# Nomad: A Pithos Based P2P Distributed Storage Network Implementation

I. T. de Villiers
17502292


Thesis presented in partial fulfilment of the requirements for the degree of Master in Engineering (Electronic) in the Faculty of Engineering at Stellenbosch University.

Supervisor: Prof H. A. Engelbrecht
Department of Electrical and Electronic Engineering

October 2021

# Acknowledgements

Throughout the course of this project, I have received a great deal of support and assistance.

I would first like to thank my supervisor, Professor Herman Engelbrecht, who secured funding for this project and provided me with support and guidance throughout the last three years, never losing faith in me.

I would like to acknowledge my colleagues at bol.com for their incredible support. A special thanks to Niels Basjes, who helped me to refine the scope of my project and provided me with invaluable guidance. To my team, Nathalie Neesen, Annelies Schreuder Hoogveld, Nick Tinnemeier, Dennis de Bode and Sajid Mohideen, without your support, guidance and not to mention all the extra thesis time, I would not have been able to succeed - thank you.

In addition, I would like to thank my parents, Francois and Marietjie, for always believing in me and their relentless support. A special thanks to my brothers Ian, Francois and Jacques, for their support, guidance and sympathetic ear. Lastly, I would like to thank my friends, especially Hanko and Karla Swart, for their support and providing me with much needed distraction through the last few years.

To my incredible partner, Clara Chennells, for all your love, support and understanding, I am truly grateful.

UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

# Plagiaatverklaring / *Plagiarism Declaration*

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
   *Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.*

2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
   *I agree that plagiarism is a punishable offence because it constitutes theft.*

3. Ek verstaan ook dat direkte vertalings plagiaat is.
   *I also understand that direct translations are plagiarism.*

4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
   *Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism*

5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
   *I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.*

| | |
|---|---|
| Studentenommer / *Student number* | Handtekening / *Signature* |
| Voorletters en van / *Initials and surname* | Datum / *Date* |

# Abstract

Massively multi-user virtual environments (MMVEs) can be defined as virtual environments where thousands of users are able to simultaneously interact with one another or with a virtual world. A popular implementation of an MMVE is massively multi-user online games (MMOGs/MMOs), such as *World of Warcraft* (WoW). In order for an MMO to be successful, it needs to be reliable, responsive, scalable, secure, and fair. In recent years, there has been much research and development surrounding distributed systems, in particular peer-to-peer (P2P) systems, although the topic of P2P MMVEs remains comparatively unexplored.

Research by Gilmore and Engelbrecht in 2013 identified an aspect of P2P MMVEs that had not received sufficient attention, namely *state consistency*. This observation led to the creation of a novel state management and persistence (SMP) architecture, called Pithos, specifically designed to satisfy the key requirements of P2P MMVE storage.

This study uses Nomad, a P2P distributed storage network (DSN) based on the Pithos architecture, to verify the simulated results obtained by Pithos, in a real-world environment. In order to verify the Pithos architecture, the key functional requirements of Pithos were identified and used to design a reliable, responsive, scalable, secure, and fair DSN, which was then implemented as a standalone Java application.

After evaluating Nomad against Pithos, it was found that both systems satisfy the storage requirements of P2P MMVEs. Nomad was found to be reliable, responsive and secure, and although scalability and fairness were not explicitly tested, these requirements were inherently satisfied due to Nomad's scalable components and load-balancing techniques. The evaluation of Nomad further proved the accuracy of the Pithos results, and indicated that Pithos may be a suitable storage architecture for P2P MMVEs.

# Uittreksel

Massiewe multigebruiker-virtuele omgewings (MMVO's) kan gedefinieer word as virtuele omgewings waarin duisende gebruikers gelyktydig met mekaar of met 'n virtuele wêreld in wisselwerking kan wees. 'n Gewilde voorbeeld van MMVO's is massiewe multigebruiker-aanlynspeletjies (MMA's) soos *World of Warcraft* (WoW). Vir 'n MMA om suksesvol te wees, moet dit betroubaar, responsief, skaalbaar, veilig en regverdig wees. Die afgelope paar jaar is daar baie navorsing en ontwikkeling gedoen oor sg. verspreide stelsels, veral ewekniestelsels ("peer-to-peer"/P2P), maar eweknie-toepassings spesifiek van MMVO's is nog nie tot dusver ondersoek nie.

Navorsing deur Gilmore en Engelbrecht in 2013 het een aspek van eweknie-MMVO's geïdentifiseer wat nog nie voldoende aandag ontvang het nie, naamlik *toestandsbehoud*. Hierdie waarneming het gelei tot die skepping van 'n nuwe argitektuur vir toestandsbestuur en -behoud, genaamd Pithos, wat spesifiek ontwerp is om die sleutelvereistes vir 'n eweknie-MMVO te bevredig.

Hierdie studie gebruik Nomad, 'n eweknie- gedesentraliseerde bergingsnetwerk gebaseer op die Pithos-argitektuur, om Pithos se gesimuleerde resultate in die regte lewe te bevestig. Ter stawing van die Pithos-argitektuur is die belangrikste funksionele vereistes daarvan geïdentifiseer en gebruik om 'n betroubare, responsiewe, skaalbare, veilige en regverdige bergingsnetwerk te ontwerp, wat toe as 'n alleenstaande Java-toepassing geïmplementeer is.

Nadat Nomad teenoor Pithos beoordeel is, het dit duidelik geblyk dat albei stelsels aan die bergingsvereistes vir 'n eweknie-MMVO voldoen. Dit is bevind dat Nomad betroubaar, responsief en veilig is. Alhoewel skaalbaarheid en regverdigheid nie uitdruklik getoets is nie, voldoen Nomad inherent aan hierdie vereistes as gevolg van sy skaalbare komponente en lasbalanseringstegnieke. Hierdie evaluering van Nomad staaf dan ook die akkuraatheid van die Pithos-resultate, en dui daarop dat Pithos moontlik 'n geskikte bergingsargitektuur vir eweknie-MMVO's kan wees.

# Contents

# List of Figures

# List of Tables

# Nomenclature

### Variables and functions

| | |
|---|---|
| $\mathcal{R}$ | Replication factor |
| $\mathcal{O}(N^2)$ | Big $O$ notation - quadratic complexity |
| $\mathcal{O}(1)$ | Big $O$ notation - constant complexity |
| $\mathcal{O}(logN)$ | Big $O$ notation - logarithmic complexity |
| $(\mathcal{R}/2)+1$ | Nomad quorum algorithm |
| $reliability = \frac{successful\ responses}{total\ requests}$ | System reliability |
| $future = current \times 2^{\left(\frac{num\ years}{2}\right)}$ | Moore's law (prediction) |

**Acronyms and abbreviations**

| | |
|---|---|
| AI | artificial intelligence |
| AoI | area of Interest |
| AWS | amazon Web Services |
| C/S | client/Server |
| CAST | group anycast and multicast |
| CPU | central processing unit |
| DCS | decentralised cloud storage |
| DHT | distributed hash table |
| DAO | data access object |
| DSN | decentralised storage network |
| GCP | Google Cloud Platform |
| gRPC | Google remote procedure call |
| HA | high availability |
| HTTP | Hypertext Transfer Protocol |
| IaaS | infrastructure as a service |
| ID | identifier |
| IDL | interface definition language |
| IM | interest management |
| IP | Internet Protocol |
| JAR | ava ARchive |
| K8s | Kubernetes |
| KBR | key-based routing |
| LAN | local area network |
| MiB | mebibyte |
| MMVE | massively multiplayer virtual environment |
| MMOG (MMO) | massively multiplayer online Game |
| ms | milliseconds |
| NIO | non-blocking input/output |
| NPC | non-player character |
| OSI | Open Systems Interconnection |

| | |
|---|---|
| P2P | peer-to-peer |
| PB | petabytes |
| PoC | proof-of-concept |
| POJO | Plain Old Java Object |
| Protobuf | protocol buffers |
| QoS | quality of service |
| RAM | random access memory |
| RDB | relational database |
| REST | representational state transfer |
| RPC | remote procedure call |
| RPS | requests per second |
| Rq | request |
| Rs | response |
| RTD | round-trip delay |
| RTL | round-trip Latency |
| SMA | simple Moving Average |
| SMP | state management and Persistence |
| TB | terabyte |
| TCP | Transport Control Protocol |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| TTL | time-to-live |
| UDP | User Datagram Protocol |
| UML | Unified Modeling Language |
| VE | virtual environment |

# Chapter 1

# Introduction

## 1.1. Background

Massively multi-user virtual environments (MMVEs) can be defined as virtual environments (VEs) where thousands of users are able to simultaneously interact with one another or with a virtual world [1]. A popular implementation of an MMVE is massively multi-user online games (MMOGs/MMOs), such as *World of Warcraft* (WoW) [2]. In order for an MMO to be successful, it needs to be reliable, responsive, scalable, secure, and fair [3]. In recent years, there has been much research and development surrounding distributed systems, in particular peer-to-peer (P2P) systems; however, the topic of P2P MMVEs remains comparatively unexplored.

P2P architectures, due to their distributed nature, are able to solve many of the challenges faced when designing an MMO. However, the benefits of using a distributed architecture come at the cost of, amongst other things, added complexity to state consistency and scaling challenges [3]. This work reviews the key design challenges of P2P MMVE architectures. From the identified design challenges, this project identifies state consistency as an interesting challenge in P2P MMVEs. This work presents the design and verification of a P2P storage network, specifically designed for P2P MMVEs. The proposed system aims to satisfy the state consistency requirement of P2P MMVEs, by making use of modern development techniques combined with established P2P storage architectures.

## 1.2. Virtual Environments

In the context of this project, a virtual environment (VE) can be defined as a digital world consisting of various objects, characterised by an environment state [1]. In MMOs, VE objects can be categorised into four object types, namely *immutable objects*, *mutable objects*, *player characters/avatars* and *non-player characters* (NPCs).

Immutable objects are objects in the VE that hold a static state and cannot be altered in any way by players or NPCs. Examples of immutable objects include world terrain or any object that does not require any logic. Mutable objects are objects that may be

altered via interaction from a player or a NPC. Mutable objects are accompanied by game logic, which ensures any interaction is handled deterministically and follows the logic of the VE. Avatars are special objects that are player controlled and allow players to interact with and alter the VE. Game logic is specifically catered for avatars, to ensure that their interactions are secure, deterministic and abide by the game logic. NPCs are characters, typically controlled by the game logic or artificial intelligence (AI), that serve as a means to make the VE a more interactive for players [3].

## 1.3. Massively Multi-user Virtual Environments

MMVEs were previously introduced as VEs where thousands of users are able to simultaneously interact with one another or with a virtual world. In the online gaming world, MMVE are extremely popular with the implementation of MMOs. One of the most popular MMOs to date is *World of Warcraft* [2], with an estimated peak of over 12 million subscribers in 2010 [4], and an estimated subscriber count of 4.47 million player in 2021 [5], [1]. This serves as an example of how successful MMVEs can be, if implemented correctly.

Design requirements that need to be taken into account when designing an MMO include:

1. Environment logic, which allows players interaction.
2. State consistency, which ensure all users have a consists view of the VE
3. Object replication, which is used to distribute the game logic and resources across multiple machines.
4. Interest Management (IM), which determines which interactions and events are important to specific users.
5. An underlying network architecture, which allows users to connect to the VE.

### 1.3.1. User Interactions

Users in an MMVE typically interact either with the VE or with other users. The environment logic is therefore required to support multiple user interactions. Example of typical player interactions include user updates, which are only of interest to the user themselves; player-to-player updates, which impact the user and the users they interact with; and player-to-object updates, which impact the VE [3].

### 1.3.2. State Consistency

Any VE typically consists of two logical states, a global state and a local state. The global state ensures that players' data and previous interactions are persisted. In contrast,

the local state is used for display purposes and allows for low latency user and player-to-object updates. As users connected to the MMVE are geographically dispersed, a state consistency architecture is required. State consistency ensures a consistent global state and defines the criteria and scope of the local state [1], [3]. State consistency is discussed in more detail in Chapter 2.

### 1.3.3. Object Replication

Players' local VE state can also be seen as copies or replicas of the global state. This means that each object has a master copy or *authoritative object*, typically stored on the global state, and a replicated copy or *replica*, stored in a user's local state. Seeing as MMVEs are extremely large and players are assumed to have a limited frame of reference, it is desired that only a portion of the world be stored on each user. This requires an object replication strategy and interest management (IM), which determines which objects are stored where [3].

### 1.3.4. Interest Management

IM defines a set of rules, by which the VE is able to determine which interactions affect which players. As players are assumed to have a limited frame of reference, a subset of VE can be made available to a user, depending on their location in the VE or other defining criteria. IM allows for better resource management, and has led to many useful segmenting approaches, which when applied to MMVEs, can improve overall performance, simplify logic and improve networking strategies [3].

### 1.3.5. Networking

All MMVEs require a networking layer, which allows users to connect to the virtual environment. Most successful MMVEs are implemented using a classic C/S architecture. This choice of architecture is mainly due to the simplicity of network administration and security provided by C/S systems [1].

As illustrated by Figures 1.1, 1.2 and 1.3, many different C/S implementations exist. The C/S model has its limitations when used as network architecture in MMVEs, specifically when it comes to population balancing and server size. Scaling in a C/S system means adding more resources to the system. Whether it is scaling vertically, by adding more CPU or RAM, or scaling horizontally, by adding more server instances, maintaining and scaling C/S systems are expensive. With the progress made in containerisation and system orchestration, with tools like Docker [6] and Kubernetes [7], the challenge of scaling C/S systems in the cloud are significantly reduced. However, the cost at which this comes is still cause for concern.

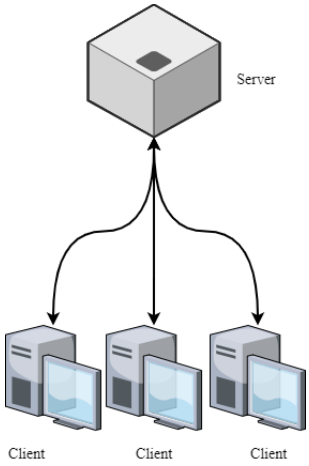


**Figure 1.1:** C/S model

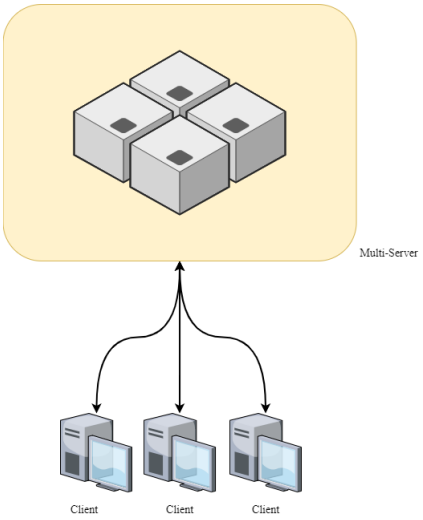


**Figure 1.2:** C/MS model

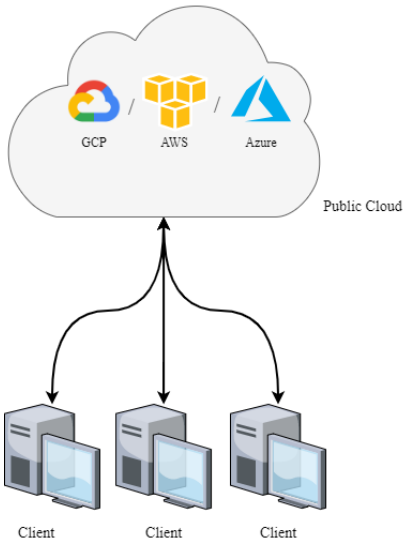


**Figure 1.3:** Cloud C/S model

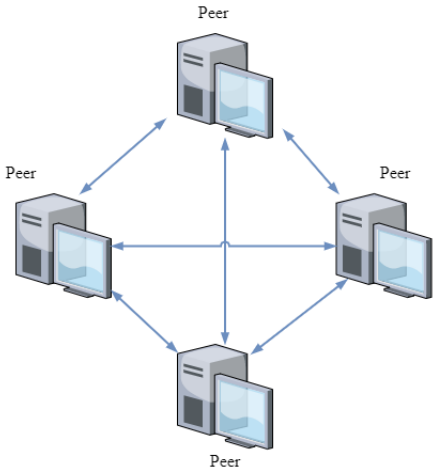


**Figure 1.4:** Peer-to-Peer model

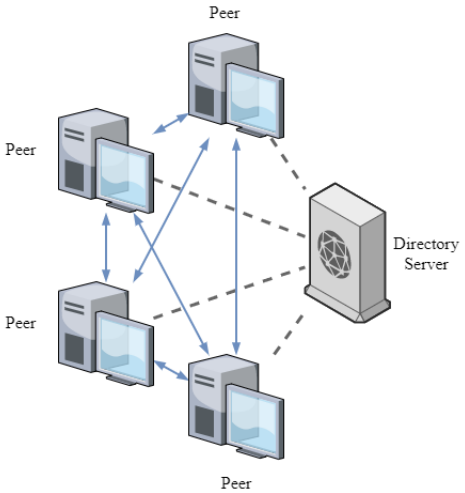


**Figure 1.5:** Hybrid peer-to-peer model

The P2P model, illustrated by figures 1.4 and 1.5, are especially interesting for MMVEs, since they solve many of the challenges of the C/S model. A comparison between C/S, P2P and hybrid P2P systems is further discussed in Section 1.4.1.

