmed while using real traces. Figure 6.8 shows the evolution of the mean routing latency while using churn traces from Overnet. The graph distance estimation provided by DONUT is almost as efficient as the use of a real-time graph. One may notice that DONUT is sometimes even better than the Near Optimal strategy. However, the difference is very slight, and is probably due to the standard deviation of the measurements. Figure 6.9 recapitulates the average latencies and the standard deviations for all the strategies while injecting the Overnet churn trace.

6.5.3 Global map propagation evaluation

The first part of the evaluation showed that our density map was helpful to the rewiring of the overlay. However, the construction of a *global* map may seem to be a costly process.

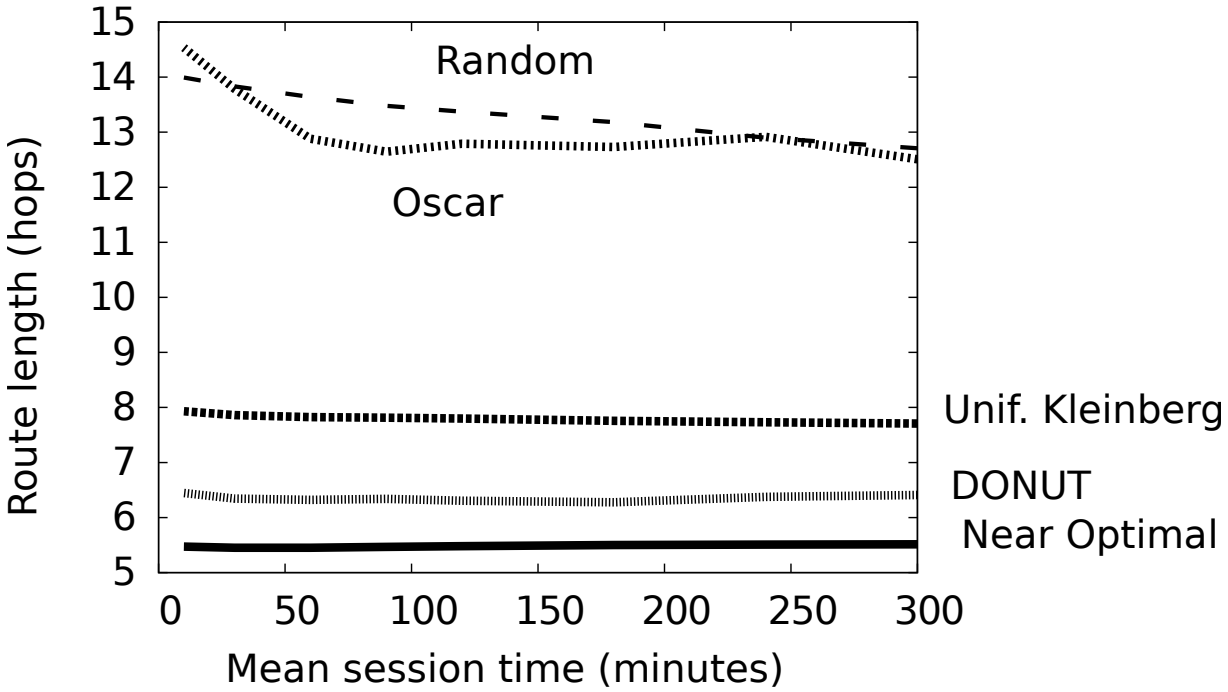


Figure 6.7: *Mean route length as a function of the mean session time, lower is better. Lower session time means higher churn.*

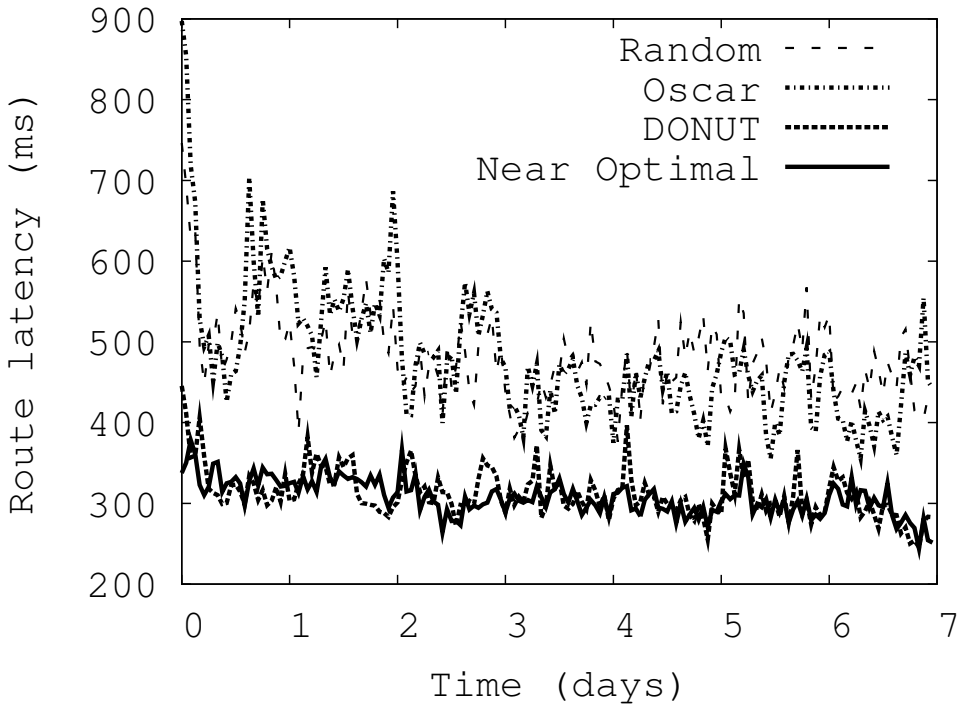


Figure 6.8: *Evolution of routing latency with the Overnet trace.*

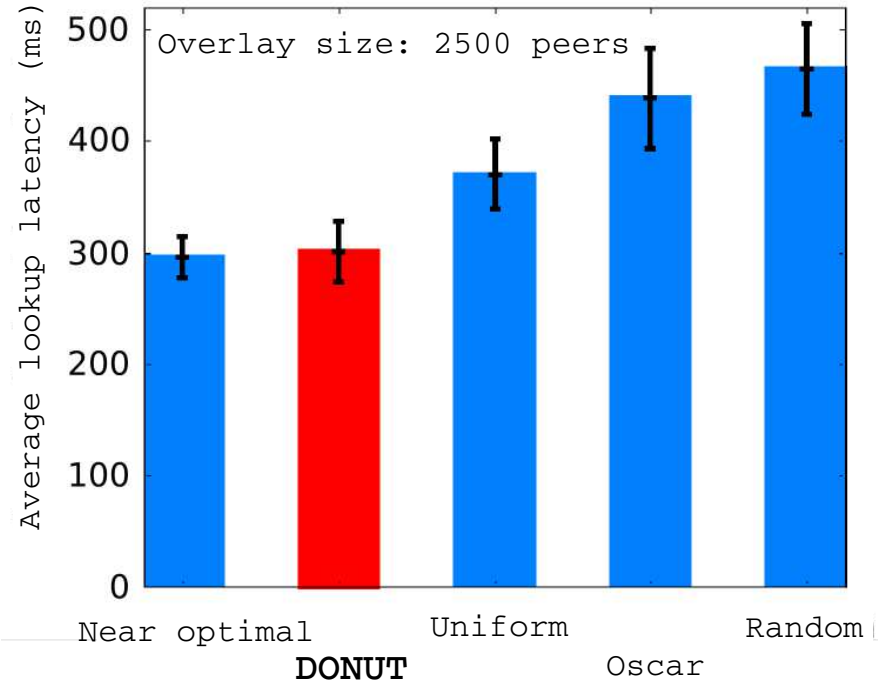


Figure 6.9: *Average latency for all the rewiring strategies with the Overnet trace and the real latency matrix.*

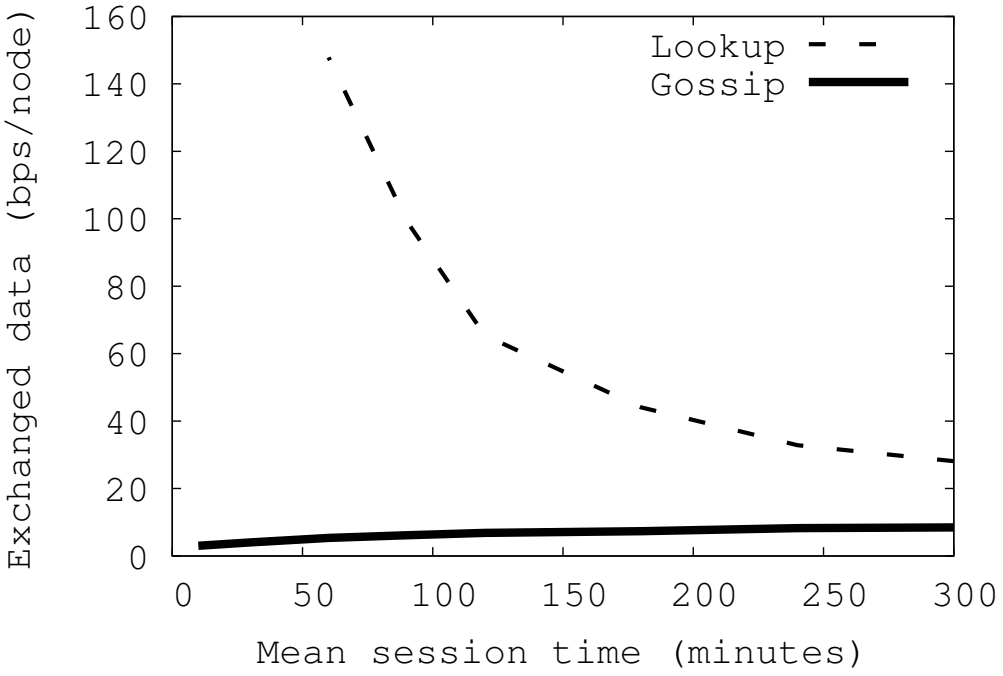


Figure 6.10: *Sent data per node in function of the mean session time.*

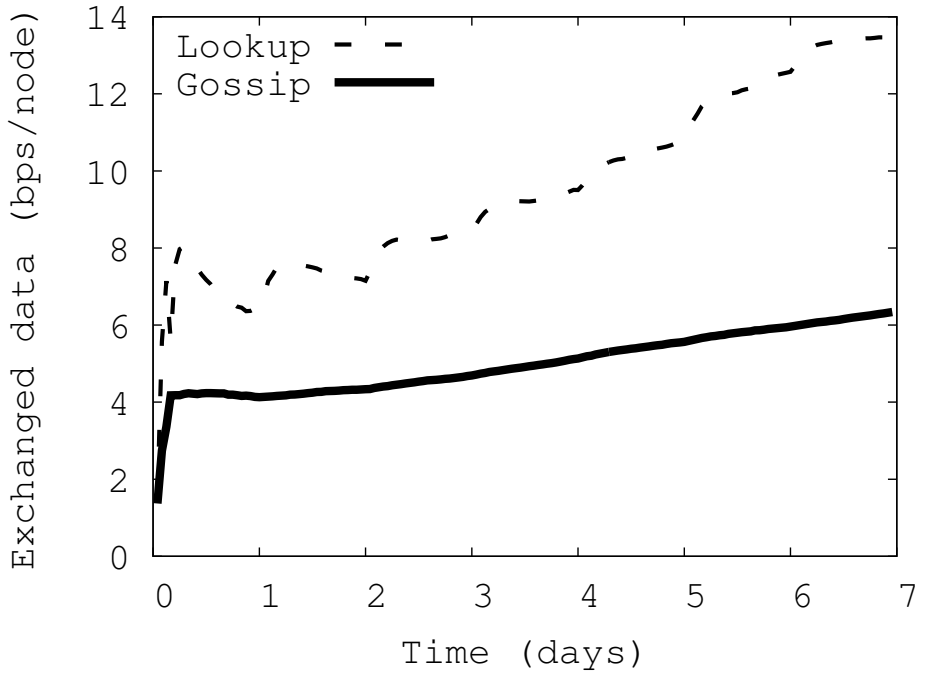


Figure 6.11: *Evolution of the network load with the Overnet trace.*

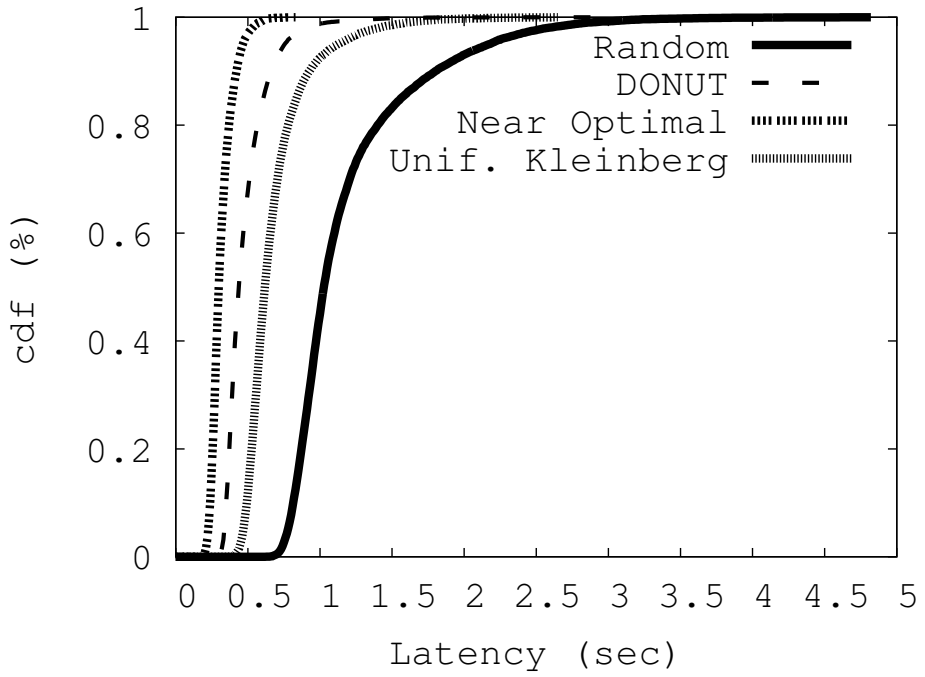


Figure 6.12: *Cumulative distribution function (cdf) of mean routing latencies for an overlay of 2500 nodes.*

Therefore, it is important to measure the network cost of DONUT.

The coordinates and the size of each region are deterministically defined by the path in the tree from that region to the root. One integer value is sufficient to store that information for most of the paths. In addition to that, each leaf of the tree stores a double value representing the density of its region. For an 2500-node overlay and the density distribution described above, the average quadtree contains 77 nodes and 232 leaves. As a regular quadtree node is stored on 4 bytes, and a leaf is stored on 8 bytes the average quadtree size is about 2.2 Kbytes only.

This size is comparable to other maintenance meta-information such as for instance, bloom filters in PAST [170]. Moreover, a peer is able to control the fuzziness of the map in order to optimize its size. A quadtree node is responsible for a region formed by the union of its children's regions. If the densities of the subregions are equivalent, the children may be suppressed and their mean density value may be affected to their parent.

Figure 6.10 shows the network load of DONUT maintenance for the two different propagation strategies described in Section 6.3. We can see that the Lookup Propagation strategy significantly increases as the session time decreases. This is due to the fact that the information is propagated inside join messages, which number increases with churn. Moreover, at each step of the join-message routing, the forwarding node adds its local information even if it has been already propagated before. Therefore, the lookup messages are always full of density information. For a mean session time equal to 30 minutes, the network load of the propagation is of 0.6 Kbytes per second per node. This cost is still affordable, and the approach does not require the implementation of a dedicated dissemination protocol. For these reasons, the lookup propagation may interest designers of systems with mean session times below 30 minutes.

On the other hand, the modified gossip algorithm has a near-constant network load because: 1) the protocol is not related to join messages and 2) peers only propagate *new* information: the gossip messages most of the time contain much less information than the maximal allowed size. Thanks to that, regardless of the churn rate, the network cost of gossiping the density map is below 10 bytes per second per node, which is very low.

Figure 6.11 shows the evolution of the network load while using the Overnet churn trace. Here again, the Lookup propagation uses more bandwidth than the gossip algorithm. Moreover, the lookup propagation strategy induces important variations of the network load over time, which may be harmful to the overlay. Results of evaluations with other real churn traces are omitted due to a lack of space but exhibit the same characteristics.

Figure 6.12 has been realized by using the real latency traces. It shows the cumulative distribution function of routing latencies. As expected, the Near Optimal strategy exhibits the best distribution characteristics. This is due to the fact that *all* peers have the same accurate local graph. DONUT significantly outperforms the other techniques, approaching the Near Optimal strategy[8]. The mean routing latency of DONUT is less than $500ms$ for 67% of the peers. On the other hand, only 0.006% have the same mean latency with the random approach. The Uniform Kleinberg approach behaves slightly better than random: 11% of the peers have less than a $500ms$ mean routing latency, which is still much less than DONUT. 98% of the peers implementing DONUT have a mean routing latency less or equal to $1s$,

[8] as the lookup propagation and the gossip algorithm appear to be both equally efficient in our system, Figure 6.12 makes no distinction between the two strategies.

while less than a half (45%) exhibit the same characteristics for the random strategy. These values show that the propagation of the density map is efficient for a large population subset. Thanks to that, and regardless to their position in the keyspace, most of the peers are able to build efficient shortcuts in the overlay.

6.6 Conclusion

This chapter presented DONUT, a mechanism that improves routing performance of nearest-neighbor overlays. On each peer, DONUT monitors the density of the NVE surrounding its avatar. Then, DONUT peers exchange information to collectively build an approximation of the global density distribution of the NVE. This distribution is then used to maintain the small-world property in presence of highly skewed distributions.

The lessons learned from our work are:

1. Graph distances in nearest-neighbor overlays can be estimated with an acceptable precision by using an approximation of the NVE density distribution. This estimation is sufficient to increase the efficiency of long range links by 20% compared to existing techniques.

2. Such estimation can be aquired with a very reasonable overhead: the average map size in our simulations is of only 2.2 Kbytes and dynamic management of the map can be achieved with less than 10 bps of traffic on each peer. This cost is very low even for peers with limited capacity.

3. DONUT is generic because it does not rely on any overlay-specific mechanism. It is thus topology-oblivious and could increase the routing performance of any nearest-neighbor overlay.

Our work once again demonstrates that incorporating knowledge about player behavior into the infrastructure can increase its performance with very low overhead.

Chapter 7

Scaling out virtual battlegrounds with in-memory databases

Contents

This chapter presents our **ongoing work** to improve adaptability and scalability characteristics of server-based MMOFPS. For that purpose, we decouple data management from the other NVE mechanisms by incorporating an in-memory database in the design of MMOFPS. Our goal is to foster the developement of virtual battlegrounds with orders of magnitude higher populations than current commercial proposals, enabling epic-scale battles at affordable development cost.

7.1 Introduction

7.1.1 Problem Statement

First Person Shooters (FPS) are an important class of online games with millions of players worldwide. Commercial hits like Counter-Strike, Battlefield or Quake III emerged in the late nineties and are still extremely popular. FPS are virtual battlegrounds in which players fight against each other in a succession of team-based or free-for-all combats. The gameplay is fast-paced, and unlike in other MMOGs, does not require time-consuming tasks such as complex character management.

Due to the nature of their gameplay, virtual battlegrounds have very tight applicative requirements. End-to-end latencies above 150 ms result in a severe degradation of the gaming experience [44], and gameplay inconsistencies or lack of availability are known to be extremely frustrating for the gamers [67]. Maintaining consistency, responsiveness and availability at large scale is untrivial. Most of the FPS, including the most popular ones, avoid the challenge by supporting only small, single-server arenas with populations limited to 64 players.

Only very few MMOFPS truly address the scalability problem by distributing the load of the NVE across a set of servers[1]. These architectures rely on game partitioning to achieve scalability. Each server is responsible for a region of the NVE, storing objects and serving clients located inside its region only. If the player's AOI spans over multiple regions, the servers need to exchange information in order to enable the illusion of a seamless NVE. As the population increases, additional servers are added to handle the load, resulting in an extremely *fragmented* NVE. The game objects are spread among a large number of servers, increasing inter-server communication and eventually reducing the overall performance of the scheme. Consequently, these complex, ad-hoc architectectures are expensive to maintain and require players to pay a monthly subscribtion which contradicts with the casual nature of the FPS gameplay [113].

Because of that, the success of MMOFPS is limited, and all the most popular FPS still have a simple server-based design. This restricts the game populations to a few dozens of players at most and epic-scale battles remain an unfulfilled gamer's dream.

7.1.2 Contribution

The work presented in this chapter addresses the lack of efficiency of existing architectures for MMOFPS. We performed the scalability analysis of Quake III Arena, a popular FPS with a conservative design composed of a unique server storing the entire game state.

The game server is responsible for the execution of two orthogonal tasks. Periodically: (i) it updates the state of all the game objects (including players) according to the modifications brought by the participants. We call this task *Game progression*; (ii) Then, it sends to each player a *snapshot* of the NVE. This snapshot contains all the objects that are visible by the player. We call this task *Snapshot propagation*.

The results of our analysis show that the load of snapshot propagation grows much faster than the load of game progression as the game population increases. Similarily to Quake III,

[1] mainly Planetside (1 & 2) and WWII Online

existing FPS and MMOFPS perform game progression and snapshot propagation on the same physical host. Therefore, as the load increases, the host becomes overwhelmed by snapshot propagation. The solution brought by the existing architectures to handle the load is to reduce the size of the region managed by the server. This technique mechanically decreases the cost of player management, but with unnecessary fragmentation of the NVE as a side effect.

Instead, we propose to manage the two tasks separately. In our architecture, both processes are decorrelated and are able to scale out at their own rythm. Game progression is ensured by the game server with the help of an in-memory distributed database that acts as a backbone and provides responsiveness and robustness to our system. Snapshot propagation is performed by a set of distinct entities called *Snapshot Mirrors*. They collectively retrieve the game state by querying the database, then perform calculation and propagation of consistent snapshots of the game state to the players. As the player population increases, new mirrors can be added on demand to handle the load without creating unnecessary fragmentation of the NVE.

This is not the first proposal to use databases in the design of MMOGs. Darkstar uses a central database to store the game state with a set of applicative servers being client of the database [50]. However, the database is not in-memory nor distributed, thus limiting scalability and responsiveness of the system. To improve responsiveness, Darkstar enables a complex consistent cache mechanism that further harms to its scalability. MiddleSIR stores the whole game state in a database and uses state machine state replication to maintain several consistent replicas [159]. Once again, the consistency mechanism is the limiting factor to the scalability of the systm.

In this chapter, we demonstrate the benefits of using an in-memory distributed database to scale out while preserving responsiveness. We adapted Quake III, a popular FPS to run on top of VoltDB, a low latency, high throughput distributed in-memory database. QuakeVolt, our proposal, turns Quake III, that originally supports at most 64 players, into a truly massive experience with many hundreds of participants. VoltDB successfully handles concurrent accesses, persistence and fault tolerance, practically demonstrating the feasibility of our architecture. This is, to the best of our knowledge, the first proposal to explicitly highlight the necessity of decoupling data management from snapshot calculation and to exploit in-memory distributed databases for that purpose.

QuakeVolt development and maintenance are extremely cheap. Servers all run on top of commodity hardware, and the adaptation of Quake III to VoltDB requires only minor modifications to the original code of the Quake III server. Furthermore, Quake III clients can interchangeably connect to Quake III and QuakeVolt: they are completely unaware of the modification in the architecture.

The contributions of our work are: (i) a scalability analysis of the standard client/server FPS scheme, and (ii) a novel scalable architecture for MMOFPS. Our architecture exhibits the following benefits:

- **Scalability:** It enables to scale without inducing unnecessary fragmentation of the NVE, improving the overall performance of the system at large scale.
- **Simplicity:** complex data management is left to the database, and adapting Quake III to VoltDB required the addition of less than 300 lines of the original server code, which

represents less than 1% of the overall code size.

- **Low cost:** all the infrastructure can run on top of standard, commodity hardware.
- **Transparency:** the modification of the architecture is transparent to the gamers since no client code needs to be modified.
- **Robustness:** Storing the game state in the database enables persistence and availability in presence of failures. No complex ad-hoc mechanism is required to enable these properties: they come in a bundle with the database.

7.2 Scalability analysis of classic FPS architectures

7.2.1 FPS architecture overview

The architecture of most FPS relies on a pure client/server model and mostly did not change since the mid-nineties. A single server maintains the whole virtual environment by storing a record of every mutable object of the NVE. Most of the time, the objects are stored in DRAM as a set of ad-hoc C/C++ structures. Clients are connected only to the game server and periodically upload keyboard and mouse events produced by the player. With a fixed interval, the server ensures the progression of the game by computing *frames*. A frame computation takes as an input:

- the current state of all the objects in the game,
- the events produced by the clients since the last frame,
- the game logic (*e.g.*, physics of the NVE, behavior of non-player objects).

According to this input, it produces the next state of the game. The frequency of the frame computation is called the *framerate* and is usually at least 20 *fps* (frames per second) to ensure a smooth gameplay. At the end of each frame, the server computes a *snapshot* of the virtual environment for each participating player. The snapshot of a player represents her view of the NVE and is composed of the new states of all the objects that are in her AOI. Upon reception, the new snapshot is used by the client application to render the game. For correct rendering, the client should receive snapshots in the order in which they were computed by the server. Since most of the FPS use UDP for communication, the client incorporates a mechanism to deal with reordering of updates.

This design is relatively simple and is generally adequate for small-scale FPS. Consistency is ensured by serialization of updates on the server. Players have almost no persistent state, so data loss is not critical, especially as the probability of the failure on a single server is relatively low. Finally, the maintenance cost is low, since a very common configuration is able to handle up to 64 simultaneous players, and no multi-server communication is required. It seems pretty obvious that the main drawback of this classical client/server scheme is its lack of scalability. However, a more detailed analysis deserves to be provided. In the rest of the section, we relate the scalability analysis we performed on a Quake III server to identify its bottlenecks.

7.2.2 Evaluation Methodology

A trustworthy analysis of the scalability limitations of Quake III requires in the first place the generation of realistic game activity at large scale. Our goal is to create a workload that will consume bandwidth, memory and cpu similarily to a real game session, but at a larger scale. In the second part, we also need to use this workload to fairly compare the scalability of Quake III and QuakeVolt. The generated workload therefore needs to be (i) realistic, (ii) scalable, and (iii) deterministic and reproducible. Generating a workload having these nice properties is a non trivial task, because game sessions with human players are not reproducible nor large scale. Moreover, bots (*i.e.,* automatic players) for Quake III run *server-side* and are not representative of real players since they do not consume server bandwidth.

Workload generation

We thus decide to produce the game workload by collecting activity traces from a game instance with real players and replaying them for the evaluation. During the record phase, we instrument the code of the server to record all the events received from each client into a file on disk. During the replay phase, the source code of the client is instrumented to replay recorded events. Each client then progressively replays its log to obtain game activity. This way, we obtain both a realistic and reproducible workload.

The trace file of a client is composed of the keyboard and mouse events produced by the client's player. These events are sent to the server, which uses it to compute the player's trajectory and its actions inside the game.

It is important to notice that the replayed game session will most likely not be exactly identical to the recorded one. During the recording phase, the server receives and serializes the events to form a consistent history. However, during the replay phase, the order in which the replayed updates are received may be different because the clients are not globally synchronized.

As an example, the events $fire_a$ and $fire_b$ simultaneously triggered by players a and b shooting at each other happen to be serialized $fire_a \prec fire_b$ during the recording phase, while they could be serialized $fire_b \prec fire_a$ during the replay phase, resulting in a totally different history.

Unfortunately, the gameplay of FPS requires a very high precision because the state of an entity can radically change because of a single event. Refering to our experience, such inversions tend to occur relatively often, and are likely to result in a radical modification of the game state. A rocket can miss a player because of a time shift of the firing event. Instead of falling, a player can catch an elevator because of a time shift of the movement event. As a result, the player remains alive in the replayed phase, while she should be dead in the original game state, dramatically changing the game.

In practice, such shifts lead to an inconsistency between the trace recordings and the replayed game state. We faced three fundamental kinds of inconsistencies. First, a player can die while her trace still contains actions that can be performed only if she is alive. Second, a living player executes totally inconsistent actions because the recorded trace tells her that she died. Finally, the player can get lost by dramatically modifying her trajectory, *e.g.,* falling into a pit, taking an elevator...

To solve these problems, we incorporate a *loopback mechanism* in the client source code. The mechanism monitors the current state of the player, and seeks for inconsistencies between her state and the trace. To ensure an accurate detection, we incorporate information about the state of the player in the trace log during the record phase. Namely, in addition to player events, the server periodically writes the absolute player position into the file. During the replay phase, the loopback mechanism measures the distance between the recorded and the current coordinates. If the delta exceeds a threshold, it means that the trace diverged from the recorded state. If such a divergence is detected, the mechanism first tries to reconcile the trace with the current state: the player attempts to reach her recorded position. If the reconciliation fails after a given time, it means that the trace irremediably diverged from the game. In that case, the loopback mechanism asks the server to respawn the player at her initial position and starts reading the trace file from the beginning. The loopback mechanism ensures that players will most of the time have a consistent behavior and will not be stuck in some corner of the map.

Because of these inconsistencies, replaying the exact same virtual fight is very challenging. However, thanks to our mechanisms, the behavior of the players remains globally consistent. As confirmed by our observation of the experiment, trace-players run, jump and shoot at each other, giving an impression of a real game play. Thanks to the traces, density hotspots and player trajectories are preserved. We believe that this level of accuracy produces a sufficient approximation of a real game workload.

Scalability concerns

In order to scale, we allow several clients to reuse the same trace file. For games with 78 players, we used 16 trace files containing game activity of 16 different players, each trace thus being reused by 5 clients. Clients using the same trace start the game with a randomly distributed time shift, so that their avatars do not execute the same actions simultaneously.

For the scalability test of Quake III, we use a large battleground-map designed for up to 64 players called Miasto[2] and representing a small city. This map was chosen because it is really used by players and has large open spaces that are able to fit many dozens of players in realistic conditions.

Hardware configuration

We perform measurements of CPU, DRAM, and bandwidth consumption of the server under different load configurations. A single server is started, and the number of clients is varied from 1 to 78. The evaluations are performed on a cluster composed of 25 Quad-core Intel Xeon@3.40GHz with 4 GB of DRAM interconnected with a Gigabit Ethernet network. We consider that running the server *and* the clients on the same cluster does not bias the evaluation since: (i) each client only uploads player events, thus consuming only a tiny fraction of the Gigabit network; (ii) clients are single-threaded so the 78 clients were able to run at full speed on 20 hosts.

[2]http://q3a.ath.cx/map/q3shw13/?lang=en

7.2.3 Results

The first thing to be noticed is that *DRAM is not the bottleneck of the system*. In Quake III, and generally in modern FPS, the memory consuming entities such as level design and textures are not mutable and are therefore stored client-side. A typical arena is composed of less than a thousand mutable objects (including players), each of them being represented by a C structure that does not exceed 500 bytes in memory. Therefore, even the most pessimistic memory footprint estimations do not exceed a very few Gigabytes of DRAM, which is very affordable even for relatively old hardware configurations.