### 1.3.6. Requirements of Massively Multi-user Online Games

Gilmore and Engelbrecht [1] summarise the classic single player game design requirements as the following: a graphics and physics engine, storage mechanisms, NPCs, game logic, sound and music and networking.

MMOs are required to satisfy the same key requirements of classic single player games. However, due to the scale of the virtual environment and large user count, the networking infrastructure is far more complex and scalability is crucial. One of the greatest challenges of MMVEs is that a large number of users should be able to interact with one another in the VE in a consistent and deterministic manner. This requirement is called the *consistency architecture* [1], which ensures all users share an identical view of the virtual world they inhabit.

The consistency architecture is responsible for interest management by relaying actions of other clients to those who are impacted or have a view of the manipulated, added or removed objects. This is known as *state management*. In addition to ensuring consistency between users, the consistency architecture is also responsible for object and information persistence, which ensures long term storage of data. An example of this is a user's avatar and world state. Between sessions, a user's profile should remain consistent, as well as the state of the VE before session termination. This is known as *state persistence* [1].

Another way of describing the responsibilities of the consistency architecture, is in terms of a non-authoritative and authoritative object store. The non-authoritative store is responsible for ensuring consistency between users and the world, i.e. for *state management*. Users normally hold a non-authoritative view of the world and other users. The authoritative store is responsible for ensuring consistency between sessions, or, if two users disagree on world state, the authoritative mechanism is used as the true state. The authoritative object store is therefore responsible for *state persistence* [1].

### 1.3.7. C/S MMVE Network Architectures

As previously discussed, most modern MMVEs, specifically MMOs, are built on a C/S network architecture. In the C/S MMVE architecture, the server contains the authoritative object store and will be in charge of state persistence of the VE. The server is also responsible for relaying connections between users and settling disagreements between users. Clients manage their own non-authoritative object store, and process all requests locally using the game/VE logic and their local copy of the VE. In section 1.4.1 the advantages and disadvantages of C/S systems are briefly discussed, from which we can conclude that

security, robustness, server crowding, high operating costs and scaling are weaknesses of C/S MMVEs.

### 1.3.8. The Peer-to-Peer MMVE Network Architecture Proposal

P2P MMVEs are relatively unexplored, due to the added complexity that P2P networking introduces. However, implementing MMVEs with P2P networking architecture has the potential to solve many of the issues relating to C/S MMVEs. P2P MMVE implementations have better robustness, scaling, lower operating costs and improved latencies when compared to C/S MMVEs [1].

## 1.4. Peer-to-Peer Systems

The term "Peer-to-Peer" (P2P) is certainly not a new concept; it is often used in many contexts to mean different things. Only with the appearance of the music and file sharing application Napster [8] in the 1990s, the term "peer-to-peer" was first used to describe a decentralised network architecture [9]. The P2P model is fundamentally different to the well known client-server (C/S) model. P2P networks can be defined as systems for which all content, services and other resources, are provided by the peers that form the P2P network. A peer in this context, is often just an application running on a machine. In contrast to a C/S networks, a peer can both serve content to other peers and request content from other peers in the network. Participants in a P2P network can access resources directly from other peers with little to no centralisation [10].

### 1.4.1. Peer-to-Peer versus Client-Server Model

The C/S model is probably the most well known network architecture. Its popularity is mostly due to its centralised approach, which simplifies network administration and security. The C/S model can be defined as a network architecture where participants are either a *client* or a *server*. Clients primarily generate requests for services or resources, whereas servers provide the requested resources or functionality. Table 1.1 provides a summary of some fundamental differences between P2P and C/S systems. In reality, all three models have their strengths and weaknesses depending on the use case.
Additionally, hybrid P2P networking approaches exist, which may inherit the advantages of both P2P and C/S approaches. Hybrid network models, like centralised P2P and hybrid P2P, allow certain system components or functionality to be centralised in order to simplify network administration. In centralised P2P for example, a single bootstrap or directory server is used to bootstrap peers to the P2P network. Similarly, hybrid P2P models may promote certain peers to act as centralised authoritative nodes, which facilitate various network operations and optimise routing within the network [10].

| Criteria | Client/Server | Peer-to-Peer | Hybrid |
|---|---|---|---|
| Participants | Clients and servers | Peer | Peers and centralised servers |
| Hierarchy | The network has a clear hierarchy: either a client, which generates requests, or a server, which servers requests. | All participants in the network are seen as equals, and can both generate requests to others or serve requests from other peers. | A clear distinction is made between centralised components and peers in terms of responsibility within the network. Centralised components are generally used to optimise network administration or to allow bootstrapping [10]. |
| Scalability | Weakly scalable, forced to scale vertically (up) or horizontally (out) [11]. | P2P networks are considered scalable. | Hybrid P2P networks are scalable, but may be bottlenecked by their centralised components. |
| Security | Security is easy to maintain and enforce. Security is generally weak, seeing that if a server is compromised, all clients are potentially at risk. Risk can however be mitigated if proper procedure and security standards are followed. | Security is more complex as peers do require additional security mechanisms for authentication, authorisation and traffic encryption. P2P security is strong, as networks require a small number of functional peers to ensure security. | Hybrid P2P approaches are slightly easier to maintain from a security standpoint, due to a degree of centralisation. However, a single point of entry generally sees a higher number of hacking attempts in comparison to a fully distributed approach. |
| Robustness | C/S systems are generally less robust, as they have a single point of failure. Robustness can easily be improved with High-availability (HA) techniques. | P2P systems are robust and fault-tolerant as they require a small number of functional peers to ensure correct functionality. | Hybrid P2P systems are generally just as robust as P2P systems, although centralised components require additional fail-over mechanisms to enable disaster recovery. |
| Resource distribution | A server(s) provides *all* the computational resources within the network. | Peers contribute their resources to host themselves within the network. Additionally, computational load is distributed amongst peers. | Peers contribute enough resources to host themselves in the network, whereas centralised components provide all the resources to fulfil its network functionality. Computational load is distributed amongst peers and centralised components. |
| Running costs | High maintenance costs. | Extremely low maintenance costs. | Might require some maintenance costs. |

**Table 1.1:** Summary of peer-to-peer, client-server and hybrid network models.

### 1.4.2. Structured and Unstructured P2P Overlays

It is generally understood that P2P networks establish an overlay network, mostly based on TCP or HTTP connections. An overlay network does not reflect the physical connections, due to the abstraction layer of the TCP protocol stack [10], as indicated by Figure 1.6.

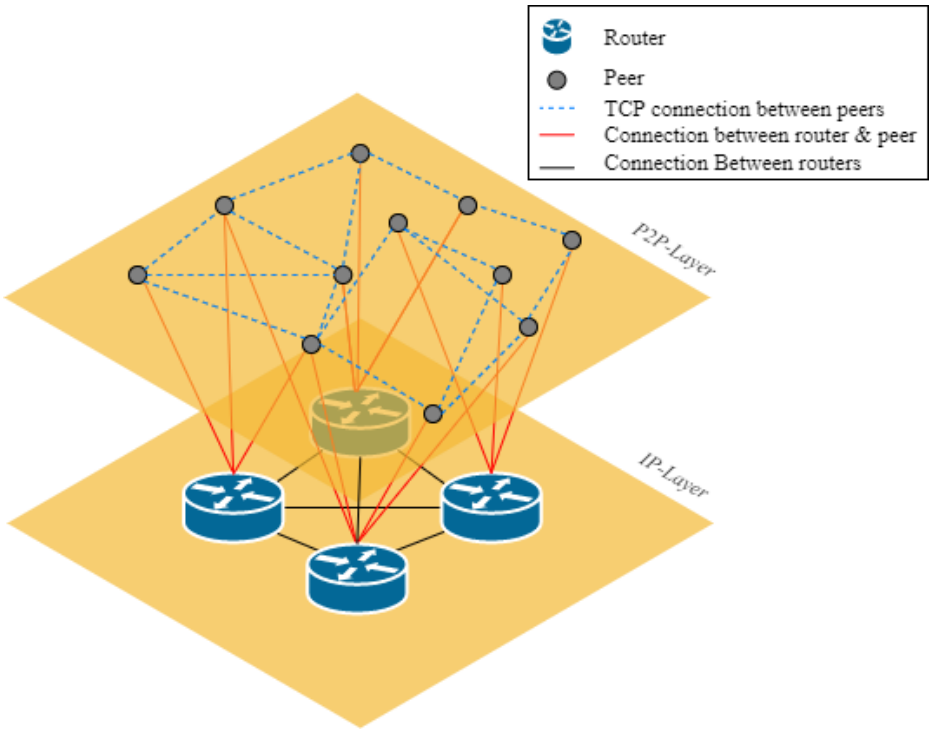


**Figure 1.6:** Schematic view of physical and virtual overlay network topology [10].

According to Aberer et al. [12], P2P overlays can generally be classified into two categories, namely structured and unstructured overlays. Each overlay network is characterised by the decisions made on six key design aspects, namely definition of the identifier space, mapping of resources and peers within the identifier space, network topology, routing strategy, and maintenance strategies and mechanisms.

Design decisions made in these six design aspects impact the overall performance and stability of the overlay network in terms of [12];

- **Efficiency**: Bandwidth usage for constructing and maintaining the overlay.
- **Scalability**: The number of peers in the overlay should not have a significant effect on performance.
- **Self-organisation**: The network should reorganise itself to a stable configuration under network churn.
- **Fault tolerance**: The overlay should still maintain basic functionality regardless of partial network failure.

- **Cooperation**: Peers within the network should maintain a degree of security and trust between each other.

#### 1.4.2.1. Features of Structured Overlays

Structured overlays map keys to overlay nodes, where each node can be identified by a specific *nodeId* selected from an identifier space. Similarly, objects within the overlay network are assigned keys from the same identifier space. A common application programming interface (API) exists for structured overlays namely the key-based routing (KBR) [13] API. Distributed hash tables (DHTs) like Kademlia [14] and Chord [15] are good examples of structured overlays [16]. Appendix A provides more information on the KBR API and how it is implemented to satisfy use cases.

#### 1.4.2.2. Features of Unstructured Overlays

In contrast to structured overlays, unstructured overlays are more flexible in terms of node relationships and resource lookup. Unstructured overlays organise nodes into a random graph, and each node maintains a neighbour table with the network addresses of its neighbours. Neighbour tables are symmetric, which means that if *peer-x* contains *peer-y* in its neighbour table, *peer-y*'s neighbour table will contain *peer-x*. This flexible relationship between nodes ensures a higher degree of resilience against node failure and network churn. Gnutella [17] is a good example of an unstructured overlay [16].

#### 1.4.2.3. Features of Peer-to-Peer Overlays

P2P Overlays typically have five distinctive design features that determine the overall resource usage, namely [1] [15]:

- **Network topology**: Determines the network structure and allows for deterministic routing. The topology also determines the available routing table maintenance strategies.
- **Routing algorithms**: Determines the amount of network hops required to deliver messages to a target node.
- **Join mechanism**: When a peer joins the overlay, the existing peers/neighbours need to be notified that a peer has joined the network. The join mechanism determines the manner in which a peer joins the P2P overlay network and the maintenance required on the routing table.
- **Leave mechanism**: When a peer leaves the overlay, the existing peers/neighbours need to be notified that a peer has left the network. The leave mechanism determines

the manner in which a peer leaves the P2P overlay network and the maintenance required on the routing table.

- **Bootstrapping mechanism**: Once a peer has joined the overlay network, the bootstrapping mechanism allows the peer to determine how it fits into the network, and ensures that a peer can become a functioning member of the P2P overlay.

### 1.4.3. Peer-to-peer Massively Multi-user Virtual Environments

As discussed in Section 1.4, P2P systems are completely distributed and, depending on the implementation, have a minimal level of centralisation. In P2P MMVE systems, individual peers contribute the required resources to the network to host itself, such as memory, processing power, storage and bandwidth [10]. Since peers act as both a server and a client, the function of the consistency architecture also becomes distributed amongst peers. This means that each peer will share the responsibility of the authoritative state of the environment; and, similarly to classic C/S MMVEs, peers are solely responsible for their own non-authoritative state [1].

#### 1.4.3.1. Advantages

- P2P MMVEs are considered robust, as a minimum number of peers are required to maintain system functionality [1].
- The system is also considered more secure, as object state and peer integrity can be verified against multiple peers in the network. This means that malicious peers that attempt to illegally alter the object state outside of the game logic parameters can easily be detected by other peers.
- P2P systems are also seen as scalable, as each peer in the network provides the required computational and storage resources to host itself [10].
- In contrast to C/S systems, from an operator perspective, operational costs for doing maintenance on the network are extremely low. Certain P2P systems still require some degree of centralisation, but usually centralised mechanisms hold no logic or state and therefore have low resource requirements [10]. Bandwidth requirements are also distributed amongst peers, which means that little to no bandwidth costs are incurred by operators.
- In P2P networks, peers can communicate directly with one another, instead of using centralised servers to access resources or interact with others. Less network hops are required to execute resource lookups or trigger logic events, allowing for a higher degree of responsiveness.

#### 1.4.3.2. Disadvantages

- Added complexity.
- Consistency becomes difficult to achieve without various synchronisation mechanisms.
- Since P2P networks are sensitive to network churn and MMVE networks are known to be high churn environments, a higher degree of replication is required.
- More object replicas require more sophisticated repair mechanisms to ensure replicas do not go missing due to network churn.
- Security in P2P systems is more complex. Cheating mitigation is therefore a challenge, since it becomes harder to identify users.
- MMVEs require highly responsive network interactions. Since two users connecting might have incredibly high latency between them, mechanisms are required to ensure that the highest performing connections are preferred.

### 1.4.4. Requirements and Key Challenges of P2P MMVEs

Design requirements for P2P MMVEs are similar to those of classic MMVEs. However, when looking at their respective key challenges, a clear distinction between the two architectures can be made. Since classic MMVEs are built on a C/S network architecture, many design requirements are trivial to implement. In contrast, P2P MMVEs are built on a distributed network architecture. This means many of the design requirements, become more complex and harder to manage. Previous research has identified the key design challenges pertaining to the design of a P2P MMVE architecture, as [1] [3]:

- Interest management.
- Event dissemination.
- Distributed computation amongst peers.
- State consistency.
- Security.
- Resource efficiency and incentive mechanisms.

Chapter 2 defines each of the key design challenges of P2P MMVEs and provides a more detailed discussion of each. From the identified key challenges, state consistency in P2P MMVEs has received the least research exposure [1]. The primary focus of this work is therefore to find a modern solution for state consistency and state persistence within P2P MMVEs.

## 1.5. State Consistency in P2P MMVEs

The *state consistency* architecture of an MMVE dictates how clients interact with one another and the VE. The state consistency architecture has two primary responsibilities, *state management* and *state persistence* [1]. These two concepts often go hand-in-hand and are therefore referred to as *storage* in this chapter.

### 1.5.1. Key Requirements for P2P MMVE Storage

In order to provide a robust solution for state consistency in P2P MMVEs, the requirements of such an architecture needs to be defined. Gilmore and Engelbrecht [1] define the key requirements for P2P MMVE storage as:

- Responsiveness
- Reliability
- Scalability
- Security
- Fairness

In order to ensure that potential storage solutions are suitable for P2P MMVE storage, each option will be evaluated in terms of these key requirements. In Chapter 2 these options are introduced and discussed.

## 1.6. Primary Project Objective

The primary objective of this work is to design, implement and evaluate a P2P distributed storage network that satisfies the key functional requirements of a P2P MMVE storage.

## 1.7. Thesis Overview

In Chapter 1, the various concepts relating to this project have been introduced and research objectives were motived. Chapter 2 introduces the design challenges and storage requirements of P2P MMVEs and multiple suitable storage options are introduced and discussed. Chapter 3 provides an overview of the Pithos architecture and the key challenges of P2P MMVEs. Chapter 4 provides a detailed description of the implemented DSN, called Nomad. Various design decisions are motivated and points that aim to improve the original Pithos architecture are described. In Chapter 6, the implemented system is evaluated and the various test scenarios and key test criteria for each are described. The test results are then visualised and evaluated. Chapter 6 also provides a comparison between Nomad's performance results and Pithos' simulated performance results. Finally, in Chapter 7 some

pertinent conclusions are presented. This last chapter also offers suggestions for future work for the Nomad DSN.

# Chapter 2

# State Consistency in Virtual Environments

In order to understand the Pithos P2P MMVE state management and persistence architecture, the challenges and requirements of such a system should first be understood. This chapter provides the required background information on P2P MMVE design challenges. More specifically these design challenges are related to the design challenges and key requirements of MMOGs [1]. As Pithos is specifically designed to solve the key challenges of persistence and state management in MMVEs, an overview of key P2P MMVE storage requirements is also provided. The key challenges and requirements identified in this section are later used to evaluate all proposed P2P MMVE storage models, introduced in Chapter 3.

## 2.1. Definition of State Management and Persistence

As mentioned in section 1.5, the *state consistency* architecture of an MMVE dictates how clients interact with one another and with the VE. The state consistency architecture has two primary responsibilities, *state management* and *state persistence* [1].

### 2.1.1. State Management

State management, also known as non-authoritative object storage, acts as the primary storage layer of objects within the system. State management is responsible for managing object state locally, usually in-memory. In P2P MMVEs, the state management layer is also used to resolve object state conflicts between peers. In this context, state refers to the state of the perceived VE world, with which all clients interact. Since the environment state is displayed to the user, state management requires real-time (responsive) storage and retrieval [1].