The scalabilty analysis therefore focuses on CPU and bandwidth measurement. Figure 7.1 plots the evolution of the consumed *incoming* (clients → server) and *outgoing* (server → client) bandwidth in function of the load. We can see that the incoming bandwidth linearly grows with the number of players. This result is straightforward, since each player generates only a bounded number of events per period of time, regardless of the number of participants. The downloads performed by the server are very far from saturating the the bandwidth: a seventy-player-load consumes about 0.00025% (250 Kbit/s) of the available Gigabit bandwidth. At this rate, the download bandwidth saturates only with 250000 clients, which is far beyond the targeted scale.

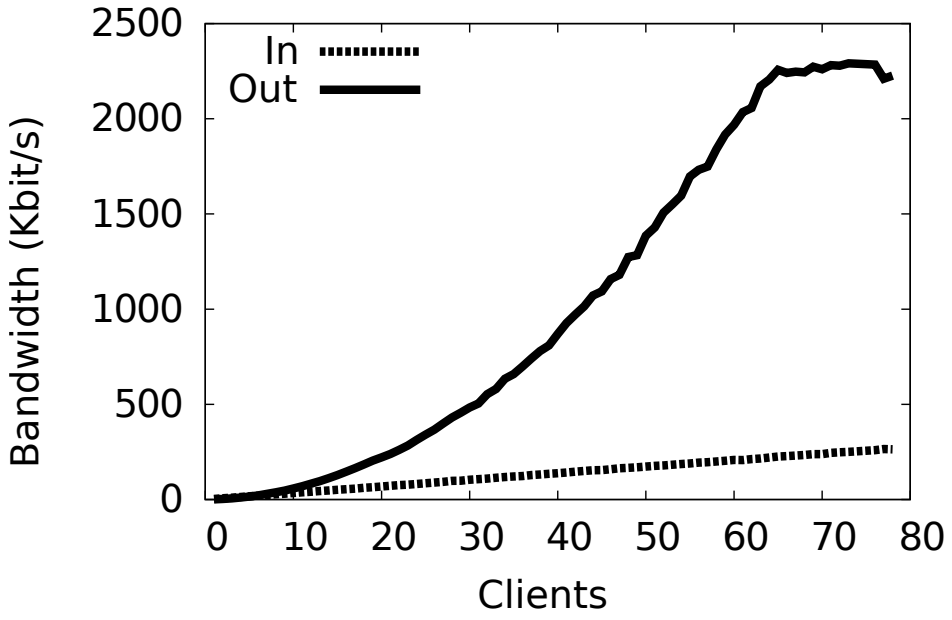


Figure 7.1: **Server bandwidth consumption.** *Incoming: linear, Outgoing: supra-linear growth.*

The results are however very different for the outgoing bandwidth, which has clearly a supra-linear growth. For 70 clients, it consumes approximately 10x more bandwidth that the downloaded information. This comes from the fact that the server needs to compute snapshots of the virtual environment for each player. That is, the server sends to each player *all* the objects located inside its AOI. When the density of the NVE increases, the server needs to send more objects to more players. As a result, the number of players to which each object is sent increases with the density of the NVE and the amount of uploaded information grows supra-linearly. Uplink bandwidth can rapidly become a bottleneck for the server. However,

it is still not the case, since the plot shows that the amount of uploaded information stagnates at approximately 2.3 Mbit/s from 65 players.

The previous result suggests that the CPU is the actual bottleneck for the Quake III server. This is confirmed by the results shown in Figure 7.2. The CPU utilization reaches 25% at 65 players, which corresponds to a 100% utilization of one core. The server is single-threaded, so the only server thread fully uses its core. We instrumented the server code to trace the CPU consumption. As expected, the frame calculation consumes the whole CPU time. In particular, more than 95% of the CPU time is spent in only one function called `SV_SendClientSnapshot`. At each frame, this function is used by the server to build and upload a consistent snapshot of the AOI of each player. This mechanism significantly limits the ability of the system to scale. Indeed, as depicted in Figure 7.3, after 70 players, the framerate of the server abruptly drops, showing that the server is no longer able to cope with the load of the system.

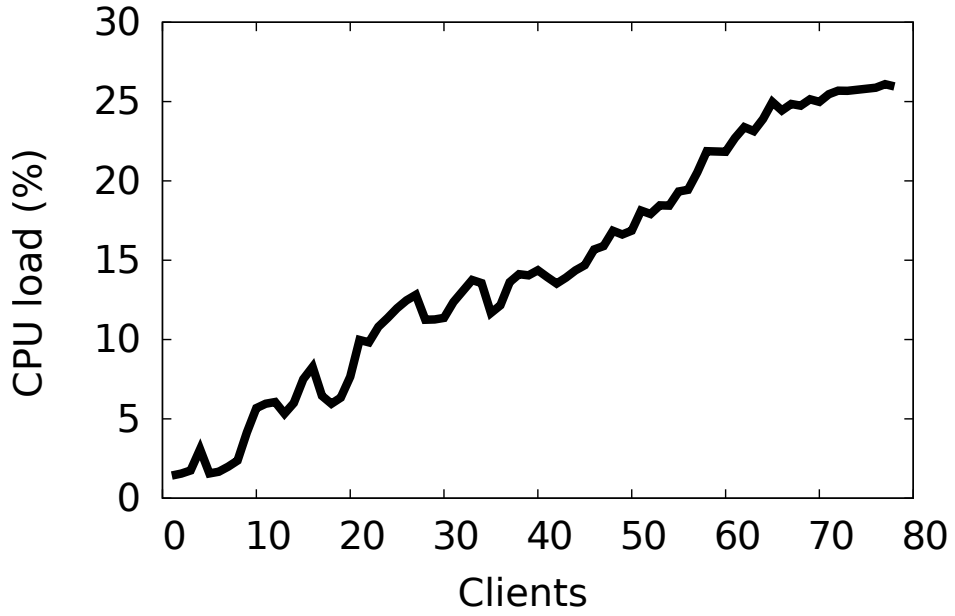


Figure 7.2: **Server CPU load.** *100% of CPU load accounts for full usage of the four cores. The Quake III server uses one core only.*

The conclusion of our evaluation is therefore that **the generation and propagation of player snapshots is the true bottleneck** of the Quake III server. This constatation is probably also true for most of the FPS servers, since the described design is commonly adopted in modern FPS. This result is corroborated by other studies [49, 124] and has a direct impact on the efficiency of FPS architectures.

7.3 Architecture

This section provides a detailed overview of the QuakeVolt architecture. QuakeVolt is designed to overcome the bottlenecks described in the previous section. Previous proposals

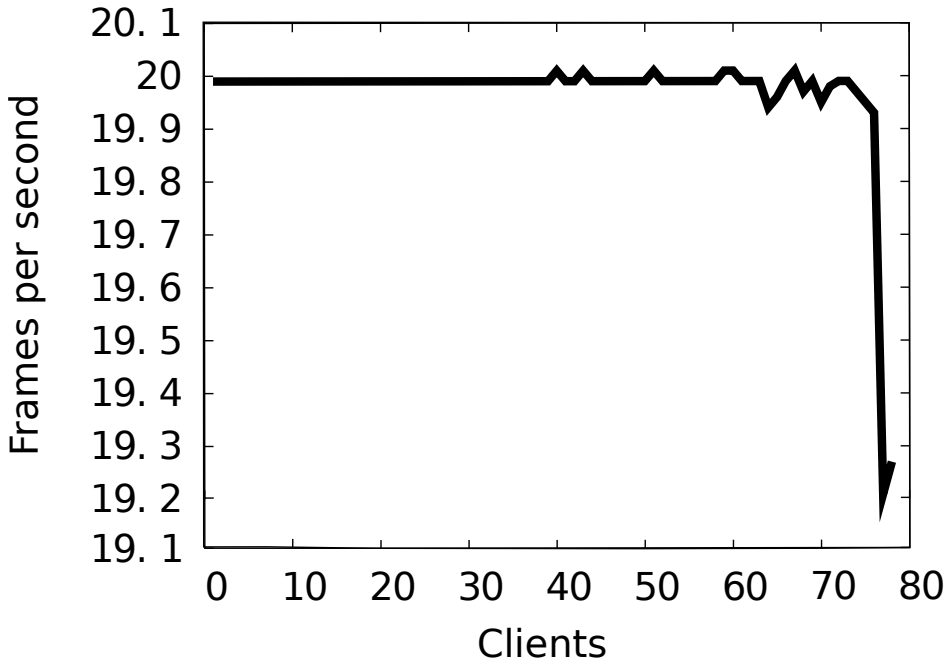


Figure 7.3: **Server frames per second.** *Rates below 20 fps degrade gaming experience.*

attempted to avoid the snapshot calculation bottleneck by proposing a completely different architecture [49, 47]. In contrast, we address this bottleneck *directly*, *i.e.,* by bringing as less modifications as possible to the native architecture of the servers. The motivation is to demonstrate that only minor modifications of existing game source code can result in a dramatic scalability increase. One of our concerns is to ensure retro-compatibility of the game protocol, *i.e.,* the new architecture must work with already existing game clients.

7.3.1 Overview

The essential purpose of our architecture is to *decorrelate* game progression from snapshot propagation. Our motivation is that the load of data management linearly grows following the population increase, whereas the snapshot propagation cost rise is supra-linear (see Section 7.2.3). We relieve the game server by delegating the snapshot calculation and propagation to a set of separate processes that we call *snapshot mirrors*.

While the game progression task is handled by the game server for the whole NVE, the snapshot propagation is ensured *collectively* by the snapshot mirrors. Each snapshot mirror is responsible for a part of the NVE and only serves clients that are inside its region (see Fig. 7.4). The communication between all the entities of the architecture is exclusively performed through an in-memory distributed database that acts as a low-latency backbone for the system. The responsibility of each entity of the architecture is as follows:

- **The game server** is used as a frontend to which all the clients connect upon log-in. It monitors client sessions, deciding if players are allowed to play, and detecting fraud attempts. During the game, it receives incoming player events, handles the game

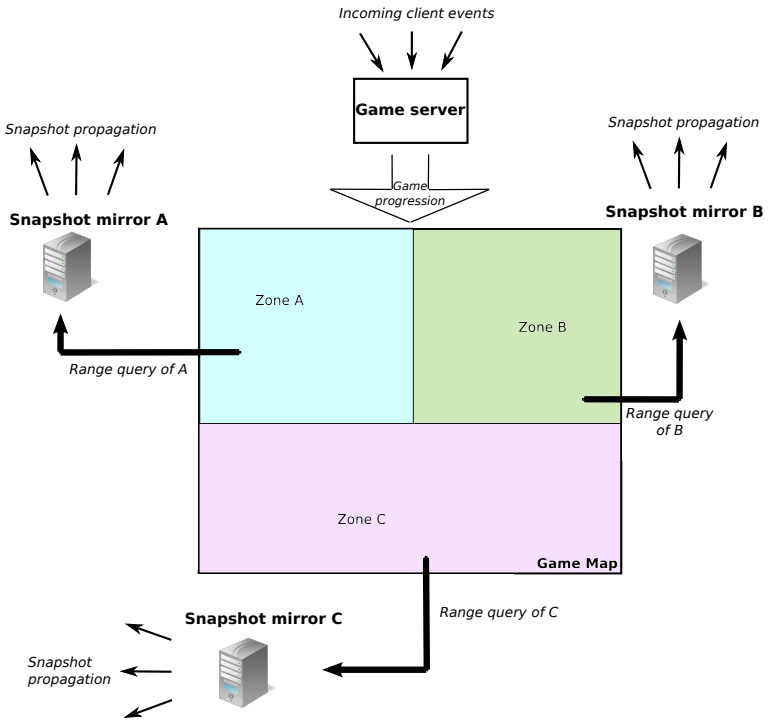


Figure 7.4: *The load of snapshot propagation is shared through NVE partitioning.*

progression by periodically computing new frames. All these features are inherited from the original Quake III server and do not need to be modified. In fact, the native Quake III protocol is preserved, and clients are unaware of the backend of the system while sending updates to the server. However, instead of computing client snapshots, it sends the new game state to the database. Each game state update is composed of the new versions of the NVE objects. They are timestamped with the sequence number of the frame in which they were computed and stored in the database as new *versions* of the objects.

- **The database** stores the game state and provides concurrent, transactional access to the data with ACID guarantees.
- **Each snapshot mirror** retrieves from the database a portion of the game state using a range query corresponding to its responsibility zone. Then, it computes snapshots for clients in its zone and propagates the snapshots.

As the NVE population increases, additional mirrors can be added on demand to cope with the load, scaling out the costly snapshot computation and propagation process. It is important to notice that adding new snapshot mirrors is likely to not fragment any data because the size of the distributed database does not directly depend on the number of mirrors. The server, the mirrors and the database are *logical* entities and can be placed on different datacenter-hosts. However, depending on the load, some of them may share one physical location. In the rest of the paper, we consider that each entity runs on a distinct host[3].

[3]The optimal placement of the entities inside the datacenter is left to a future work.

7.3.2 Distributed frame computation

In contrast with the original Quake III server, the frame computation is distributed in QuakeVolt because of the task sharing. Figure 7.5 details the network exchanges occuring during the distributed frame computation.

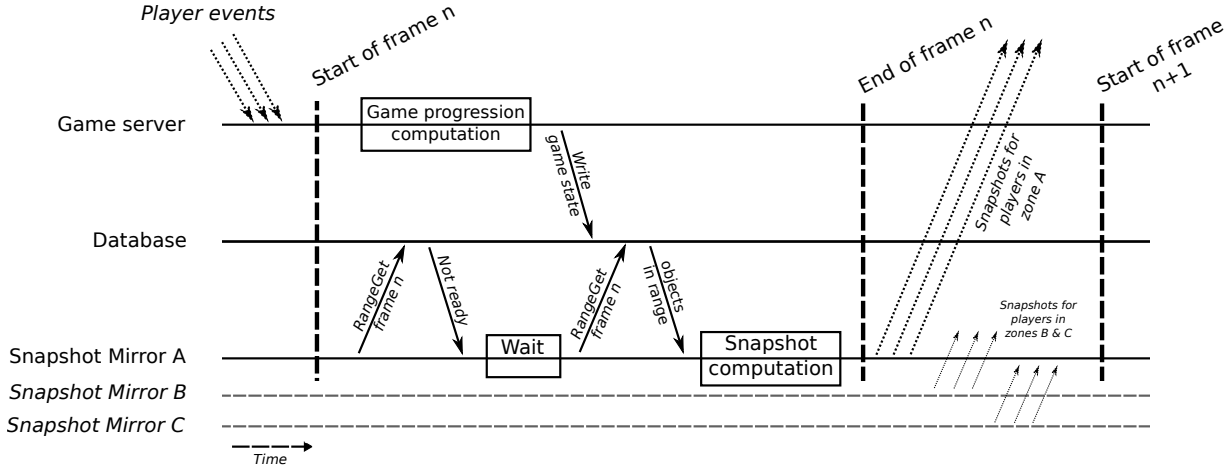


Figure 7.5: *Exchanges during frame computation (for clarity, only the queries of Snapshot mirror A are represented)*

Periodically, the game server initiates the frame computation. It parses all the player events received since the last frame, and updates the states of all the objects according to these events. The server stores the states of all the objects in DRAM, making the process relatively inexpensive. Once all the objects are in the new state in the memory of the game server, the new game state is written to the database.

Once the entire game state has been successfully written, each snapshot mirror performs a range query to retrieve objects located inside its responsibility zone. This implicitly means that each snapshot mirror needs to be synchronized with the game server in order to propagate the correct game state to the clients. In our design, all the synchronization tasks are performed through the database. This is motivated by the desire to keep the architecture simple. This way, the snapshot mirrors are unaware of each other, and can be easily added at runtime to absord population peaks. However, this choice implies that the snapshot mirror does not receive any notification about the availability of a new game state. Therefore, it periodically needs to check if a new state is available. For that purpose, the backbone contains a counter which indicates the sequence number of the last frame that was *fully* received by the database. This counter is updated by the game server at the end of each game state flush. Meanwhile, each mirror holds the sequence number of the last snapshot successfully sent to the clients. It performs a range query with the new sequence number, and if the backbone counter has not yet been updated, the range query fails. Such an attempt is represented in Figure 7.5. In case of failure, the Snapshot mirror waits a small amount of time and retries until success.

Once the new state of the objects in its zone has been retrieved, the snapshot mirror computes the snapshots for each player in its zone. Namely, it finds out which objects are

in the player's AOI, and sends her the new states of these objects. When all the snapshot mirrors have propagated their snapshots, the system is ready for computation of the next frame.

7.3.3 NVE partitioning

In order to distribute the load of snapshot propagation, the snapshot mirrors need to partition the NVE. This section details the partitioning mechanisms that enable a good quality, seamless gaming experience.

Database organization

The state of the game is stored in a table partitioned accross the database nodes. The partitioning scheme of the table is database-dependent and obviously impacts the performance of data retrieval. The complete evaluation of different data placement schemes is left to a future work. Each object has a unique identifier and is represented by a set of row-records in the table. For an object, one record corresponds to the base state and the rest of the records contain successive delta-encoded updates. The delta-updates are timestamped, and it is therefore possible to rebuild the full state of the object at any time by aggregating its updates with the base state. The table is indexed by the identifier and the timestamp attributes.

To avoid accumulation of the object versions in the database, deprecated delta-updates are periodically deleted by a garbage collecting process. The collecting process accurately identifies deprecated versions because the database maintains a record about the last frame consumed by each mirror. For each object, at each collection, the process therefore removes all the delta-updates which timestamp is inferior or equal to the minimum of the frame-numbers consumed by the mirrors. The removal is performed by aggregating the delta-updates and replacing the old base state by the new aggregated state for each object.

The database is accessed through two main stored procedures: `send_gamestate` and `receive_gamestate`. The former is used by the game server to flush a new game state at the end of each frame calculation. The latter is basically a range query used by the snapshot mirrors to retrieve objects contained in their region.

Data retrieval through range querying

It is important to notice that a snapshot mirror only needs to retrieve the *latest* updates for the objects which it already knows, whereas the *full* state needs to be acquired for new entities. Typically, the mirror needs to acquire the full state of a player once it entered its region, while intra-region entities only require a single delta update.

The naive technique to retrieve a consistent snapshot of a mirror-region is to perform two queries. A first range query to obtain all the latest updates for the objects in the region, and a second selective query to retrieve full states of incomplete objects. However, this technique requires two queries for each state retrieval and thus degrades responsiveness.

Instead, we provide each mirror with an object-ownership mechanism. The mirror stores the identifiers of all the known objects in a bit map. At each call to `receive_gamestate`, the mirror sends its ownership-map to the database. The database locally checks whether

the retrieved objects are new to the mirror. It returns the latest record for all the objects in the bit map and an aggregated state for all the new objects. Once the set of updates is received by the mirror, the bit map is updated by adding new objects and removing missing ones.

This technique is more efficient because only one query is performed, and all the verifications are performed by the stored procedure in the address space of the database. However, if the number of objects of a region is high, transmitting the bit map at each query may become expensive. In that case, the map can be replaced by more space-efficient structures such as bloom filters [57].

Handover mechanism

The role of the handover mechanism is to ensure a seamless gaming experience when the player moves from one NVE region to another. The traditional architectural approach of server-based MMOGs is to couple player and data management. As the player AOI overlaps two regions, some of the AOI objects need to be retrieved from the neighboring server to provide a seamless view. If NVE fragmentation is high, the handover mechanism incurs significant data traffic between the servers. Moreover, the consistency of the players' AOI must be preserved, so servers have to collectively enable efficient ad-hoc consistency mechanisms.

In QuakeVolt, when a player moves from a region to another, the responsibility of snapshot propagation to that player is transfered from one mirror to another. Thanks to the database backbone, our handover mechanism is very simple and requires little synchronization between the affected mirrors. As a player crosses the border between two regions, the game server simply updates its coordinate in the database. Since the new player state coordinate is located in a new region, it is automatically retrieved by the range query of the new mirror. Because of the object ownership mechanism described previously, the old mirror notices that the player is no longer in its region, and the new mirror receives the full state of the player. Slight direct synchronization between the two mirrors is however required, because the current upstream sequence number is only maintained by the mirrors and needs to be transmitted from the old to the new one. In contrast with the traditional scheme, AOI consistency is ensured by the database and is cheap, because data fragmentation is low.

Placement of snapshot mirrors

In order to cover the whole NVE, the snapshot mirrors need to partition the game space. This is the only strong requirement of our scheme. Since handover is cheap, all the placement configurations that form a partition of the NVE space are allowed. Currently, our infrastructure enables simple, static partitioning. Each mirror is responsible for a fixed-size rectangular area. However, the distribution of avatars across the map tends to be skewed, and static partitioning can result in a load imbalance on the mirrors [49, 196]. Other placement strategies, including dynamic partitioning are thus being considered.

7.3.4 Propagation of information

Bandwidth cost of game state propagation

Propagating the whole state of the game at each frame *i.e.,* many times per second may sound costly. However, for FPS games, it is not. The first reason is that the most voluminous entities, such as textures and level design, are immutable and are kept on the clients. Second, one nice property of NVEs is that the state of a mutable object rarely changes *completely* from one frame to another. The modifications are rather very progressive: player's coordinates undergo only a slight shift from one frame to another because the framerate is high compared to the velocity of player movements. Similarily, a projectile hitting a player will decrease its health in a limited fashion. Thanks to that, *delta encoding*[4] of updates is extremely efficient and widely used by the FPS applications. We therefore employ this technique to propagate the state update from the game server to the database. As a result, propagating the state of 30 Quake III objects including one constantly moving player requires approximately 400 bytes per second, which is less than 15 bytes per entity per second in average. Regarding the fact that the entities are running inside a datacenter with at least a Gigabit interconnection, this cost is very reasonable.

Communication with the clients

In Quake III, all the communication with the client is maintained by the game server through message passing over UDP. Since no reliable transport level protocol is used, loss detection, retransmission and rate control are ensured by the application. The game server stores the IP address and the port of the client, aswell as a set of variables to ensure consistent communication. For each client, the server maintains the sequence number of the last received update, the sequence number of the last snapshot sent, and the sequence number of the last acknowledged snapshot. It is important to notice that the last received update and last acknowledged snapshot sequence numbers are *downstream* variables: they are sent by the client to the server. In contrast, the last snapshot sent is an upstream variable: it is maintained by the game server.

In QuakeVolt, downstream and upstream communication with the client are managed by different entities. Session management and reception of player updates are performed by the game server, whereas snapshot propagation is collectively ensured by the snapshot mirrors. Connection information about a client must be shared between the different entities communicating with the client. In particular, downstream variables of a client must be propagated from the game server to the mirror of the region where client's avatar is located. This exchange is entirely performed through the database. The complete connection information is stored in the C structure of the client in the memory of the game server. At the end of each frame, the server flushes the structure, including the connection information, into the database. The information is then retrieved by the corresponding mirror and used to communicate with the client without breaking the message sequence. As a result, the client sends updates to the game server but receives snapshots from the mirror without any client-side modification of the original Quake III protocol.

[4]Delta encoding is a compression technique that relies on sending only differences between two consecutive states instead of propagating the whole state.

7.4 Implementation: from Quake III to QuakeVolt

Our decision to choose Quake III to evaluate our architecture is driven by the extreme popularity of the game. Thirteen years after the release, the Quake III community is still very active. Nowadays, the game is used for cyber sport competitions and despite its venerable age, the gameplay remains extremely entertaining. The game is now open source and has a relatively straightforward architecture we believe to be representative of multiplayer FPS of the last decade.

We choose to use VoltDB because it is a recent representative of the new generation of horizontally scalable, high performance in-memory databases. Furthermore, VoltDB creators claim that their architecture is well suited to support games [23].

7.4.1 Adapting Quake III source code

Quake III has a traditional client/server design and is written in C. The server is single-threaded and is mainly composed of an infinite loop that computes frames at a default rate of 20 fps.

The frame computation process is composed of an ordered set of operations:

1. Management of client connections (connect, disconnect players).
2. Reception of player events from the connected players.
3. Computation of the new state for each object in the NVE
4. Calculation and dispatching of snapshots to the players.

The latest task is responsible for the scalability bottleneck of the server and is entirely performed by a function called `SV_SendClientSnapshot`. The only modification we bring to the Quake 3 server code is the replacement of `SV_SendClientSnapshot` by a function called `SV_SendGameState` that flushes the state of all the objects to VoltDB.

A set of C structures kept in DRAM represent the current state of the NVE-entities. The structures `client_t` and `playerState_t` keep the state of the players (including the state of the connection), while `svEntity_t` and `sharedEntity_t` account for non-player objects (*e.g.*, ammunition, projectiles). The Quake III source code embeds a set of functions for delta encoding/decoding data structures. We use these functions to efficiently propagate the state of the abovementioned structures to the database. The whole state propagation process required coding less than 300 lines of C code, most of it being straightforward delta-encoding functions. The modifications we bring to the server source code represent approximately 0.03% of the overall Quake III code size, and the source code of the client is not modified.

7.4.2 Snapshot mirror source code

The snapshot mirror source code reuses a part of the Quake III code. In particular, it incorporates the `SV_SendClientSnapshot` function used in the native Quake III server code to

propagate client snapshots. To maintain the state of the entities in its region, a mirror reuses the functions we created to decode delta-encoded updates of the entity structures.

Thanks to that, the snapshot mirrors are able to communicate with the client by using the native Quake III communication protocol. This protocol incorporates mechanisms to ensure the security of the client/server communication:

- At the beginning of the client session, the client and the server agree on a random value which is then used to XOR-encode the communication messages.
- For each received message, the client checks if the emitter of the UDP packet is the game server. If the packet was sent from another IP address, the message is discarded.

The snapshot mirror entity needs to override these mechanisms in order to communicate with the client. To ensure correct XOR encoding at the mirror, the server propagates the XOR-key-value of each client in the database along with other client connection information (the value is kept in the `client_t` structure). It is important to notice that the database cannot be accessed directly by the clients, only serving Snapshot Mirror read queries, so storing confidential data in the database is not an issue. Finally, the snapshot mirror enables UDP-spoofing to change the emitter address field of the UDP packet to the address of the game server. As a result, the UDP message received by the client from the mirror is indistinguishable from a message sent by the server and is accepted by the client.

7.4.3 Linking with VoltDB

VoltDB is an in-memory distributed database written in java that provides transactional ACID access to data. It enables relational management of the data through stored procedures using a subset of the SQL language. The main particularity of the database is that on each node, the data processing is single-theaded in order to avoid the cost of locking and latching.

We define seven stored procedures (about 1000 lines of java code) responsible for data initialization, write access, synchronized read access and garbage collection (refer to Section 7.3.4 for details). The game server and the snapshot mirrors distantly execute the stored procedures through a reliable TCP connection by using a C++ driver provided with the database.

7.5 Performance issues

QuakeVolt is still in developement[5] and a thorough evaluation is yet to be performed. Although its behavior at large scale is unknown, the prototype has been tested with a few gamers and appears to be fairly playable. In particular, the handover latency seems to be acceptable and switching zones does not degrade the gameplay. We currently use the original Quake III client without any modification, proving the good support of the legacy protocol. The evaluation is planned to start in the next few days with the generated workloads described in Section 7.2.2. The important characteristics to evaluate are:

[5] as of mid-september 2012

- *Scalability*: In/Out bandwidth, CPU and DRAM consumption in function of the population size and the number of snapshot mirrors (test of horizontal scalability).
- *Responsiveness*: average client-side latency variance during game session in function of the number of snapshot mirrors.
- *Game quality*: server-side frames per second in function of the population size.
- *Fault tolerance*: a consistent state of the game is stored in VoltDB. VoltDB has fault tolerance mechanisms, making our system potentially fault tolerant. It would be interesting to evaluate the capacity of the system to support snapshot mirror or even game server failures.

Finally, QuakeVolt is expected to have much better scalability than Quake III, but is not intended to scale indefinitely, because the Game server eventually will become the bottleneck of the system. We would thus like to find the maximal population size of our architecture.

7.6 Conclusion

In this chapter, we described QuakeVolt, an novel scalable architecture for MMOFPS. First, we perform a scalability analysis of standard FPS architectures that shows that client view calculation is the most expensive task performed by the servers. To overcome this bottleneck, QuakeVolt decouples data management from view calculation, allowing the architecture to scale out without excessive fragmentation of the NVE data. It is, to the best of our knowledge, the first proposal that uses recently emerged in-memory distributed databases to consistently manage the NVE data. The adaptation of existing FPS to our design is straightforward: transforming Quake III into QuakeVolt required the modification of less than 1% of the original server code. Furthermore, our design is retro-compatible: gamers with native Quake III clients are able to log-in and successfully play QuakeVolt.

This is an ongoing work, and a thorough evaluation of the benefits brought by our architecture is currently being performed. However, we believe that thanks to the abovementioned benefits, our architecture can potentially foster the emergence of popular, large-scale MMOFPS.

Chapter 8

Conclusions

Contents

8.1 Synthesis

We have described our work to improve distributed architectures for MMOGs. Our proposals aim to enhance the performance of MMOG systems by increasing their "awareness" of the workload characteristics. This is achieved by monitoring the player behavior at runtime, and by interpreting the collected data through simple behavioral models. Such awareness allows the system to (pro)actively adapt the infrastructure to better accommodate the workload.

We base our approach on a wide range of studies that show that the MMOG workload is shaped by player behavior. These studies point out that such workloads are qualitatively non-random and can be approximated with relatively simple models. Our work addresses two aspects of the MMOG workload: player mobility and player distribution. This choice is motivated by the importance of the impact that these aspects have on the underlying architectures. Rather than focusing on a particular infrastructure or application, we propose 3 generic mechanisms that facilitate the integration of workload knowledge at the system level.

Blue Banana, our first contribution is intended to increase the resilience of nearest-neighbor overlays to avatar mobility. High mobility speeds are known to degrade the topology of the system [37]. To address that problem, Blue Banana detects fast movement and predicts the avatar trajectory by using a simple mobility model integrated at the system level. This knowledge allows the overlay to proactively adapt its topology to better handle movement.

DONUT, our second contribution, helps to improve greedy routing in overlays with heterogeneous peer distributions. Such distributions are very common in existing MMOGs and degrade the Small-World property of the underlying p2p infrastructure. By monitoring the player distribution at runtime and exchanging their knowledge, peers acquire a view of the global density distribution. This view is used to readapt overlay shortcuts, preserving polylogarithmic routing.

QuakeVolt, our third contribution is a cluster-based architecture for MMOFPS that enables the game to scale out without unnecessary fragmentation of its NVE. The design of the QuakeVolt architecture facilitates dynamic adaptation to the evolutions of the game population. Additional hosts can easily be added to the architecture during a population peak and servers can be reorganized at runtime to fit with the population distribution.

A non-negligible part of our work is dedicated to the creation of realistic workloads. We believe it to be a necessary condition to any valuable contribution in the domain of distributed MMOGs. This is the last, but not the least of our contributions.

The lessons learned from this thesis are:

- Player movement can be approximated by very simple markov chains.
- This player movement model is sufficiently accurate to make trustworthy short term predictions of the movement.
- Movement prediction can be exploited to improve the robustness of content dissemination p2p overlays.
- Dynamic monitoring of large-scale player distributions in a fully decentralized context is possible at low bandwidth cost.
- Knowledge of the global distributions enables a peer to accurately estimate graph distances between any points of the key space.
- Graph distance estimation can be used to dynamically draw density-aware small-world shortcuts in the p2p overlay graph at runtime, allowing efficient routing despite skewed peer-distributions.
- The true bottleneck of server-based MMOFPSs is player-view computation.
- Decoupling player-view computation from other tasks has two main benefits: (i) it allows the architecture to scale-out without inducing unnecessary fragmentation of the data; and (ii) it enables cheap server placement for dynamic load-balancing.

8.2 Future work

Our future work on MMOGs follows two main directions: improvement of behavioral models and integration of these models at the architecture level.

Modeling complex behavior

The mobility model presented in Chapter 4 is extremely simple and deserves to be refined to capture more complex behavioral patterns. As suggested by Suznjevic *et al.*, other gameplay-dependent states can be introduced in the Markov chain [188]. More expressive models that take the *history* of the player mobility into account could provide an extremely accurate analysis, allowing mid-term prediction of player trajectory.

The global distribution monitoring mechanism introduced in Chapter 6 enables the system to capture collective dynamics of the MMOG population. These observations can be used to build a model of the population evolution. The motivation is to predict population peaks, emergence and vanishing of density hotspots, allowing the system to automatically allocate and place ressources to cope with the load.

Such sophisticated behavior models can be integrated into the infrastructure, increasing its ability to better accommodate complex workloads. Alternatively, the models can be used for realistic trace generation.

Self-adaptable architectures

We plan to improve the Blue Banana prefetching mechanism by taking trajectories of other players into account. By loosely maintaining the state of the peers in the prefetching set, it is possible to evict links that are irrelevent because of their relative movement to the player. More sophisticated history-based models can help the prefetching mechanism to retrieve frequently accessed entities. These improvements can boost the performance of the prefetching mechanism, ensuring the resilience of nearest-neighbor overlays to any player movement.

The density monitoring mechanism of DONUT can be employed as a component to build efficient decentralized maintenance mechanisms. For example, it can be used to enable dynamic load balancing in p2p overlays for range queries. Each peer could maintain two DONUT maps: one for the objects and the other for the peers. The idea of the load balancing mechanism is to maintain the peer distribution in adequation with the object distribution. When an object hotspot appears on the object map, peers move to the hotspot coordinate in order to relieve peers supporting that region of the environment. The DONUT map can also be used to enable hotspot-aware greedy routing. At each hop, the peer locally consults its density map and propagates the message to the peer which minimizes the *graph* distance to the destination[1].

Our work on QuakeVolt is still in progress, and a set of placement problems still need to be addressed. In the first place, we are currently designing dynamic snapshot mirror placement mechanisms. The novelty resides in the fact that thanks to the QuakeVolt architecture, mirror movement does not incur any data transfer. We enable dynamic player distribution monitoring and fine-grain server placement. Mirrors can be dynamically added to support emerging hotspots and change their NVE location to follow group movements with low overhead.

The impact of data placement inside the distributed database also needs to be investigated. Currently, we use the default VoltDB placement scheme, which uniformly distributes

[1] in contrast, traditional greedy routing minimizes the keyspace distance and is not guaranteed to be efficient with skewed keyspaces.

data with a hash function. However, we are planning to replace this strategy by locality-aware data placement (the source code of VoltDB is available) and measure the impact on data access.

Finally, the QuakeVolt process placement inside the datacenter also needs to be studied. In order to reduce network traffic inside the datacenter, some snapshot mirrors can be placed on the physical nodes that host the distributed database, making the database access local to the node. Used together with locality aware data placement, such configurations could dramatically reduce the bandwidth consumption of the architecture.

8.3 General perspectives

The dynamic and heterogeneous nature of MMOG workloads is currently inducing substantial financial ovehead to the game providers [123]. The infrastructures need to be overprovisioned to absorb extreme concurrent user peaks. This is however not sufficient to guarantee good game conditions to all the players, because load-balancing is static, and some regions of the NVE may become rapidly overloaded. Self-adaptable and dynamically reconfigurable architectures are the right approach to overcome these issues.

The results presented in this thesis show that player behavior monitoring can be used as a corner stone for efficient dynamic reconfiguration. The fact that even very simple behavior-aware mechanisms allow to improve the performance of the systems at low cost is particularily promising. We are thus convinced that integration of player behavioral models will be one of the characteristics of future MMOG architectures.

However, at the current state of the research on MMOGs, the proposed enhancements have only a limited impact on commercially popular games. A significant amount of work still has to be performed to enable the integration of research prototypes into real games. We isolated three main research efforts that should, in our opinion foster the emergence of self-adaptable commercial architectures: (i) Development of workload modeling, (ii) spread of evaluation with realistic workloads and (iii) design of self-adaptable systems.

Development of workload modeling

A thorough survey of the different MMOG workload characteristics still has to be performed. It should provide a classification of the collective dynamics of player populations, including mobility, distribution, interest management and population evolution in function of their predictability. The study must also supply a comprehensive summary of the impact of these dynamics on the underlying infrastructure.

Parts of this study have already been conducted. Evolution of the global MMOG population has been shown to be predictable through the observation of player session lengths [67]. Accurate interest management algorithms taking into account the specificities of human perception have been proposed [47, 75]. Simple properties of the evolution of player distributions have been observed [196].

The goal of this research is to isolate characteristic predictable patterns that can be described by reasonably simple models from fundamentally random behavior. We believe this to be necessary to highlight and solve the real challenges of the MMOG workloads.

Spread of evaluation with realistic workloads

Substantial effort has to be made to endow the research community with a universally adopted, realistic evaluation testbed. Currently, publicly available data-sets are uncommon, because trace collection from commercial MMOGs is hard, and only a few studies focused on realistic trace generation. As a result, many architectures for MMOGs have been evaluated under unrealistic workload assumptions, making real contributions hard to understand [157].

To solve this problem, the community needs to adopt a set of trustworthy benchmarks based on real-application datasets and realistic trace generators. This would facilitate a fair comparison of the different proposals, increasing the visibility of the research effort which is, in our opinion, mandatory to impact real-world architectures[2].

Design of self-adaptable systems

The final research direction is the integration of behavioral models in the architectures for MMOGs, which requires the development of novel monitoring techniques and the increase of the architectural elasticity.

Current research prototypes of MMOG architectures are separated in two main families: server-based and peer-to-peer. Peer-to-peer architectures are inherently better suited for elasticity, but monitoring is easier in server-based systems. We thus believe that self-adaptation can be enabled for both approaches. Being convinced that both paradigms will participate to the emergence of future MMOGs, we took care in this thesis to step aside from stating in favor of one of them.

[2]This opinion has also been exposed by Miller *et al.* [146].

Part II

Appendix I: Relax DHT

Chapter 9

Churn resilient strategy for distributed hash tables

Contents

9.1 Introduction

Distributed Hash Tables (DHTs), are distributed storage services that use a structured overlay relying on key-based routing (KBR) protocols [170, 184] (see Chapter 2). DHTs provide the system designer with a powerful abstraction for wide-area persistent storage, hiding the

complexity of network routing, replication, and fault-tolerance. Therefore, DHTs are increasingly used for dependable and secure applications like backup systems [133], distributed file systems [85, 59], multi-range query systems [173, 108, 71], and content distribution systems [119].

A practical limit in the performance and the availability of a DHT relies in the variations of the network structure due to the unanticipated arrival and departure of peers. Such variations, called *churn*, induce at least performance degradation, due to the reorganization in the set of the replicas of the affected data that consumes bandwidth and CPU cycles, and at worst the loss of some data. In fact, Rodrigues and Blake have shown that using classical DHTs to store large amounts of data is only viable if the peer lifetime is in the order of several days [168]. Until now, the problem of churn resilience has been mostly addressed at the peer-to-peer (P2P) routing level to ensure the reachability of peers by maintaining the consistency of the logical neighborhood, *e.g.*, the leafset, of a peer [164, 66]. At the storage level, avoiding data migration is still an issue when a reconfiguration of the peers has to be done.

In a DHT, each peer and each data block is assigned a key (*i.e.*, an identifier). A data block's key is usually the result of a hash function performed on the block. Each data block is associated a *root* peer whose identifier is numerically the closest to its key. The traditional replication scheme uses the subset of the root's leafset containing the closest logical peers to store the copies of a data block [170]. Therefore, if a peer joins or leaves the leafset, the DHT enforces the placement constraint on the closest peers and may migrate many data blocks. In fact, it has been shown that the cost of these migrations can be high in terms of bandwidth consumption [133].

This chapter proposes RelaxDHT, a variant of the leafset replication strategy designed to tolerate higher churn rates than traditional DHT protocols. Our goal is to avoid data block migrations when the desired number of replicas is still available in the DHT. We relax the "logically closest" placement constraint on block copies and allow a peer to be inserted in the leafset without forcing migration. Then, to reliably locate the block copies, the root peer of a block maintains replicated localization metadata. Metadata management is integrated to the existing leafset management protocol and does not incur substantial overhead in practice.

We have implemented both PAST and RelaxDHT, on top of PeerSim [118]. The main results of our evaluations are:

- We show that RelaxDHT achieves higher data availability in presence of churn, than the original PAST replication strategy. For a connection or disconnection occurring every six seconds (for 1000 peers) our strategy looses three times less blocks than the PAST's one.

- We show that our replication strategy induces less unnecessary block transfers than the PAST's one.

- If message compression is used, our maintenance protocol is more lightweight than the maintenance protocol of PAST.

9.2 Background and motivation

DHT based P2P systems are usually structured in three layers: 1) a routing layer, 2) the DHT itself, 3) the application that uses the DHT. The routing layer is based on keys for peer identification and is therefore commonly qualified as *Key-Based Routing* (KBR). Such KBR layers hide the complexity of scalable routing, fault tolerance, and self-organizing overlays to the upper layers. In recent years, many research efforts have been made to improve the resilience of the KBR layer to a high churn rate [164]. The main examples of KBR layers are Pastry [169], Chord [184], Tapestry [204] and Kademlia [144]. The DHT layer is responsible for storing data blocks. It implements a distributed storage service that provides persistence, fault tolerance and can scale up to a large number of peers. DHTs provide simple `get` and `put` abstractions that greatly simplify the task of building large-scale distributed applications. PAST [170] and DHash [86] are DHTs respectively built on top of Pastry [169] and Chord [184]. Thanks to their simplicity and efficiency, the DHTs became standard components of modern distributed applications. They are used in storage systems supporting multi-range queries [173, 108, 71], mobile *ad hoc* networks [203], aswell as massively multiplayer online gaming [193], or robust backup systems [133].

In the rest of this section we present replication techniques that are used to implement the DHT layer. Then, we describe the related work that considers the impact of churn on the replicated data stored in the DHT.

9.2.1 Replication in DHTs

In a DHT, all the peers are arranged in a logical structure according to their identifiers, commonly a ring as used in Chord [184] and Pastry [169] or a d-dimensional torus as implemented in CAN [163] and Tapestry [204].

A peer possesses a restricted local knowledge of the P2P network, *i.e.*, the leafset, which amounts to a list of `L` close neighbors in the ring. For instance, in Pastry the leafset contains the addresses of the `L/2` closest neighbors in the clockwise direction of the ring, and the `L/2` closest neighbors counter-clockwise. Periodically, each peer monitors its leafset, removing peers that have disconnected from the overlay and adding new neighbor peers as they join the ring.

In order to tolerate failures, each data block is replicated on `k` peers which compose the *replica set* of a data block. Two protocols are in charge of the replica management, the initial placement protocol and the maintenance protocol. We now describe existing solutions to implement these two protocols.

9.2.1.1 Replica placement protocols

There are two main basic replica placement strategies, leafset-based and multiple-key-based:

Leafset-based replication The data block's root is responsible for storing one copy of the block. The block is also replicated on the root's closest neighbors in a subset of the leafset. The neighbors storing a copy of the data block may be either immediate successors of the root in the ring as in DHash [86], immediate predecessors or both as in

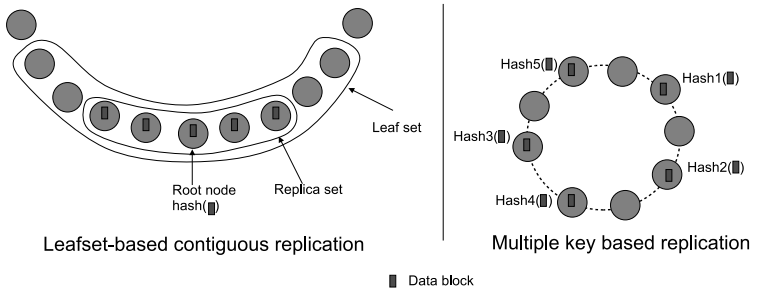


Figure 9.1: *Leafset-based and multiple-key-based replication ($k = 5$).*

PAST [170]. Therefore, the different copies of a block are stored *contiguously* in the ring as shown by Figure 9.1.

Multiple key replication This approach relies on computing k different storage keys corresponding to different root peers for each data block. Data blocks are then replicated on the k root peers. This solution has been implemented by CAN [163] and Tapestry [204]. GFS [101] uses a variant based on random placement to improve data repair performance. *Path* and *symmetric replication* are variants of multiple-key-based replication [102, 128].

Lian *et al.* propose a hybrid stripe replication scheme where small objects are grouped in blocks and then randomly placed [138]. Using an analytical framework, they show that their scheme achieves on near-optimal reliability. Finally, several works have focused on the placement strategies based on availability of nodes. Van Renesse [191] proposes a replica placement algorithm on DHT by considering the reliability of nodes and placing copies on nodes until the desired availability is achieved. To this end, he proposes to track the reliability of each node such that each node knows the reliability information about each peer. In FARSITE [31], dynamic placement strategies improve the availability of files. Files are swapped between servers according to the current availability of these latter. With theses approaches, the number of copies can be reduced. However, such approaches may lead to a high unbalanced distribution whereby highly available nodes contain most of the replicas and can become overloaded. Furthermore, such solutions are complementary, and taking nodes-availability into account could be done on top of RelaxDHT.

9.2.1.2 Maintenance protocols

The maintenance protocols have to maintain k copies of each data block without violating the initial placement strategy. It means that the k copies of each data block have to be stored on the root-peer-contiguous neighbors in the case of the leafset-based replication scheme and on the root peers in the multiple-key-based replication scheme.

The leafset-based maintenance mechanism is based on periodic information exchanges within the leafsets. Leafset-based maintenance mechanisms have good scalable properties: the number of messages of the protocol does not depend on the number of data blocks managed by a peer, but only on the leafset size. Yet, even if the overall quantity of maintenance data linearly grows with the number of blocks, it is possible to efficiently aggregate and

compress the data. For instance, in the fully decentralized PAST maintenance protocol [170], each peer sends a bloom filter to all the peers in its leafset. The bloom filter is a compact probabilistic data structure containing the set of block identifiers stored by the peer [57]. When a leafset peer receives such a request, it uses the bloom filter to determine whether it stores one or more blocks that the requester should also store. It then answers with the list of the keys of these blocks. The requesting peer can then fetch the missing blocks listed in all the answers it receives. Notice that the maintenance interval at the KBR layer is much smaller than the maintenance interval of blocks (for instance in PAST the default value of leafset maintenance interval is 1 minute whereas the data block interval is set to 10 minutes).

In the case of the multiple-key-based replication strategies, the maintenance has to be done on a "per data block" basis. For each data block stored in the system, it is necessary to periodically check if the different root peers are still alive and are still storing a copy of the data block. This replication method has the drawback that a maintenance message has to be sent to each root of each data block, which means that the number of messages linearly grows with the number of blocks. Therefore, it seems impossible to aggregate maintenance information in order to optimize its propagation ,*e.g.*, by compressing the messages like in leafset-based replication. For backup and file systems that may store up to thousands of data blocks per peer, this is a severe limitation.

9.2.2 Impact of the churn on the DHT performance

A high churn rate induces consequent changes in the P2P network, and the maintenance protocol must frequently adapt to the new structure by migrating data blocks. While some migrations are mandatory to restore `k` copies, some others are only necessary to enforce placement invariants.

For example, as a new peer joins the system, if its identifier is closer to a block's key than the identifier of the current block's root, the data block needs to be migrated on the new peer. A second example occurs in leafset-based replication if a peer joins the DHT between two nodes storing replicas of the same block. One of the replicas then needs to be migrated on the new peer in order to maintain the contiguity between replicas. It should be noticed that larger the replica set is, higher is the probability for a new peer to induce migrations. Kim and Park try to limit this problem by allowing data blocks to interleave in leafsets [125]. However, they have to maintain a global knowledge of the complete leafset: each peer has to know the content of all the peers in its leafset. Unfortunately, the maintenance algorithm is not described in detail and its real cost is unknown.

In the case of the multiple-key-based replication strategy, a new peer may be inserted between two replicas without requiring the migration of data blocks, as long as the new peer identifier does not make it one of the data block roots.

9.3 Relaxing the DHT's placement constraints to tolerate churn

The goal of this work to is to design a DHT that tolerates a high rate of churn without degradation of performance. In order to achieve this, we avoid to copy data blocks when this is not mandatory to restore a missing replica. We introduce a leafset based replication that

relaxes the placement constraints in the leafset. Our solution, named RelaxDHT, is presented thereafter.

9.3.1 Overview of RelaxDHT

RelaxDHT is built on top of a KBR layer such as Pastry or Chord. Our design decisions are to use replica localization metadata and separate them from data block storage. We keep the notion of a root peer for each data block. However, the root peer does not necessarily store a copy of the blocks for which it is the root. It only maintains metadata describing the replica set and periodically sends messages to the replica set peers to ensure that they keep storing their copy. It is possible, but not mandatory, for a peer to be both root and part of the replica set of the same data block. Using localization metadata allows a data block replica to be anywhere in the leafset; a new peer may join a leafset without necessarily inducing data block migrations.

We choose to restrain the localization of replicas within the root's leafset for two reasons. First, we believe that it is more scalable, because the *number* of messages of our protocol does not depend on the number of data blocks managed by a peer, but only on the leafset size [1]. Second, since the routing layer already induces many exchanges within leafsets, the local view of the leafset at the DHT-layer can be used as a failure detector. We now detail the salient aspects of the RelaxDHT algorithm.

9.3.1.1 Insertion of a new data block

To be stored in the system, a data block `b` with key `key` is inserted using the `put(k,b)` operation. This operation produces an "insert" message which is sent to the root peer through the KBR layer. Then, the root randomly chooses a replica set of `k` peers around the center of the leafset. The peer belongs to the center of a root's leafset if its hop-distance to the root is less than a fixed value ϵ_1. The algorithm chooses the replica-peers inside the center to reduce the probability that a chosen peer quickly quits the leafset due to the arrival of new peers. Finally, the root sends to the replica set peers a "store message" containing:

1. the data block itself,
2. the identity of the peers in the replica set,
3. the identity of the root.

As a peer may be root for several data blocks and part of the replica set of other data blocks, it stores:

1. a list `rootOfList` of data block identifiers with their associated replica-set-peer list for blocks for which it is the root;
2. a list `replicaOfList` of data blocks for which it is part of the replica set. Along with data blocks, this list also contains: the identifier of the data block, the associated replica set peer-list and the identity of the root peer.

[1] see section 9.2.1.2.

A *lease counter* is associated to each stored data block. This counter is set to the value "`Lease`" which is a constant. It is then decremented at each leafset maintenance time. The maintenance protocol described below is responsible for periodically resetting this counter to "`Lease`".

9.3.1.2 Maintenance protocol

The goal of this periodic protocol is to ensure that: 1) a root peer exists for each data block. The root is the peer that has the closest identifier to the data block's one; 2) each data block is replicated on `k` peers located in the data block root's leafset.

At each period T, a peer `p` executes Algorithm 4, so as to send maintenance messages to the other peers of the leafset. It is important to notice that Algorithm 4 uses the leafset knowledge maintained by the peer routing layer which is relatively accurate because the inter-maintenance time of the peers is much smaller than the DHT-layer's one.

Algorithm 4: RelaxDHT maintenance message construction.

```
   Result: msgs, the built messages.
 1 for data ∈ rootOfList do
 2     for replica ∈ data.replicaSet do
 3         if NOT isInCenter (replica,leafset) then
 4             newPeer =choosePeer (replica,leafset);
 5             replace (data.replicaSet, replica,newPeer);
 6     for replica ∈ data.replicaSet do
 7         add(msgs [replica ],<STORE, data.blockID, data.replicaSet >);
 8 for data ∈ replicaOfList do
 9     if NOT checkRoot (data.rootPeer,leafset) then
10         newRoot =getRoot (data.blockID,leafset);
11         add(msgs [newRoot ],<NEW ROOT, data.blockID, data.replicaSet >);
12 for p ∈ leafset do
13     if NOT empty (msgs [p ]) then
14         send(msgs [p ],p);
```

The messages constructed by Algorithm 4 contain a set composed of the following elements:

STORE elements for asking a replica node to keep storing a specific data block (*i.e.*, resetting the lease counter).

NEW ROOT elements for notifying a node that it has become the new root of a data block.

Each element contains both a data block identifier and the associated replica-set-peer list.

Algorithm 4 sends at most one single message to each leafset member.

Algorithm 4 is composed of three phases: the first one computes STORE elements using the `rootOfList` structure (lines 1 to 7), the second one computes NEW ROOT elements using the `replicaOfList` structure (from line 8 to 11), the last one sends messages to the destination peers in the leafset (line 12 to the end). Message elements computed in the two first phases are added to *msgs[]*. *msgs[q]* is a message containing all the elements to send to node q at the last phase.

Therefore, each peer periodically sends a maximum of `leafset-size` maintenance messages to its neighbors. The size of these messages depends on the number of blocks. However, we will show in the performance section that our approach remains scalable even if the number of blocks per node is huge.

In the first phase, for each block for which the peer is the root, it checks if all the replicas are placed sufficiently far from the extremity of the leafset (line 3) using its local view provided by the KBR layer. We define the *extended center* of a root's leafset as the set of the peers which distance to the root is less than a system-defined value ϵ_2 (with $\epsilon_2 > \epsilon_1$, *i.e.*, the center of the leafset is a subset of the extended center of the leafset[2]). If a peer is out of the extended center, its placement is considered to be too close to the leafset extremity. In that case, the algorithm replaces the peer by randomly choosing a new peer in the center of the leafset. It then updates the replica set of the block (lines 4 and 5). Finally, the peer adds a STORE element in each replica-set-peer message (lines 6 and 7). In the second phase, for each block stored by the peer (*i.e.*, the peer is part of the block's replica set), it checks if the root node did not change.

This is done by checking that the identifier of the current root is still the closest to the block's key (line 9). If the root has changed, the peer adds a NEW ROOT message element to announce to the future root peer that it is the new root of the data block. Finally, from line 12 to line 14, a loop sends the computed messages to each leafset member.

Note that it is possible to temporarily have two different peers acting as a root peer for the same data block. However, this phenomenon is rare. This may happen when a peer joins, becomes root of a data block and then receive a NEW ROOT message element from a replica node for this data block *before* the old ROOT (its direct neighbor in the leafset) detects its insertion. Moreover, even if this happens, it does not lead to data loss. The replica set of the root will simply receive more STORE messages, and the anomaly will be resolved with the next maintenance of the wrong root (*i.e.* at most 10 minutes later in our system).

9.3.1.3 Maintenance message treatment

Algorithm 5: RelaxDHT maintenance message reception.

```
   Data: message, the received message.
1  for elt ∈ message do
2      switch elt.type do
3          case STORE
4              if elt.id ∈ replicaOfList then
5                  newLease(replicaOfList,elt.id);
6                  updateRepSet(replicaOfList,<elt.id,elt.replicaSet >);
7              else
8                  requestBlock(elt.id,elt.replicaSet);
9          case NEW ROOT
10             addRootElt(rootOfList,<elt.id,elt.replicaSet >);
```

For a STORE element (line 3), if the peer already stores a copy of the corresponding data block, it resets the associated lease counter and updates the corresponding replica set

[2] see section 9.3.1.1 for the definition of the leafset center

if necessary (lines 4, 5 and 6). If the peer does not store the associated data block (*i.e.*, it is the first STORE message element for this data block received by this peer), it fetches the latter from one of the peers mentioned in the received replica set (line 8).

For a NEW ROOT element , a peer adds the data block id and the corresponding replica set in the `rootOfList` structure (line 10).

9.3.1.4 End of a lease treatment

If a data block lease counter reaches 0, it means that no STORE element has been received for a long time. This can be the result of numerous insertions that have pushed the peer outside the leafset center of the data block's root. The peer sends a message to the root peer of the data block to ask for the authorization to delete the block. Then, the peer receives an answer from the root peer, either allowing to remove the data block or asking to put the data block again in the DHT (if the data block has been lost).

9.3.1.5 Impact of the ϵ_1 and ϵ_2 values on the protocol performance

The placement constraints of RelaxDHT are defined by the parameters ϵ_1 and ϵ_2. The first parameter defines a set of nodes around a block's root (called the center of the leafset) on which the replicas of the block are initially placed. The second one defines the tolerance threshold of the protocol: the replacement of the replica inside the center is performed only its hop distance to the root exceeds ϵ_2 due to node arrival.

It is thus important to set optimal values for ϵ_1 and ϵ_2. We varied these parameters to study how they affect the churn resilience of the protocol. As it can be seen on the figure 2, ϵ_1 has the most important impact on the churn resilience. Low values of ϵ_1 reduce the churn tolerance of the DHT, because they strengthen the initial placement constraints. Strong placement constraints increase the correlation between the blocks stored by a node, thus reducing the number of sources that can be used to restore replicas when a node fails. For example, if $\epsilon_1 = k$, the replicas of a block are necessarily placed on the root and its $k - 1$ contiguious neighbors. It means that in case of the failure of a node n, the replicas of all the blocks stored on n are located on $k + 1$ nodes, which is less than for larger values of ϵ_1.

The ϵ_2 parameter should not be too low because it would reduce the placement tolerance of the protocol, and should not be to high, because if a replica shifts out the extended center *and* out of the leafset, its lease is no more renewed, which may lead to the loss of the replica. However, the figure 2 shows that the parameter has less influence on data loss than ϵ_1, because its value is important only when the leafset faces an important node arrival. For our evaluation, we choose $\epsilon_1 = 4$ and $\epsilon_2 = 8$.

9.3.2 Side effects and limitations

By relaxing placement constraints of data block copies in leafsets, our replication strategy for DHTs significantly reduces the number of data blocks to be transferred when peers join or leave the system. Thanks to this, we show in the next section that our maintenance mechanism allows us to better tolerate churn. However, this enhancement has several possibly inconvenient effects. The two main ones concern the data block distribution on the peers and

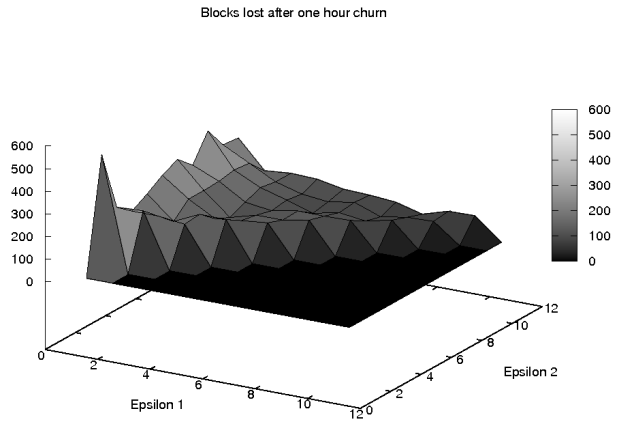


Figure 9.2: *Number of blocks lost on a 100 peer-DHT after one hour churn in function of the ϵ_1 ans ϵ_2 parameters.*

the lookup performance. While the changes in data block distribution can provide positive effects, the lookup performance can be slightly reduced.

9.3.2.1 Data blocks distribution

With usual replication strategies in DHT's, the data blocks are distributed among peers according to some hash function. Therefore, if the number of data blocks is big enough, data blocks should be uniformly distributed among all the peers. When using RelaxDHT, this remains true if there are no peer connections/disconnections. However, in presence of churn, as our maintenance mechanism does not transfer data blocks if it is not necessary, new peers will store much less data blocks than peers involved for a longer time in the DHT. It is important to notice that this side effect is rather positive: the more stable a peer is, the more data blocks it will store. Furthermore, it is possible to easily counter this effect by taking into account the quantity of stored data blocks while randomly choosing peers to add in replica sets.

9.3.2.2 Lookup performance

Our strategy also induces additionnal delay while retrieving blocks from the DHT because the placement of the data block on its root is no more mandatory. During a lookup, if no replica is stored on the root, the root has to forward the request to one of the k replica nodes. This adds one hop to the request, and the latency of the last link is added to the overall lookup latency. This never happens in PAST, because the root of a block necessarily stores it.

We compared the lookup performance of both strategies by simulating a DHT with 1000 nodes in presence of churn. As we can see in figure 3.a, the lookup latency is in average about 13% lower with PAST than with RelaxDHT. The figure 3.b shows that the percentage of failed lookups increases the same way for the two strategies as the churn intensifies. RelaxDHT has a slightly better failure percentage, because all the nodes of the replica set (*i.e.*, the root

and the k replica nodes) have the complete list of the replicas, thus allowing to re-route the request if the root does not have the block. In PAST, if the block on the root is missing, the lookup fails.

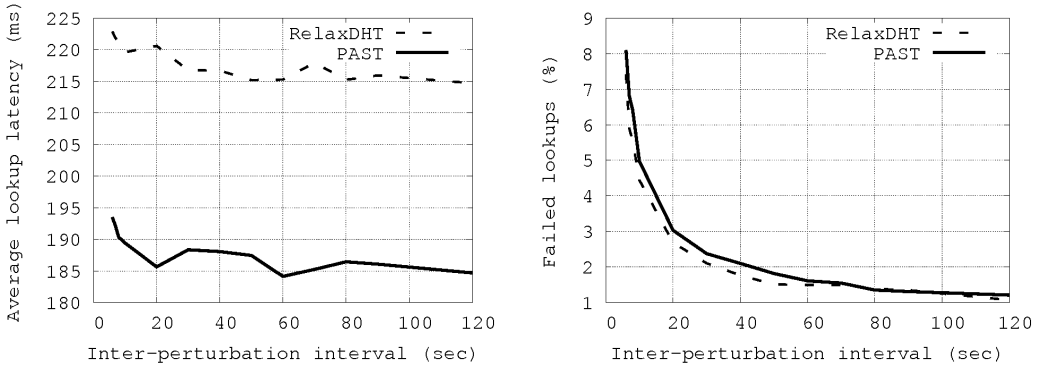


Figure 9.3: *For 1000 nodes, varying the churn rate: (a) Average lookup latency, (b) Percentage of failed lookups.*

The observed latency overhead is constant and relatively low, because the root has the list of all the replica-nodes, and is able to choose the least expensive link for the last hop. As the main concern of DHTs is to provide large scale persistent data storage, we consider that the slight latency overhead in data retrieval is affordable in most of the cases. Moreover, if the lookup performance is essential to the application, it is possible to lower the probability that a root does not to store a replica by choosing small values for ϵ_1[3]. Small values of ϵ_1 increase the placement constraints, reducing the churn resilience, but a tradeoff between the lookup efficiency and the churn resilience can be found.