### 2.1.2. State Persistence

State persistence, also known as authoritative object storage, acts as a secondary storage layer in the system. State persistence refers to the long term storage of data, that persists regardless of system failure. State persistence is responsible for persisting crucial data, like player information, or fall-back world state, effectively acting as a disaster recovery layer. State persistence is not required to be highly responsive, as it is mostly used as long term and backup storage or conflict resolution. High storage and retrieval reliability is therefore a strict requirement for state persistence [1]. This requirement ensures that critical data is always stored successfully.

## 2.2. P2P MMVE Key Design Challenges

In order to provide the necessary background information, this section aims to identify and discuss the key design challenges for P2P MMVEs. Fan et al. [18] provide a comprehensive overview of the six key design issues for P2P MMOGs. Solving these six design issues will pave the way for a smooth transition from classic C/S MMOG architectures to P2P MMOG architectures. The key design issues are: interest management, event dissemination, NPC host allocation, game state persistence, cheating mitigation and incentive mechanisms. The following subsection will provide an overview of each of the aforementioned design issues. The specifics of the mentioned solutions are beyond the scope of this work and will therefore not be elaborated on.

### 2.2.1. Interest Management

One of the main challenges of distributed P2P MMOs is maintaining state consistency among all peers within the MMVE, without the use of major centralisation mechanisms. Peers within an MMO are interested in an immersive experience by interacting with each other and the world. This requires state consistency and interest management.

IM originates from two observations: (1) That a single player in a VE has a limited frame of reference and does not need to know about events that do not alter its state or view of the world; and (2) that a player has limited sensing and movement capability and is therefore limited to a fairly static area of interest (AoI) [18]. IM therefore dictates which objects peers are interested in and need to subscribe to in order to maintain consistency.

Multiple IM approaches exist that can be used to address IM in P2P MMVEs, namely spatial models [19], region based publish/subscribe models [20] and hybrid communication models [21]. The spatial model allows *fine-grained* IM, which means that peers know exactly which objects they are interested in and only receive updates for those objects. A drawback of fine-grained IM is that all objects need to exchange positional updates in order to avoid AoI collisions. This leads to excessive communication overhead. Region based

publish/subscribe models support *coarse-grained* IM, by partitioning the VE into static regions and only sending updated events to players in the region of interest. Coarse-grained IM holds several advantages over fine-grained IM, in that it is generally cheaper, more efficient to send to players, and allows players to perform local IM since AoI collisions are not of concern. Drawbacks of coarse-grained IM are that it does not work well if objects are not evenly distributed within the VE and that region size is hard to determine [18].

### 2.2.2. Event Dissemination

IM dictates which objects a peer subscribes to. Event dissemination refers to how object events are sent to the right peers within a VE. It is important to note that event dissemination relies on the underlying IM model used [18]. Event *unicast* [22] and *multi-cast* [23] have been proposed to address the event dissemination design issue.

When fine-grained IM is used, peers are usually interested in a small subset of objects. This means that a peer can form a direct connection to all objects they are interested in, and gaming events are exchanged through *unicast* communication. When coarse-grained IM is used, peers usually only know of the regions they need to subscribe to and not the specific objects. Regions are represented by multi-cast groups, which provides players with a single interface to publish events to. This in turn allows gaming events to be exchanged through *multi-cast* communication [18].

### 2.2.3. NPC Host Allocation

In MMOG virtual worlds, non-player characters (NPCs) or AI-controlled characters exist. These NPCs allow for an immersive experience, by either providing a user with continuous challenges or driving a story line. In classic MMOGs, NPCs are hosted on centralised servers that store all game logic and VE state. NPCs traditionally require special logic and therefore consume network resources to function.

A challenge for P2P MMOGs is deciding which peers should host which NPCs. NPCs might require significant resources and therefore need to be hosted by reputable peers. Multiple P2P MMVE host allocation approaches exist that address the issue of NPC host allocation. Possible solutions include region based [24], virtual distance based [25] and heterogeneous task sharing [26]. Multiple NPC host allocation approaches are often used in a single MMOG to address different scenario requirements.

The *region based* NPC host allocation approach partitions the world into several logical regions, each region being represented by a super-peer. The selected super-peer acts as the authoritative node, which hosts all the NPC objects within the region. Drawbacks of the region based approach include, unfair resource requirements for super-peers and that quality of service (QoS) cannot be guaranteed, since most super-peer selection strategies do not take host resource availability into consideration.

The *virtual distance based* NPC host allocation approach ensures that NPC objects are stored on the peer closest to it in the virtual world. This is based on the assumption that the closest peer to the NPC is the most likely to interact with it. Several sub-approaches include Voronoi distance based approaches, where each peer only hosts objects and NPCs within its Voronoi region. Drawbacks of this approach include expensive NPC host calculations and reduced security, since malicious peers have full authority over NPC objects.

The *heterogeneous task sharing* NPC host allocation approach distributes NPC objects amongst peers, based on their available computational and network resources. The heterogeneous task sharing model consists of three parties: workers, resource providers and matchmaker super-peers. Workers generate NPCs, resource providers share their additional resources and matchmaker super-peers manage workers and resource providers. This approach has proven to be more resource efficient than other approaches, although it still has some issues when global NPCs are required [26].

### 2.2.4. Game State Persistence

Another of the requirements of the Pithos architecture is to address game state persistence in P2P MMVEs. Game state persistence refers to the ability of a P2P MMVE to persist user information between active sessions. In MMOGs the world state does not remain static when a user is inactive. MMOGs therefore require mechanisms that store a user's previous session information, to ensure continuity.

The usage of a distributed storage infrastructure like overlay storage can be used to address the requirement of game state persistence. Overlay storage is discussed in more detail in section 3.3.2. P2P MMVE storage has specific requirements, discussed in Section 2.3, that distributed storage infrastructures do not yet meet. In 2009 it was noted by Fan et al. [18] that game state persistence in P2P MMVEs is still a relatively unexplored topic. In the present day, P2P MMVE state persistence has still not received much research attention.

Cloud storage options have become very popular and more affordable in recent years. One viable approach could be to separate the game persistence layer from the state management layer. For state management, P2P storage techniques can be used to ensure responsive and reliable object state management, whereas for object persistence a centralised cloud storage solution can be used. This hybrid approach is, however, not suitable for small operators, since as discussed in Section 2.5.3, MMVEs have demanding storage requirements and are expensive to host in the cloud.

### 2.2.5. Cheating Mitigation

Another challenge for P2P systems in general, is security. P2P systems are by definition distributed, and therefore have no central authority that controls or verifies system interactions. This poses a serious risk. In P2P MMOGs, malicious users might try to alter their own VE state and consequently the VE state of others, to gain an advantage [1].

To mitigate cheating in P2P MMOGs, multiple security mechanisms can be put in place, like redundancy and certification mechanisms. Cheating mitigation approaches can generally be classified as either preventative (proactive) or remedial (reactive).

### 2.2.6. Incentive Mechanisms

Incentive mechanisms refer to mechanisms that incentivise users to share their available bandwidth, storage, CPU and memory resources. These resources are in turn used for IM, event dissemination, state persistence, NPC hosting and cheating mitigation.

In modern blockchain-based systems like those discussed in Section 2.4, users are incentivised to continuously contribute to the P2P network through cryptocurrency micro payments, which hold real-world financial value. P2P MMOGs are voluntary by nature, and require some degree of cooperation in order to maintain stability. Users often have a disregard for the collective welfare, which means that P2P MMOGs are susceptible to uncooperative behaviour. Incentive mechanisms are therefore required to ensure cooperation [18].

## 2.3. P2P MMVE Storage Requirements

As previously stated, the focus of this work, is to provide state management and persistence in P2P MMVEs using the Pithos architecture. This section will define and elaborate on the key storage requirements for P2P MMVEs.

As described by Gilmore and Engelbrecht [1] [27], P2P MMVEs are required to be scalable, reliable, fair, secure, and responsive.

### 2.3.1. Scalability

Scalability is the most important requirement for P2P MMVEs, as it underpins all other requirements [27]. In order for a system to be considered scalable, performance should not diminish as the size of the network increases. The authors argue that overall system scalability is determined by the scalability of individual components. Testing all other acceptance criteria with a sufficiently large number of peers inherently takes scalability into account.

### 2.3.2. Responsiveness

Since MMOGs are real-time systems, storage and retrieval requests need to be handled in real time, i.e. within a specific latency range. Variance in response times are required to be small. Typically in RTS games, the latency is required to be less than a second [27].

Normoyle et al. [28] argue that in multi-player platform games, latencies up to 300 ms barely affect player experience. Only when latencies above 500 ms are present, player experience is significantly altered. An Ealier study suggests that fast paced games require latencies of below 100 ms, whereas third-person strategy games can tolerate latencies of up to 500 ms [29].

### 2.3.3. Reliability

Storage reliability is defined as the robustness and availability of resources stored within the system. Robustness refers to resilience against network churn, whereas availability means that an object should be available to any peer in the network with the correct permissions [27].

### 2.3.4. Security

Security in P2P MMVEs is crucial to the integrity of the network. Resources stored in the system should therefore be resilient against malicious peers that may attempt to alter their state and contradict the VE logic. The system should therefore be able to detect and identify malicious peers in the network, which requires certification mechanisms [27].

### 2.3.5. Fairness

P2P systems require distributed computing, as each peer is required to contribute their available resources to the network. Ensuring fairness in the system requires load to be distributed equally among all peers within the network. Fairness within the system also promotes high availability and resilience against network churn [27]. For the purpose of this work, fairness refers to the system's ability to balance load across multiple peers within the system. The terms load balancing and fairness are therefore used interchangeably.

## 2.4. Modern Peer-to-Peer Decentralised Storage Networks

In order to ensure that the chosen storage solution is suitable for P2P MMVE storage, each potential storage option will be evaluated in terms of the key requirements introduced in section 2.3. Subsequent sections introduce potential distributed storage options.

The unprecedented boom in the value of certain cryptocurrencies has put a spotlight on distributed systems and more specifically blockchain technology [30]. This invigoration of P2P systems has produced many decentralised storage networks (DSNs) built with blockchain technology. The following sections provide a brief introduction and assessment of potential DSNs, that could be used in P2P MMVEs.

### 2.4.1. Pithos: Distributed Storage for Massively Multi-user Virtual Environments

Pithos is a reliable, responsive, secure, load-balanced and scalable distributed storage system, designed specifically for P2P MMVEs. The Pithos architecture addresses deficiencies such as lack of load balancing, responsiveness, and scalability in previously reviewed state management and persistence (SMP) architectures. Pithos uses two different storage layers, namely **group storage** and **overlay storage**, to achieve low latency, object state management and persistence. Group storage splits the VE into various segments, and groups peers into logical geographical groups. On a network layer, group storage can be seen as a fully connected overlay implementation with $\mathcal{O}(1)$ resource lookup. Group storage is responsible for object state management. The overlay storage is an implementation of an existing structured overlay, such as a DHT, with $\mathcal{O}(logN)$ resource lookup [1]. The overlay's main responsibility is ensuring object persistence [31].

#### 2.4.1.1. Peer Types

On a network level, group storage consists of three peer types, namely *peers, super-peers* and a single *directory server*. A directory server provides a well-known entry point to the network, effectively allowing peers to bootstrap to a known member of the network. Peers, also called storage peers, contribute to both group and overlay storage and handle all storage, replication and repair requests. Additionally, peers are responsible for maintaining a group ledger that provides a consistent view of all objects and peers within its group. Super-peers are authoritative peers in the network, that serve a more administrative role within each group. Super-peers are mainly used to ensure object and group consistency for a group. They handle network join and leave requests and initiate object repair. Super-peers also facilitate peer migration and maintain a group ledger of their own.

Figure 2.1 provides a simplified illustration of the Pithos architecture. The authors implemented a Pithos system in simulation and found that the architecture increased reliability and responsiveness compared to classic P2P MMVE state persistence methods.

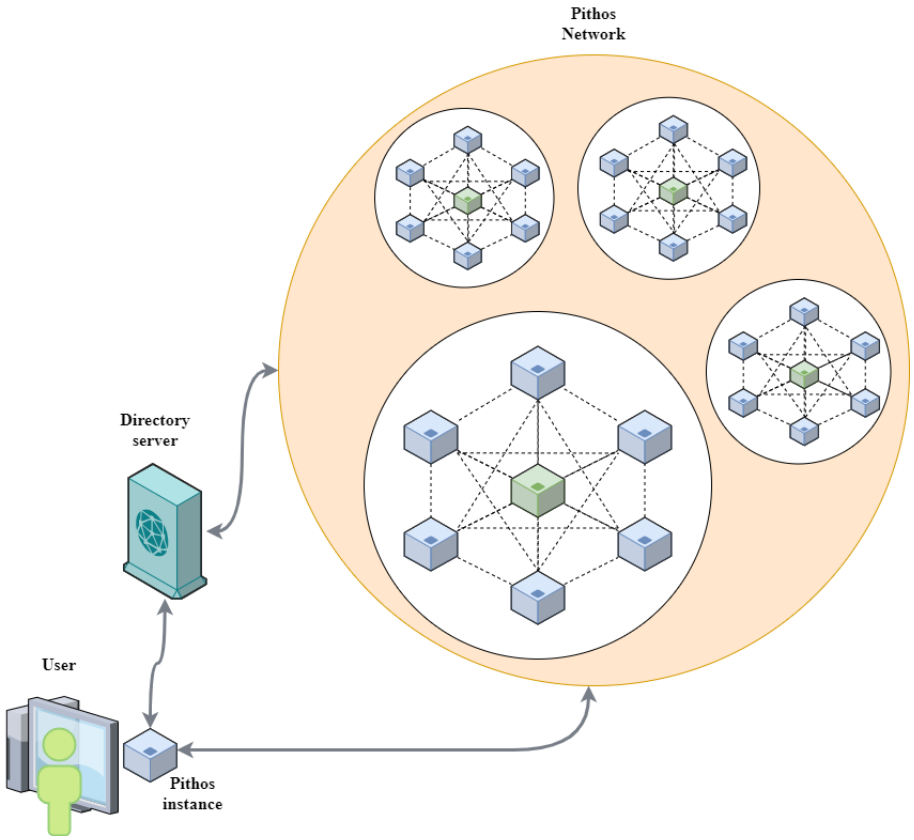


**Figure 2.1:** Simplified illustration of the Pithos architecture.

#### 2.4.1.2. Storage Procedure

Storage on a Pithos network follows the following procedure [1]:

1. A storage request is received from a higher layer, like game logic.
2. The resource is stored locally in memory.
3. The host informs all peers (peers and super-peer) within the group that it has stored an object.
4. All peers add an entry to their group ledger
5. The peer replicates the resource amongst other peers in the network for redundancy.
6. Each peer successfully storing a replica, informs all peers that it has stored an object.
7. All peers add an entry to their group ledger.
8. Super-peers perform scheduled repairs to ensure sufficient replication.
9. If a peer leaves the network, a repair is triggered by the super-peer

In Chapter 3, the Pithos architecture is discussed in detail.

#### 2.4.1.3. Pithos Suitability

As Pithos is specifically designed for P2P MMVEs, it is undoubtable a highly suitable option to be implemented as state consistency architecture in a P2P MMVE. However, since Pithos was designed in 2013 and only evaluated through simulation, the state of modern storage solutions should first be assessed. Only with an overview of available solutions, can it be determined whether Pithos is still the most suitable solution, and if so, whether its design can be improved.

### 2.4.2. Sia

Sia is a modern DSN launched in March of 2015 [32], making it one of the first decentralised storage platforms. According to the Sia whitepaper, "Sia is a decentralised cloud storage platform that intends to compete with existing storage solutions. At both P2P and enterprise level" [33]. Peers on the Sia network do not rent storage from a centralised provider, but instead rent storage from each other. Sia stores only the contracts formed between peers, which define the storage terms. These contracts state that a peer agrees to store another peer's data, periodically submitting proof of storage, until the contract between the peers expires.

Sia uses contracts in the form of a blockchain, similar to Bitcoin [30]. The Bitcoin blockchain makes use of a scripting system, which enables a range of transaction types such as pay-to-public-key-range and pay-to-script-hash. Sia reduces implementation complexity and attack surface, by using an M-of-N multi-signature system instead of a scripting system. Sia also extends classic transactions to enable the creation and enforcement of contracts.

The main idea behind Sia is to leverage underutilised storage capacity on devices all around the world. Users are incentivised to contribute their resources to the Sia network. The incentive is fulfilled in the form of payment in *SiaCoin* tokens, the Sia network cryptocurrency [33].

#### 2.4.2.1. Sia Storage Procedure

Storage on the Sia blockchain follows the following procedure:

1. A resource is split into 30 segments prior to upload, using erasure encoding [34], where any 10 of the 30 can fully recover the resource.
2. Segments are encrypted using the Threefish algorithm [35].
3. Resource segments are distributed amongst hosts using smart contracts.
4. Renters pay hosts for storage using SiaCoin.
5. Hosts add collateral to each segment as a disincentive to go offline.

6. Contracts are renewed over time until they expire.
7. Hosts submit "proof-of-storage" which form Merkle trees.

#### 2.4.2.2. Sia Skynet

From the official Skynet documentation, "Skynet is an open protocol for hosting data and web applications on the decentralized web using Sia" [36]. Skynet decentralises the cloud in such a way that no single central authority holds a user's data. This in turn allows resources to be available across the globe and be accessed on any device, by any application.

Skynet is, however, not yet fully decentralised, as it makes use of centralised Skynet portals. These portals can be seen as an abstraction layer which abstracts away the complexities of dealing directly with the Sia network. Figure 2.2 provides an oversimplified illustration of the Skynet architecture.

In general, Skynet portals are very responsive and object retrieval is usually expected to start about 500 milliseconds after an object was requested. One downside of the Skynet portals is that they are currently rate limited, i.e. that above a certain request rate, a user will be blocked from making more requests.

Related projects like SkyGameSDK [37] are attempting to harness the Skynet platform to allow game developers to build games on the Skynet platform [38].

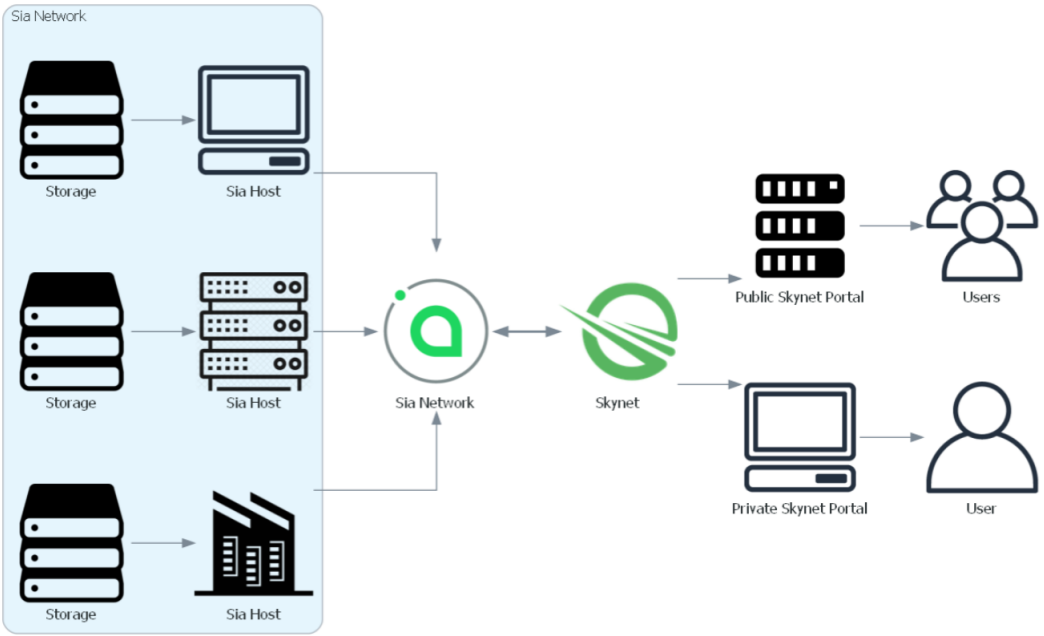


**Figure 2.2:** Skynet simplified architecture [39]

#### 2.4.2.3. Sia and Sia Skynet Suitability

Sia and Sia Skynet are secure, reliable and intrinsically fair. However, P2P MMVEs require a high degree of responsiveness. Due to Sia Skynet's use of centralised portals to access the Sia storage network, the system does not provide the required degree of responsiveness. Response times of 500 ms will reduce user experience in P2P MMVEs. Sia Skynet also applies rate limiting to its public portals, which will drastically limit the speed at which a user can interact with the VE. Additionally, Sia peers are required to be vetted before participating in the storage network. Within a high churn environment, reputation based storage as used in Sia is not suitable for P2P MMVEs. For these reasons Sia will not be considered any further.

### 2.4.3. Storj

Storj is a modern DSN launched in 2014. According to the Storj whitepaper, "the Storj network is a robust object store that encrypts, shards, and distributes data to nodes around the world for storage" [40]. The Storj network consists of three peer components, namely (1) storage nodes, which allow users to share excess hard drive capacity and network bandwidth; (2) Uplink clients, which are developer tools that can be used to upload and download resources; (3) Satellites, which consist of a hosted set of services. Satellites can be seen as authoritative nodes in the network that deal with access management, metadata management, storage node reputation, data repair and billing. Figure 2.3 provides a simplified illustration of the Storj architecture.

Peers are financially incentivised to contribute their resources to the network. The incentive is fulfilled in the form of payment in *STORJ* tokens, the Storj cryptocurrency. Renters pay a fixed monthly amount to store their data on the Storj Decentralised Cloud Storage (DCS) network.

#### 2.4.3.1. Storj Storage Procedure

Storage on the Storj blockchain follows the following procedure:

1. Resources are encrypted using AES-256-GCM symmetric encryption [41].
2. Encrypted resources are split into 80 segments, using erasure encoding [34], where any 29 of the 80 can fully recover the resource.
3. Segments are then distributed across the Storj network, amongst reputable nodes, which also means that segments are distributed globally.
4. Resources can be retrieved as needed. Renters pay a fixed cost for storage.
5. If too many hosts leave the network, an automatic repair mechanism redistributes segments amongst new nodes.

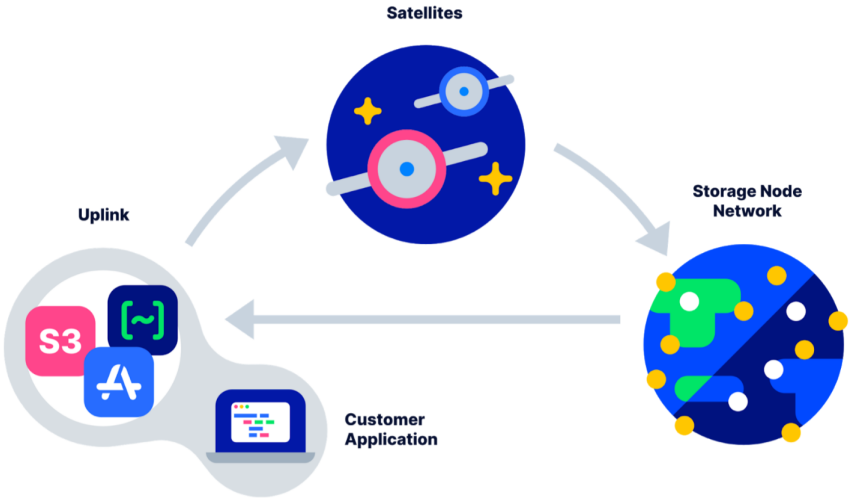


**Figure 2.3:** Storj DCS simplified architecture [42]

#### 2.4.3.2. Storj Suitability

Storj is secure, reliable and intrinsically fair. However, P2P MMVEs require a high degree of responsiveness. Due to Storj's use of centralised portals or Satellites to access the Storj storage network, the system does not provide the required degree of responsiveness. Additionally, Storj uses reputation based peer storage, since peers that enter the network are required to be vetted before participating. Within a high churn environment, reputation based storage as used in Storj is not suitable for P2P MMVEs, since practically no peers will be deemed reputable for storage. For these reasons Storj will not be considered any further.

### 2.4.4. Filecoin

Filecoin is a DSN, built on the InterPlanetary File System (IPFS) [43]. According to the Filecoin whitepaper, "Filecoin is a decentralized storage network that turns cloud storage into an algorithmic market. The market runs on a blockchain with a native protocol token (also called *Filecoin*), which miners earn by providing storage to clients. Conversely, clients spend Filecoin hiring miners to store or distribute data" [44]. Figure 2.4 provides a simplified illustration of the Filecoin architecture.

The Filecoin DSN consists primarily of four component types, namely (1) chain verifiers which do not participate in the network, but constantly synchronise and verify the blockchain; (2) clients that pay to store and retrieve data; (3) Storage Miners, which are peers that earn tokens by offering their storage resources; (4) Retrieval Miners, which earn coins by serving data to clients. The protocol is built upon four novel components, namely [44]:

1. A decentralised storage network (DSN).

2. A novel Proof-of-Storage mechanism, consisting of:
   (a) Proof-of-Replication, which allows storage providers to prove that they store a replica of a certain resource on their own physical drive.
   (b) Proof-of-Spacetime, which allows storage providers to prove that they have stored a certain resource for a specified amount of time.
3. Verifiable Markets, a Storage market and a Retrieval market, where miners and clients can submit storage and retrieval orders. The Verifiable Market ensures that miners get paid when a service is successfully provided.
4. Useful Proof-of-Work, which is based on Proof-of-Spacetime.

The expected storage response time for a 1 mebibyte (MiB) file, is around the 5 to 10 minute mark. For retrieval requests, similar response times are expected [45].

#### 2.4.4.1. Filecoin Storage Procedure

Storage on the Filecoin blockchain follows the following procedure:

1. A client submits a storage or retrieval order for a resource.
2. The system matches the client to a Miner willing to accept the order.
3. A deal is created between the client and the miner.
4. Both parties sign the deal.
5. The deal is submitted to the blockchain as proof.
6. The client pays a micro fee to the miner for storing or retrieval of the resource.
7. The network constantly verifies that miners are storing resources correctly, ensuring that a sufficient replica count exists.

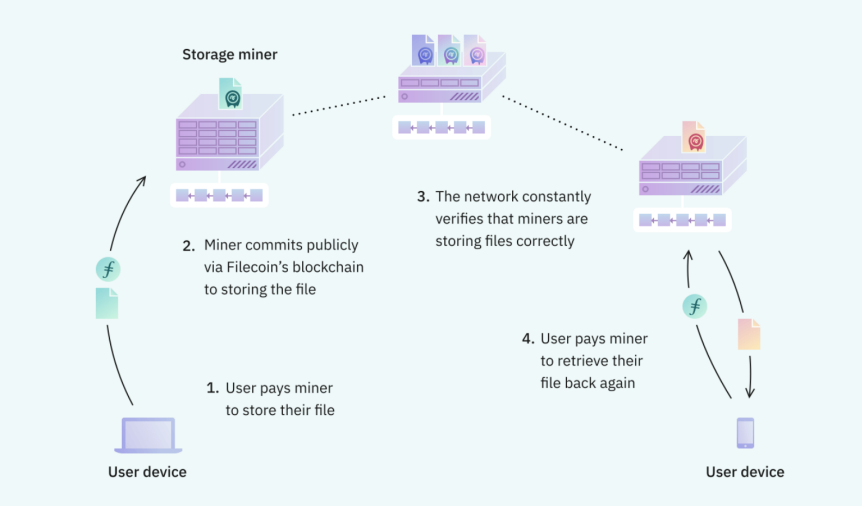


**Figure 2.4:** Filecoin simplified architecture and storage flow [46]

#### 2.4.4.2. Filecoin Suitability

Filecoin is secure, reliable and intrinsically fair. However, P2P MMVEs require a high degree of responsiveness. Due to Filecoin's use of Proof-of-Work, the system does not provide the required degree of responsiveness. For this reason Filecoin will not be considered any further.

## 2.5. Modern Cloud Storage Networks

The internet is one of the cornerstones of modern society, and helped to usher in the Secondary Information Age [47]. It is simultaneously an invaluable repository of generational knowledge, and a tool for mass misinformation and destruction. By design, the internet is distributed, consisting of thousands of petabytes of data. Shockingly, the vast majority of public data is hosted by five key players, namely Amazon, Microsoft, Alibaba, Google and Huawei. Amongst these giants, Amazon has the greatest infrastructure-as-a-service (IaaS) market share, with about 40.8% in 2020 [48]. These companies play a big role in keeping the C/S network architecture popular, by providing cloud solutions that are both cheap and extremely easy to use.

With this popularity of cloud services comes a plethora of robust storage options for both consumers and developers. Historically, using cloud services for game storage has been known for being expensive, hard to integrate, hard to scale and region specific, forcing most game developers to invest in their own data centres. With the great leaps in cloud technology, it is now possible and completely viable to develop and host a game using exclusively cloud services. AWS GameLift [49], Google Game Servers [50] and Microsoft Azure for Gaming [51] are a great cloud native game infrastructure examples. These services simplify dealing with many complexities like gateway protection, matchmaking, session directory and scaling of game servers. Figure 2.5 illustrates the AWS GameLift system architecture. Other cloud providers offer similar gaming infrastructure solutions.

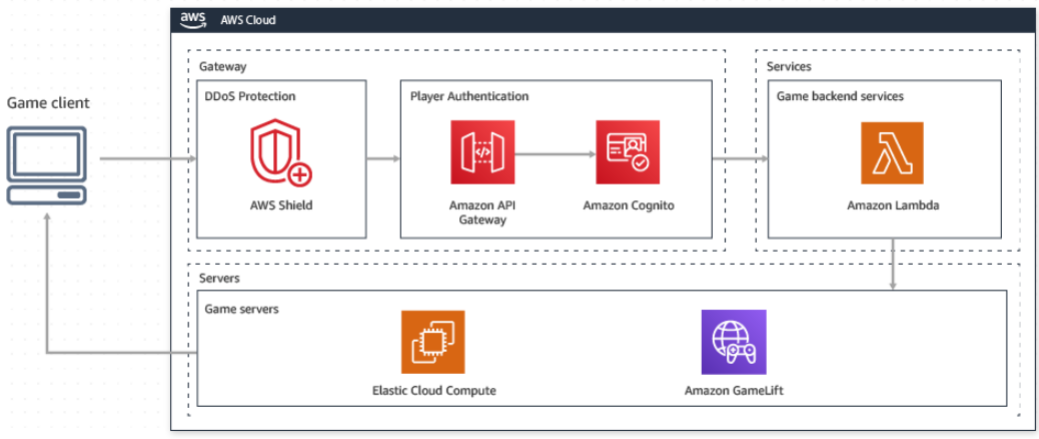


**Figure 2.5:** AWS GameLift simplified architecture [49]

For the purpose of this work however, we will focus on viable storage solutions for P2P MMVEs. These include:

- AWS S3 (S3 buckets)
- AWS DynamoDB
- GCP Cloud Storage
- GCP Spanner
- Azure Disk Storage

Many cloud storage providers offer similar solutions, but for the sake of simplicity we will only review two of the most popular solutions, AWS S3 [52] and its low latency alternative AWS DynamoDB [53].

## 2.5.1. Amazon Web Services: S3 storage

Amazon S3 is a highly secure, durable and scalable object store specifically designed for low cost object storage. S3 has many usage patterns, ranging from serving static content on the web to solutions for backups and archiving of data. This vast range of usage patterns coupled with cross-region hosting and replication make S3 a viable candidate as a storage solution for P2P MMVEs.

### 2.5.1.1. S3 Advantages

- **Durability**: S3 is designed to be highly reliable, with 99.99% uptime in a given year, and an ever higher level of availability if cross-region replication is enabled.
- **Scalability and Elasticity**: S3 is designed to store a practically unlimited amount of bytes in a bucket, without losing any storage or retrieval performance.
- **Security**: All AWS services are highly secure and offer fine grain access management.
- **Interfaces**: S3 interfaces provide standard REST web service APIs, to access and manage buckets.

### 2.5.1.2. S3 Disadvantages

- **Provider centralisation**: S3 is highly available across multiple instances and geographical zones, but relying on a single operator means a high degree of centralisation.
- **Operational costs**: Storage costs incurred for storing data across multiple regions for extended periods will contribute to high operational cost.

- **Read and write latencies**: Even though S3 has many useful usage patterns for rapidly changing data, S3 is not suitable considering high read write latencies. For services with such demands, like MMVEs, AWS DynamoDB is more suitable.

#### 2.5.1.3. S3 Suitability

AWS S3 is secure, durable, scalable and intrinsically fair. However, P2P MMVEs require a high degree of responsiveness and low operating costs. Since S3 does not offer these, it will not be considered any further.

### 2.5.2. Amazon Web Services: DynamoDB

Similar to S3, DynamoDB is designed specifically for high availability and durability with the added requirement of consistently low response times. According to DeCandia et al. [53], DynamoDB is "a highly available key-value storage system that some of Amazon's core services use to provide an 'always-on' experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use."

Appendix F provides more information on the various P2P storage techniques used within DynamoDB.

#### 2.5.2.1. DynamoDB Advantages

- **Responsiveness**: DynamoDB is extremely responsive. Under a load of 500 requests per second (RPS), the response time's 99.9th percentile is under 100 milliseconds.
- **Durability**: DynamoDB is designed to be highly reliable, with 99.99% uptime in a given year, and an ever higher level of availability if cross-region replication is enabled.
- **Incremental scalability**, DynamoDB can scale instantly with minimal impact on operator and system.
- **Node symmetry**: All nodes in DynamoDB are symmetric, meaning peers have the same responsibilities.
- **Decentralised storage techniques**: DynamoDB favours P2P techniques above centralised techniques, improving availability and robustness of the network. Appendix F provides an overview of the P2P techniques used.
- **Heterogeneity**: Distribution of work is determined on an individual node level.

#### 2.5.2.2. DynamoDB Disadvantages

- **Provider centralisation**: DynamoDB is highly available across multiple instances and geographical zones, but relying on a single operator means a high degree of centralisation.
- **Operational costs**: Storage costs incurred for storing data across multiple regions for extended periods will contribute to high operational costs.
- **Data replication**: To provide high availability, data is replicated across nodes, which requires greater storage volumes.
- **Consistency**: One of the trade-offs of a highly responsive, distributed object store like DynamoDB is that it only provides eventual consistency. This means that it takes time for object updates to be propagated to all nodes in the network.

#### 2.5.2.3. DynamoDB Suitability

DynamoDB is a responsive, reliable, secure, durable, scalable and intrinsically fair DSN that employs many P2P techniques. However, P2P MMVEs require low operator costs in order to ensure longevity of the system. Since the use of DynamoDB incurs substantial operator costs, DynamoDB will not be considered any further. It should be noted that DynamoDB has significant potential and its use in a P2P MMVE state consistency architecture should be investigated.