9.4 Evaluation

This section provides a comparative evaluation of RelaxDHT and PAST [170]. This evaluation, based on discrete event simulations, shows that RelaxDHT provides a considerably better tolerance to churn: for the same churn levels, the number of data losses is divided by more than two when comparing both systems.

9.4.1 Experimental setup

To evaluate RelaxDHT, we have built a discrete event simulator using the PeerSim [118] simulation kernel. We have implemented both PAST and RelaxDHT strategies. It is important to note that all the different layers and all message exchanges are simulated. Our simulator also takes into account network congestion because DHT maintenance during churn incurs a lot of simultaneous downloads that are likely to congest the links. Moreover, we used

[3] *e.g.*, if $\epsilon_1 = k$, k being the replica rate, all the replicas are contiguously stored around the root, forcing the root to store a replica.

real-life latency traces. Measurements performed by Madhyastha et al. [143] between DNS servers in 2004 were injected in the simulation to simulate a realistic latency distribution.

For all the simulation results presented in the section, we used a 1000-peer network with the following parameters (for both PAST and RelaxDHT):

- a leafset size of 24, which is the Pastry default value;
- an inter-maintenance duration of 10 minutes at the DHT level;
- an inter-maintenance duration of 1 minute at the KBR level;
- 100,000 data blocks of 10,000 KB replicated 3 times;
- network links of 1 Mbits/s for upload and 10 Mbits/s for download.
- for RelaxDHT maintenance protocol, the leafset center is set to the 8 central nodes, while the extended center is set to the 16 central nodes of the leafset[4].
- the replica set lease is set to 5 DHT maintenance periods, *i.e.*, 50 minutes.

We have injected churn following two different scenarii:

One hour churn This scenario allows us to study 1) how many data blocks are lost after a perturbation period and 2) how long it takes to the system to return to a state where all remaining/non-lost data blocks are replicated k times. It consists of one perturbation phase with churn during one hour followed by another phase without connections/ disconnections. In real-life, there are some period without churn within a leafset, and the system has to take advantage of them to converge to a safer state.

Continuous churn This scenario focuses on the perturbation period: it provides the ability to study how the system resists when it has to repair lost copies in presence of churn. For this set of simulations, we focus on phase one of the previous case observing a snapshot of the DHT during churn.

During the churn phase at each *perturbation period* we randomly choose either a new peer connection or a peer disconnection. This perturbation can occur anywhere in the ring (uniformly chosen). We have run numerous simulations varying the inter-perturbation delay.

9.4.2 Single failure

In order to better understand the simulation results using the two scenarii, we start by measuring the impact of a single peer failure/disconnection. When a single peer fails, data blocks it stored have to be replicated on a new one. Those blocks are transferred to the new peer in order to rebuild the initial replication degree k. In our simulations, with the parameters given above, it takes 4609 seconds to PAST to recover the failure: *i.e.*, to create a new replica for each block stored on the faulty peer, while with RelaxDHT, it takes only 1889 seconds. The number of peers involved in the recovery is indeed much more important. This gain is due to the parallelization of the data block transfers:

[4] i.e., $\epsilon_1 = 4$ and $\epsilon_2 = 8$, see section 9.3 for the description of these sets.

- in PAST, the content of contiguous peers is really correlated. With a replication degree of 3, only peers located at one or two hops of the faulty peer in the ring may be used as sources or destinations for data transfers. In fact, only `k+1` peers are involved in the recovery of one faulty peer, where `k` is the replication factor.
- in RelaxDHT, all of the peers in the extended center of the leafset (the extended center contains 16 peers in our simulations) may be involved in the transfers.

9.4.3 One hour churn

We first study the number of lost data blocks (data block for which the 3 copies are lost) in PAST and in RelaxDHT under the same churn conditions. Figure 9.4.a shows the number of lost data blocks after a one-hour churn period. The inter-perturbation delay is increasing along the X axis. With RelaxDHT and our maintenance protocol, the number of lost data blocks is 2 to 3 times lower than with the PAST's one.

The main reason of the result presented above is that, using the PAST replication strategy, the peers have more data blocks to download. It implies that the mean download time of one data block is longer using PAST replication strategy. Indeed, the maintenance of the replication scheme location constraints generate a continuous network traffic that slows down critical transfers, preventing efficient data block copy restoration.

Figure 9.4.b shows the total number of blocks exchanged in both cases. There again, the X axis represents the inter-perturbation delay. The figure shows that with RelaxDHT the number of exchanged blocks is always smaller than with PAST. This is mainly due to the fact that in PAST's case some transfers are not critical and are only done to preserve the replication scheme constraints. For instance, each time a new peer joins, it becomes root of some data blocks [5] or is inserted within replica sets that should remain contiguous. As a consequence, a reorganization of the storage has to be performed. In RelaxDHT, most of the non-critical bandwidth consumption is replaced by critical data transfers: the maintenance traffic is more efficient.

Using PAST replication strategy, a newly inserted peer may need to download data blocks during many hours, even if no failure/disconnection occurs. During all this time, its neighbors need to send it the required data blocks, using a large part of their upload bandwidth.

In the case of RelaxDHT, *no* or very *few* data block transfers are required when new peers join the system. It becomes mandatory only if some copies become too far from their root-peer in the logical ring (*i.e.*, they leave the leafset center, which is, in our simulation, formed of the 16 peers that are the closest to the root peer). In this case, they have to be transferred closer to the root before their hosting peer leaves the root-peer's leafset. With a replication degree of 3 and a leafset size of 24, many peers can join a leafset before any data block transfer is required.

Finally, we have measured the time the system takes to return to a normal state in which every remaining data block is replicated `k` times. Blocks for which all copies are lost can not be recovered and are thus not taken into account. Figure 9.4.c shows the results obtained

[5] because its identifier is closer than the current root-peer's one

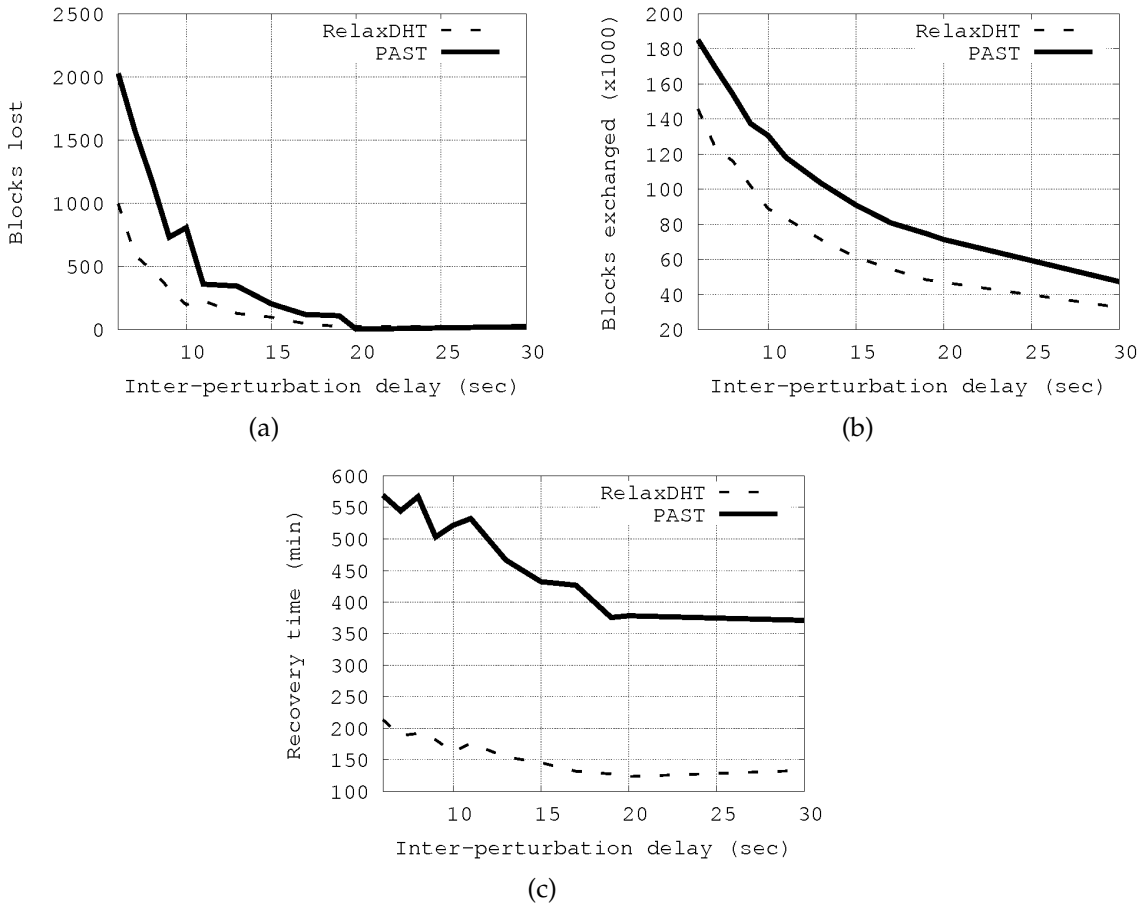


Figure 9.4: *(a) Number of data block lost (i.e., all copies are lost) after one hour of churn, (b) Number of exchanged data blocks to restore a stable state after one hour of churn, (c) Recovery time: time for retrieving all the copies of every remaining data block.*

while varying the delay between perturbations. We can observe that the recovery time is four to five times longer in the case where PAST is used compared to RelaxDHT. This result is mainly explained by the efficiency of the maintenance protocol: RelaxDHT transfers only very few blocks for placement constraints compared to PAST's one.

This last result shows that the DHT using RelaxDHT repairs damaged data blocks (data blocks for which some copies are lost) faster than PAST. It means that it will be promptly able to cope with a new churn phase. The next section describes our simulations with continuous churn.

9.4.4 Continuous churn

The above simulation results show that: 1) RelaxDHT induces less data transfers, and 2) remaining data transfers are more parallelized. Thanks to this two points, even if the system remains under continuous churn, RelaxDHT provides a better churn tolerance.

Figure 5.a shows the number of data block losses under continuous churn using the parameters described at the beginning of this section. Here again, we can see that PAST starts to loose data blocks with lower churn rate than RelaxDHT. This delay needs to be of less than 20 seconds [6] for RelaxDHT to loose a significative amount of blocks, whereas PAST continues to loose blocks even for inter-perturbation intervals greater than 40 seconds. If the inter-perturbation delay continues to decrease, the number of lost data blocks using RelaxDHT strategy remains less than the third of the number of data blocks lost using PAST strategy.

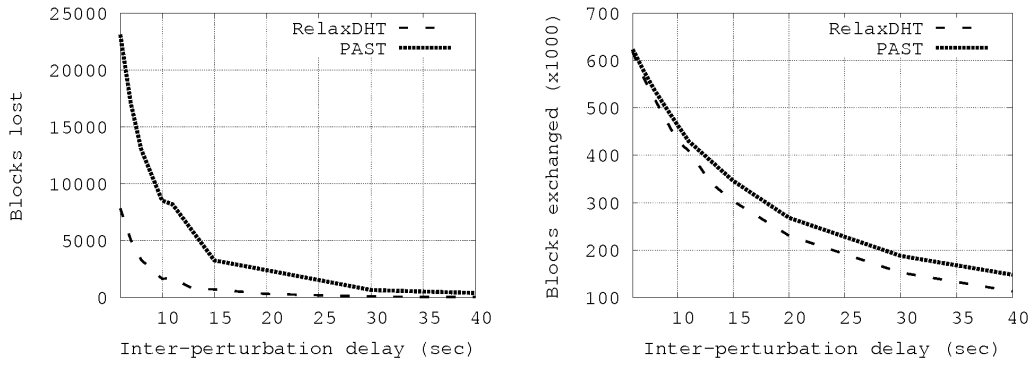


Figure 9.5: *While the system is under continuous churn: (a) Number of data block transfers, (b) Number of data block losses (all k copies lost)*

Finally, Figure 5.b confirms that even with a continuous churn pattern, during a 5 hour run, the proposed solution behaves better than PAST. The number of data transfers required by RelaxDHT is still always smaller than the number of data transfers induced by the PAST's replication strategy (about 10% smaller for a 10 second inter-perturbation delay and 50% smaller for a 35 second delay). In that case, the continuous churn rate induces more critical

[6] for 1000 nodes

data exchange than in the first scenario. For RelaxDHT, the bandwidth gained by relaxing the placement constraints is mostly re-used to exchange critical data. Therefore, the bandwidth consumption of our protocol is closer to the PAST's one than in the first scenario, but the bandwidth management is more efficient.

9.5 Maintenance protocol cost

In the simulation results presented above, we have considered that the maintenance protocol cost was negligible. This section evaluates the network cost of PAST and RelaxDHT maintenance protocols. Both RelaxDHT and PAST peers send at most one maintenance message to each leafset member, that is why it is appropriate to compare the *global* amount of data to be sent by a node in order to perform a maintenance. Then, we evaluate the optimizations that can be made to reduce network cost of the protocols.

9.5.1 Amount of exchanged data

Let: 1) M be the overall number of blocks in the DHT, 2) N be the number of DHT nodes, 3) m be the number of peers in the center of the leafset RelaxDHT uses to replicate a block, 4) k be the mean replication factor and 5) $|leafset|$ the size of the leafset of the DHT (PAST and RelaxDHT have the same $|leafset|$).

Let S be the set of blocks a node has to maintain. As the blocks are uniformly distributed by the hash function, we have in average $|S| = \frac{M \times k}{N}$.

PAST maintenance cost

While performing a maintenance, a PAST peer sends all the identifiers of blocks it stores to every member of its leafset. Therefore, the average cost of the maintenance is $Maintenance_{PAST} = |S| \times |leafset| = \frac{M \times k}{N} \times |leafset|$ identifiers to send to the leafset neighbors.

RelaxDHT maintenance cost

In RelaxDHT, on each node, S can be partitioned in 3 subsets:

1. **Subset R.** Data blocks for which the node is the root and is *not* part of the replica set. Since the DHT hash-function is uniform, and each block has a root, $|R| = \frac{M}{N}$ blocks.
2. **Subset T.** Data blocks for which the node is not the root of the block, but is part of its replica set. Let T be the subset of S formed of such blocks. Since $S = R \cup T \cup (T \cap R)$, $|T| = \frac{M \times k}{N} - \frac{M}{N} - |T \cap R|$.
3. **Subset $T \cap R$.** Data blocks for which the node is both the root of the block and part of the replica set. As a block is inserted, the root chooses k replica-peers among m central leafset-members. Let p be the probability for the root to choose itself as a replica[7]. Thus, $|T \cap R| = \frac{M}{N} \times p$ blocks.

[7] in our case, the choice is made at random with a uniform distribution, therefore $p = \frac{k}{m}$.

Each maintenance time, Algorithm 4 computes STORE and NEW ROOT elements: 1) For each block of the set R, k STORE elements are created. 2) For each block of the set $T \cap R$, only $k-1$ STORE elements are created. 3) There are no STORE elements for blocks that belong to the set T.

Therefore, to perform a maintenance, a node has to send $\#ST = |R| \times k + |T \cap R| \times (k-1)$ STORE elements.

Moreover, depending on the churn rate, some NEW ROOT elements are sent for the members of the set T. If there is no churn at all, no NEW ROOT elements are sent. On the other hand, in the worst case, it could be mandatory to send a NEW ROOT element per member of T. In the worst case, the number of NEW ROOT elements is therefore: $\#NR = |T| \approx \frac{M}{N} \times (k-1)$. It is important to notice that this occurs only in case of a massive node failure. In practice, the amount of NEW ROOT messages induced by churn is considerably smaller.

Each element contains $k+1$ identifiers: the identifier of the block and the k identifiers of the replica set members. Thus, in average, a RelaxDHT node has to send $Maintenance_{RelaxDHT} = (\#NR + \#ST) \times (k+1) \approx \frac{M \times k}{N} \times p \times (k+1)$ identifiers per maintenance.

Comparison

Putting aside all optimisations that are made for both RelaxDHT and PAST, the cost of both protocols can now be compared. As we usually have $k \ll |leafset|$ (for example, in our simulations, $k = 3$ and $|leafset| = 24$) , $Maintenance_{RelaxDHT} < \frac{M \times k}{N} \times p \times |leafset|$

Therefore, since $p < 1$, $Maintenance_{RelaxDHT} < Maintenance_{PAST}$.

This result is mainly due to the fact that PAST peers send their content to all the members of their leafset while RelaxDHT peers use extra metadata to locally compute the information that needs to be transferred from one peer to another. Moreover, as there are less NEW ROOT messages when there is less churn means that the RelaxDHT protocol is able to adapt itself to be more lightweight as churn drops, whereas PAST protocol cost is constant, even if there is no churn.

We now discuss the impact of the optimizations that can be made to both protocols on their network load.

9.5.2 Optimization of maintenance message size

In PAST, the optimization of maintenance traffic relies on the usage of bloom filters. This space-efficient probabilistic data structure helps each peers to propagate the information about the data blocks it stores. Given a data block identifier, the bloom filter is used to determine whether or not this identifier belongs to the set of identifiers from which the bloom filter has been formed. With a certain probability, depending on its size and on the size of the set, the bloom filter allows false positives [57]. It means that a peer examining a neighbor's bloom filter searching for missing data blocks could decide that this neighbor stores a data block, while it is actually missing. In order to minimize the probability of false positives, the size of the bloom filter needs to be increased. For example, allocating an

average size of 10 bits per element in the bloom filter provides approximatively 1% of false positives [57].

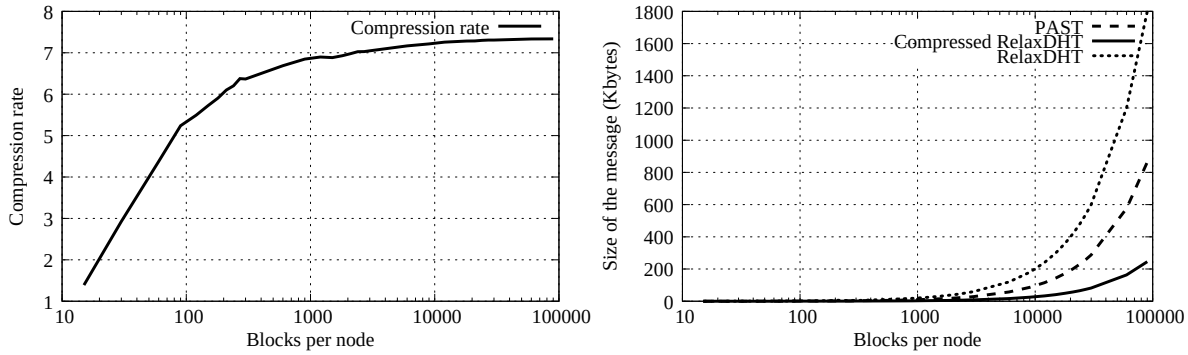


Figure 9.6: *(a) Compression rate of RelaxDHT maintenance messages in function of the number of blocks per node, (b) Compressed RelaxDHT maintenance message size compared to PAST maintenance message size using bloom filters with 1% false positive rate (9.6 bits per element).*

RelaxDHT is unable to use bloom filters for traffic optimization, because its maintenance messages have a well defined structure. Each data block key is associated a set of its replica-peers, whereas bloom filters are only capable of providing information about the belonging of a key to a key-set. Therefore, RelaxDHT uses non-probabilistic lossless compression, such as dictionary coders [206]. This kind of compression is efficient for maintenance message compression, because they contain many redundant digit sequences. This is due to two main factors: 1) All peer identifiers and data block keys are close in the DHT ring because they are located in the same leafset. It means that digital distance between two identifiers is low and decreases as the number of blocks/peers increases. 2) All the replica-peers are located inside the middle of the block's root leafset. Therefore, the sets of peer identifiers associated with different STORE and NEW ROOT elements are likely to be close to each other. This phenomenon also increases with the number of blocs.

Thanks to the two factors, the compression rate of messages increases with the load of the DHT. Figure 9.6.a recapitulates the results obtained while compressing maintenance messages generated by the simulation. Assuming that a standard DHT node is able to allocate at least 10Gbytes of local storage space, it is able to store more than 1000 blocks of 10Mbytes each. Therefore, it is reasonable to suppose that a compressed message is commonly 6 to 8 times smaller than the original one[8].

We compared the cost of both maintenance protocols, using 1% acceptable false-positive bloom filter rate for PAST. Figure 9.6.b shows that uncompressed RelaxDHT messages are more voluminous than PAST's ones. However, if compression is activated, RelaxDHT messages are 3 to 4 times smaller than PAST's ones. Furthermore, it is important to notice that in absence of churn (*i.e.*, when the leafset does not change between two maintenance times), a RelaxDHT root node may replace a regular maintenance message by a simple ping to

[8]We used the gzip software to compress messages [207].

perform maintenance. Therefore, taking this optimization into account, the average maintenance message size should be very limited. In other words, RelaxDHT maintenance protocol is more efficient than the PAST's one because: 1) in absence of churn, the maintenance is almost negligible: the protocol is able to adapt itself to the churn rate; 2) in presence of churn, its messages are less voluminous than PAST ones; 3) there is no information loss (no false positives as in PAST).

Regardless of the data compression efficiency, the size of the RelaxDHT maintenance messages linearly grows with the overall number of data blocks stored in the DHT (cf. logarithmic-scaled figure 9.6.b). However, in order to provide an acceptable rate of false positives induced by the bloom filters, PAST maintenance algorithm also should gradually increase the size of the its bloom filters as the number of stored blocks grows. It means that PAST maintenance message size also linearly grows with the number of stored blocks.

Finally, the linear growth of maintenance message size is not a problem for RelaxDHT. Indeed, consider that 1,000,000 blocks stored on a 100-peer RelaxDHT with replica rate of 3 induce 70Kbytes of data per maintenance message (see Figure 9.6.b). Knowing that these messages are sent by a peer with a sparse interval (the default PAST value is 10 minutes) and only to peers located "in the middle" of its leafset (*i.e.*, its 16 closest neighbors in our simulations), this size may be considered as negligible compared to data block transfer.

9.6 Conclusion

Distributed Hash Tables provide an efficient, scalable and easy-to-use storage system. However, existing solutions loose many data in presence of a high churn rate or are not really scalable in terms of stored data block number. We have identified one of the reasons why they do not tolerate high churn rate: they impose strict placement constraints that induce unnecessary data transfers.

In this chapter, we propose a new replication strategy, RelaxDHT that relaxes the placement constraints: it relies on metadata (replica-peers/data identifiers) to allow a more flexible location of data block copies within leafsets. Thanks to this design, RelaxDHT entails fewer data transfers than classical leafset-based replication mechanisms. Furthermore, as data block copies are shuffled among a larger peer set, peer contents are less correlated. This benefits to RelaxDHT because in case of failure, more data sources are available for the download of a missing block, which makes the recovery more efficient and thus the system more churn-resilient. Our simulations, comparing the PAST system to ours, confirm that RelaxDHT 1) induces less data block transfers, 2) faster recovers lost data block copies and 3) looses less data blocks. Furthermore, we show that the churn-resilience does not involve a prohibitive maintenance overhead.

Part III

Appendix II: Jekyll and Hyde

Chapter 10

High performance key-value store for mixed workloads

Contents

10.1 Introduction

In this chapter we describe an asymmetric cluster architecture for key-value stores. The architecture is driven by the combination of two hardware trends - big memory and micro-clusters - and by the increasing need to support a mix of transactional and scan-based workloads on a single instance of the store. We explore this architecture with a clean-slate design of an in-memory cluster key-value store: the Jekyll and Hyde store. The aim is to exploit heterogeneity and asymmetry in cluster hardware to support mixed workloads more effciently than homogeneous hardware. Specifically the aim is to effciently execute multi-key random access operations with basic transactional support, while also providing high throughput for scan-based analytical workloads operating on the key-value pairs.

The first hardware trend motivating this approach is the current rapidly declining cost of memory DIMMs combined with the current generation of servers supporting large memory sizes. As we will show, it is now financially feasible to put 100s of GB of DRAM on a commodity server. The second trend is the emerging popularity of high-density microservers that package a modest number of low-power processors and a relatively small amount of RAM into a small form factor. Using custom interconnects these are then scaled out into "microclusters" with 10s to 100s of processors within a single 1 or 2U unit.

These two trends lead to two different server architectures. The first is a scale-up solution where the entire data set can be held in the memory of a single server. This is ideal for operations that are difficult to partition and run on a scale-out system (e.g. OLTP transactions) and hence are more efficient to execute on a single, centralized, in-memory copy of the key-value store. However running large scan-dominated analytics on such a centralized store is suboptimal for two reasons. First, the scans are relatively slow due to the large size of the memory relative to the memory bandwidth of a single server. Second, the scans interfere with the transactional workload by destroying cache locality and competing for memory bandwidth. Many scans, on the other hand, parallelize easily and hence will run efficiently on a scaleout cluster such as a microcluster.

Unfortunately many real-world workloads are mixed, *i.e.,* have both transactional and analytic portions. In partic-ular there is an increasing need for "real-time analytics", *i.e.,* analytics that run on the latest (or almost-latest) transactional data and hence cannot use traditional warehousing solutions. Instead there is a trend towards running the entire mixed workload on the same in-memory data [106, 122]. However this introduces a tension when designing the architecture of the system. A centralized big-memory ("scale-up") solution will give the best performance for transactions and random-access queries. However, a decentralized ("scaleout") solution will give higher scan throughput since it can effectively aggregate the memory bandwidth of a large number of servers. Microservers will make the scale-out solution even more competitive on space, power, and cost.

We address this tension by rethinking the system architecture from the ground up to accommodate heterogeneity. Traditionally a compute cluster is composed of a single type of server, with the same CPU and memory footprint. We have built a cluster that is asymmetric in terms of both memory and processor. There are a few servers with a modern state-of-the-art high-end CPU and a large amount of DRAM. The other servers in the cluster have a lower computational power processor and orders of magnitude less DRAM. In our current prototype they are traditional rackmount servers rather than microservers; however

they are representative of microservers in having much less memory and processing power individually than the big servers.

In its simplest form the Jekyll and Hyde store consists of two *heterogeneous* replicas of the same key-value store maintained in this asymmetric cluster. Both replicas are in memory, but the first replica (the Jekyll) is stored entirely in the memory of a single large server. The other replica is partitioned across several smaller servers (the Hyde). Versioning is used to support both serializability and snapshot isolation for transactions, and snapshot semantics for readonly scans. In theory either replica can support any query. However for efficiency, point queries and transactions are directed to the Jekyll and scans are handled by the Hyde. The experimental evaluation of the Jekyll and Hyde store demonstrates the benefits of implementing the key-value store on top of an heterogeneous cluster.

10.2 Background

In this section we provide the background on the two hardware trends that we are exploiting when designing the cluster: memory prices and high-density micro servers. We then describe a workload that is typical of the type of mixed workload we wish to support with Jekyll and Hyde.

10.2.1 Memory

The price of DRAM is dropping as it benefits from Moore's Law plus the increased consumer demand from servers for larger memory DIMM form factors. Up until recently the cost of DRAM meant that provisioning a server with 0.25+TB of DRAM was expensive enough that there needed to be considerable performance benefit to justify the expenditure. However, currently a 16GB ECC DIMM can cost as little as US$220[1]. We should also stress that this is the retail street price for a large commodity vendor, and does not include bulk or preferred-vendor discounts.

Based on this memory module, provisioning a server with 192GB - commonly supported on motherboards supporting 12 or more DIMMs - costs approximately $2640. To put this into perspective, a low end commodity rack-based server is advertised at $6638[2]. Upgrading this server to 192GB would cost $2640, i.e. only 40% of the server price, or $5280 to upgrade to 384GB which is still less than the price of a single server.

10.2.2 High-density microservers

We use the term high-density microserver to refer to an entire class of server that is composed of lower power processors. There are different specific designs, but they all share some common features. They have small per-server form factors and support memory in the range of 216 GB. As the processors are low-power processors, they allow the servers to be physically packed at much higher density than traditional servers. This is because the

[1]Kingston 16GB 240-Pin DDR3 SDRAM ECC Registered DDR3 1066 Server Memory QR x4 w/TS Model KVR1066D3Q4R7S/16G from `http://www.newegg.com/` accessed on 6th January 2012

[2]HP ProLiant DL360 G7 X5650 1U Rack Server with 2 Xeon X5650s (2.66 GHz - 6 cores), 12 GB of RAM (max 384 GB) from `http://www.newegg.com/` accessed on 6th January 2012.

low power processors generate less heat and therefore require less cooling. The servers often have minimal storage etc and are often presented as basically CPUs with memory.

The idea of partitioning a key-value store over a Fast Array of Wimpy Nodes (FAWN) - a cluster of low-end, lowpower processors - to give higher performance per joule was introduced by Anderson et al. [33]. Today there are already commercial products that assemble microservers into "microclusters" in different ways.

The best example of a commercially available platform is the AMD SeaMicro[3] SM1000-64HD which is a 10U chassis with 384 dual-core 1.66 Ghz Intel Atom processors connected using a custom interconnect. The chassis contains 64 compute cards with each card containing 6 processors and 24GB of DRAM. The Dell PowerEdge C5125 Microserver[4] fits up to 12 servers in a 3U form factor. Each microserver can have up to 16 GB memory and supports a single 2- or 4-core AMD Phenom II or AMD Athlon II processors. Each server has a standard Ethernet NIC.

Similar ideas are being explored using low-power ARM processors. Calxeda[5] makes the EnergyCore ECX-1000 Series which provides quad ARM Cortex-A9 cores (which are 32-bit processors that use the ARMv7 instruction set). They have five 10 Gbps integrated communication channels with an integrated fabric crossbar switch. This allows them to fit four processors on a small circuit board (that can be fitted vertically in a 1U form factor). The HP Project Moonshot [10] is building an example platform, the Redstone Development Server, which has 72 such cards for a total of 288 processors (1152 cores) in a 4U form factor. Using 32- bit technology in the data center is limiting, but ARM has released the spec for the ARMv8 64-bit server class cores and, as of 26th April 2012, AppliedMicro has successfully demonstrated the X-Gene which is a server-on-a-chip using multiple ARMv8 64-bit cores.

Some companies are pushing these ideas even further to build systems-on-a-chip (SoCs). The Intel SCC is a single chip that connects 24 dual-core tiles in a 2-D mesh topology, with no cache-coherent shared memory across the tiles but instead explicit point-to-point message-passing supported by the hardware.

The common theme of all these architectures is that they offer a large number of low-end, low-power processors with a relatively high latency of communication across nodes and no hardware support for shared memory. Thus they are ideally suited for running data-parallel operators such as scans over a shared-nothing partitioning. For such workloads, they promise greater throughput per dollar, per joule, and per unit of rack space. The aim of Jekyll and Hyde is to exploit these advantages when running a mixed transactional and scan workload.