### 2.5.3. Estimated MMO Cloud Costs

There is much speculation surrounding the hardware requirements for hosting a modern MMO like *World of Warcraft* (WoW). In 2009, with an estimated active player count of 10 million [5], the cumulative resource requirements to host the entire WoW infrastructure were reported as [54] [55];

- **Storage:** 1.3 petabytes
- **Memory:** 112.5 terabytes
- **CPU:** 75,000 CPU cores
- **Servers:** 13,250

From teardowns of older regional WoW Servers or "BladeServers" [1] [56], the hardware specifications were documented as;

[1] WoW seems to make use of a region based C/MS approach, with the use of separate regional servers.

- **Storage:** Not specified
- **Memory:** 6 GB
- **CPU:** 2 x dual core CPUs

In 2021 WoW has an estimated active player count of 6.7 million [57]. If identical game instances were hosted on machines, with similar hardware to the old "BladeServers", then only 67% of previously estimated resources would be required, since only 67% of the active player base remains [2]. This would amount to:

- **Storage:** 0.871 petabytes (PB) (871 terabytes)
- **Memory:** 75.36 terabytes (TB)
- **CPU:** 50,250 CPU cores

For the purpose of this cost estimate, we will use only AWS as the cloud provider, since it is the most popular.

#### 2.5.3.1. AWS Cloud Storage

Table 2.1 provides a monthly cost estimate for running an MMO like WoW's persistence layer in the cloud in 2021. If such a persistence layer was stored on cloud storage, an operator might incur the following costs for storing 871 TB on S3 and DynamoDB respectively:

| Storage Service | Monthly Cost (USD) | Notes |
|---|---|---|
| S3 | 19,300 | 2 million storage interactions per month |
| DynamoDB | 223,000 | Dedicated instance object size of 1 KB |

**Table 2.1:** Storage cost estimate for hosting 871 TB of data on AWS (1) S3, (2) S3 Glacier and (3) DynamoDB. All estimates were made using the AWS cost calculator [59].

#### 2.5.3.2. Virtual Machine Instance

Table 2.2 provides a monthly cost estimate for running an MMO like WoW in the cloud in 2021. Costs are estimated for 12,562 a1.xlarge EC2 instances, which would be required to satisfy the required 871 TB of storage, 75.36 TB RAM and 50,250 CPU cores. EC2 instances are standard AWS virtual machines, which are highly configurable in terms of CPU core count, memory, storage and other resources.

[2]Note that Moore's law [58] was not taken into account, since it is assumed that similar CPU and memory modules are being used.

Some hosting companies provide out-of-the-box private game servers. HostBarrel for example, provides pre-configured private WoW servers, which can be rented on a monthly basis. However, this does not come cheap, in order to satisfy the hardware requirements calculated in 2.5.3, approximately 6,700 WoW ultra instances would be required. This would cost on average 1,340,000 USD.

| VM Resource | Monthly Cost (USD) | Hardware Description |
|---|---|---|
| AWS EC2 | 677,815.00 | 12,562 a1.xlarge instances [59]:<br>CPUs: 4<br>RAM: 8 GB<br>Storage: 70 GB (HDD) |
| HostBarrel | 1,340,000.00 | 6,700 WoW ultra instances [60]:<br>CPUs: 4<br>RAM: 24 GB<br>Storage: 160 GB (SSD) |

**Table 2.2:** Resource estimate to host WoW in 2021, using extrapolated hardware requirements from 2009. Costs were estimated using hardware requirements calculated in 2.5.3. All AWS estimates were made using the AWS cost calculator [59].

#### 2.5.3.3. Conclusion

Due to their immense scale and demanding availability requirements, MMOs are expensive to host and maintain. Managing the persistence layer of an MMO like WoW in AWS cloud storage would cost between 19,300.00 and 223,000.00 USD per month, depending on the storage type used. In comparison, running all game servers in the cloud could amount to a monthly cost of 677,815.00 USD when using AWS or even up to 1,340,000.00 USD if a private hosting company like HostBarrel is used.

These cost estimates may seem excessively high, but they are far lower than the speculated operating costs of WoW, which are believed to be in the range of *5 million* USD per month [61]. With this kind of cost involved, building MMO titles is not feasible for many game developers.

P2P MMOs are therefore promising, since the underlying P2P network architecture would negate the high operating costs classic MMOs incur.

## 2.6. Storage Choice

After considering various other modern distributed storage systems, a few suitable storage solutions like DynamoDB and Pithos have been identified. Yet because DynamoDB requires excessive operating costs, it is less suitable for P2P MMVE storage. In comparison, Pithos

is specifically designed for P2P MMVEs and is therefore the only storage solution that satisfies all the requirements of P2P MMVE storage. For this reason, Pithos was selected as the state management and consistency architecture for this project.

## 2.7. Project Approach

Gilmore and Engelbrecht [1] introduced a novel SMP architecture for P2P MMVEs. As part of their research, a simulation was implemented in C++ and simulated in OverSim [62] a flexible overlay network simulation framework. The primary objective of this work, introduced in section 1.6, is to verify the results obtained in the Pithos research project.

In order to achieve this project's primary objective, a number of project approaches were investigated,

1. **Approach 1**: Wrap the existing system functionality in a Java plugin, and use Pithos as consistency architecture in a real-world MMVE like Minecraft [63]. To verify Pithos' performance, storage metrics should be measured and compared to Pithos' simulated results.

2. **Approach 2**: Extract the existing Pithos C++ logic and implement the system functionality that was previously simulated as a storage layer within a simple game. To verify Pithos' performance, storage metrics should be measured and compared to Pithos simulated results.

3. **Approach 3**: Design and implement a DSN, based on the Pithos architecture, from the ground-up. To verify Pithos' performance, performance tests should be executed on the implemented DSN, and performance should be compared.

On investigation of the simulated Pithos implementation, it was concluded that the current implementation was too entangled with the simulation framework OverSim, which is no longer actively supported. This rendered the current implementation unusable, which excluded approaches 1 and 2. Approach 3 was therefore used as a means to satisfy the project objectives.

To design and implement a fully functional DSN, the Pithos research [1] was used as reference architecture. The secondary objectives of this project were formulated as follows:

1. Design a DSN.

   (a) The design should implement all P2P techniques specified in the Pithos architecture.

   (b) The design should fulfil the requirements of scalability, responsiveness, security, reliability and fairness, identified in the Pithos research.

2. Implement the designed DSN.
   (a) The implemented system should fulfil all the functional and structural requirements of a Pithos system.
   (b) The implemented system should be testable.
3. Performance test the DSN.
   (a) The implemented system's various components should be tested and evaluated.
   (b) The implemented system's performance metrics, including response times, bandwidth usage and reliability, should be collected.
   (c) The implemented system's performance metrics should be compared to Pithos' metrics.

## 2.8. Project Objective

The objectives of this work are therefore to:

1. Design and implement a peer-to-peer (P2P) distributed storage network (DSN) suitable for P2P MMVEs.
2. Evaluate the performance of the implemented DSN in terms of the identified key storage requirements.
3. Compare the evaluated performance results against those of a comparable DSN, such as the simulated results of the Pithos architecture.

## 2.9. Conclusion

This chapter provided an explanation of the key design challenges of P2P MMVEs. From these challenges, state persistence was identified as the focus of this study. In order to evaluate a Pithos implementation, the key storage requirements of a P2P MMVE were identified, namely scalability, reliability, fairness, security and responsiveness. These requirements are used throughout this project as evaluation criteria for the proposed Pithos implementation.

# Chapter 3

# Pithos Architecture Overview

Chapters 1 and 2 introduced various concepts and requirements relating to P2P MMVEs. This should provide sufficient background information to understand Pithos' design, introduced in section 2.4.1.

In the research by Gilmore and Engelbrecht, the key design challenges of P2P MMVEs were identified [1]. It was determined by the authors that the biggest contribution to a P2P MMVE architecture, would be the design of a state management and persistence architecture. The main goal of the designed system was to meet the storage requirements of P2P MMVEs, namely load-balancing or fairness, reliability, responsiveness, scalability and security.

Specifically designed to satisfy all requirements of P2P MMVE storage, Pithos is a responsive, reliable, secure, fair and scalable state management and persistence architecture. Pithos uses a *group-based distance-based* storage approach to support fair and responsive storage operations [1].

In this chapter, the Pithos architecture is discussed by taking an in-depth look at each of the system's use cases and key mechanisms.

## 3.1. Pithos

Pithos uses two separate storage layers, namely **group storage** and **overlay storage**, to achieve low latency, object state management, and persistence. Group storage splits the VE into various segments, and groups peers into logical geographical groups. On a network layer, group storage can be seen as a fully connected overlay implementation with $\mathcal{O}(1)$ resource lookup. Group storage is responsible for object state management. The overlay storage is an implementation of an existing structured overlay, such as a DHT, with $\mathcal{O}(logN)$ resource lookup [1]. The overlay's main responsibility is ensuring object persistence [31].

### 3.1.1. Background

The Pithos architecture was conceptualised and published in 2013. After designing a generic state consistency model for both C/S and P2P models and performing a review of

SMP architectures, the authors identified the key challenges of P2P MMVEs. In particular it was found that research regarding *state consistency* in P2P MMVEs was lacking [1]. This observation led to the design and implementation of a novel state management and persistence architecture, called Pithos. Figure 2.1 provides a simplified illustration of the Pithos architecture. The authors implemented Pithos as a simulation and found that the architecture increased reliability and responsiveness compared to classic P2P MMVE state persistence methods.

### 3.1.2. Characteristics

The Pithos architecture consists of three main peer components, namely (1) a directory server, which allows peers to bootstrap to a known member of the network; (2) storage peers, which contribute to both group and overlay storage and handle all storage, replication and repair requests; and (3) super-peers, which are the authoritative peers in a group and serve a more administrative role to ensure group consistency. Super-peers handle network join and leave requests, and initiate object repair and peer group migration.

Additionally, all peers and super-peers are responsible for maintaining a group ledger, which provides a consistent view of all objects and peers within its group. The Pithos architecture has the following unique characteristics compared to classic P2P MMVE state persistence methods [1]:

- Distance-based storage on a group level, instead of a peer level.
- Two types of storage: safe and fast.
- Three types of retrieval: fast, parallel and safe.
- Two types of repair: periodic and leaving.

### 3.1.3. Network Topology

Pithos' network topology is illustrated by figure 3.1. The network topology can be described as a multi-layer structured overlay network of fully connected peers. The fundamental overlay (grey circle), consists of multiple groups (blue circle) of fully connected peers.

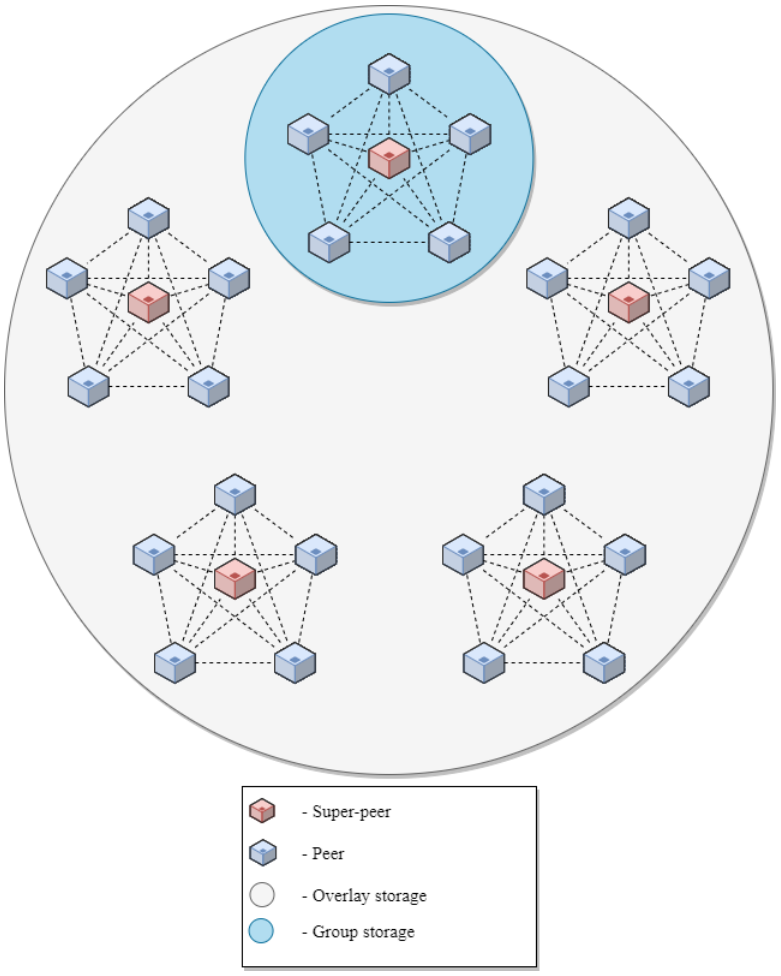


**Figure 3.1:** Pithos network topology [1]

## 3.2. Pithos Use Cases

Pithos was designed specifically for P2P MMVEs, for efficient and reliable object storage and retrieval. Pithos therefore provides an application programming interface (API) that facilitates interactions with the storage network. The interface consists primarily of four operations: **store, retrieve, modify** and **remove**. These four storage operations can be described as the basic use cases of the system. Figure 3.2 illustrates how a virtual environment's logic interacts with the Pithos storage API for state management and persistence of game objects and user information.

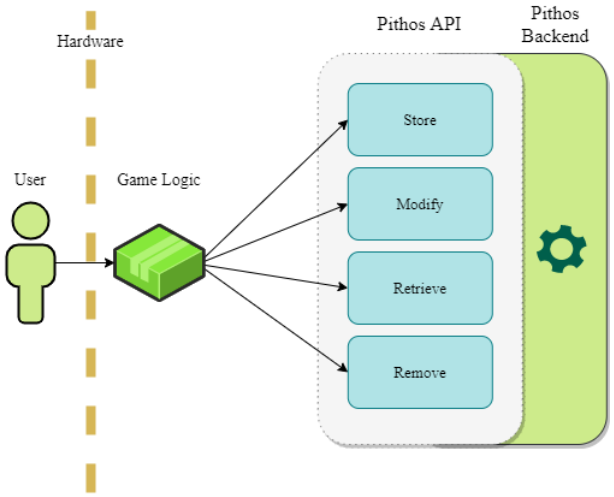


**Figure 3.2:** Pithos Storage API operations [1]

### 3.2.1. Store

A store operation is invoked by the VE logic when new objects are created. When a new object is stored on a peer, the object is assigned a time-to-live (TTL). When the TTL of an object expires, the object will automatically cease to exist. Object creation events in a VE include the following scenarios: [1];

- Initial map (VE) generation.
- Spawning of loot boxes.
- Spawning of NPCs.
- Player interactions (e.g. throwing a grenade, summoning a skeleton).

### 3.2.2. Retrieve

A retrieval operation is invoked on all VE interactions, namely self, player-to-object or player-to-player interactions. VE interactions require object retrievals, as interactions can effect the state of objects within their AoI. Within P2P MMVEs, the object state is usually served from the non-authoritative object store, which manages the VE state for each user. Object retrieval can occur, for example if a user throws a hand grenade at another object within the VE [1].

### 3.2.3. Modify

Interactions often lead to object modification. Hence a modify operation is invoked in the case where an interaction alters the state of an object. The state of the modified object is updated in both the authoritative and non-authoritative object stores. Object modification can occur during self, player-to-object or player-to-player interactions [1].

### 3.2.4. Delete

A delete operation is invoked in the case where an object needs to be removed from the VE. Traditionally in a DSN, delete operations are not strictly necessary, as object "liveness" is determined by the TTL attached to the object. However, a delete operation might be called if, for instance, an interaction causes the object to be completely destroyed. For the purpose of Pithos' design, no exclusive API endpoint object removal exists, since DSNs rely on object TTLs to correctly expire and remove objects [1].

## 3.3. Pithos: Underlying Persistence Models

In order to understand Pithos on a fundamental level, the architecture will now be described in terms of its underlying P2P persistence models.

### 3.3.1. Object Consistency

As explained in section 1.2, VE objects can be categorised into four object types, namely *immutable objects*, *mutable objects*, *player characters/avatars* and *non-player characters* (NPCs). For the interest of state consistency and persistence, all four object types are important. However, as mutable objects are crucial to user interaction and game logic, state consistency for these objects is of higher importance. Mutable objects occur in two forms within a VE: the root form, i.e. the original object, and the replicated form, which are copies of the original object. Replicas are used to achieve low-latency system-wide consistency. Object replicas are copied to each user that is interested in the object, in order to process interactions locally. These local object state changes and interactions are used for display purposes [3].

It is worth noting that the state persistence model is determined by the high-level consistency model used. The following sections will describe the different consistency models used in Pithos.

### 3.3.2. Overlay Storage Persistence Model

The Pithos architecture makes use of overlay storage as a fall-back storage layer. Additionally, overlay storage is used to satisfy the requirement for state persistence. For the purpose of this work, overlay storage refers to a structure P2P overlay, like a DHT, as a storage network. Overlay storage as a P2P persistence layer is justifiable, since any P2P distributed storage network can be used to store only VE objects [1].