10.2.3 Amadeus Workload

In order to motivate the design of Jekyll and Hyde key-value store we describe an example workload that is based on the Amadeus workload [190]. For completeness we summarize the Amadeus workload as described by Unterbrunner et al. [190].

[3] `http://seamicro.com`

[4] `http://i.dell.com/sites/content/shared-content/data-sheets/en/Documents/poweredge-c5125-spec-sheet.pdf`

[5] `http://calxeda.com/`

Amadeus is a service provider for managing travel-related bookings. It provides a service called the Global Distribution System (GDS) which is an electronic marketplace for the travel industry used by the airline carriers and travel agencies. The GDS database contains millions of flight bookings stored as key-value pairs. Keys are unique and the values are denormalized binary objects of a few kilobytes. For the bookings kept online, this results in a single table of several hundred gigabytes in size. This BLOB table currently sustains a workload of several hundred updates and several thousand key-value look-ups per second. The largest data set is the denormalization of flight bookings which contains one record for every ticket which can contain hundreds of millions of records, each 350 bytes in length. The mean write rate is a few hundred updated per second, but the peak can be many times higher. A typical query might be: *give the number of first class passengers in a wheelchair, who depart from Tokyo to a destination in the US tomorrow*.

Unterbrunner et al. [190] observe that many services are dependent on querying this data, and that both average query and update rate are increasing, to the extent that it is no longer feasible to materialize views for each of the queries. They also observe that "key-value access is sufficient for all transactional workloads faced by the system, but is illsuited to real-time, decision-support queries that usually select on non-key attributes. These decision-support queries to support real-time operational decisions are becoming increasingly common and have tight latency constraints." All queries must be answered within 2 s regardless of their selection predicates and all updates made visible within 2 s. They observe that the average update load is 1 GB/s with a peak of 20 GB/s. Peak load needs to be sustained for at least 30 s. Hence Unterbrunner et al. [190] propose a solution based on shared-nothing data partitioning, and continuous circular scans of each partition or segment. This gives high parallelism and hence throughput while bounding latency. Scan throughput and hence analytic query performance will scale with the aggregate memory throughput of the system. However transactional semantics are limited to a single scan segment (partition) and read/write transactions cannot span segments.

We observe that such a design is ideally suited to running scan workloads on a scale-out hardware architecture based on clusters, microclusters, or systems-on-a-chip. However, it does not suffice for running mixed transactional/scan workload. This is the goal of the Jekyll and Hyde architecture.

10.3 Design

At a high level, our goal is to design a key-value store that can efficiently support a mixed transactional/scan workload over a data set such as the Amadeus store. When considering implementing such a key-value store today there are two obvious choices: *centralized* and *distributed*. The centralized approach exploits the memory trends we have described. A single large-memory server can be configured to hold all the key-value pairs in memory. On today's servers, storing hundreds of millions of 350 byte records in memory is feasible, and cheaper than building a cluster. This configuration has the benefits that multi-key operations (such as transactions) are highly efficient: the DRAM and large caches on the server provide great random access performance, and a single multicore server with cache coherency and fast inter-core interconnects is also efficient for operations such as locking or compare-and-swap which are required for transactional support.

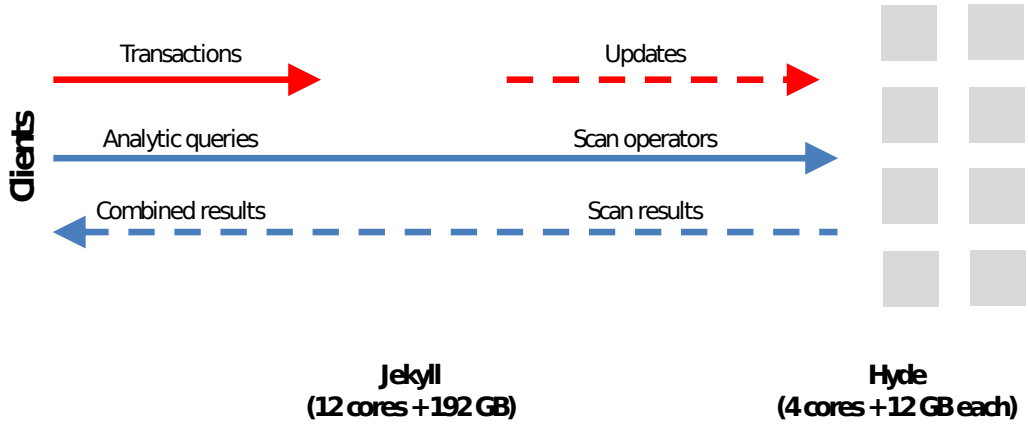


Figure 10.1: *Jekyll and Hyde architecture.*

However, the key challenge with such a configuration is trading-off the performance of the transaction workload for the decision-support workload. The decision-support workload touches a large number of cache lines, and therefore, destroys locality. Processor caches therefore provide less benefit. The use of CPU and memory bandwidth resources impacts the throughput of the foreground transactional workload. Memory bandwidth in particular is now the critical resource for data processing, a fact that was recognized more than a decade ago [52] but is ever more relevant as memory capacities and core counts continue to increase. The other design point uses a cluster where the key-value pairs are sharded and each server manages a single shard. This scale-out approach has been widely adopted. It provides great support for exploiting parallelism to perform scans over the store. The key challenge for this approach is the overhead of implementing distributed transactions. The (multi-round) communication required between servers storing the shards touched during a transaction that touches several keys induces latency. Therefore, most efficient scalable key-value stores do not support transactions. A good design for a key-value store would achieve the high transactional throughput achievable for the single server design while also achieving the high scan throughput achieved by a cluster. The hybrid approach of Jekyll and Hyde is aimed at precisely this. The basic idea is to store two replicas of the key-value store. The replication is heterogeneous, i.e. the replicas are laid out differently to efficiently support different workloads.

Specifically, one replica of the key-value store is maintained in-memory within a single address space, and the other is sharded across a set of n servers, each storing approximately $1/n$th of the key-value store in memory. We refer to the centralized replica as the Jekyll and the sharded replica as the Hyde, one key-value store with two very different personalities. The Jekyll is optimized for transactional and random-access queries, and the Hyde for scans and decision-support workloads.

In the default configuration and when there are no failures, the Jekyll functions as a master and the Hyde as a set of slaves (we will cover operation under failures in Section

3.3). The master stores all key-value pairs, but the key-value pairs are uniformly partitioned across the slaves. Clients send requests to the master, which acts as a global coordinator. It executes all transactions, updates, and point queries locally. Analytic queries consist of a scan operator, which is executed at all slaves in parallel, and a "reduce" operator that runs at the master and combines the results from all the slaves. Figure 10.1 shows this basic architecture.

Updates are logged synchronously by the Jekyll to a local write-ahead log that is used for failure recovery. Updates are sent asynchronously to the slaves, thus the Hyde is always a consistent (but potentially stale) snapshot of the state whereas the Jekyll always reflects the latest updates. It is important to note that the replication even if asynchronous is continuous (i.e. not a batch ETL process), and hence the freshness of the Hyde is limited only by the network latency of replication.

Internally both master and slave are structured as a versioned, in-memory, key-value store. Each key maps to a list of values ordered by version. Each transaction has a globally unique version and versions are monotonically increasing. Stale versions are garbage-collected lazily. Versions are maintained by the master, i.e. the master also acts as a timestamp coordinator.

For simplicity, both Jekyll and Hydes use the same internal layout. It offers efficient random-access operations by key as well as sequential scanning of values. Keys are stored in a concurrent dictionary data structure and values as inmemory byte arrays. Our current implementation of scans is a read/write elevator scan [190]. The data are partitioned into mutually exclusive scan segments, and each segment is continuously scanned by a dedicated thread. Queries can enter the scan at any point and they exit when they are completed. Multiple queries thus run concurrently within a single segment, and also each query is parallelized across the segments. Each scan thread reads every key-value pair in its segment (there are no indexes) and passes it to all the currently active queries before processing the next key-value pair: this maximizes cache locality and hence throughput.

We note that several variations are possible on this basic design. For example, the Jekyll could run part of the scan workload when the Hydes are overloaded. Alternatively, the Hydes could run point queries, or even transactions using distributed transactions. Transactions could be snapshot isolation rather than serializable. Replication could be synchronous rather than asynchronous (this has implications for fault tolerance that are discussed later in this section). We have not evaluated all these options: however we note that they all share the same basic hybrid design. The evaluation in this chapter is based on the default or base configuration described above.

10.3.1 Operators

The lowest-level abstraction offered by Jekyll and Hyde is a versioned key-value store that supports the operators shown below. The first three can be executed on either the master or the slaves but the last two are only executed on the master by default:

- `put(key, value)` atomically inserts a key-value pair. The value is assigned a new global version number and if the key is not present a new entry is created. The value is appended to the list of values associated with the key. The version assigned to the value is returned.

- `get(key)` retrieves the value with the highest version number for the key.
- `get(key, version)` returns the value associated with a key. The version returned will be the value with largest version number that is less than or equal to the specified version.
- `cas(readset, writeset)` is a multi-key compare-andswap operation. The readset is composed of a set of keys each with a version number and the writeset is composed of a set of keys and values. The same key can appear in the readset and writeset. The operation checks that the version number specified for each key in the readset is the highest version number. If it is not then the operation fails. If the readset checks pass then the writes are performed. The entire operation is preformed atomically.
- `version()` returns a version greater than or equal to that of the the last write operation globally.

These basic operations are then wrapped to allow us to implement simple transactions on the master.

- $transaction_{begin}()$ starts a transaction which retrieves the current version number, which is used for all read operations thereby providing a snapshot of the keyvalue store. It also creates empty read and write sets.
- $transaction_{write}(key, value)$ stores the key and value in the write set.
- $transaction_{read}(key)$ checks the write set to see if it contains the key, and if so returns the associated value. If it is not present then the `get(key, version)` function is used to retrieve the value associated the key. The read set is augmented with the key and version of the value read.
- $transaction_{abort}()$ Flushes the read and write set and all further operations performed are dropped.
- $transaction_{commit}()$ uses the CAS operation with the generated read and write set. If the CAS operation succeeds then the transaction has successfully completed. If not, then the function calling this operation should handle the failure. The readset contains all the keys read or written by the transaction (if we need serializability) or just the keys written (if using snapshot isolation).

These operations provide basic support for the key-value store for supporting the transactional workload. To support scan-based queries we provide a scan operation:

- `scan(result, MapAndCombine(), Reduce())` The scan operator is a generic operator which allows for scan operations over the values. `result` is a data structure which stores the intermediate and final result. The `MapAndCombine` function takes a key-value pair, extracts the relevant information from the value (the "map"), and then integrates this into a result object ("combine"). The `Reduce` function takes two or more result objects and produces a single result object. Intuitively, the `MapAndCombine` function is applied to all the key-value pairs. When the data is sharded then the results of each shard can be combined using one or more application of the `Reduce` function.

The scan operates over a snapshot of the data in order to ensure that it sees a consistent version of the key-value store. When the scan is started it reads the current version number on the master, and all read operations on each key are performed on the snapshot corresponding to this version. Hence, the scan operation can be performed concurrently with updates and these will not be visible to the scan operator. Scans are not supported within transactions and are read-only: we do not support key-value updates in the `MapAndCombine` or `Reduce` functions. The aim is to support decision-support queries, which do not need updates or transactional support, without incurring any extra overheads for supporting updates or transactions. Scans are supported on both the master and the slaves but typically only execute on slaves.

Finally, we implement higher-level application functionality as stored procedures. A user provides a set of functions that they would like to execute, and these functions perform multiple operations. For the transactional workload each transaction is a stored procedure. For scan workloads the mappers and reducers are stored procedures.

10.3.2 Implementation

The Jekyll and Hyde replicas use the same core in-memory data structures to store the key-value pairs for both the Jekyll and Hyde replicas. The store treats all keys and values as byte arrays. We use three data structures. The first is a small data structure for short-lived fine-grained locks that is designed to be small enough to be kept in the processor cache. The second data structure is an array of linked lists of value-version pairs. The final data structure is a concurrent hash table that maps keys to indexes into this array. A background process periodically removes stale and deleted value-version pairs from these linked lists. Currently all the data structures are written in C# and we rely on the .NET runtime for memory allocation, garbage collection, and defragmentation.

We also implemented a stored procedure like functionality that allows a service using the store to embed higherlevel operations in the key-value store. This removes the overheads of round-trip times, makes failure handling easier, and reduces the average duration of a transaction. We expect that each transaction will execute in a single stored procedure on the Jekyll, and each analytic query will use a `MapAndCombine` stored procedure on the Hyde servers and a `Reduce` on the Jekyll.

The implementation of the operators running entirely locally on the master, i.e. transactions, is straight-forward. The scan operator is slightly more complex. It runs locally against each shard, using the `MapAndCombine` function against each relevant value. Within each shard we use a simple collaborative elevator scan over the data structure storing the values. Each scan operation is associated with a version number v, and the value passed to the `MapAndCombine` function for each key is the one with the highest version that is less than or equal to v. Finally, the results generated by each shard are sent to a final server to the `Reduce` function to be applied. In the examples we have been looking at the final result per server is small and therefore we send the result to the server hosting the Jekyll to perform the final result aggregation. It would also be feasible to send all the results to any server in the cluster or, indeed, to partition the results per server and have multiple servers perform aggregation of parts of the results.

10.3.3 Handling failure and recovery

There are several different failures that the Jekyll and Hyde store must handle. We first describe the default fault tolerance configuration for our prototype, and then some alternatives which give different tradeoffs between availability and performance.

We have already mentioned that Jekyll and Hyde logs write operations. In fact, any write operation will not return to the client until the Jekyll has written the operation to disk, *i.e.*, the logging is synchronous on the Jekyll. Each Hyde server also keeps a write log. These logs allow the system to be halted and restarted in a consistent state and guarantees durability, *i.e.*, committed updates are never lost.

In server deployments today, availability is as important as durability. Jekyll and Hyde maintains availability when one or more Hyde servers fails. The load for the failed Hyde servers is simply shifted to the Jekyll, which holds the "master copy" of the data and hence can answer all queries. Of course, this will impact performance until the Hydes can be recovered, but availability will be maintained.

Failure of the Jekyll is a more challenging scenario. Since by default replication from the Jekyll to the Hydes is asynchronous, the Hydes hold a consistent snapshot but this snapshot does not necessarily reflect the most recent committed updates. Thus any further transactional activity must wait for the Jekyll to be recovered from the write log.

Synchronous replication from Jekyll to Hyde would avoid this problem by making the Hyde be an up-to-date replica. Availability can now be maintained during a Jekyll failure by electing one of the Hyde servers as the new master (*i.e.*, Jekyll). However since the new master will be underpowered compared to the original Jekyll, will not hold the entire data set, and will have interference from scan workloads, there will be a performance penalty until the Jekyll can be recovered.

A third option is to maintain two Jekyll replicas synchronously, and have synchronous or asynchronous replication to the Hyde. This allows transactional workloads to be isolated from scan workloads even when the Jekyll fails.

We are currently evaluating the tradeoffs for these different options, which impact availability as well as failure recovery. Given (possibly stale) in-memory replicas as well as on-disk logs, these could be combined in different ways to recreate the state of a failed server. Recovering from inmemory replicas over the network can potentially be much faster, especially with 10 Gbps networks and if the recovery be parallelized over many servers. On the other hand, recovery from local disk or SSD while potentially slower reduces the load on the shared network resource.

10.3.4 Discussion

The current Jekyll and Hyde prototype represents one point in the broader design space for heterogeneous replication. The prototype design could be generalized along different axes. We have already discussed the different options for distributing queries between the Jekyll and the Hyde (e.g. off-loading scans activity to the Jekyll from overloaded Hyde servers). We have also discussed how replication and logging can be configured to give different tradeoffs between performance and availability.

A third axis is *layout heterogeneity*. For simplicity, we use the same in-memory layout for both the Jekyll and the Hyde implementation. We could also use different data structures, for example a record or object based layout on the Jekyll but a column-based layout on the Hyde.

Finally, the current implementation assumes a flat network topology, i.e. Jekyll and Hydes are all connected to a single crossbar switch. However with micro-servers it is possible to have a different topology, e.g. the 3-D torus used in Camdoop [81] and the SeaMicro SM1000-64HD. This has implications for both data layout and the execution of analytic queries (e.g. by doing in-path aggregation), however a detailed discussion of these issues is beyond the scope of this work.

10.4 Evaluation

Next we evaluate the performance of Jekyll and Hyde Key- Value store using a prototype asymmetric cluster.

10.4.1 Hardware

We use an asymmetric heterogeneous cluster architecture, where the master (Jekyll) replica is stored on a high-end large memory server. The server has two Xeon X5650 6-core 2.66 GHz processors with 192GB of DRAM on 6 DRAM channels in two NUMA domains. It also has an Intel SSD to store the recovery log. It has a 10 Gbps NIC connected to a 10 Gbps switch.

The Hyde replica is sharded over a set of slaves with lower specification processors (a single quad-core Intel Xeon E5520 2.27 GHz) and only 12GB of DRAM per server. In our base configuration we use 20 such servers. Each Hyde server has a standard hard disk for the local recovery log and is connected via a 1 Gbps NIC to the same switch as the Jekyll server.

Ideally, we would have liked to have used microservers to implement the Hyde replica, but they were not available when we first started to build this cluster. However the Hydes use a a processor that was released in Q1 2009, and is therefore over 3 years old. Therefore, the 21 server cluster that we used is asymmetric in terms of processor performance and memory per server, exploiting the two hardware trends described earlier.

10.4.2 Software configuration

All the servers run the C# based implementation described in Section 3.2. To make measurements more repeatable, in the experimental evaluation we disable the garbage collector thread as it runs infrequently. In the experimental configuration we enable synchronous logging to the local recovery log on the Jekyll as well as the Hydes, and asynchronous replication from the Jekyll to the Hydes.

10.4.3 Experimental setup

We used two base configurations to compare performance, one we call *centralized* and the other we call *Jekyll and Hyde*. In the centralized configuration we just use a single replica

stored on the single 192GB server to service the entire workload. In the distributed version we maintain the two replicas and partition the workload over the replicas.

In all the experiments, unless otherwise stated, we used key-value pairs consisting of 4-byte keys and 350-byte values. We preload 230 million key-value pairs representing 75.8 GB of raw data. All experimental results reported are the median of three runs unless otherwise stated. Each experimental run lasts for 5 min.

The experimental workload consists of a transactional and a scan workload. The transactional workload is implemented using a stored procedure that executes transactions in a loop. The stored procedure is then executed concurrently by multiple threads. Each transaction is a single operation (either a `get` or a `put`) on a randomly chosen key. The workload is parametrized by the read/write ratio required. The workload avoids read/write conflicts to maximize transactional throughput: writes always insert a new key-value pair and reads always access one of the pre-loaded key-value pairs. The default read ratio used is 0.95 (95% reads); with this ratio approximately 7GB of new key-value pairs are added during a 5 min run.

Unless otherwise stated, the scan workloads use a `MapAndCombine` function that updates a single integer stored in the result object. The computational overhead of this is minimal and the performance is dominated by the cost of scanning the in-memory data. The `Reduce` function simply sums the integers across all the result objects.

Scans are run with both operator-level paarellelism (multiple scans are run concurrently) and data parallelism (there are multiple concurrent scan threads, each responsible for a disjoint subset of the data). Each scan thread continuously iterates over the key-value pairs in the range assigned to it, passing each key-value pair to each scan operator in turn.

Thus the scan workload has two tuning parameters: the number of concurrent scan operations kscan and the number of scan threads t_{scan}. We have empirically found that $k_{scan} = 6$ gives the best throughput and so we use this value throughput. For the centralized configuration, t_{scan} is a tuning knob that affects the tradeoff between transactional and scan throughput. By default we set tscan = 6, but we vary it in different experiments to trade off scan throughput against the impact on the transactional workload. For Jekyll and Hyde, we use $t_{scan} = 12$, but this parameter has no effect since the scan threads on the Jekyll are almost always blocked, with the actual scan work done on the Hydes. On each of the Hydes we use 8 scan threads, one per logical processor (hyperthread) as the Hydes each have a single quad-core hyper-threaded processor.

For both configurations, we allocate 6 dedicated threads for the transactional workload on the centralized/Jekyll server. However we will see that in the centralized case the actual amount of resources available to the transactional workload, and hence its performance, depends on t_{scan}.

During each experiment the primary metric gathered for the transactional workload is the number of transactions successfully performed per second. For the scan workload the primary metric is the number of times a `MapAndCombine` function was invoked per second. Note that each invocation of `MapAndCombine` happens on a single key-value pair. Under normal operation each key-value pair retrieved by a scan thread will cause at most k_{scan} invocations of `MapAndCombine`. In general we report transactional performance in Mops/s (millions of transactions per second) and scan performance in Mkeys/s (millions of keys passed to a `MapAndCombine` function per second).

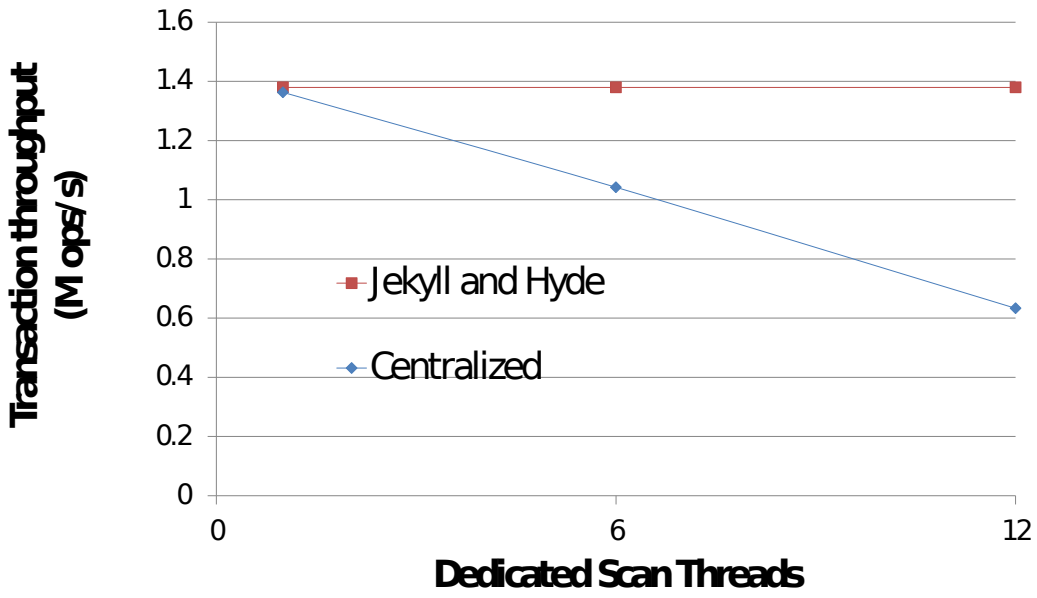


Figure 10.2: *Transactional throughput as a function of scan threads.*

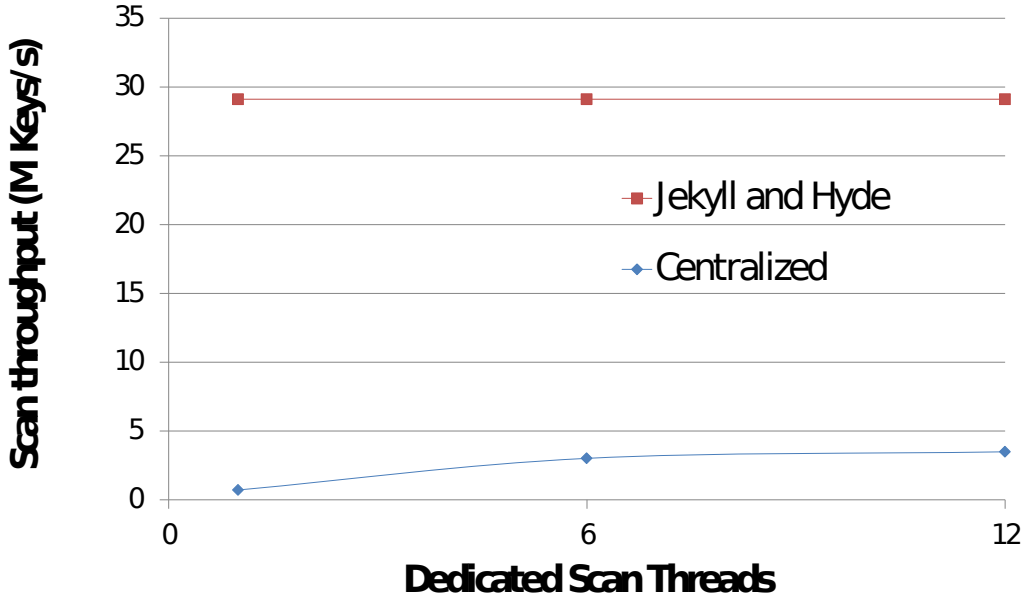


Figure 10.3: *Scan throughput as a function of scan threads.*

10.4.4 Effect of mixed workload

The benefit of using Jekyll and Hyde is the ability to handle mixed transactional/scan workloads and the first set of experiments demonstrates the performance increase achieved.

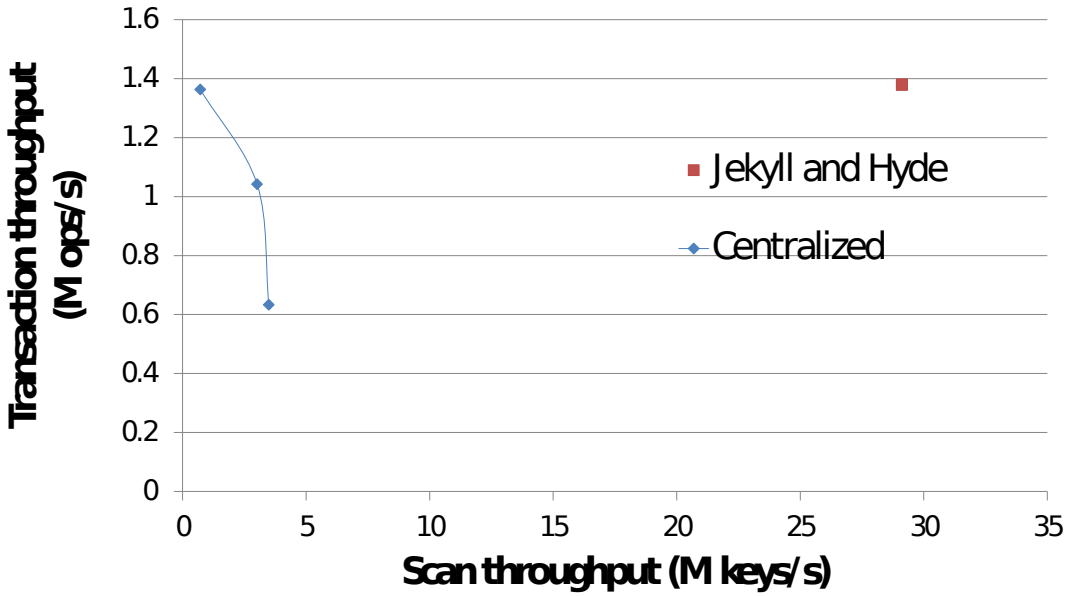


Figure 10.4: *Tradeoff between scan and transactional throughput.*

Figures 10.2, 10.3 and 10.4 show the effect of running scan and transactional workloads on the same server in the centralized configuration, compared to separating them in the Jekyll and Hyde configuration. For these experiments we fix the read/write ratio at 5% writes and the number of dedicated transaction threads $t_{transaction}$ at 6, and vary the number of dedicated scan threads t_{scan} on the centralized server. A higher t_{scan} gives more resources to the scan workload and correspondingly fewer to the transactional workload. For Jekyll and Hyde the two workloads run on different servers. Hence the parameter tscan is irrelevant for Jekyll and Hyde: the performance is the same for all values of t_{scan}.

Figure 10.2 shows the transactional performance. We see that centralized shows a linear drop in transactional performance as more resources are given to the scan workload. Jekyll and Hyde on the other hand always affects the full transactional performance as scans do not compete for the same resources.

Figure 10.3 shows the scan performance. This initially increases with the amount of resources assigned to the scan but soon flattens out as it reaches the limit of performance achievable on a single machine. For Jekyll and Hyde the scan performance is constant and much higher, showing the much higher aggregate memory bandwidth across the 20 Hyde servers.

Figure 10.4 shows these same results as a tradeoff between scan and transactional performance. We see clearly that the centralized configuration must pay a significant price in lost transactional performance as soon as it starts running any scan workloads. In contrast the Jekyll and Hyde keeps the two workloads isolated and achieves high performance for both.

10.4.5 Effect of read/write ratio

When running with Jekyll and Hyde there is the overhead of maintaining two replicas compared to the centralized version, because of the need to propagate updates from the Jekyll to the Hyde. We would expect this to appear as reduced transactional throughput for Jekyll and Hyde and for the impact to increase as the update ratio of the transactional workload increases. To measure this, in the next set of experiments, we show the performance of both configuration while varying the update ratio.

We compared the two configurations with the transactional workload only, i.e. with the scan workload disabled. We varied the ratio of updates from 0% to 100%, ran each configuration for 5 min, and measured the number of read and write transactions performed per second.

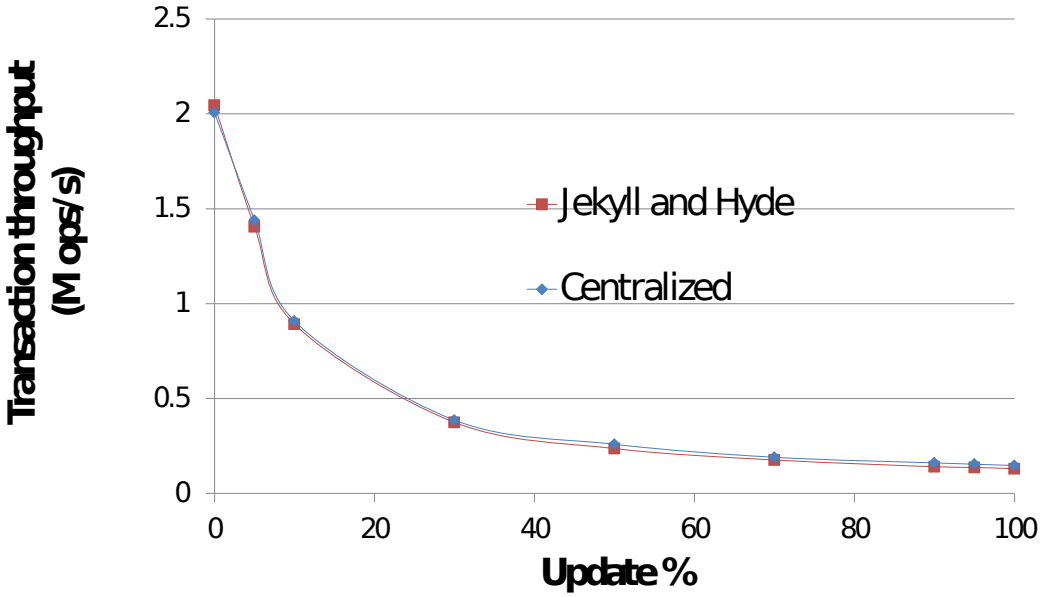


Figure 10.5: *Overall transaction throughput as a function of update ratio.*

Figure 10.5 shows the total (read and write) transactions per second as the read write ratio is varied for both the centralized and Jekyll and Hyde configurations. We see that both configurations achieve comparable throughput. We also see that both configurations' throughput drops significantly as the percentage of updates increases. Finally we see a very small difference in performance at extremely high update ratios.

In order to understand this difference more, we examined the performance of the read and write operations individually. Figure 10.6 shows the read throughput and Figure 10.7 shows the write throughput as the read write ratio is varied. From Figure 10.6 we can see that there is no difference between running the two configurations for the read throughput. This is due to the fact that all read operations are being serviced locally from the Jekyll and requires no communication with the Hydes.

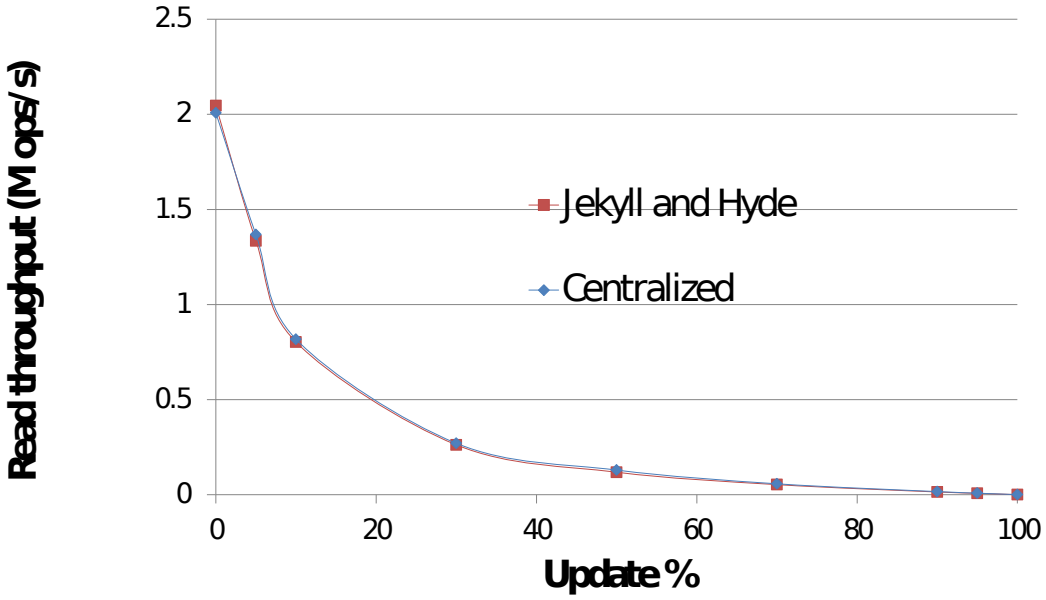


Figure 10.6: *Read transaction throughput as a function of update ratio.*

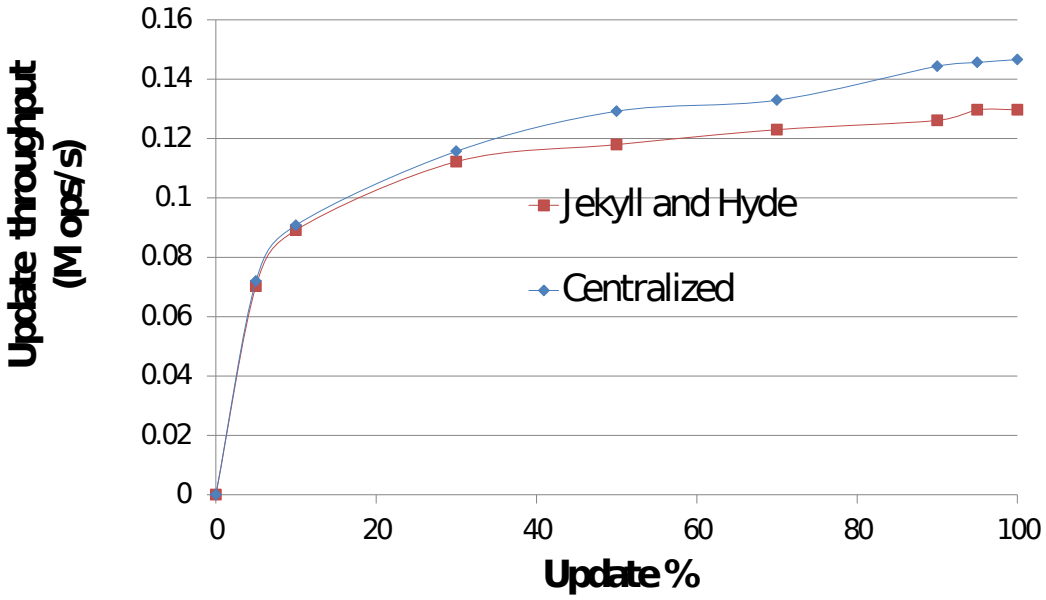


Figure 10.7: *Write transaction throughput as a function of update ratio.*

However, from Figure 10.6 we can see that as the ratio of writes increases the performance tails off because of the higher overhead of performing writes. Writes are logged to the local SSD but in the Jekyll and Hyde case they are also asynchronously sent to the Hyde.

Thus the degradation in performance as update ratio increases is slightly more pronounced for Jekyll and Hyde, with approximately 20% less throughput than centralized when the workload is 100% writes. When a write operation is performed the replica being maintained by the Hydes needs to be also updated which requires the information to be marshaled. On each write operation a single Hyde server needs to be informed. Since writes are synchronously logged on the master in both the centralized and Jekyll and Hyde configurations, they have the same level of failure resilience. We could in addition update the Hydes synchronously rather than asynchronously but this would decrease throughput further.

These set of results have demonstrated the performance degradation that is observed when using the Jekyll and Hyde over simply using the centralized configuration. What we have shown is that Jekyll and Hyde effectively isolates transactional workloads from scan workloads. It gives far higher scan performance than a centralized version, and equal transactional performance except at extremely high update ratios, when there is a modest overhead of up to 20% for a 100% write workload. The overhead at realistic update ratios is smaller: under 10% for 50% writes and under 2% for 10% writes. For comparison, the update ratio for TPC-C is 34% and for TPC-E is 9% [74].

10.4.6 Understanding scan throughput

Scale-out

Jekyll and Hyde allows us to get the maximum possible transactional throughput from a single large server, by moving the scan workload to the Hyde replica. In addition, it also allows easy scaling of the scan performance by scaling the number of servers in the Hyde replica. Figure 10.8 shows the impact on the scan workload of varying the number of Hyde servers. In the previous experiments we used 20 Hyde servers by default. Here we add servers up to a total of 27 (the size of our clusters). The results show that the performance scales with the number of servers used to partition the Hyde replica, as would be expected.

Processor architecture

A second advantage of Jekyll and Hyde is that the Hyde servers can not only be less powerful but in fact can use a different architecture from the Jekyll. In particular they could use ARM or Atom-based microservers which have the potential to be extremely competitive in terms of power, space, and cost per unit of computation. Since each individual processor or micro-server is much less powerful than a single Jekyll-type server, we would need a large number of such servers, and hence the workload needs to be highly partitionable and parallelizable: an ideal fit for shared-nothing scans.

Processor type	Clock speed	# processors	DRAM	Scan performance	Power	Cost[†]
Intel Xeon X5650	2.67 GHz	2	192GB	10.86Mkeys/s	190W	$1992
Intel Xeon E5520	2.27 GHz	1	12GB	2.91Mkeys/s	80W	$373
Intel Atom CPU D510	1.66 GHz	1	2GB	0.37Mkeys/s	13W	$63

[†] Cost of processors in US$, as of 7 June 2012 on ark.intel.com

Table 10.1: Scan performance of different processor architectures

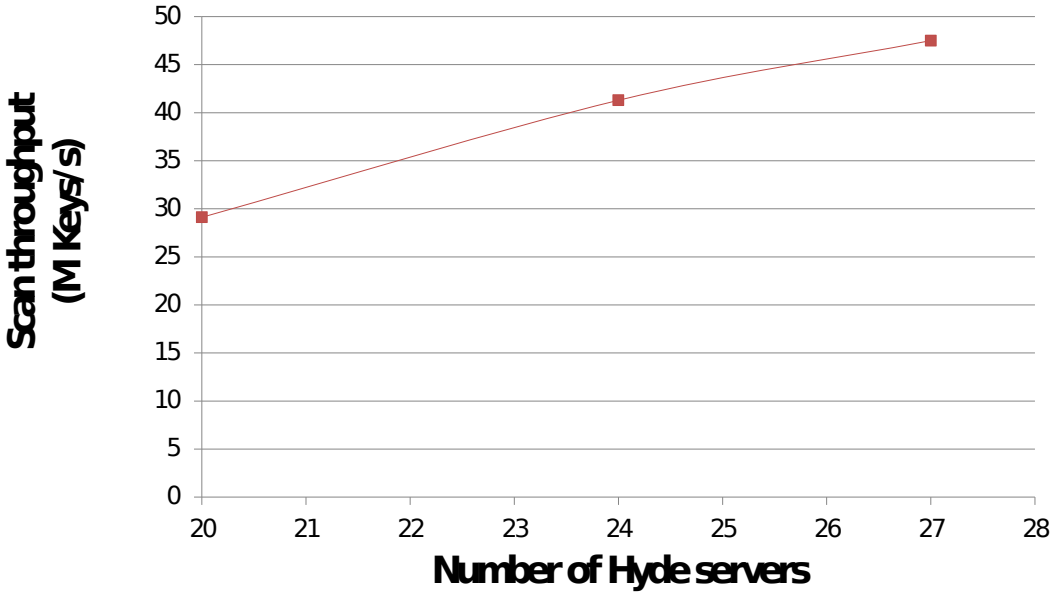


Figure 10.8: *Scan throughput versus number of Hyde servers.*

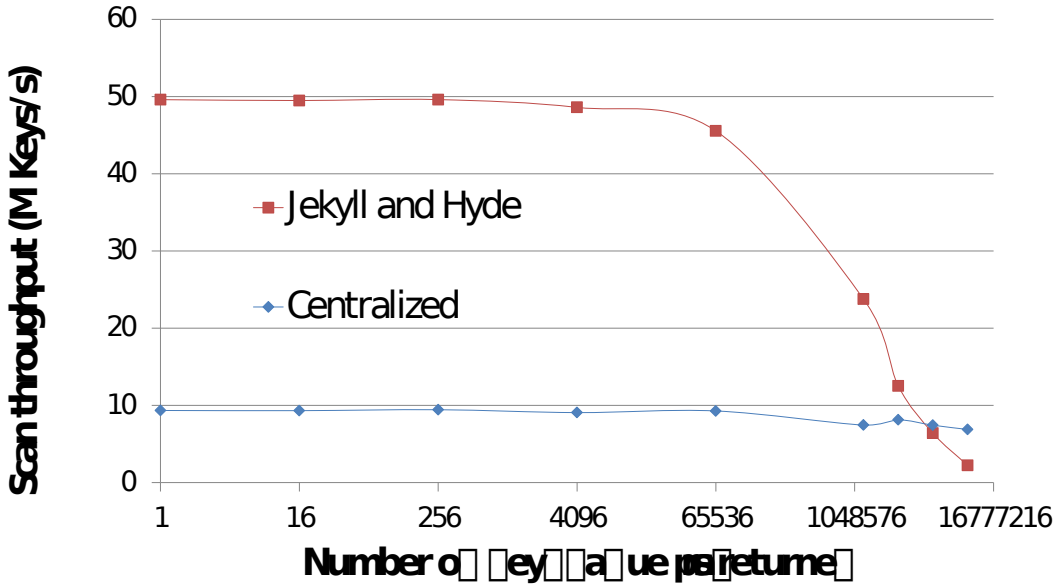


Figure 10.9: *Network effects for centralized and Jekyll and Hyde configurations.*

The best choice for a Hyde at any given point in time depends on the relative scan performance as well as other factors such as cost. We used a micro-benchmark to compare the raw scan throughput of the Intel Xeon X5650 2-socket machine (our current Jekyll) with that

of the Xeon E55220 1-socket machine (1 of our current 20 Hyde servers), as well as an Intel ATOM D150 2-core machine (representative of a low-power microserver). Because the memory sizes of the servers vary we configured the test to use only 1 million keyvalue pairs which fits in memory for all three servers. We configured each server to maximize throughput, and Table 10.1 shows the results. Currently the E5520s are most competitive in terms of performance per dollar, and hence the best option for the Hydes, but over the next 13 years we expect the microservers to improve in performance (e.g. as 64-bit ARM based designs come out) as well as reduce overall system cost by packing a large number of microservers into a limited amount of rack space.

Effects of selectivity

So far we have used a very simple scan that generates a single integer result from the `MapAndCombine` function. This means that the result object generated for each shard on the Hyde replica is only a single integer. We believed, based on the description of the Amadeus workload, that many of the decision-support queries produce responses that are small, *i.e.*, they are highly selective and/or aggregation queries. However, there is clearly a trade-off between performance and the size of the result. At the limit, if the results returned every key-value pair then this would require all the data to be transferred across the network and combined.

In order to understand this trade-off we created a greplike scan. We inserted 250 million key-value pairs with the value length being 350 bytes as before. In the first 4 bytes of the value we embedded a unique integer between 0 and 250 million. The scan returns all key-value where this integer is below a threshold parameter T. Thus by varying T we are able to effectively vary the selectivity of the query and hence the amount of data returned.

We configured the experiment to run three scan operations concurrently and varied the number of key-value pairs returned. Each key-value pair is approximately 350 bytes, hence returning 65536 key-value pairs, for example, corresponds to sending 23MB in aggregate. Figure 10.9 shows the scan throughput versus the number of key-value pairs returned for both the centralized and Jekyll and Hyde configurations. Note that the x-axis is on a log scale but the y-axis is linear. The maximum throughput achieved by Jekyll and Hyde is higher than in the previous experiments, as in this experiment we have no transactions running and hence no updates on the Hydes and the scans touch much less data in memory: only a single integer is examined for keys that are not selected, and the entire 350-byte value is touched only for keys that are selected.

The graph clearly shows the trade-off. At around 65536 key-value pairs (23MB) returned, Jekyll and Hyde starts showing the overhead of transferring the data over the network. At approximately 5 million key-value pairs (1.6 GB) the centralized version starts to outperform the Jekyll and Hyde implementation. In the centralized version the cost of performing the scan is largely independent of the amount of data returned.

Further optimization of scan workload

We believe that the queries that will be performed will usually be calculating statistics over the data, and therefore will generate small results (much smaller than 0.4 GB).

If this assumption does not hold, we could distribute the load of performing the aggregation and final generation of results from the Jekyll server to the Hyde servers. This can be achieved in a number of ways, from simply partitioning the result keys to aggressive in-network aggregation [81].

At the limit it may also be better to perform some of the scan operation on the Jekyll replica. We have been exploring the development of an optimizer that is able to do this. Although beyond the scope of this paper, the intuition is that normally optimization of an individual query is performed prior to execution of query. The challenge is that prior to execution we have little visibility into runtime operation of the scan. Unlike SQL queries our `MapAndCombine` functions are opaque C# functions and hence traditional techniques for estimating predicate selectivity etc. before running the query can not be applied. We have therefore been developing an online optimizer that monitors the performance of each scan on the Hyde replicas. It is able to monitor the size of the result object and the fraction of key-value pairs scanned. If we assume that each Hyde server stores a subset of the keys, r, the scan operator can be applied to this set of keys r either on the Hyde replica or on the Jekyll replica. At any point in execution the scan operator can be migrated from the Hyde replica to Jekyll. The result object for the two parts can be merged using the Reduce function. The optimizer therefore continuously estimates the likely network overhead of transferring the result, and migrates the execution to the Jekyll when it is advantageous.

10.5 Discussion

So far we have focused on the design and performance of the Jekyll and Hyde key-value store and how it leverages current hardware trends. We now consider three other very practical issues: *cost*, *energy* and *form factor*.

It is very hard with emerging technologies to estimate the likely cost points as many factors impact it, including things like market size. We have already shown that the cost of buying a server today with quarter of a terabyte of DRAM costs less than twice the price of buying a server with 48GB of DRAM (the change in cost is 16 versus 3 x 16GB DIMMs, which is $2860).

Estimating the cost of a micro server is much harder, as they are still in prototype form. We believe that in a year or so it will be feasible to buy a bare bones micro server with a processor, 16GB of DRAM and 16GB flash for less than $400, assuming that components like the power supply are shared by all micro servers in the rack. Twenty such servers would provide an aggregate memory footprint of 320GB of DRAM and persistent storage for less than $8000. We believe this Figures are realistic and, in fact, pessimistic. The FAWN paper [33] from 2009 states the price of a FAWN server with 2GB of DRAM is $150 at volume (column 1, page 13). At this price, the cost of 20 microservers would be $3000, which is comparable to the price of a single server today. As evidence of how cheap low-power nodes can be, we see ARM-based systems-on-a-chip retailing for under $50 in the client and device space. The recently announced Raspberry Pi, which used a Broadcom BCM2835 system-on-chip, which provides a low-end ARM11 core, a GPU and 1 Gbps NIC, plus 256MB of DRAM and retails for only $35. This is somewhat less RAM per core than we expect for a microserver, but on the other hand the price includes circuitry for connectors and functionality that would not

be required in a micro server. Similarly the VIA APC is a similar system with twice as much DRAM as the Pi and is expected to retail for $49[6].

The energy requirements are increasingly important, and are a proxy for likely processor density as well as likely cooling costs. The energy requirements per micro server are likely to be less than 10-15 W, meaning the entire cluster (the main server and the micro servers) will require less than 800 W, which can be provided by a single power supply. This is important as it impacts the cost of buying the device, as well as the operating expenses.

Finally, another issue is the form factor, and we estimate that such a cluster will be between 3-5U depending on density with which the micro servers can be packed. For example, the current SeaMicro servers have between 64 and 300+ processors in a 10U form factor.

We believe that in terms of energy, costs and form factor (which translate into capital and operating expenditure) a heterogeneous cluster would be cheaper than running a symmetric cluster to support this key-value store, as well as higher-performance than a single large server.

10.5.1 Architectural isolation

In the evaluation of Jekyll and Hyde we have focused on the performance isolation between two different workloads using two replicas stored on different server configurations.

However one of the benefits of heterogeneous replication is that it also achieves *architectural isolation*. The Jekyll and Hyde have a narrow, well-defined interface between them (Figure 10.1) and little visibility into or dependence on each other's internal architecture. The architecture specific knowledge is limited to an understanding of how keys are partitioned across the Hyde slaves. This gives several benefits. First, the Hyde can be provisioned and scaled differently from the Jekyll. This means not only that they can take advantage of different hardware trends (scale-up versus scaleout) but also that the overall system can easily be provisioned for any ratio of transactional to analytic load. Second, the Hyde can use a different processor architecture and even operating system, which allows us to mix, for example, Intel-based Jekylls with ARM-based Hydes. An improvement on one side (*e.g.*, adding support for hardware transactional memory support on the Jekyll) requires no changes on the other side. Finally, it allows the software and data layouts to be diffxerent optimized, e.g., by using compression and/or columnar layouts on the Hyde.

We can trivially replace the micro servers with another single server, or replace the large memory server with another set of micro servers.

10.6 Related Work

Jekyll and Hyde uses heterogeneous replication to achieve three key aims - in-memory data processing, centralized (scale-up) transactions, and scale-out for analytics. To our knowledge this is the first work to combine all three in a single store. However there is a large body of existing work that combines one or two of these three ideas. Here we briefly mention the most recent and relevant work, focusing mainly on in-memory stores and databases.

[6] `http://www.apc.io` accessed June 28, 2012.

10.6.1 Centralized stores

The need to support mixed workloads is widely recognized in the database community and often expressed as the need to run both OLTP and near-real time OLAP on the same database. While Jekyll and Hyde is a "noSQL" key-value store rather than a relational database, it addresses the underlying challenge that all such systems have, *i.e.*, that transactional and analytic workloads have different requirements and interfere with each other's performance.

HyPer [122] is a centralized in-memory DBMS designed for mixed workloads. Both OLTP and OLAP workloads run on the same physical machine but on different snapshots: the OLAP workload uses a periodically taken copy-on-write virtual memory snapshot of the OLTP database. While this provides a logical separation of the two workloads, they still compete for physical resources on a single centralized machine. HYRISE [106] automatically optimizes the in-memory layout for a given mixed workload and thus achieves better overall performance for a centralized configuration than an all-row or all-column design. Both designs are variants of the "centralized" configuration, with the corresponding disadvantages: interference between the workloads, lack of scale-out for analytics, and a single layout for both transactions and analytics. By separating the two workloads at the architectural level, Jekyll and Hyde avoids these problems.

Multimed [171] is an interesting approach which uses a share-nothing architecture within a single multicore machine, by running one replica on each core. A single replica functions as a master and handles the transactional load, and the remaining replicas apply updates asynchronously and handle read-only queries. Thus this resembles Jekyll and Hyde in separating the transactional and the read-only queries. However, the key difference is that in Jekyll and Hyde there is a single read-only replica (Hyde) that is partitioned across many slaves. This allows Hyde to scale out well beyond a single multicore machine and to take advantage of microcluster architectures. This in turn makes all the cores of the single large machine (Jekyll) available for transactions without competition from the scan workload.

10.6.2 Scale-out without transactions

In recent years there has been an explosion in scale-out storage designs especially in the world of "noSQL" key-value stores. They can be disk based [70, 88, 4, 14, 3], SSD based [141, 180] or in-memory [62, 97, 132, 153].A widely used example of the last is `memcached`, an in-memory key-value store used as a distributed cache [97] that is used to speed up high traffic web sites such as LiveJournal.com, Wikipedia, or Slashdot. Although `memcached` is technically a cache, many workloads rely on a large percentage of the data being resident in memory, with block storage used only for failure recovery. Recent research on failure recovery in RAMCloud [153] demonstrates that in-memory key-value stores can provide availability and fast recovery at low cost.

In general, such systems are designed to support largescale MapReduce style analytics: some also support point queries or single-key lookups. However they do not support effcient multi-key queries or transactions: transactional support is either nonexistent or limited to single-key atomic operations. By contrast Jekyll and Hyde is designed to support multi-key transactions as well as analytics. While it cannot scale to petascale data like Google's

BigTable [70], being limited by the size of the Jekyll, we believe it is the right architecture for terascale applications such as Amadeus requiring transactional and analytic support.

10.6.3 Scale-out with distributed transactions

Recent work has combined in-memory operation with a scale-out architecture as well as transactional support using distributed transactions. H-Store [120] and VoltDB [197] are two examples that implement distributed transactions over a shared-nothing in-memory partitioning of the data. The proportion of distributed transactions can be reduced with careful partitioning of the data [83] but they still remain complex, expensive, and hard to scale.

Crescando [190] is a scan-based relational engine which achieves both scale-up and scale-out for scans. The implementation of the Hyde replica in Jekyll and Hyde is modeled on Crescando. However the Crescando engine by itself does not support mixed workloads: without heterogeneous replication the workload must either contend for limited resources on a single machine or suffer the overheads of distributed transactions.

10.7 Conclusions

In this paper we have described the Jekyll and Hyde keyvalue store. The design of this key-value store is interesting as it leverages hardware trends that are likely to disrupt the way clusters are designed for the data center. We have shown that by embracing these hardware trends and thinking how they can impact the design it is possible to build services on these clusters that have good performance.

To demonstrate this we have implemented a key-value store that is able to service two workloads concurrently, an online transactional workload and a real-time analytics workload. The transactional workload is best serviced by using a single (in-memory) replica offering low-latency random access to the data structured. The analytics workload is better serviced by a distributed (in-memory) replica. By building a single cluster that can support both, we provide a system with low-latency and good consistency across the replicas. Both replicas store the same information, there is no transcoding the data between the two replicas, and they are updated either synchronously or asynchronously but continuously. There is no batch processing of updates to the primary to update a secondary replica.

The advantages of a heterogeneous cluster architecture are not limited to performance. We believe that the capital and operating expenses of such a cluster will be much lower.

More generally, we believe that designing clusters from the ground up to leverage new hardware trends and to support particular workload classes is a promising way forwards. With the popularity of rack-scale appliances we are already seeing some limited forms of alternative design: for example, clusters that have both Ethernet and Infiniband networks and partitioning traffic between the two. We believe there is much more scope for interesting new hardware architectures based on commodity (but heterogeneous) compute, storage, and interconnect technologies. The observation that “one size does not fit all” [185] is now true for hardware architectures as much as for software.

Quatrième partie

Résumé de la thèse

Chapitre 11

Résumé de la thèse

Contents

11.1 Introduction

Le succès d'un jeu vidéo dépend de sa capacité à fournir une expérience immersive à ses joueurs. Les jeux massivement multijoueurs (*Massively Multiplayer Online Games* ou MMOGs) créent l'immersion en offrant un monde virtuel détaillé et hautement interactif. De tels mondes virtuels, appelés NVEs pour *Networked Virtual Environments*, sont peuplés par un grand nombre de joueurs ayant une totale liberté de mouvement. Chaque joueur est représenté par un *avatar* et peut interagir avec les autres avatars et objets du monde virtuel.

La popularité des MMOGs a significativement augmenté au cours de la dernière décennie. Aujourd'hui, des dizaines de mondes virtuels sont quotidiennement visités par des millions de joueurs [12]. La population cumulée de ces mondes virtuels dépasse actuellement 20 millions de joueurs à travers le monde.

Les NVEs trouvent sans cesse de nouvelles applications dans des domaines très divers. Utilisés à des fins publicitaires [39], médicales [56], sportives [160] ou même diplomatiques [129], ils deviennent progressivement une composante importante de la société moderne.

11.1.1 Définition du problème

L'architecture des jeux en ligne traditionnels se résume à un unique serveur qui stocke l'ensemble des objets du NVE et gère les interactions entre tous les joueurs. Cependant, une telle configuration est insuffisante pour supporter un MMOG. World of Warcraft, le plus populaire des MMOGs actuels, comptabilise plus de 10 millions d'abonnés, dont plusieurs dizaines de milliers de membres actifs [12, 196]. La charge induite par de telles populations dépasse clairement la capacité d'un seul serveur, forçant les concepteurs des MMOGs à adopter une architecture distribuée capable de passer à l'échelle.

La charge de travail des MMOGs dépend grandement du comportement des joueurs au sein du monde virtuel. La distribution des avatars dans le monde virtuel ainsi que la quantité d'information qu'ils échangent en interagissant varie en fonction des intérêts respectifs des joueurs. Le comportement des joueurs détermine également la popularité des objets du NVE et donc le profil d'accès aux données stockées par l'infrastructure. Or, le comportement des joueurs a été identifié comme étant très hétérogène et hautement dynamique par de nombreuses études [205, 47, 165, 196]. En conséquence de quoi, les évolutions de la charge applicative des MMOGs est difficilement prévisible, ce qui complexifie grandement le dimensionnement de l'infrastructure, l'allocation de ressources et l'équilibrage de charge. De plus, les MMOGs imposent les fortes contraintes suivantes sur l'architecture sous-jacente :

- **Réactivité :** la portion du monde virtuel vue par le joueur doit être mise à jour avec une faible latence. Les études montrent que pour être jouable, le temps entre deux mises à jour ne doit pas excéder quelques centaines de millisecondes tout au plus [44].
- **Cohérence :** les joueurs doivent avoir une vue cohérente du monde virtuel. Ils ne doivent pas voir : (i) de vues obsolètes, (ii) d'objets manquants ou dédoublés, (iii) d'objets dont l'état est en violation avec les règles du jeu. De plus, les vues des differents joueurs doivent être mutuellement cohérentes : si les joueurs accèdent un même objet de façon concurrente, les mises-à-jour de l'objet doivent être vues dans le même ordre par tous les joueurs.
- **Persistance :** pour une bonne immersion, le monde virtuel doit progresser même en l'absence de joueurs. Par conséquent, l'état de toutes les entités du NVE doit être stocké et maintenu malgré les déconnexions et les fautes.
- **Disponibilité :** les joueurs doivent pouvoir rejoindre le NVE à tout moment.
- **Élasticité :** la population des MMOGs varie grandement au cours du temps car les joueurs se connectent généralement en même temps (*e.g.*, pendant les week-ends) [67]. L'architecture doit donc s'adapter pour absorber les pics de charge.
- **Sécurité :** triche et autres comportements malveillants doivent être empêchés afin d'assurer de bonnes conditions de jeu aux joueurs honnêtes.

Satisfaire toutes ces contraintes pour des applications ayant un comportement aussi complexe et dynamique est extrêmement difficile. Actuellement, la plupart des MMOGs populaires ont une architecture rigide [2, 11, 19, 25]. L'énorme masse des joueurs est divisée en sous-ensembles disjoints appelés *shards*, et répartie sur un ensemble de serveurs. Chaque shard représente une instance de jeu totalement indépendante avec une population limitée

et aucune communication n'est possible entre les shards. Au sein d'un shard, l'univers est statiquement divisé en zones disjointes et la gestion de chaque zone est déléguée à un serveur [20]. De telles architectures sont capables de passer à l'échelle, mais ont deux défauts majeurs qui dégradent significativement l'expérience de jeu. Premièrement, un NVE divisé en shards n'est pas uni : deux joueurs qui veulent interagir doivent rejoindre le même shard. Cela nuit à l'expérience de jeu, car les joueurs ne sont pas garantis de pouvoir jouer ensemble à cause des limitations de population. Deuxièmement, le partitionnement statique des MMOGs actuels est peu adapté à la distribution de population non uniforme observée dans les mondes virtuels. Lorsque la population d'une zone dépasse la capacité de son serveur, une dégradation significative de la réactivité du jeu est observée [8].

De nombreuses propositions ont été faites par la communauté scientifique pour remédier aux lacunes des architectures existantes. Nombre d'entre elles sont basées sur le paradigme pair à pair, l'ensemble des pairs formant un réseau logique (*overlay*) capable de supporter le NVE [121, 42, 49, 194]. De telles architectures souffrent traditionnellement d'une réactivité faible et d'un manque de sécurité. Bien que un nombre important de travaux de recherche ont abordé le problème, aucune des propositions n'a complètement résolu les problèmes posés par les MMOGs.

11.1.2 Proposition de recherche et contributions

La charge de travail imposée par les MMOGs est fonction des interactions entre les joueurs et le NVE.

Ces interactions sont définies par les propriétés du jeu d'une part, et par le comportement des joueurs d'autre part. Les propriétés jeu étant connues dès la conception, la majeure partie de la complexité de la charge applicative des MMOGs est créée par les joueurs eux mêmes. Heureusement, le comportement des joueurs est souvent globalement prévisible. C'est pourquoi il est possible de prédire l'évolution du MMOG en observant le comportement des joueurs.