#### 3.3.2.1. Fairness

Structured overlays use an identifier space to allocate a unique ID to each peer. This same identifier space is used to allocate IDs to objects stored in the overlay. An objects is stored on the peer with the closest match (distance-based) to its own ID. The identifier space is distributed in a random fashion, which means overlay storage can be deemed fair [10].

#### 3.3.2.2. Reliability

To ensure reliability in overlay storage, redundancy mechanisms must be implemented. Object replication is often used to ensure redundancy, fault tolerance and high availability in the overlay [9]. The presence of object replicas increases the probability that an object will always be available. In the case where a peer storing an original object leaves the network, the distance-based routing mechanism will ensure that traffic is automatically routed to the nearest neighbouring peer storing a replica of that object [27]. With the addition of redundancy, overlay storage is deemed reliable.

#### 3.3.2.3. Security

Overlay storage is generally secure, due to the distributed nature of the system. To improve security, redundancy and quorum techniques can be implemented. Object replication ensures that more than one copy of an object exists across multiple peers. A quorum mechanism can then be used on object retrieval to request a set number of replicas, to determine if received objects are identical. However, security and trust are still challenges for overlay storage implementations like DHTs [9], and the suggested measures should be used in conjunction with other security mechanisms.

#### 3.3.2.4. Responsiveness

Responsiveness is one of the weaknesses of overlay storage, and is caused by the computational overhead for storage and retrieval requests. Overlay storage networks are not fully connected, and therefore require on average more than one hop to retrieve or store data. On average, $\mathcal{O}(logN)$ hops are required to complete a request [27].

#### 3.3.2.5. Scalability

Structured overlay storage is considered to be sufficiently scalable. Since peers and objects are normally distributed within an identifier space, it achieves logarithmic scalability [27].

### 3.3.3. Distance-based Persistence Model

Pithos makes use of distance-based storage on a group level. With a distance-based storage approach, objects are stored on peers that are geographically close to the object in the

virtual world. A distance metric usually determines on which specific peer an object should be stored. Voronoi storage [64] is an example of a distance-based storage approach [27].

#### 3.3.3.1. Fairness

Distance-based storage is known to be relatively fair, since objects are relatively evenly distributed amongst peers within the VE. There are, however, similar concerns to region-based storage, where some peers might be close to a large number of objects, effectively overloading that peer. This reduces the fairness of distance-based storage [27]. Certain mechanisms like peer overload/underload detectors and aggregators can be put in place that can determine a peers current load and take over the responsibility of overloaded peers [64].

#### 3.3.3.2. Reliability

Distance-based storage has some reliability concerns, due to its weakness against network churn. Similarly to region-based storage, redundancy in the form of replication is used to improve reliability. An object can be stored on multiple peers that are geographically close to that object. Object replication therefore ensures high availability of objects [27].

#### 3.3.3.3. Security

Similarly to region-based storage, the main concern of distance-based storage is security. Malicious peers are very difficult to detect within the network. Redundancy in the form of replication allows for the use of quorum mechanisms which improve security. Quorum mechanisms do, however, increase resource usage and reduce responsiveness [27]. Certification mechanisms or reputation mechanisms are other well known security implementations that can be used to ensure only authorized peers contribute to object storage.

#### 3.3.3.4. Responsiveness

Distance-based storage can be very responsive. When a user interacts with an object, the probability of the object being stored locally is quite high. Yet There are cases where multiple peers interact with the same object, and this has two possible outcomes. The best case scenario is that all peers are storing the object, in which case every peer hosting the object becomes a server for that object. In the worst case, a single peer stores the object and becomes the server for the object, which still only requires a single hop to serve requests [27]. One challenge introduced by distance based storage is that if the authoritative object store of the object changes too frequently, distance based storage becomes unresponsive. In distance based storage, an object's authoritative store can change if the peer hosting the object is deemed to have left the object's AoI.

#### 3.3.3.5. Scalability

The scalability of distance based storage depends heavily on the use case. For large VEs with few peers, a peer might be required to host many objects. With more peers joining the network, storage might become more responsive, but if peers are constantly moving, the overhead required to update the authoritative object store of an object will reduce responsiveness. It can therefore be said that distance based storage does not scale well.

## 3.4. Peer Types

Pithos' three main peer types are **peers**, **super-peers** and a **directory server**. This section provides an overview of each peer type and its responsibilities.

### 3.4.1. Directory Server

In order for a new peer to join a virtual network, it needs to know at least one IP address of a node already participating in the network. Some distributed networks make use of host address caching, which relies on previous session connection addresses to join the network. Unfortunately, this is failure-prone and unreliable. Another way of providing an entry point to a distributed network, is through the use of centralised directory or bootstrap servers. Directory servers are well-known hosts within the system, with a static IP or domain name. The directory server's main purpose in a distributed system is to cache the IP addresses of other peers in the network. Peers can use the directory server to determine the bootstrap IPs of other peers [10].

Pithos makes use of a directory server to store super-peer IP addresses. Apart from providing super-peer hostnames, the directory server also keeps track of super-peer changes. If a super-peer leaves the network, the directory server is responsible for updating the group's super-peer hostname. The directory server serves two purposes in Pithos, namely facilitating new peers joining the network, and group migration [1]. The join and migration mechanisms are discussed in subsequent sections.

### 3.4.2. Peer

The Pithos peer module is one of the fundamental modules of the Pithos architecture. Peer nodes can be seen as storage nodes that are responsible for both group and overlay storage. The Pithos peer module additionally contains all the required logic to participate in the Pithos storage network.

#### 3.4.2.1. Peer Logic

A peer's responsibilities can be summarised as follows [1]:

1. Joining and leaving the group.
2. Migrating to another group.
3. Handling all store, retrieve and modify requests for both group and overlay storage.
4. Keeping track of requests.
5. Implementing quorum mechanisms.
6. Forwarding requests to the required modules.

### 3.4.3. Super-peer

The super-peer module is another fundamental module. In contrast to normal storage peers, super-peers do not contribute to object storage. Super-peers are responsible for various administrative tasks that ensure group and object consistency.

#### 3.4.3.1. Super-peer logic

A super-peer's responsibilities can be summarised as follows [1]:

1. Handling all group leave and join requests.
2. Facilitating group migration.
3. Ensuring group consistency.
4. Initiating object repair.
5. Maintaining its own super-peer ledger, which stores object and peer IDs.

## 3.5. Group Configuration

Pithos' groups consist of a combination of its three peer types: A global directory server to enable network bootstrapping, a super-peer to allow group bootstrapping and perform maintenance tasks, and multiple storage peers that participate in group and overlay storage. This section provides an overview of Pithos' grouping mechanisms and leader election.

### 3.5.1. Grouping Mechanism

As Pithos makes use of group-based storage, it requires a grouping mechanism to logically group peers to ensure optimal group storage performance. Pithos uses a super-peer centred distance-based grouping approach, which groups peers that are geographically close to one another in the virtual world. This mechanism satisfies the IM requirement of MMVE

design, by ensuring that objects that peers are likely to interact with, are stored within the group.

The Pithos architecture does not explicitly specify a grouping mechanism to use, although it does mention *distributed peer clustering techniques* and *dynamic Voronoi regioning* as possible grouping techniques [1]. Using distributed peer clustering techniques, the distance between peers could be used to dynamically group peers. In order to avoid group overloading, peer density techniques can be used to split and merge groups as the density changes [1]. Dynamic Voronoi regioning divides the virtual world into regions [64] that can be dynamically split or merged in order to maintain a constant peer density [1].

### 3.5.2. Super-peer Selection Mechanism

In order to assign a super-peer to each group, a leader election mechanism is required. The Pithos architecture does not specify any super-peer selection strategy, since the topic had not received sufficient research at the time of writing. Pithos therefore assumes that a utility function exists that selects super-peers within the system [1].

## 3.6. Group Join, Leave and Migrate Mechanisms

Previous sections have provided detail on Pithos' main peer types and group configuration. This means that the topic of network and group bootstrapping can be discussed next. This section provides more information on Pithos' group join, leave and migration mechanisms.

### 3.6.1. Group Join Mechanism

Super-peers represent groups within the Pithos network and are required to act as group gatekeepers. This means that super-peers determine which peers may join their group. Each super-peer shares its bootstrap hostname with the directory server. When new peers send a join request to the directory server, the directory server determines which group the peer should join and responds with the group and its corresponding super-peer hostname. The peer then sends an identical join request to the super-peer, which can either reject or accept the peer into its group.

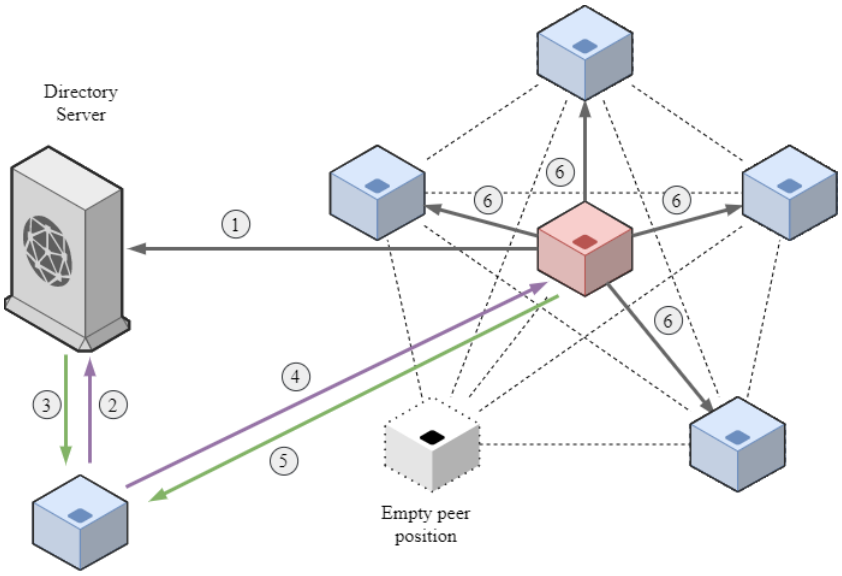


**Figure 3.3:** Pithos network join [1]

Figure 3.3 illustrates the procedure that allows peers to join the Pithos network [1].

1. A super-peer shares its bootstrap hostname with the directory server. This can be used to join that super-peer's group. The directory server stores this information.
2. A new peer sends a join request to the directory server. The join request contains some peer information, along with the peer's required location in the VE.
3. The directory server responds to the request with the hostname of the super-peer of the group the peer should join.
4. The peer sends the same join request to the super-peer.
5. The super-peer decides whether it wants to accept or reject the peer, based on the current group size and the peer's VE location.
   (a) If the super-peer rejects the peer, it provides the peer with a bootstrap hostname of a more suitable group.
   (b) If the super-peer accepts the peer, it provides the new peer with a list of peers and objects in the group. The new peer uses this information to populate its own group ledger.
6. (Assuming the peer was accepted) The super-peer notifies all peers within the network that a new peer was added.

### 3.6.2. Group Leave Mechanism

If a peer ends its session, either gracefully or unexpectedly, a leave mechanism is required to maintain group consistency. If a peer leaves the network gracefully, it notifies the super-peer that it is leaving the network. The super-peer then broadcasts to the rest of the group that the peer has left. The same leave mechanism is used during group migration, discussed in section 3.6.3.

In some cases a peer might leave the network unexpectedly, which may result in group inconsistency. To ensure group consistency in Pithos, peers send keep-alive messages, in the form of network *pings*, to each other. If a keep-alive message is not acknowledged with a *pong* response, a special leave mechanism attempts to verify whether a peer has indeed left the network. If the peer is not reachable, the super-peer will notify all other peers to remove it. Super-peers do not originate keep-alive pings themselves, to prevent super-peer overloading. The rate at which keep-alive pings are executed is kept relatively low, to prevent excessive network usage [1].

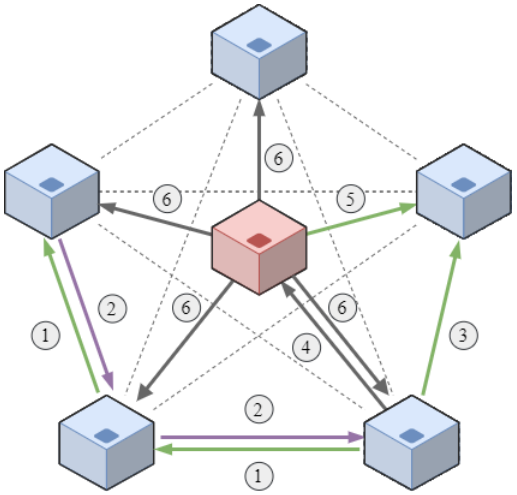


**Figure 3.4:** Pithos leave mechanism [1]

Figure 3.4 illustrates the group leave procedure, which contributes to group consistency if a peer leaves the group unexpectedly [1].

1. A peer routinely sends a keep-alive ping request to another random peer.
2. If a peer receives a keep-alive ping, it acknowledges the message with a pong.
3. If a peer does not acknowledge the keep-alive ping within a specific time window, the peer is considered to have left the network.
4. The original sender of the keep-alive ping notifies the super-peer that a peer is unreachable.

5. The super-peer sends a ping of its own to the peer in question, to verify that the peer has actually left the network.

6. If a peer does not acknowledge the super-peer ping within a specific time window, the peer is considered to have left the network. The super-peer then informs all peers that the peer has left the group and should be removed from all group ledgers.

### 3.6.3. Group Migration Mechanism

In MMVEs, peers are able to move within the virtual world. This means that peers may leave the AoI of a group. When a peer leaves the AoI of a group, it is no longer interested in the objects hosted in that group, and therefore require a migration mechanism to enable it to join a suitable group. In Pithos, the directory server contributes to group migration, by enabling super-peers to communicate with one another.

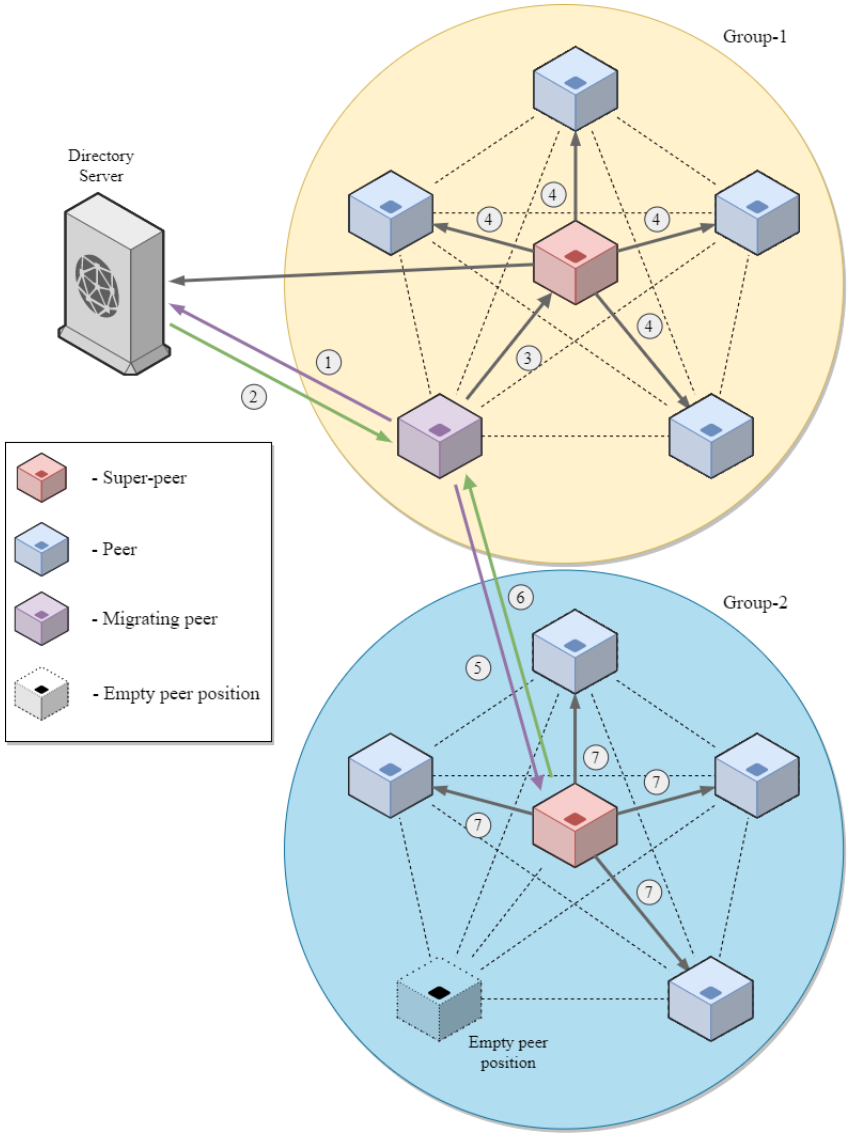


**Figure 3.5:** Pithos group migration [1]

Figure 3.5 illustrates the group migration procedure that allows peers to migrate from one geographical group to another [1].

1. A peer routinely sends positional updates to the directory server.
2. The directory server evaluates the peer's position and responds with the super-peer ID for the group within which the peer's position falls.
   (a) If the super-peer ID matches the peer's current group super-peer ID, the peer is still in the correct group. The peer does not need to be migrated.
   (b) If the super-peer ID does not match the peer's current group super-peer ID, the peer is no longer in the correct group. The peer needs to be migrated.
3. The peer sends a leave request to its current super-peer.
4. The super-peer notifies all peers in the group that the peer has left.
5. The peer sends a join request to the new super-peer. (The join request procedure is followed)
6. If the new super-peer accepts the peer, it provides the peer with a list of peers and objects in the group. The peer uses this information to populate its own group ledger.
7. (Assuming that the peer was accepted) The new super-peer notifies all peers within the network that a new peer was added.