Dans cette thèse, nous affirmons que la prise en compte du comportement des joueurs *à l'exécution* peut accroitre la performance des architectures pour MMOGs. Notre objectif est de proposer des mécanismes **autonomes** capables de **dynamiquement** adapter l'architecture aux évolutions de la charge applicative. Le système peut alors, à l'aide de ces mécanismes, mieux anticiper les besoins applicatifs complexes des MMOGs. Grâce à cela, l'infrastructure distribuée gagne en efficacité sans surcoût en terme de charge.

Le manuscrit de thèse effectue d'abord une analyse complète de traces de mobilité des joueurs collectées dans Second Life, un NVE populaire [19]. L'analyse isole les propriétés caractéristiques de ces traces et propose un générateur capable de produire des traces synthétiques à large échelle ayant les caractéristiques des traces de Second Life. Enfin, nous proposons diverses techniques qui permettent d'améliorer les performances d'architectures pour MMOGs existantes. Nos contributions sont donc :

1. Une analyse de traces extraites de Second Life par Michiardi *et al.*

2. Un générateur de traces pouvant produire des traces de n'importe quelle taille ayant des caractéristiques proches de celles observées dans Second Life.

3. Un mécanisme qui adapte la topologie des NVE pair à pair pour mieux gérer les mouvements des avatars.

4. Une technique d'estimation de la distribution des joueurs dans le monde virtuel à très large échelle dans les MMOGs totalement décentralisés.

5. Un algorithme permettant d'accroitre la réactivité des MMOGs pair à pair. L'algorithme prend en compte la densité des joueurs pour créer des raccourcis efficaces dans l'overlay pair à pair et ainsi améliorer le routage décentralisé.

6. Une architecture client/serveur qui facilite le placement dynamique de serveurs afin de mieux gérer les distributions hétérogènes de joueurs. Notre architecture découple stockage de données et gestion des joueurs permettant ainsi au système de passer à l'échelle sans induire une fragmentation excessive de données. Une telle organisation améliore les performances générales du système ainsi que ses capacités d'adaptation à la charge.

11.2 Générateur de traces

Motivation

La conception de systèmes distribués capables de supporter efficacement la charge de travail induite par les MMOGs est une tâche complexe. Au cours de la dernière décennie, la communauté a apporté un grand nombre de solutions aux problèmes posés par les applications MMOGs émergentes. Pourtant, ces travaux de recherche ont un impact encore modeste sur les MMOGs présents sur le marché. Tous les jeux multijoueurs les plus populaires ont une architecture rigide et ne sont pas capables de correctement satisfaire les besoins applicatifs des MMOGs. Leur incapacité à dynamiquement s'adapter à la charge applicative induit de graves limitations dans l'expérience de jeu tout en étant responsable d'un important surcoût financier pour le producteur.

Miller *et al.* et Pittman *et al.* soulignent que l'une des raisons de ce manque de visibilité est la difficulté d'évaluation des solutions proposées [146, 157]. En effet, l'extraction de traces à partir de MMOGs présents sur le marché sans l'accord des producteurs de jeux est longue et complexe et il n'existe pas de benchmark unanimement adopté par la communauté. Par conséquent, les systèmes sont souvent testés sous des conditions irrealistes[1], rendant les contributions difficiles à évaluer.

Pourtant, l'analyse du comportement des joueurs montre de nombreuses caractéristiques communes dans tous les jeux étudiés [131, 165]. Cela suggère que ces caractéristiques sont en fait propres au comportement humain, et sont généralisables à tous les NVEs. Ainsi, il est observé que la distribution des avatars dans le monde virtuel est très hétérogène. Une grande partie de la population est rassemblée dans quelques zones de très forte densité, laissant le reste du monde quasi-désert. En outre, les études montrent que les déplacements des joueurs dans les zones désertes avaient une trajectoire plutôt rectiligne et une vitesse élevée. Par contraste, dans les zones denses, les avatars ont une trajectoire chaotique et une vitesse faible [131, 165].

[1] *e.g.*, de nombreux travaux utilisent une distribution uniforme de données lors des tests [157].

Contribution

Il nous apparait alors possible de proposer un modèle simple capable de décrire le comportement des joueurs décrit ci-dessus. Notre modèle se base sur un automate à états finis, chaque état représentant un type de comportement. L'automate, représenté sur le schéma 11.1, possède 3 états :

- **(H)**alte : le joueur ne bouge pas.
- **(E)**xploration : le joueur se déplace selon une trajectoire chaotique et a une vitesse réduite.
- **(V)**oyage : le joueur se déplace entre deux zones denses selon une trajectoire rectiligne et a une vitesse élevée.

Les trois états de l'automate décrivent les trois comportements de base des joueurs observés dans les MMOGs. Nous utilisons donc cet automate pour générer des traces de mobilité des joueurs. Après avoir distribué les joueurs sur la carte avec une distribution hétérogène, le générateur procède par itérations successives. Il affecte un état à chaque joueur selon son emplacement sur la carte. Un joueur situé dans une zone dense sera ainsi dans l'état E ou H, alors qu'un joueur dans une zone déserte sera dans l'état V. À chaque itération, le générateur fait évoluer les coordonnées des joueurs en prenant en compte leur trajectoire et leur état. L'ensemble des coordonnées des joueurs forme la trace de mobilité générée.

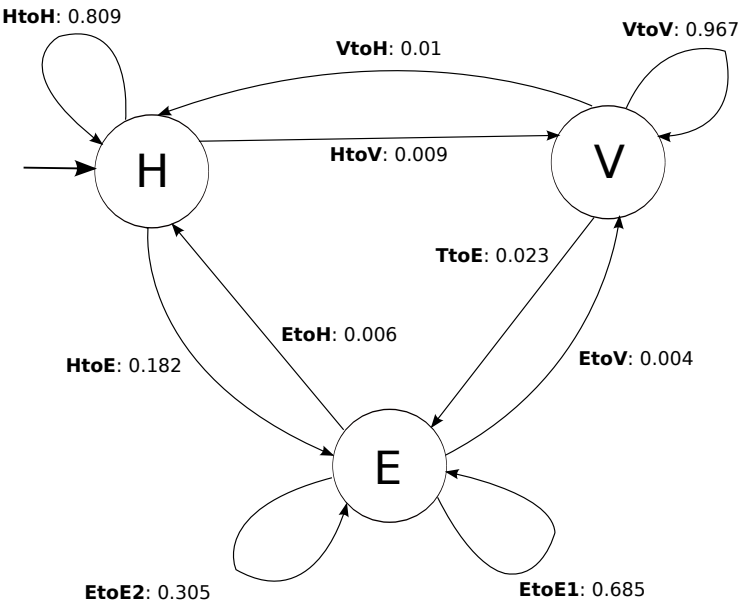


FIGURE 11.1 – *L'automate décrivant le comportement des joueurs.*

Notre modèle présente trois principaux avantages : (i) il permet de générer des traces qui décrivent un comportement proche des comportements réellement observés dans les MMOGs ; (ii) il est suffisamment simple pour être générique et (iii) il permet de créer des traces à large échelle avec ayant un échantillonage de bonne qualité. Nous pensons donc qu'il convient bien mieux qu'un modèle simplement uniforme utilisé par de nombreux travaux jusque là. Il s'agit cependant d'une première approche qui mérite d'être étendue afin de mieux décrire des phénomènes comportementaux complexes [2].

11.3 Support pour la mobilité des avatars dans les MMOGs distribués

Motivation

L'échange de données entre les joueurs d'un MMOG est essentiel au bon déroulement du jeu car chaque joueur doit propager en temps réel l'état de son avatar aux autres participants. Étant donné que les joueurs communiquent principalement avec leur entourage virtuel, les mises à jour ne sont utiles qu'à une petite fraction de tous les participants. Le MMOG doit donc disposer d'un mécanisme efficace de propagation de contenu dont la tâche est d'assurer que les mises à jour d'un joueur soient propagées avec une faible latence à ses voisins dans le monde virtuel.

Les architectures pair à pair dans lesquelles les pairs sont logiquement interconnectés avec leur voisins directs dans le NVE, appelées *topologies de proche en proche*, offrent une solution optimale à ce problème. En effet, dans ces topologies, un pair connait ses voisins proches et possède un lien direct avec chacun d'entre eux, ce qui permet une propagation rapide de l'information aux pairs intéressés. La principale difficulté est alors de maintenir la topologie à jour malgré le mouvement des avatars. Étant donné que le voisinage virtuel de l'avatar évolue au fur et à mesure de sa progression à travers le monde virtuel, son pair doit constamment mettre à jour son voisinage dans l'overlay pour incorporer les nouveaux venus. Il a ainsi été observé que certains mouvements pouvaient gravement mettre en danger la structure de la topologie [37].

En particulier, si la vitesse relative entre deux avatars est trop élevée et que la distance qui les sépare est trop faible, l'overlay n'arrive pas à mettre à jour son voisinage à temps. Ces retards peuvent produire des incohérences lors des propagations des mises à jour (appelées *fautes transitoires*), voire partitionner la topologie de façon permanente [37]. Ces phénomènes se produisent car l'overlay *n'anticipe pas* le mouvement, et a donc un délai extrêmement réduit pour réagir et modifier la topologie.

Contribution

Afin de résoudre ce problème, nous proposons un mécanisme appelé Blue Banana, qui permet d'*anticiper* les mouvements de l'avatar et d'adapter en avance l'overlay afin de préserver sa topologie. L'algorithme effectue une prédiction de la trajectoire de l'avatar, et si l'indice de confiance de la prédiction est jugé suffisamment élevé, le pair crée des liens avec les pairs

[2] Des travaux ont depuis été menés dans ce sens [187].

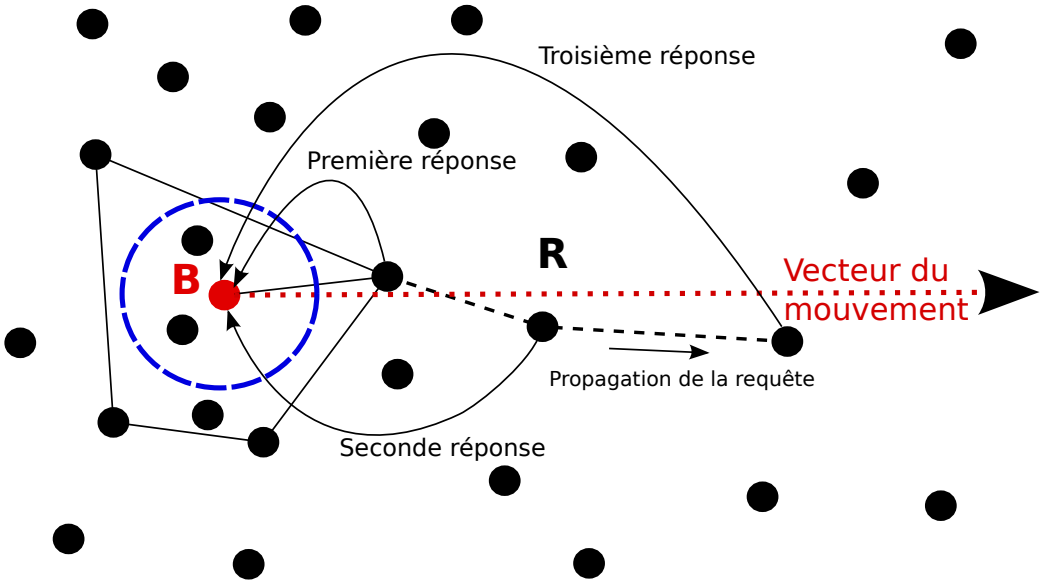


FIGURE 11.2 – *Algorithme de préchargement de voisinage : sont concernés les pairs situés le long de la trajectoire supposée de l'avatar.*

situés le long de sa trajectoire. La prédiction est basée sur l'analyse du comportement du joueur : une vitesse élevée suggère une trajectoire rectiligne, alors qu'une vitesse faible indique plutôt un mouvement chaotique. Ainsi, une grande vitesse est synonyme de bonne prévisibilité. L'estimation de la confiance de prédiction est un élément clé de notre algorithme. En effet, le préchargement de liens ne doit s'effectuer que si la confiance est bonne, car le préchargement de liens inutiles peut surcharger le pair et entrainer l'inverse de l'effet recherché. Notre algorithme permet de réduire le nombre de fautes transitoires induites par les mouvements des avatars car l'adaptation de l'overlay devance le mouvement. Lorsque deux avatars ayant une vitesse relative élevée se croisent dans le NVE, il est probable que leurs pairs se connaissent depuis suffisamment longtemps, évitant ainsi une faute transitoire.

Nous avons implémenté Blue Banana dans PeerSim, un simulateur à événements discret [118]. Notre mécanisme est utilisé pour améliorer la résistance à la mobilité de Solipsis, un overlay pour NVEs ayant une topologie de proche en proche. L'évaluation effectuée en utilisant des traces réalistes issues de notre générateur montre que Blue Banana permet d'éviter 20% des fautes transitoires avec un surcoût réseau de seulement 2%. De plus, notre mécanisme ne dégrade pas la robustesse du protocole original : lorsque le mouvement est erratique, le nombre de fautes transitoires n'augmente pas. Enfin, grâce à Blue Banana un pair connait en moyenne 7.5 pairs en aval du mouvement, alors que sans l'aide de Blue Banana, il n'en connait en moyenne qu'un seul. Une meilleure connaissance de ses futurs voisins permet au pair de précharger plus d'informations concernant le futur entourage de son avatar.

11.4 Prise en compte de la distribution des avatars dans les MMOGs pair à pair

Motivation

Les topologies de proche en proche sont largement utilisées dans les architectures pair à pair pour MMOGs car elles permettent une propagation efficace des mises à jour du monde virtuel. Le placement d'un pair dans un telle topologie est déterminé par la position de son avatar dans le NVE. Par conséquent, lorsqu'un avatar rejoint le NVE ou se téléporte dans une région éloignée du NVE, son pair doit être replacé à sa nouvelle position dans la topologie. Pour cela, une requête de connexion doit être routée à sa nouvelle position dans l'overlay. Les MMOGs existants subissent souvent des taux de connexions/déconnexions élevés, et nombre d'entre eux mettent à disposition un service de téléportation. C'est pourquoi assurer le passage à l'échelle du routage dans les MMOGs pair à pair est essentiel.

Les topologies de proche en proche garantissent la terminaison d'algorithmes de routage glouton [54]. Afin de rendre le processus efficace à large échelle, des raccourcis, ou *liens longs* sont ajoutés à la topologie. Kleinberg a montré que le processus de routage est optimal dans une grille si la probabilité pour un pair p de choisir un pair q en tant que lien long dépend de la distance en nombre de hops entre p et q [126]. La distribution de liens longs définie par Kleinberg est appelée *d-harmonique* [126]. Pour atteindre une telle distribution pour ses raccourcis, un pair doit trouver des pairs qui se trouvent à bonne distance de lui (*en nombre de hops*) dans le graphe de l'overlay.

Implicitement, cela signifie qu'un pair doit connaitre la topologie de l'overlay. L'acquisition de cette connaissance est aisée lorsque la distribution des avatars dans le NVE est uniforme : la distance dans le graphe, *i.e.,* le nombre de hops, croit linéairement avec la distance dans le NVE. Cependant, de nombreuses études montrent que la distribution des avatars dans les NVEs est hautement hétérogène. Dans ces cas, l'estimation des distances dans le graphe requiert une connaissance plus ou moins précise de la *distribution* de la population dans le NVE. Les solutions existantes utilisent des techniques d'échantillonnage aléatoire pour sonder la distribution lors du processus de création des liens longs. Or, les populations des NVEs sont fortement dynamiques et des regroupement de pairs peuvent se former ou s'effacer à tout moment, rendant les liens longs obsolètes. En l'absence d'une surveillance de la densité durant l'exécution, les systèmes existants ne s'aperçoivent du changement de densité que lorsque les performances du routage sont dégradées.

Contribution

Nous proposons DONUT, un mécanisme décentralisé qui effectue une surveillance de la population du NVE. DONUT permet à chaque pair d'acquérir et maintenir une approximation de la distribution de la population au sein du NVE. Le pair peut ainsi détecter les changements de distribution et replacer ses liens longs bien avant que les performances du routage se dégradent. Grâce à cette carte de densité, un pair peut à tout moment estimer *localement* la distance en nombre de hops qui le sépare de n'importe quel point du NVE. Les pairs utilisent cette estimation pour établir des liens longs qui permettent au graphe de l'overlay de maintenir la propriété de Kleinberg malgré des distributions non uniformes. Le processus

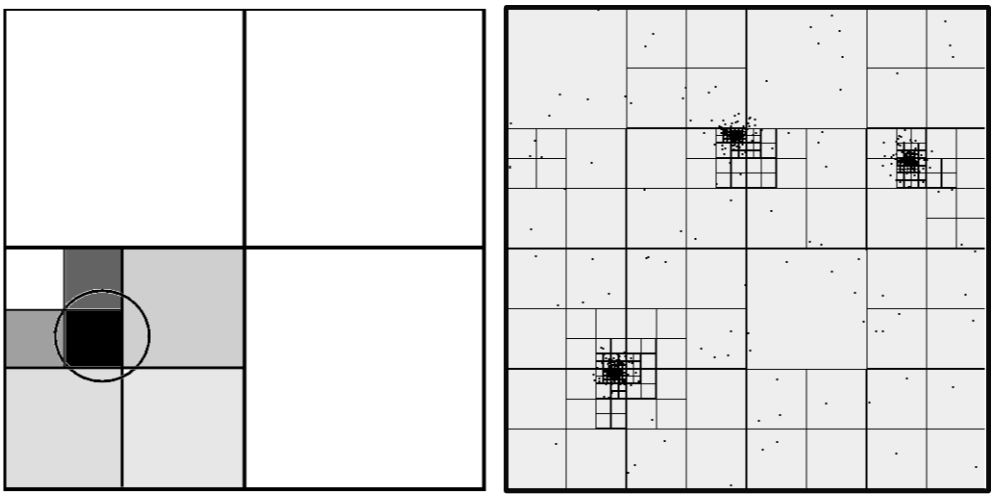


FIGURE 11.3 – ***Gauche :*** *Acquisition d'une carte de densité locale par un pair.* ***Droite :*** *Approximation globale obtenue après échange d'information.*

de replacement des liens longs consomme très peu de ressources réseau, car la localisation des futurs liens longs s'effectue localement grâce à la carte de densité.

La connaissance de la densité de population globale sur chaque pair est le résultat d'un processus collaboratif (Figure 11.3). Chaque pair surveille la densité de la portion de NVE qui entoure son avatar et propage sa connaissance locale dans l'overlay. La propagation se fait par deux moyens distincts : (i) le pair rajoute une information de densité à tous les messages transmis ; (ii) le pair utilise un algorithme de gossip. L'évaluation de DONUT montre que la carte de densité globale peut accroitre les performances du routage de 20% par rapport aux techniques existantes. Notre mécanisme a un impact réseau de seulement quelques octets par seconde par pair, ce qui le rend abordable même pour les pairs avec une faible bande passante.

Outre l'amélioration du routage, l'acquisition d'une connaissance globale du système peut s'avérer très utile pour le pair. Elle peut être exploitée par d'autres mécanismes de maintenance de l'overlay, tels que la répartition de charge ou l'estimation de la taille de l'overlay. Les contributions de ces travaux sont donc : 1) un algorithme distribué qui permet de doter chaque pair d'une connaissance de la répartition de la population dans le NVE et 2) une technique permettant d'utiliser cette connaissance pour améliorer le routage en présence de distributions hétérogènes.

11.5 Passage à l'échelle sans fragmentation de données dans les architectures pour MMOFPS

Motivation

Les jeux de tir en vue subjective (ou FPS pour First Person Shooters) sont un genre important de jeux video avec plusieurs millions d'utilisateurs à travers le monde. Des succès commer-

ciaux tels que Counter-Strike, Battlefield ou Quake III sont apparus à la fin des années 90 et sont toujours très populaires. Cette popularité puise sa source dans une mécanique de jeu très distractive et peu contraignante qui attire même des utilisateurs peu intéressés par d'autres types de jeux. Les FPS mettent en scène des champs de bataille virtuels dans lesquels les joueurs luttent les uns contre les autres dans une succession de combats.

De par la nature de leur mécanisme de jeu, les FPS imposent des contraintes très fortes sur l'infrastructure sous-jacente. Des latences supérieures à 150 ms dégradent sévèrement l'expérience de jeu [44], et les incohérences ou indisponibilités du jeu sont extrêmement frustrantes pour le joueur [67]. Maintenir simultanément cohérence, réactivité et disponibilité est non trivial à large échelle. La plupart des FPS, y compris les plus populaires, contournent le problème en limitant la population des champs de bataille à 64 joueurs au maximum.

Seule une fraction des FPS, alors appelés MMOFPS, abordent le problème en distribuant la charge du NVE sur un ensemble de serveurs. Leurs architectures reposent classiquement sur un partitionnement du NVE en zones, chacune étant gérée par un serveur distinct. Chaque serveur est alors responsable du stockage et de la maintenance des entités (objets et joueurs) présentes dans sa zone. Lorsqu'il joue, un joueur est intéressé par les objets qui entourent son avatar dans le jeu. Si le joueur se trouve à la jonction de plusieurs zones, ces objets sont stockés sur plusieurs serveurs. Ceci induit des échanges de données entre ces serveurs, car le jeu doit garantir l'illusion d'un monde continu à ses joueurs. Au fur et à mesure de l'accroissement de la population, de nouveaux serveurs doivent être ajoutés pour faire face à l'augmentation de la charge. Chaque serveur voit ainsi sa zone de responsabilité se réduire, ce qui conduit au final à un monde très *fragmenté*. Les objets du NVE sont répartis parmi un grand nombre de serveurs, augmentant la quantité d'informations devant être échangée sur le réseau. C'est pourquoi de telles architectures sont complexes et coûteuses à maintenir. Le producteur du jeu est alors obligé de mettre en place un système contraignant d'abonnement pour amortir ces coûts de maintenance, rebutant un grand nombre de joueurs occasionnels [113]. À cause de cela, le succès des MMORPGs est relativement limité, et tous les FPS les plus populaires gardent une conception client/serveur simple.

Contribution

Cette section aborde le problème de l'absence d'efficacité des architectures pour MMOFPS existantes. Nous avons effectué une analyse détaillée de la capacité de passage à l'échelle de Quake III Arena, un FPS populaire ayant une architecture client/serveur. Les résultats montrent que la gestion des joueurs devient vite beaucoup plus coûteuse que la gestion des objets du NVE lorsque la population augmente.

Les FPS existants exécutent ces deux tâches bien distinctes sur les mêmes hôtes physiques et lorsque la charge augmente, ces hôtes deviennent rapidement débordés par la gestion des joueurs. La solution apportée par les architectures existantes pour diminuer cette charge est de réduire la taille de la zone qu'il gère. Cette technique va mécaniquement réduire la charge applicative, mais à pour effet de bord l'augmentation de la fragmentation des données du NVE. Nous proposons donc de gérer ces deux tâches séparément. Dans notre architecture, les deux processus sont décorrélés et peuvent passer à l'échelle à leur propre rythme. La gestion des données est assurée par une base de données distribuée en mémoire qui forme l'épine dorsale de notre architecture, garantissant robustesse et réactivité à notre système. La gestion des joueurs est réalisée par un ensemble d'entités physiques distinctes appelées

Miroirs. Ces miroirs récupèrent collectivement l'état des objets du NVE stockés dans la base de données et choisissent les objets à envoyer à chaque joueur. Comme dans les architectures traditionnelles, chaque joueur reçoit une vue cohérente du NVE contenant tous les objets qui entourent le joueur à cet instant. Lorsque la population augmente, de nouveaux miroirs sont ajoutés à la demande pour mieux supporter la charge sans pour autant augmenter la fragmentation du NVE.

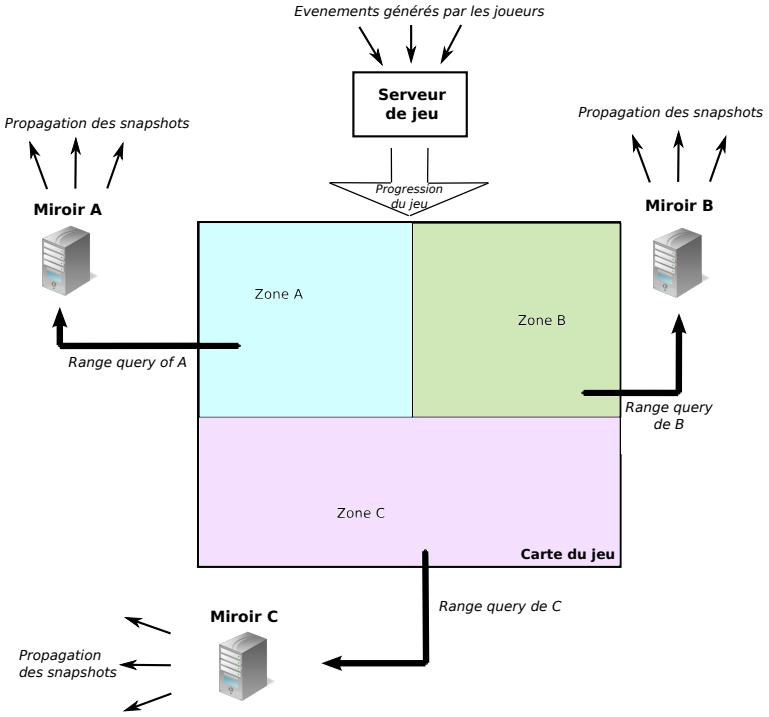


FIGURE 11.4 – *Vue d'ensemble de l'architecture de QuakeVolt partitionnant le jeu.*

Notre architecture n'est pas la première utilisant une base de données pour stocker les données des MMOGs. Darkstar utilise une base de données centralisée pour stocker l'état du jeu, avec un ensemble de serveurs applicatifs faisant interface entre la base de données et les clients [50]. Cependant, la base de données utilisée est classique, c'est-à-dire qu'elle n'est ni distribuée ni en mémoire. Pour cette raison, la base de données devient rapidement le facteur limitant du système. Pour assurer la réactivité, Darkstar met en place un mécanisme complexe de caches cohérents qui passe difficilement à l'échelle. MiddleSIR stocke toutes les données du jeu dans une base de données et utilise la réplication de machines à états afin de maintenir plusieurs répliques de données cohérentes [159]. Ce mécanisme de cohérence impose lui aussi des limites au passage à l'échelle du système.

Dans cette section, nous démontrons les avantages de l'utilisation d'une base de données distribuée en mémoire dans les MMOGs. Nous avons adapté Quake III afin qu'il puisse s'exécuter au dessus de VoltDB, une base de données distribuée en mémoire. Notre prototype, appelé QuakeVolt, transforme Quake III, originalement conçu pour 64 joueurs au plus, en un champ de bataille à large échelle, capable de supporter plusieurs centaines de joueurs simultanément. La Figure 11.4 donne une vue d'ensemble du partitionnement du NVE dans

QuakeVolt. VoltDB gère accès concurrents, persistance et tolérance aux fautes, démontrant par la pratique la faisabilité de notre approche. Il s'agit, à notre connaissance, la première proposition qui expose explicitement la nécessité de découpler la gestion des données de la gestion des joueurs et utilise pour cela une base de données en mémoire.

Le développement de QuakeVolt a demandé très peu d'efforts. Toutes les composantes de l'architecture peuvent s'exécuter sur du matériel standard, et l'adaptation de Quake III à VoltDB nécessite d'apporter très peu de modifications au code original du serveur de Quake III. De plus, les clients Quake III peuvent de façon interchangeable se connecter aussi bien à Quake III qu'à QuakeVolt : le protocole entre le client et le serveur reste inchangé.

Les contributions proposées dans cette section sont : (i) une étude approfondie des facteurs limitant le passage à l'échelle des FPS standards et (ii) une nouvelle architecture pour MMOFPS. Notre architecture possède les qualités suivantes :

- **Passage à l'échelle** : l'absence de fragmentation inutile des données du NVE permet d'améliorer les performances globales du système et de gérer potentiellement un grand nombre de joueurs.
- **Simplicité** : la gestion complexe des données avec accès concurrents est déléguée à la base de données, et l'adaptation de Quake III à notre architecture nécessite uniquement la modification de 300 lignes de code du serveur, ce qui représente moins de 1% de la taille du code original.
- **Faible coût** : l'infrastructure peut s'exécuter sur du materiel standard.
- **Transparence** : les modifications apportées à l'architecture originelle du FPS sont transparentes pour l'utilisateur. Aucune modification du client originel n'est nécessaire pour le rendre compatible avec la nouvelle architecture.
- **Robustesse** : le stockage des données de jeu dans une base de données garantit persistance et disponibilité en présence de pannes. Aucun mécanisme dédié n'est nécessaire pour assurer ces précieuses propriétés.

11.6 Conclusion

Ce chapitre a résumé nos efforts pour améliorer les architectures distribuées pour MMOGs. Nos propositions visent à rendre ces architectures *"conscientes"* des évolutions de l'application qu'elles supportent. Pour cela, nous avons conçu et évalué des mécanismes capables de surveiller l'activité de l'utilisateur pendant le jeu et d'interpréter les données récoltées selon des modèles comportementaux simples. Nous montrons qu'une telle "conscience" permet au système d'adapter (pro)activement son infrastructure afin de mieux servir l'application.

Nous basons notre approche sur un large ensemble d'études qui montrent que la charge applicative des MMOGs est façonnée par le comportement des joueurs. Ces études soulignent que malgré leur complexité, les évolutions de la charge sont qualitativement non aléatoires et peuvent être prédites avec des modèles relativement simples. Nous proposons un ensemble de mécanismes génériques qui facilitent l'intégration de ces modèles au niveau de l'infrastructure.

Nos travaux traitent deux aspects de la charge applicative des MMOGs : mobilité et distribution des joueurs. Ce choix est motivé par l'importance de l'impact que ces aspects ont sur les architectures distribuées. Des vitesses élevées lors du déplacement des avatars peuvent dégrader la topologie de l'infrastructure [37], alors que les distributions de population non uniformes compliquent une répartition équitable de la charge. Or, vitesse élevée et distribution non uniforme sont des caractéristiques très communes dans les MMOGs existants [196, 75].

Une part non négligeable de nos efforts est dédiée à la création de traces d'activité réalistes pouvant servir à une bonne évaluation de mécanismes proposés par la communauté. Nous considérons qu'il s'agit d'une condition nécessaire à une meilleure intégration des prototypes de recherche dans les MMOGs dans le futur.

Bibliographie

Bibliographie

[1] Active worlds. http://www.activeworlds.com/. Accessed : 07/2012.

[2] Aion : Free to play. http://www.aionfreetoplay.com/website/. Accessed : 07/2012.

[3] Amazon simpledb. http://aws.amazon.com/simpledb/. Accessed : 07/2012.

[4] Apache hbase. http://hbase.apache.org/. Accessed : 07/2012.

[5] Counter-strike. http://en.wikipedia.org/wiki/Counter-Strike. Accessed : 07/2012.

[6] Cyberathlete professional league. http://thecpl.com/. Accessed : 07/2012.

[7] Eve online. http://www.eveonline.com/. Accessed : 07/2012.

[8] Eve online architecture. http://highscalability.com/eve-online-architecture. Accessed : 07/2012.

[9] Folding@home. http://folding.stanford.edu. Accessed : 07/2012.

[10] Hewlett-Packard. HP Project Moonshot : Changing the game with extreme low-energy computing. http://h20195.www2.hp.com/V2/GetPDF.aspx/ 4AA3-9839ENW.pdf.

[11] Lineage 2. http://www.lineage2.com/en/. Accessed : 07/2012.

[12] Mmo data. http://www.mmodata.net/. Accessed : 07/2012.

[13] Mmorpgrealm. world of warcraft statistic in 2010. http://www.mmorpgrealm.com/world-of-warcraft-statistic-in-2010/. Accessed : 07/2012.

[14] Mongodb. http://www.mongodb.org/. Accessed : 07/2012.

[15] Opensimulator. http://opensimulator.org/wiki/Main_Page. Accessed : 07/2012.

[16] Planetside. http://planetside.station.sony.com/. Accessed : 07/2012.

[17] Planetside 2. http://www.planetside2.com/. Accessed : 07/2012.

[18] Quake 3 arena. http://en.wikipedia.org/wiki/Quake_III_Arena. Accessed : 07/2012.

[19] Second life. http://secondlife.com/. Accessed : 07/2012.

[20] Second life server architecture. http://wiki.secondlife.com/wiki/Server_architecture. Accessed : 07/2012.

[21] Seti@home. http://setiathome.berkeley.edu/. Accessed : 07/2012.

[22] There. http://www.there.com/. Accessed : 07/2012.

[23] Voltdb for online games. http://voltdb.com/voltdb-online-games. Accessed : 07/2012.

[24] World war 2 online. http://www.battlegroundeurope.com/. Accessed : 07/2012.

[25] Wow. http://eu.battle.net/wow/fr/. Accessed : 07/2012.

[26] Wow bittorrent downloader wiki. http://www.wowwiki.com/Blizzard_Downloader. Accessed : 07/2012.

[27] Wow instance wiki. http://www.wowwiki.com/Instance. Accessed : 07/2012.

[28] Wowwiki. http://www.wowwiki.com/Quest. Accessed : 09/2012.

[29] Wowwiki. http://www.wowwiki.com/Player_vs._Player. Accessed : 09/2012.

[30] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-grid : a self-organizing structured p2p system. *ACM SIGMOD Record*, 32(3) :29–33, 2003.

[31] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite : Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI '02 : Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, USA, December 2002.

[32] Matteo Agosti, Francesco Zanichelli, Michele Amoretti, and Gianni Conte. P2pam : a framework for peer-to-peer architectural modeling based on peersim. In Sándor Molnár, John Heath, Olivier Dalle, and Gabriel A. Wainer, editors, *SimuTools*, page 22. ICST, 2008.

[33] D.G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn : A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14. ACM, 2009.

[34] D.P. Anderson. Boinc : A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.

[35] M.S. Ardekani, P. Sutra, N. Preguiça, and M. Shapiro. Non-monotonic snapshot isolation. *Institut National de la Recherche en Informatique et Automatique, Tech. Rep. RR-7805*, 2011.

[36] M. Assiotis and V. Tzanov. A distributed architecture for mmorpg. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 4. ACM, 2006.

[37] H. Backhaus and S. Krause. Voronoi-based adaptive scalable transfer revisited : gain and loss of a voronoi-based peer-to-peer approach for mmog. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 49–54. ACM, 2007.

[38] N.T.J. Bailey. *The mathematical theory of epidemics*. Griffin London, 1957.

[39] S. Barnes. Virtual worlds as a medium for advertising. *ACM SIGMIS Database*, 38(4) :45–55, 2007.

[40] I. Barri, F. Giné, and C. Roig. A scalable hybrid p2p system for mmofps. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 341–347. IEEE, 2010.

[41] S.A. Baset and H. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *Arxiv preprint cs/0412017*, 2004.

[42] O. Beaumont, A.M. Kermarrec, L. Marchal, and E. Rivière. Voronet : A scalable object network based on voronoi tessellations. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[43] O. Beaumont, A.M. Kermarrec, and É. Rivière. Peer to peer multidimensional overlays : Approximating complex structures. In *Proceedings of the 11th international conference on Principles of distributed systems*, pages 315–328. Springer-Verlag, 2007.

[44] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003. In Wu chang Feng, editor, *NETGAMES*, pages 144–151. ACM, 2004.

[45] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2) :1–10, 1995.

[46] R. Bhagwan, S. Savage, and G. Voelker. Understanding Availability. In *Proceedings of IPTPS'03*, 2003.

[47] A. Bharambe, J.R. Douceur, J.R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. Donnybrook : Enabling large-scale, high-speed, peer-to-peer games. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 389–400. ACM, 2008.

[48] A.R. Bharambe, M. Agrawal, and S. Seshan. Mercury : supporting scalable multi-attribute range queries. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 353–366. ACM, 2004.

[49] Ashwin R. Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus : A distributed architecture for online multiplayer games. In *NSDI*. USENIX, 2006.

[50] T. Blackman and J. Waldo. Scalable data storage in project darkstar. 2009.

[51] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a Serverless Distributed file System Deployed on an Existing Set of Desktop PCs. In *Proceedings of SIGMETRICS*, 2000.

[52] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck : Memory access. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 54–65, 1999.

[53] F. Bonnet, A.M. Kermarrec, and M. Raynal. Small-world networks : From theoretical bounds to practical systems. *Principles of distributed systems*, pages 372–385, 2007.

[54] P. Bose and P. Morin. Online routing in triangulations. *Algorithms and Computation*, pages 113–122, 1999.

[55] J.S. Boulanger, J. Kienzle, and C. Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 6. ACM, 2006.

[56] M.N.K. Boulos, L. Hetherington, and S. Wheeler. Second life : an overview of the potential of 3-d virtual worlds in medical and health education. *Health Information & Libraries Journal*, 24(4) :233–245, 2007.

[57] Andrei Broder, Michael Mitzenmacher, and Andrei Broder I Michael Mitzenmacher. Network applications of bloom filters : A survey. In *Internet Mathematics*, pages 636–646, 2002.

[58] J.H. Burrows. Secure hash standard. Technical report, DTIC Document, 1995.

[59] Jean-Michel Busca, Fabio Picconi, and Pierre Sens. Pastis : A highly-scalable multi-user peer-to-peer file system. In *Euro-Par '05 : Proceedings of European Conference on Parallel Computing*, pages 1173–1182, August 2005.

[60] E. Buyukkaya and M. Abdallah. Efficient triangulation for p2p networked virtual environments. *Multimedia Tools and Applications*, 45(1) :291–312, 2009.

[61] E. Buyukkaya, M. Abdallah, and R. Cavagna. Vorogame : a hybrid p2p architecture for massively multiplayer games. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1–5. Ieee, 2009.

[62] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B.C. Ooi, H.T. Vo, S. Wu, and Q. Xu. Es2 : A cloud data storage system for supporting both oltp and olap. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 291–302. IEEE, 2011.

[63] M. Castro, M. Costa, and A. Rowstron. Should we build gnutella on a structured overlay ? *ACM SIGCOMM Computer Communication Review*, 34(1) :131–136, 2004.

[64] M. Castro, M. Costa, and A. Rowstron. Debunking some myths about structured and unstructured overlays. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 85–98. USENIX Association, 2005.

[65] M. Castro, P. Druschel, A.M. Kermarrec, and A.I.T. Rowstron. Scribe : A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8) :1489–1499, 2002.

[66] Miguel Castro, Manuel Costa, and Antony Rowstron. Performance and dependability of structured peer-to-peer overlays. In *DSN '04 : Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 9, Washington, DC, USA, June 2004. IEEE Computer Society.

[67] C. Chambers, W. Feng, S. Sahu, and D. Saha. Measurement-based characterization of a collection of on-line games. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 1–1. USENIX Association, 2005.

[68] A. Chan, R.W.H. Lau, and B. Ng. A hybrid motion prediction method for caching and prefetching in distributed virtual environments. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 135–142. ACM, 2001.

[69] L. Chan, J. Yong, J. Bai, B. Leong, and R. Tan. Hydra : a massively-multiplayer peer-to-peer architecture for the game developer. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 37–42. ACM, 2007.

[70] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable : A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2) :4, 2008.

[71] Yatin Chawathe, Sriram Ramabhadran, Sylvia Ratnasamy, Anthony LaMarca, Scott Shenker, and Joseph M. Hellerstein. A case study in building layered dht applications. In Roch Guérin, Ramesh Govindan, and Greg Minshall, editors, *SIGCOMM*, pages 97–108. ACM, 2005.

[72] A.V. Chechkin, V.Y. Gonchar, J. Klafter, and R. Metzler. Fundamentals of lévy flight processes. *Fractals, Diffusion, and Relaxation in Disordered Complex Systems*, pages 439–496, 2006.

[73] A. Chen and R.R. Muntz. Peer clustering : a hybrid approach to distributed virtual environments. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 11. ACM, 2006.

[74] S. Chen, A. Ailamaki, M. Athanassoulis, P.B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. Tpc-e vs. tpc-c : characterizing the new tpc-e benchmark via an i/o comparison study. *ACM SIGMOD Record*, 39(3) :5–10, 2011.

[75] E. Cheslack-Postava, T. Azim, B.F.T. Mistree, D.R. Horn, J. Terrace, P. Levis, and M.J. Freedman. A scalable server for 3d metaverses. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX*, volume 12, 2012.

[76] J.H.P. Chim, R.W.H. Lau, H. Va Leong, and A. Si. Multi-resolution cache management in digital virtual library. In *Research and Technology Advances in Digital Libraries, 1998. ADL 98. Proceedings. IEEE International Forum on*, pages 66–75. IEEE, 1998.

[77] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet : A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2001.

[78] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11) :40–45, 2006.

[79] B. Cohen. The bittorrent protocol specification, 2008.

[80] Carmela Comito, Simon Patarin, and Domenico Talia. A semantic overlay network for p2p schema-based data integration. In Paolo Bellavista, Chi-Ming Chen, Antonio Corradi, and Mahmoud Daneshmand, editors, *ISCC*, pages 88–94. IEEE Computer Society, 2006.

[81] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop : Exploiting in-network aggregation for big data applications. *NSDI, Apr*, 2012.

[82] E. Cronin, A.R. Kurc, B. Filstrup, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools and Applications*, 23(1) :7–30, 2004.

[83] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism : a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2) :48–57, 2010.

[84] F. Dabek, E. Brunskill, M.F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan. Building peer-to-peer systems with chord, a distributed lookup service. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 81–86. IEEE, 2001.

[85] Frank Dabek, Frans M. Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP '01 : Proceedings of the 8th ACM symposium on Operating Systems Principles*, volume 35, pages 202–215, New York, NY, USA, December 2001. ACM Press.

[86] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, Frans F. Kaashoek, and Robert Morris. Designing a DHT for low latency and high throughput. In *NSDI '04 : Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, San Francisco, CA, USA, March 2004.

[87] B. De Vleeschauwer, B. Van Den Bossche, T. Verdickt, F. De Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for massively multiplayer online gaming. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–7. ACM, 2005.

[88] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo : amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[89] S. Douglas, E. Tanin, A. Harwood, and S. Karunasekera. Enabling massively multiplayer online gaming applications on a p2p architecture. In *Proceedings of the IEEE international conference on information and automation*, pages 7–12, 2005.

[90] M. Driel, J. Kraaijeveld, Z.K. Shao, and R. van der Zon. A survey on mmog system architectures, 2011.

[91] A. El Rhalibi and M. Merabti. Agents-based modeling for a peer-to-peer mmog architecture. *Computers in Entertainment (CIE)*, 3(2) :3–3, 2005.

[92] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *SRDS*, pages 73–84. IEEE Computer Society, 2005.

[93] L. Fan, H. Taylor, and P. Trinder. Mediator : a design framework for p2p mmogs. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 43–48. ACM, 2007.

[94] Johannes Färber. Network game traffic modelling. In *Proceedings of the 1st workshop on Network and system support for games*, NetGames '02, pages 53–57, New York, NY, USA, 2002. ACM.

[95] W. Feng, D. Brandt, and D. Saha. A long-term study of a popular mmorpg. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 19–24. ACM, 2007.

[96] S. Fernandes, R. Antonello, J. Moreira, C. Kamienski, and D. Sadok. Traffic analysis beyond this world : the case of second life. In *17th International workshop on Network and operating systems support for digital audio and video, University of Illinois, Urbana-Champaign*, pages 4–5. Citeseer, 2007.

[97] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124) :5, 2004.

[98] A.D. Flaxman. Expansion and lack thereof in randomly perturbed graphs. *Internet Mathematics*, 4(2-3) :131–147, 2007.

[99] T. Fritsch, H. Ritter, and J. Schiller. The effect of latency and network limitations on mmorpgs : a field study of everquest2. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–9. ACM, 2005.

[100] P. Fulop, S. Szabó, and T. Szálka. Accuracy of random walk and markovian mobility models in location prediction methods. In *Software, Telecommunications and Computer Networks, 2007. SoftCOM 2007. 15th International Conference on*, pages 1–5. IEEE, 2007.

[101] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03 : Proceedings of the 9th ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, October 2003. ACM Press.

[102] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Symmetric replication for structured peer-to-peer systems. In *DBISP2P '05 : Proceedings of the 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, page 12, Trondheim, Norway, August 2005.

[103] S. Girdzijauskas, A. Datta, and K. Aberer. On small world graphs in non-uniformly distributed key spaces. In *Data Engineering Workshops, 2005. 21st International Conference on*, pages 1187–1187. Ieee, 2005.

[104] S. Girdzijauskas, A. Datta, and K. Aberer. Oscar : Small-world overlay for realistic key distributions. In *Proceedings of the 2005/2006 international conference on Databases, information systems, and peer-to-peer computing*, pages 247–258. Springer-Verlag, 2005.

[105] L. Gong. Jxta : A network programming environment. *Internet Computing, IEEE*, 5(3) :88–95, 2001.

[106] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise : a main memory hybrid storage engine. *Proceedings of the VLDB Endowment*, 4(2) :105–116, 2010.

[107] Saikat Guha, Neil Daswani, and Ravi Jain. An experimental study of the skype peer-to-peer voip system. In *IPTPS*, February 2006.

[108] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Approximate range selection queries in peer-to-peer systems. In *CIDR*, 2003.

[109] N. Gupta, A. Demers, J. Gehrke, P. Unterbrunner, and W. White. Scalability for virtual worlds. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 1311–1314. IEEE, 2009.

[110] M.P. Herlihy and J.M. Wing. Linearizability : A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3) :463–492, 1990.

[111] S. Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4) :439–562, 2006.

[112] D. Horn, E. Cheslack-Postava, B.F.T. Mistree, T. Azim, J. Terrace, M.J. Freedman, and P. Levis. To infinity and not beyond : Scaling communication in virtual worlds with meru. Technical report, Stanford Computer Science Technical Report, CSTR 2010-01, 2010.

[113] W.H. Hsu. Business model developments for the pc-based massively multiplayer online game (mmog) industry. 2005.

[114] S.Y. Hu, J.F. Chen, and T.H. Chen. Von : a scalable peer-to-peer network for virtual environments. *Network, IEEE*, 20(4) :22–31, 2006.

[115] S.Y. Hu and G.M. Liao. Scalable peer-to-peer networked virtual environment. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 129–133. ACM, 2004.

[116] 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 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 116–120. ACM, 2004.

[117] J. Jardine and D. Zappala. A hybrid architecture for massively multiplayer online games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 60–65. ACM, 2008.

[118] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim simulator. http://peersim.sourceforge.net/.

[119] Jimmy Jernberg, Vladimir Vlassov, Ali Ghodsi, and Seif Haridi. Doh : A content delivery peer-to-peer network. In *Euro-Par '06 : Proceedings of European Conference on Parallel Computing*, page 13, Dresden, Germany, September 2006.

[120] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E.P.C. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store : a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2) :1496–1499, 2008.

[121] J. Keller and G. Simon. Solipsis : A massively multi-participant virtual world. In *Proc. of PDPTA*, pages 262–268, 2003.

[122] A. Kemper and T. Neumann. Hyper : A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011.

[123] J. Kesselman. Server architectures for massively multiplayer online games. In *Session TS-1084, Javaone conference*, 2005.

[124] J. Kim, J. Choi, D. Chang, T. Kwon, Y. Choi, and E. Yuk. Traffic characteristics of a massively multi-player online role playing game. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–8. ACM, 2005.

[125] Kyungbaek Kim and Daeyeon Park. Reducing data replication overhead in DHT based peer-to-peer system. In *HPCC '06 : Proceedings of the 2nd International Conference on High Performance Computing and Communications*, pages 915–924, Munich, Germany, September 2006.

[126] J. Kleinberg. The small-world phenomenon : an algorithm perspective. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 163–170. ACM, 2000.

[127] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1. Ieee, 2004.

[128] Salma Ktari, Mathieu Zoubert, Artur Hecker, and Houda Labiod. Performance evaluation of replication strategies in DHTs under churn. In *MUM '07 : Proceedings of the 6th international conference on Mobile and ubiquitous multimedia*, pages 90–97, New York, NY, USA, December 2007. ACM Press.

[129] S. Kumar, J. Chhugani, C. Kim, D. Kim, A. Nguyen, P. Dubey, C. Bienia, and Y. Kim. Second life and the new generation of virtual worlds. *Computer*, 41(9) :46–53, 2008.

[130] K. Kutzner and T. Fuhrmann. Measuring large overlay networks - the overnet example. In *Kommunikation in Verteilten Systemen (KiVS)*, pages 193–204. Springer, 2005.

[131] C.A. La and P. Michiardi. Characterizing user mobility in second life. In *Proceedings of the first workshop on Online social networks*, pages 79–84. ACM, 2008.

[132] A. Lakshman and P. Malik. Cassandra : A structured storage system on a p2p network. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 47–47. ACM, 2009.

[133] Martin Landers, Han Zhang, and Kian-Lee Tan. Peerstore : Better performance by relaxing in peer-to-peer backup. In *P2P '04 : Proceedings of the 4th International Conference on Peer-to-Peer Computing*, pages 72–79, Washington, DC, USA, August 2004. IEEE Computer Society.

[134] Y.T. Lee and K.T. Chen. Is server consolidation beneficial to mmorpg ? a case study of world of warcraft. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 435–442. IEEE, 2010.

[135] Y.T. Lee12, K.T. Chen, Y.M. Cheng, and C.L. Lei. World of warcraft avatar history dataset. 2011.

[136] Sergey Legtchenko, Sebastien Monnet, and Gael Thomas. Blue banana : resilience to avatar mobility in distributed mmogs. *DSN*, 0 :171–180, 2010.

[137] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. Deconstructing the kazaa network. In *Internet Applications. WIAPP 2003. Proceedings. The Third IEEE Workshop on*, pages 112–120. IEEE, 2003.

[138] Qiao Lian, Wei Chen, and Zheng Zhang. On the impact of replica placement to the reliability of distributed brick storage systems. In *ICDCS '05 : Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 187–196, Washington, DC, USA, June 2005. IEEE Computer Society.

[139] J. Liang, R. Kumar, and K.W. Ross. The fasttrack overlay : A measurement study. *Computer Networks*, 50(6) :842–858, 2006.

[140] J. Liebeherr, M. Nahas, and W. Si. Application-layer multicasting with delaunay triangulation overlays. *Selected Areas in Communications, IEEE Journal on*, 20(8) :1472–1488, 2002.

[141] H. Lim, B. Fan, D.G. Andersen, and M. Kaminsky. Silt : A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM, 2011.

[142] J.C.S. Lui and MF Chan. An efficient partitioning algorithm for distributed virtual environment systems. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3) :193–211, 2002.

[143] Harsha V. Madhyastha, Thomas E. Anderson, Arvind Krishnamurthy, Neil Spring, and Arun Venkataramani. A structural approach to latency prediction. In Jussara M. Almeida, Virgílio A. F. Almeida, and Paul Barford, editors, *Internet Measurement Conference*, pages 99–104. ACM, 2006.

[144] P. Maymounkov and D. Mazieres. Kademlia : A peer-to-peer information system based on the xor metric. *Peer-to-Peer Systems*, pages 53–65, 2002.

[145] S. Milgram. The small world problem. *Psychology today*, 2(1) :60–67, 1967.

[146] J. Miller. Distributed virtual environment scalability and security. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games*, 2011.

[147] J.L. Miller and J. Crowcroft. Avatar movement in world of warcraft battlegrounds. In *Proceedings of the 8th annual workshop on Network and systems support for games*, page 1. IEEE Press, 2009.

[148] J.L. Miller and J. Crowcroft. The near-term feasibility of p2p mmog's. In *Proceedings of the 9th Annual Workshop on Network and Systems Support for Games*, page 5. IEEE Press, 2010.

[149] S. Monnet, R. Morales, G. Antoniu, and I. Gupta. Move : Design of an application-malleable overlay. In *Reliable Distributed Systems, 2006. SRDS'06. 25th IEEE Symposium on*, pages 355–364. IEEE, 2006.

[150] K.L. Morse et al. *Interest management in large-scale distributed simulations*. Information and Computer Science, University of California, Irvine, 1996.

[151] Rajeev Motwani and Prabhakar Raghavan. Randomized algorithms. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 141–161. CRC Press, 1997.

[152] M.A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the FREENIX Track : 1999 USENIX Annual Technical Conference*, pages 183–192, 1999.

[153] D. Ongaro, S.M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[154] L. Pantel and L.C. Wolf. On the suitability of dead reckoning schemes for games. In *Proceedings of the 1st workshop on Network and system support for games*, pages 79–84. ACM, 2002.

[155] S. Park, D. Lee, M. Lim, and C. Yu. Scalable data management using user-based caching and prefetching in distributed virtual environments. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 121–126. ACM, 2001.

[156] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1) :71–98, 2003.

[157] D. Pittman and C. GauthierDickey. A measurement study of virtual populations in massively multiplayer online games. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 25–30. ACM, 2007.

[158] D. Pittman and C. GauthierDickey. Characterizing virtual populations in massively multiplayer online role-playing games. *Advances in Multimedia Modeling*, pages 87–97, 2010.

[159] A. Ploss, S. Wichmann, F. Glinka, and S. Gorlatch. From a single-to multi-server online game : a quake 3 case study using rtf. In *Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*, pages 83–90. ACM, 2008.

[160] L. Rai and G. Yan. Future perspectives on next generation e-sports infrastructure and exploring their benefits. *International Journal of Sport Science and Engineering*, 3(01) :027–033, 2009.

[161] S. Ramabhadran, S. Ratnasamy, J.M. Hellerstein, and S. Shenker. Brief announcement : prefix hash tree. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 368–368. ACM, 2004.

[162] A.H. Rasti, D. Stutzbach, and R. Rejaie. On the long-term evolution of the two-tier gnutella overlay. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–6. IEEE, 2006.

[163] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[164] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the 2004 USENIX Technical Conference, Boston, MA, USA*, June 2004.

[165] I. Rhee, M. Shin, S. Hong, K. Lee, S.J. Kim, and S. Chong. On the levy-walk nature of human mobility. *IEEE/ACM Transactions on Networking (TON)*, 19(3) :630–643, 2011.

[166] M. Ripeanu. Peer-to-peer architecture case study : Gnutella network. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 99–100. IEEE, 2001.

[167] R. Rivest. The md5 message-digest algorithm. 1992.

[168] Rodrigo Rodrigues and Charles Blake. When multi-hop peer-to-peer lookup matters. In *IPTPS '04 : Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, pages 112–122, San Diego, CA, USA, February 2004.

[169] A. Rowstron and P. Druschel. Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[170] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 188–201. ACM, 2001.

[171] T.I. Salomie, I.E. Subasu, J. Giceva, and G. Alonso. Database engines on multicores, why parallelize when you can distribute ? In *Proceedings of the sixth conference on Computer systems*, pages 17–30. ACM, 2011.

[172] S. Saroiu, P.K. Gummadi, S.D. Gribble, et al. A measurement study of peer-to-peer file sharing systems. In *proceedings of Multimedia Computing and Networking*, volume 2002, page 152, 2002.

[173] Cristina Schmidt and Manish Parashar. Enabling flexible queries with guarantees in p2p systems. *IEEE Internet Computing*, 8(3) :19–26, 2004.

[174] R. Schmidt et al. Gridella : an open and efficient gnutella-compatible peer-to-peer system based on the p-grid approach. *Master's Thesis, Technical University of Vienna, October 2002*, 2002.

[175] F.B. Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys (CSUR)*, 22(4) :299–319, 1990.

[176] F.B. Schneider. Replication management using the state machine approach. *Distributed systems*, 2 :169–198, 1993.

[177] Shervin Shirmohammadi and Carsten Griwodz, editors. *10th Annual Workshop on Network and Systems Support for Games, NetGames 2011, Ottawa, Ontario, Canada, October 6-7, 2011*. IEEE, 2011.

[178] M.K. Sinha, PD Nandikar, and SL Mehndiratta. Timestamp based certification schemes for transactions in distributed database systems. In *ACM SIGMOD Record*, volume 14, pages 402–411. ACM, 1985.

[179] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *Network Computing and Applications, 2001. NCA 2001. IEEE International Symposium on*, pages 298–309. IEEE, 2001.

[180] V. Srinivasan and B. Bulkowski. Citrusleaf : A real-time nosql db which preserves acid. *Proceedings of the VLDB Endowment*, 4(12), 2011.

[181] M. Steiner and E.W. Biersack. Ddc : a dynamic and distributed clustering algorithm for networked virtual environments based on p2p networks. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–6. IEEE, 2006.

[182] R.C. Steinke and G.J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM (JACM)*, 51(5) :800–849, 2004.

[183] J.Q. Stewart and W. Warntz. Physics of population distribution. *Journal of regional science*, 1(1) :99–121, 1958.

[184] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4) :149–160, 2001.

[185] M. Stonebraker. Technical perspective-one size fits all : An idea whose time has come and gone. *Communications of the ACM*, 51(12) :76, 2008.

[186] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202. ACM, 2006.

[187] M. Suznjevic, O. Dobrijevic, and M. Matijasevic. Hack, slash, and chat : a study of players' behavior and communication in mmorpgs. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games*, page 2. IEEE Press, 2009.

[188] M. Suznjevic, I. Stupar, and M. Matijasevic. Mmorpg player behavior model based on player action categories. In *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games*, page 6. IEEE Press, 2011.

[189] P. Svoboda, W. Karner, and M. Rupp. Traffic analysis and modeling for world of warcraft. In *Communications, 2007. ICC'07. IEEE International Conference on*, pages 1612–1617. Ieee, 2007.

[190] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proceedings of the VLDB Endowment*, 2(1) :706–717, 2009.

[191] Robbert van Renesse. Efficient reliable internet storage. In *WDDDM '04 : Proceedings of the 2nd Workshop on Dependable Distributed Data Management*, Glasgow, Scotland, October 2004.

[192] M. Varvello, E. Biersack, and C. Diot. Dynamic clustering in delaunay-based p2p networked virtual environments. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 105–110. ACM, 2007.

[193] M. Varvello, C. Diot, and E. Biersack. P2p second life : experimental validation using kad. In *INFOCOM 2009, IEEE*, pages 1161–1169. IEEE, 2009.

[194] M. Varvello, C. Diot, and E. Biersack. A walkable kademlia network for virtual worlds. In *INFOCOM Workshops 2009, IEEE*, pages 1–2. IEEE, 2009.

[195] M. Varvello, S. Ferrari, E. Biersack, and C. Diot. Distributed avatar management for second life. In *Network and Systems Support for Games (NetGames), 2009 8th Annual Workshop on*, pages 1–6. IEEE, 2009.

[196] M. Varvello, F. Picconi, C. Diot, and E. Biersack. Is there life in second life ? In *Proceedings of the 2008 ACM CoNEXT Conference*, page 1. ACM, 2008.

[197] LLC VoltDB. Voltdb technical overview, 2010.

[198] S. Voulgaris, D. Gavidia, and M. Van Steen. Cyclon : Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2) :197–217, 2005.

[199] D.J. Watts and S.H. Strogatz. Collective dynamics of "small-world" networks. *nature*, 393(6684) :440–442, 1998.

[200] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 464–474. IEEE, 2000.

[201] A. Yahyavi, K. Huguenin, and B. Kemme. Antreckoning : dead reckoning using interest modeling by pheromones. In *Network and Systems Support for Games (NetGames), 2011 10th Annual Workshop on*, pages 1–6. IEEE, 2011.

[202] 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 the international workshop on Network and operating systems support for digital audio and video*, pages 99–104. ACM, 2005.

[203] Thomas Zahn and Jochen H. Schiller. Dht-based unicast for mobile ad hoc networks. In *PerCom Workshops*, pages 179–183. IEEE Computer Society, 2006.

[204] B.Y. Zhao, J. Kubiatowicz, A.D. Joseph, et al. Tapestry : An infrastructure for fault-tolerant wide-area location and routing. 2001.

[205] G.K. Zipf. Human behavior and the principle of least effort. 1949.

[206] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23 :337–343, 1977.

[207] Gzip. http://www.gzip.org/.

[208] NoSQL DBMS. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/Home\%20Page.

Index

Sergey LEGTCHENKO

Exploiting player behavior in distributed architectures for online games

Abstract

Massively Multiplayer Online Games (MMOGs) are a very popular class of distributed systems with more than 20 millions of active users worldwide. MMOG have strong applicative requirements in terms of data consistency, persistence, responsiveness and scalability. Shaped by the behavior of the players in-game, MMOG workloads are data-intensive and hardly predictable. Despite extensive research in the area, none of the currently existing architectures is able to fully satisfy all the requirements in presence of such complex workloads.

This thesis investigates the ability of MMOG architectures to better accomodate the workload by monitoring player activity at runtime. By doing that, the system is able to detect evolutions that are hard to foresee at startup, and dynamically allocate ressources to handle the load. We describe different techniques of runtime player monitoring and propose mechanisms to incorporate user behavior in the architectural design of MMOGs. Our experimentations are based on realistic workloads and show that our mechanisms have negligible overhead and improve global performances of MMOG distributed architectures.

Résumé

Durant la dernière décennie, Les jeux massivement multijoueurs (MMOGs) sont devenus extrêmement populaires et comptent désormais plus de 20 millions d'utilisateurs actifs à travers le monde. Les MMOGs sont des systèmes distribués ayant des contraintes applicatives fortes en terme de cohérence de données, persistance, réactivité et passage à l'échelle. L'évolution des besoins applicatifs du MMOG au cours du temps est difficilement prévisible car dépendante du comportement des joueurs dans le monde virtuel. C'est pourquoi, malgré un important effort de recherche dans le domaine, aucune des architectures proposées ne satisfait pleinement toutes les contraintes requises.

Cette thèse explore les capacités des architectures distribuées à s'adapter à la charge applicative grâce à une prise en compte du comportement des joueurs lors de l'exécution. Le système est alors capable de détecter des évolutions qui sont difficiles à prévoir à priori, et dynamiquement allouer les ressources nécessaires à l'application. Nous décrivons différentes techniques de surveillance des joueurs et proposons des moyens de prendre en compte ces informations au niveau de l'architecture. Nos expériences, effectuées dans des conditions réalistes, montrent que nos mécanismes ont un surcoût limité et permettent d'améliorer les performances globales du système.