## 3.7. State Management and Persistence

As discussed in Section 2.1, state consistency architectures have two responsibilities, namely state management and state persistence.

Pithos uses three components to achieve state management and consistency within the network, namely overlay storage, group storage, and group ledgers. The following subsections provide information on each of these components.

### 3.7.1. State Persistence

State persistence is responsible for persisting objects in long term storage. The Pithos architecture achieves state persistence by making use of an existing structured overlay implementation in the form of a DHT.

### 3.7.2. Overlay (DHT) Storage

The term overlay storage in Pithos refers to a higher layer of storage in the form of a DHT, a structured overlay implementation. Group storage attempts to maximise the amount of in-group requests, but depending on the grouping algorithm used, the ratio between in-group and out-of-group requests could vary. The system therefore requires a mechanism for handling out-of-group requests. A DHT fulfils this requirement in Pithos.

In order to support out-of-group communication, the overlay storage is used as the backup storage layer. This means that all objects are stored to the overlay network. Every peer in the DSN participates in the overlay network. This enables peers to request objects from the overlay.

The peer module is also responsible for overlay storage. The overlay storage is an $\mathcal{O}(logN)$ implementation of a structured overlay. As all peers participate in overlay storage, the overlay storage module is responsible for serving overlay requests on behalf of the peer.

The Pithos overlay module makes use of an existing structured overlay implementation in the form of a DHT. A peer's overlay storage responsibilities can be summarised to the following [1]:

1. Handling overlay store, retrieve and modify requests.
2. Maintaining an object store, with all objects.
3. Maintaining a consistent view of all peers within the overlay.
4. Maintaining a consistent view of all objects within the overlay.
5. Handling of overlay join and leave requests.
6. Routing of requests to the responsible peers.

The Pithos research provides an overview of multiple DHT implementations, as seen in table 3.1. From these results it can be concluded that DHTs with lower reliability are also less responsive, as peers take longer to retrieve objects. For the Pithos simulation, Chord was used as the preferred overlay, since it provides the best simulation times compared to other overlays [1].

| DHT | Reliability (%) | | Responsiveness (s) | | Bandwidth (Bps) | |
|---|---|---|---|---|---|---|
| | store | retrieve | store | retrieve | in | out |
| Chord high | 98.31 | 93.65 | 1.217 | 1.745 | 2175 | 2189 |
| Chord medium | 96.90 | 93.20 | 1.214 | 1.582 | 1183 | 1197 |
| Chord low | 79.08 | 62.16 | 1.245 | 2.071 | 301 | 314 |
| Pastry | 98.97 | 94.90 | 0.625 | 1.182 | 1979 | 2088 |
| Kademlia | 45.53 | 35.13 | 0.908 | 4.604 | 512 | 498 |

**Table 3.1:** Pithos project evaluation of overlay storage implementations [1].

### 3.7.3. State Management

State management is responsible for managing object state in the VE. The Pithos architecture achieves responsive and reliable state management by making use of group ledgers, group storage and overlay storage. Since overlay storage has already been discussed, only group storage and group ledgers will be covered here.

### 3.7.4. Group Storage

Group storage is an $\mathcal{O}(1)$ implementation of a structured overlay, as the group consists of fully connected peers. Group storage is therefore highly responsive and has constant lookup time for objects contained in the group. Pithos' peer module is responsible for group storage.

Group storage's responsibilities can be summarised as the following [1]:

1. Handling group store, retrieve and modify requests.
2. Maintaining an object store, with a subset of group objects.
3. Maintaining the peer's group ledger.
   (a) Maintaining a consistent view of all peers within the group.
   (b) Maintaining a consistent view of all objects within the group.
4. Maintaining the peer's group membership.
5. Monitoring peer availability in the group by making use of scheduled network pings.
6. Notifying peers and the super-peer of new objects.
7. Handling repair requests from the super-peer.

### 3.7.5. Group Ledgers

Pithos makes use of group ledgers in order to track all peers and objects within a group. The group ledger acts as a sort of routing table, effectively facilitating constant lookup time within the group. A group ledger is an abstract data type, in the form of a map. It can be described as a unification of two separate data structures, an *object ledger map* and a *peer ledger list*. Figure 3.6 provides an illustration of the group ledger data structure.

The object ledger map consists of multiple object ledgers. Each object ledger entry consists of object information and a list of peers that store the object. Each peer information entry stored in the object ledger is a reference to the peer information contained within the actual peer ledger.

The peer ledger list consists of peer information, where each entry in the peer ledger consists of a peer ID and a list of objects it stores. The object information stored in the peer ledger is a reference to the object information contained in the object ledger.

Since object and peer ledger entries refer to one another, each entry is only stored once within the data structure. This allows for efficient object lookup. As group storage is designed to be highly responsive, the group ledger is a crucial module to satisfy this requirement. Apart from efficient object lookup, the group ledger is also required by the super-peer module that maintains object replicas [1].

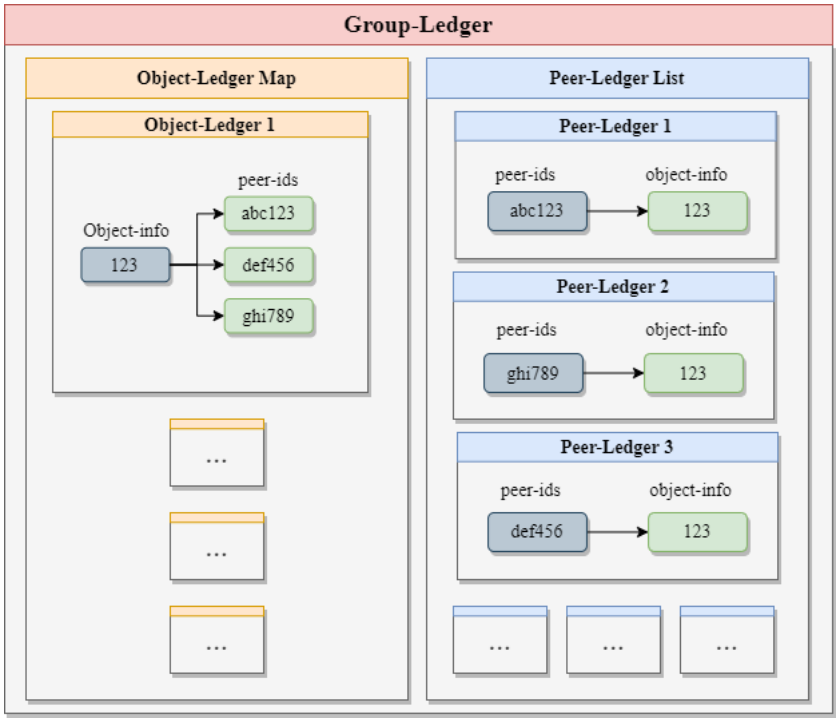


**Figure 3.6:** Pithos group ledger data structure [1]

#### 3.7.5.1. Super-peer Ledger

As the super-peer is responsible for maintaining group consistency and object repair, a super-peer ledger is required to maintain a consistent view of all peers and objects within the group. The super-peer ledger is identical to a normal group ledger, with the added ability to collect group-wide storage statistics [1].

## 3.8. Redundancy

Pithos makes use of object replication and repair mechanisms to provide a level of redundancy in the storage network. These measures are discussed in the following subsections.

### 3.8.1. Replication Mechanism

Object replication serves multiple purposes in Pithos, one of which is ensuring high availability of data under constant network churn. This in turn improves security against malicious peers, and ensures better object storage and retrieval times.

For every object stored in the Pithos network, $\mathcal{R}$ object replicas are created. $\mathcal{R}$ is known as the replication factor. The replication factor is determined by certain conditions. If high network churn is expected, a high replication factor is required. As object replicas can be lost due to network churn, a repair mechanism is required to ensure that $\mathcal{R}$ replicas are always available [1].

If a high number of malicious peers are expected in the network, a high replication factor is required, to ensure that object quorum can be reached. If a sufficiently high replication factor is not used, the system becomes susceptible to object manipulation.

Replication also improves responsiveness in Pithos, as multiple object replicas ($\mathcal{R}$), enable the use of parallel object storage and retrieval, which can improve latency. However, to ensure that object replicas are maintained, a repair mechanism is required.

### 3.8.2. Repair Mechanism

As just described, objects are stored on multiple peers to improve object high availability. This form of redundancy is useful, but is fallible under network churn, as peers leaving the network reduce the number of available replicas. If no mechanism is put in place to repair missing object replicas, an object could eventually cease to exist. For this reason, Pithos makes use of an object repair mechanism to ensure that missing object replicas are repaired.

Pithos' repair mechanism consists of *periodic (scheduled) repair* and *leave repair*. During scheduled repair, a super-peer periodically checks that $\mathcal{R}$ object replicas are stored in the

group, by checking its group ledger. If an object is found to have insufficient replicas, a repair request is sent to peers that do not already store the object. Scheduled repair is useful as peers might not leave the group gracefully, which means that leave-repair is never executed [1].

Leave repair is initiated when a peer leaves the group. This is required, since when a peer leaves the group, object replicas are destroyed. During leave repair, super-peers pre-emptively create new replicas of all objects stored by the peer leaving the group. Using leave repair ensures that $\mathcal{R}$ object replicas are always available if peers leave the group gracefully. The repair mechanism can therefore be seen as an extension of the leave mechanism [1].

Figure 3.7 illustrates the object repair procedure. Note that as this is an extension of figure 3.4, steps 1 to 6 are not included in the description [1].

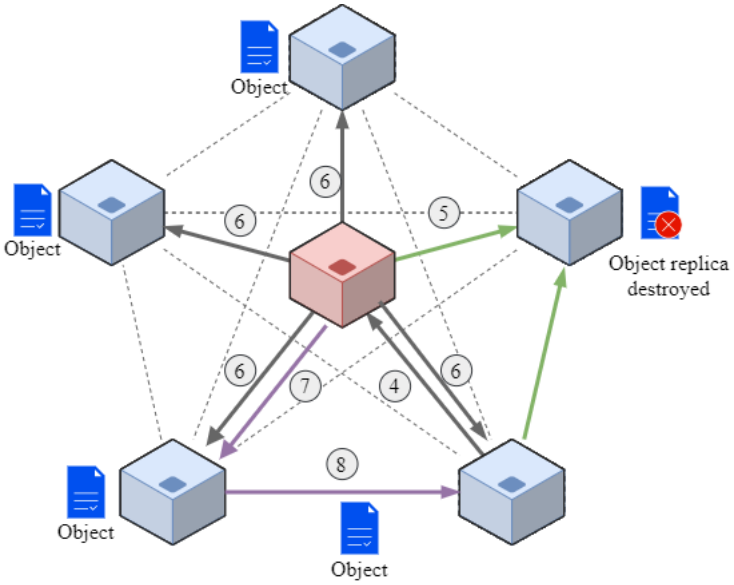


**Figure 3.7:** Pithos repair mechanism [1]

1. After a super-peer has notified all other group peers that a peer has left the group, the super-peer selects a peer that contains a replica of the object that was destroyed, and requests that this peer replicates the object.

2. The peer hosting the replica object then selects a random peer in the group that does not store the object, and sends a storage request to that peer, effectively replacing the missing object replica.

## 3.9. Object State Consistency and Security

Pithos primarily makes use of its group ledgers to ensure group consistency. However, object state can also vary between peers as a result of network errors or malicious peers.

This means that a mechanism to ensure object state consistency and validity in the network is required. Pithos primarily makes used of a quorum mechanism to achieve object state consistency. Additionally, Pithos' quorum mechanism and a certification mechanism are used to increase security within the network.

### 3.9.1. Quorum Mechanism

Pithos makes use of a security mechanism, called quorum, to perform basic object verification. When active, Pithos executes multiple object retrievals from unique peers. The quorum mechanism then compares objects and verifies that the object replicas are consistent. The object value that occurs the most amongst all responses is considered to be the true object value. A simple formula can be used to determine if quorum can be reached: $(\mathcal{R}/2) + 1$, where $\mathcal{R}$ is the replication factor [1].

### 3.9.2. Certification Mechanism

Certification mechanisms ensure peers are identifiable and cannot generate their own IDs. Pithos implements its own certificate authority (CA), which allows it to generate peer IDs and signed certificates. When a node requests to join the network, the certification server assigns the peer a valid ID and a signed certificate, which are both required to participate in the Pithos network. Additionally, whenever an object is stored or updated, peers are required to sign the object. This allows Pithos to keep track of changes during an object's lifetime [1].

## 3.10. Satisfying the Use Cases

This section demonstrates how Pithos satisfies the required use cases for store, retrieve, modify and delete operations, introduced in section 3.2. All storage requests are assumed to originate from some higher layer, like the VE or game logic.

### 3.10.1. Store

Two storage modes exist in Pithos, namely *fast* and *safe* storage. Fast storage is intended for highly responsive storage operations. In fast storage mode, Pithos sends a storage command to multiple group peers and to overlay storage. The first successful response is propagated to the higher layer. Fast storage operations only fail if all storage requests fail. Sending requests to multiple peers improves the responsiveness of storage commands, since some peers may have a lower latency connection due to being geographically closer to the requesting peer. Fast storage therefore favours peers with lower latency connections. Fast

storage is no less reliable than safe storage, however the likelihood that a false positive is sent to the higher layer is greater [1].

Safe storage is intended for highly secure storage operations. In safe storage mode, Pithos sends a storage command to multiple group storage peers, as well as the overlay storage. Only after all responses are received from both group and overlay storage a decision is made whether the request was successful. If the majority of responses are successful, the response is deemed successful. The likelihood that a false positives is sent to the higher layer is much lower with safe storage. Compared to fast storage, however, safe storage is much less responsive [1].

Figure 3.8 illustrates Pithos' storage procedure [1].

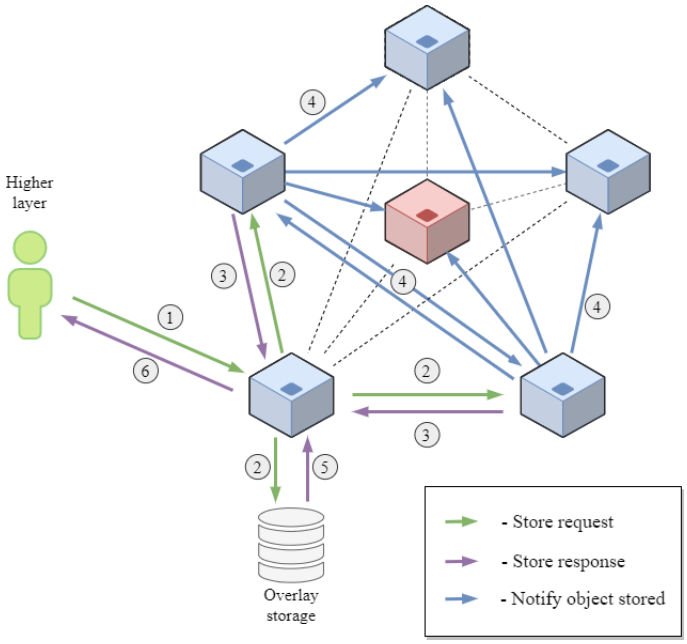


**Figure 3.8:** Pithos storage procedure [1]

1. A storage request is generated from the higher layer to the Pithos storage interface.
2. Pithos propagates the storage request to both the group storage and overlay storage peers. As each peer participates in both group and overlay storage, overlay storage is represented by a single entity in Figure 3.8. In reality, the storage request is propagated to another peer in the network.
3. The group storage module stores the object in its local object store and acknowledges the store request by responding with a message to the requesting peer.
4. Peers that are successfully storing the object add the object to their group ledger and notify all other peers and the super-peer that they are storing the object.
5. Once the overlay storage operation completes, it responds with success or failure.

6. Pithos responds to the higher layer:

   (a) **Fast storage:** Pithos responds with the result from either overlay or group storage, whichever succeeds first.

   (b) **Safe storage:** Pithos monitors responses from group storage and overlay storage. If the majority of responses are successful, the response is deemed successful.

### 3.10.2. Retrieve

Three retrieval modes exist in Pithos, namely *fast*, *safe* and *parallel* retrieval [1]. Fast retrieval mode is similar to fast storage mode. In fast retrieval mode, Pithos sends a retrieval command to a single peer that participates in group storage, as well as the overlay storage. The first successful response object that it receives is then propagated to the higher layer. Fast retrieval is generally very responsive and requires minimum bandwidth, but it is more susceptible to malicious peers and object manipulation [1].

Safe retrieval is intended for highly secure retrieval operations. In safe retrieval mode, Pithos sends a storage command to multiple peers that participate in group storage, as well as the overlay storage. After a sufficient number of response objects is received, a quorum mechanism (3.9.1) is used to determine the original object. If the majority of responses are successful, the response is deemed successful. Safe storage is resilient against malicious peers, although similarly to safe storage, it is less responsive compared to fast and parallel retrieval [1].

Parallel retrieval is intended for reliable and responsive object retrieval. In parallel retrieval mode, Pithos sends a storage command to multiple group storage peers, as well as the overlay storage. Similar to fast storage, parallel retrieval sends the first successful response object to the higher layer. Parallel retrieval is generally responsive and reliable, but is more susceptible to malicious peers than safe retrieval [1].

Object retrieval mainly fails due to network churn. As safe and parallel retrieval both send retrieval requests to multiple peers, the probability of successfully retrieving an object increases. Multiple retrieval requests also increase the probability of contacting peers that are geographically closer, which increases responsiveness.

Figure 3.9 illustrates Pithos retrieval procedure [1].

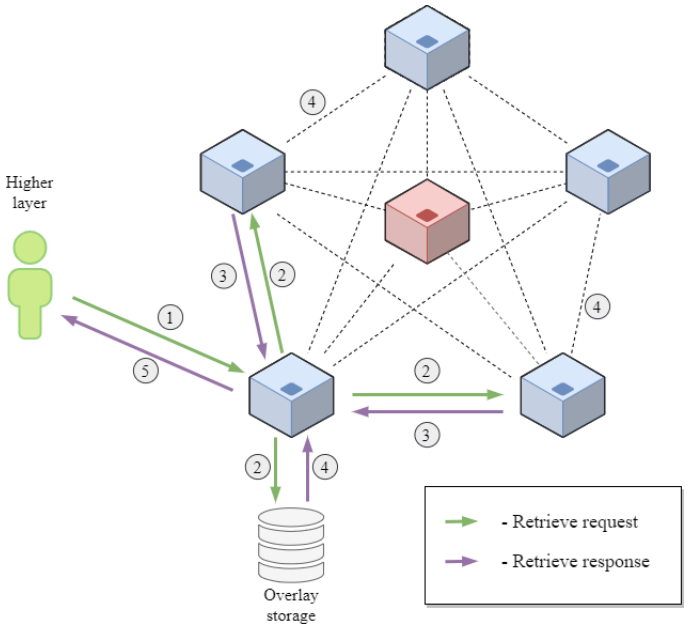


**Figure 3.9:** Pithos retrieval procedure [1]

1. A retrieval request is sent from the higher layer to Pithos' storage interface.
2. Pithos first determines whether the object is stored within the group. If it is, Pithos propagates the retrieval request to both group storage and overlay storage peers. If it is not, a retrieval request is only sent to overlay storage. This is known as an *out-of-group* request.
3. Peers that receive the retrieval request, retrieve the object from their local storage. This object is returned to the originating peer, along with the appropriate response message.
4. Generally, overlay retrieval is less responsive than group storage. When overlay retrieval completes, the result is also sent to Pithos.
5. Pithos responds to the higher layer:
   (a) **Fast retrieval:** Pithos responds with the retrieved object from either overlay or the group storage peer, whichever successful response it receives first.
   (b) **Safe retrieval:** Pithos monitors responses from multiple peers in group storage and overlay storage. A quorum mechanism determines if the request was successful and which object should be sent to the higher layer.
   (c) **Parallel retrieval:** Pithos responds with the retrieved object from either overlay or group storage, whichever succeeds first.

### 3.10.3. Modify

Modify requests are treated similarly to store requests, except that the full object payload is not required. The only parameters that are required for modifying an object is its ID, the field that needs to be modified and its value. Similar to all other storage operations, modify requests are received from the higher layer. Pithos propagates the modify request to both group and overlay storage [1].

Peers receiving the modify request verify that the object is stored in the group. If it is not, the object is only modified in overlay storage. If it is, the object is modified within both group storage and overlay storage.

In order to keep track of object changes, objects are assigned a version number. On each object modification, the version number is incremented within the peer receiving the modify request. In safe retrieval mode, where a request produces objects with different version numbers, the set of objects with the latest version number is sent to the higher layer [1].

### 3.10.4. Delete

Object removal has no explicit interface in Pithos. Instead it relies on an object's time-to-live (TTL). When an object is created in Pithos, it is assigned a TTL, which dictates its "expiry" time. When an object's TTL period expires, the object is removed from both group and overlay storage [1].

The reason for not supporting explicit removals in Pithos is simple. When an explicit removal request is made for an object, certain peers might not receive the removal request due to connectivity issues. This means that the object might be deleted on some peers, but available on others, causing object inconsistency [1].

Object TTL is a reliable way to ensure that objects are consistently removed from the system. Object TTL ensures that stale objects are removed and that storage requirements do not increase indefinitely.

## 3.11. Satisfying Requirements for P2P MMVE Storage

Now that all Pithos' components and mechanisms have been introduced, we can discuss how Pithos satisfies the requirements of P2P MMVE storage. In chapter 2, the storage requirements of P2P MMVEs were identified as responsiveness, reliability, security, fairness and scalability. This section describes which of Pithos' components contributes to which storage requirement.

### 3.11.1. Responsiveness

Pithos achieves responsiveness by splitting the network into fully connected groups of peers. The consistency architecture uses distance-based group-based storage mechanisms to distribute and replicate objects within the system. Peers in a group are more likely to query objects within the group than outside of the group. Since peers within a group are fully connected, the lookup complexity is constant, $\mathcal{O}(1)$. Object replication also enables parallel object retrieval, which means that a peer can request an object from multiple peers simultaneously [1].

#### 3.11.1.1. Group-based Storage

As previously mentioned, fully connected networks can be extremely responsive, but are not scalable, since the required messages scale quadratically. In order to benefit from the low responsiveness of fully connected networks without sacrificing scalability, group storage is used. Group storage in Pithos is implemented by dividing peers that are geographically close to one another in the VE into fully connected groups. Group sizes are smaller than the total size of the network. This effectively achieves a scalable single hop architecture [1].

#### 3.11.1.2. Distance-based Storage

To ensure an even higher degree of responsiveness, distance-based storage is implemented on a group level. Objects that are contained in group's AoI are stored on the peers that collectively form that group. This is called group-based distance-based storage. This approach is based on the assumption that peers are more likely to request objects that are contained in the group [1]. Pithos' backup storage layer, overlay storage, ensures that out-of-group requests can be served.

#### 3.11.1.3. Replication

Objects are replicated within the group, which ensures that multiple peers host the state of a single object. This allows Pithos to implement parallel retrieval and storage requests, which, depending on the implementation, could improve responsiveness. In reality, peers could be geographically distributed all over the world, resulting in a varying degree of latency between peers. This justifies the use of parallel requests, as peers that have lower latencies between each other will be more responsive. Pithos, in *fast* storage/retrieval mode, makes use of the first response from peers to determine a higher layer response, which in general ensures a higher degree of responsiveness [1].

### 3.11.2. Reliability

Reliability is achieved in Pithos by various mechanisms. Reliability is defined as the ratio between the number of successful responses and the total number of requests.

$$reliability = \frac{successful\ responses}{total\ requests} \tag{3.1}$$

Object replication ensures that object replicas exist to provide object availability in the event that a peer hosting a certain object leaves the group due to network churn. Replicas are maintained by scheduled and leave repair mechanisms, which ensures a set number of object replicas are always available. An overlay storage component in the form of a DHT increases reliability as it is able to serve any in- or out-of-group retrieval request [1].

#### 3.11.2.1. Replication

Object replication allows for a higher degree of reliability, as all peers hosting an object have to simultaneously leave the network in order for that object to become completely unavailable. In P2P MMVE systems, resilience to network churn is an important requirement. This makes object replication an essential mechanism to ensure that valid objects are always available. One of the disadvantages of replication is that replicas have to be maintained and their state managed. This leads to higher resource usage for peers and super-peers. It is therefore necessary to take resource constraints into account when choosing a replication factor and repair schedule [1].

#### 3.11.2.2. Repair

Repair mechanisms exist with the single purpose of maintaining object replicas. The repair mechanism ensures sufficient replicas are available in a group. The Pithos architecture states that if an object replica is lost due to network churn, either a leave-repair mechanism or scheduled-repair mechanism will re-replicate the lost object to peers that do not contain the object [1].

#### 3.11.2.3. Overlay Storage

In Pithos, every storage and retrieval request is executed on a group and on an overlay level. This adds a layer of reliability, as it ensures that if an object is not contained in a peer's group storage, it can still be retrieved from the overlay storage [1].

### 3.11.3. Security

Mechanisms used in the Pithos architecture to ensure security are object replication and a quorum mechanism. Pithos additionally makes use of a certification mechanism to ensure

peers are identifiable and to sign object modifications [1].

#### 3.11.3.1. Replication

In Pithos, object replication diminishes the impact of malicious peers in the MMVE. As multiple peers store an object, a maliciously altered version of the object is easily identifiable. Multiple object replicas also enable the use of quorum mechanisms on parallel responses [1].

#### 3.11.3.2. Quorum

Pithos allows parallel object retrieval from multiple peers in a group. When parallel requests are made, depending on the retrieval mode, a quorum mechanism can be used to verify object state. This way, a peer can decide to select only the most populous version of an object. This effectively diminishes the impact of malicious peers in the system and improves security [1].

#### 3.11.3.3. Certification

Pithos requires all peers to be identifiable. This requirement allows Pithos to ensure that only known peers have authorisation to perform storage operations, and to keep track of which peers have created or updated an object [1].

### 3.11.4. Fairness

Distributed computing is one of the key requirements of the P2P MMVE architecture. It is therefore required that load be fairly distributed across the system or, in Pithos' case, the group. To ensure fairness, storage commands are load-balanced between peers in a group and the overlay [1].

#### 3.11.4.1. Group Storage

The group storage mechanism distributes object replicas in a uniformly random fashion, ensuring that no one peer is favoured for storage purposes. Peers use their own copy of the peer ledger to choose random peers in the group when (a) requesting objects or (b) storing object replicas [1].

#### 3.11.4.2. Overlay Storage

For overlay storage, a DHT is used. DHTs map objects and peers to the same identifier space. Objects are stored on peers with the closest ID match. This way, if IDs are assigned in a uniformly random fashion, storage load is also distributed in a uniformly random fashion [1].

### 3.11.5. Scalability

As previously mentioned, in fully connected networks the amount of messages required per request scales quadratically, i.e. $\mathcal{O}(N^2)$. To improve scalability, whilst still benefitting from the advantages of fully connected networks, Pithos groups peers in the VE. This design decision improves the Pithos architecture's scalability compared to fully distributed P2P networks.

One potential performance bottleneck in the Pithos architecture is the directory server. Pithos relies heavily on the directory server for bootstrapping and group migration. In a P2P MMVE where potentially thousands of peers need to be migrated simultaneously, the directory server will become a bottleneck as network participants increase.

## 3.12. Implementation Motivation

Pithos is scalable, decentralised storage network that provides responsive and reliable storage for large P2P environments. In terms of load balancing, Pithos distributes objects fairly among peers in both group and overlay storage [31].

Since none of the reviewed modern storage options meet the requirements of P2P MMVE storage, Pithos is still one of the best storage options for P2P MMVEs. Given that Pithos has previously only been implemented and evaluated as a simulation, a modern real-world implementation is required to verify Pithos' simulated results. This implemented system should be maintainable, modular and cloud ready[1].

From this real-world evaluation, a final conclusion on Pithos' viability for P2P MMVE storage can be made.

## 3.13. Conclusion

This chapter describes the Pithos architecture in detail. These modules and mechanisms ensure that Pithos' use cases can be satisfied whilst adhering to the key storage requirements for P2P MMVEs, namely responsiveness, reliability, security, fairness and scalability.

[1] Cloud ready refers to software and services that are designed to work on the internet and run in the cloud

# Chapter 4

# Nomad Design

Chapter 2 introduced various modern decentralised storage networks. These solutions provide reliable, secure and responsive storage. However, none of these meet all the requirements for P2P MMVE storage. Pithos, on the other hand, does satisfy all the requirements. So far, Pithos has only been evaluated in simulation.

In this chapter, the design of a real-world Pithos implementation, named Nomad[1], is introduced, which aims to implement the modules and mechanisms of the Pithos architecture in order to verify its simulated results. This chapter focusses on various adjustments and modifications made to the original Pithos architecture to realise a real-world distributed storage network. Subsequent chapters will discuss Nomad's implementation details and evaluation.

## 4.1. Design Requirements

To ensure that Nomad adequately corresponds to Pithos' modules and mechanisms, the following requirements should be met:

1. Network configuration
   (a) Suitable peer communication layer.
   (b) Implementation of Pithos' main peer types.
   (c) Peer discovery mechanisms.
2. Group configuration
   (a) Configurable groups.
   (b) Grouping mechanisms.
   (c) A super-peer selection algorithm.
3. Network join, leave and migration mechanisms

[1] "a member of a group of people who move from one place to another rather than living in one place all of the time" [65]

4. State management and persistence
   (a) Group ledgers
   (b) Local storage mechanism.
   (c) Group storage mechanism.
   (d) Overlay storage mechanism.
5. Redundancy
6. State consistency
7. Storage API

## 4.2. Network Configuration

This section provides an overview of Nomad's underlying communication layer, Nomad's main peer types and a description of peer discovery within the network.

### 4.2.1. Peer Communication Architecture

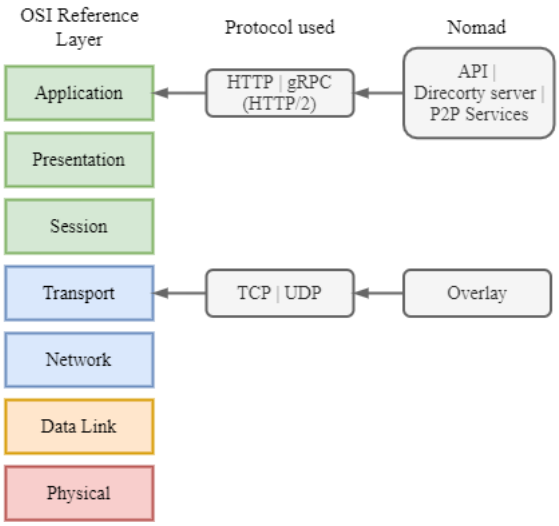


**Figure 4.1:** Nomad peer description in terms of the OSI model Application and Transport layers. [66]

Figure 4.1 provides a simplified description of a Nomad peer in terms of the Open Systems Interconnection (OSI) model [66]. Within the Nomad application different peer components communicate using different protocols. On the highest layer of the OSI model is the application layer. Nomad's application layer makes use of two main communication protocols to enable resource sharing and remote procedure calls. These protocols are Hypertext Transfer Protocol (HTTP) and google remote procedure protocol (gRPC).

HTTP connections are used to communicate with the Directory server and to access Nomad's representational state transfer (REST) application programming interface (API). HTTP is used for accessing API resources, as this is the standard communication protocol for REST APIs. Section B.2.7 provides more information on Nomad's REST API. In Nomad, peers communicate with one another using remote procedure calls (RPCs), gRPC is therefore used as the underlying peer-to-peer service communication protocol, since it provides lightweight serialisation and message compression. Section 5.1.2 provides more information on Nomad's usage of gRPC services.

Nomad's transport layer consists of two communication protocols, namely transport control protocol (TCP) and User Datagram Protocol (UDP). These two protocols are the lowest communication layers in Nomad, and allow messages to be transported from one node to another. Various TCP and UDP connections form Nomad's overlay, which is implemented as a structured overlay in the form of a DHT.

## 4.2.2. Peer Node Types

Nomad consists of three main peer types, namely **peers**, **super-peers** and a **directory server**. This section provides an overview of each peer type and its responsibilities.

### 4.2.2.1. Peer

Similar to Pithos' peer module, discussed in section 3.4.2, Nomad's peers are representative of storage nodes in the network. Peer nodes are mainly responsible for group and overlay storage, but also contain client logic for joining and leaving the network, group migration, maintaining group consistency, maintaining group ledgers and executing object repair.

Nomad's peer implementation deviates from the original Pithos architecture design. As discussed in section 3.4.2, the architecture states that the peer module is responsible for executing maintenance tasks from the super-peer and participating in group and overlay storage. Nomad's design separates these two concerns by implementing two separate modules, namely the *peer* and *peer-storage* modules, which enables the separation of a peer's maintenance and storage responsibilities. This separation of concerns simplifies the required logic classes significantly, since storage modules are now only concerned with storage operations, and maintenance modules with maintenance operations.

A Nomad peer's responsibilities can be summarised as follows:

1. Handling add peer requests from the super-peer, by adding new peers to its local group cache.

2. Handling remove peer requests from the super-peer, by removing peers that have left the group from its local group cache.

3. Handling super-peer leave requests, by closing stale super-peer connections and creating a connection to a new super-peer.
4. Handling repair requests from the super-peer.
5. Forwarding requests to relevant group and overlay storage modules.
6. Keeping track of requests.
7. Informing the directory server of any changes in group or peer server hostnames.
8. Participating in group joins, leaves and migrations.
9. Participating in group migrations.
10. Executing group keep-alive pings.
11. Updating the replication factor if required.
12. Promoting the peer to a super-peer if required.

#### 4.2.2.2. Super-peer

Nomad's super-peers are authoritative nodes in the network. Similar to Pithos' implementation, super-peers are representative of a group and effectively act as gatekeepers, by determining which peers may join their group. Super-peer nodes do not participate in group storage, but in contrast to Pithos', in Nomad super-peers participate indirectly in overlay storage. This means that they act as overlay storage nodes, but do not have an interface that supports direct overlay calls. Super-peers in the Nomad network are mainly responsible for facilitating group join, leave and migration operations, maintaining object replicas, and maintaining a consistent view of peers and objects in the network.

A Nomad super-peer's responsibilities can be summarised as follows:

1. Participating in overlay storage.
2. Handling all group leave and join requests.
3. Facilitating group migration.
4. Ensuring group consistency.
5. Initiating object repair.
6. Maintaining its own group ledger also, referred to as a super-peer ledger.
7. Keeping track of peer positions in the VE.

8. Executing final ping requests to potentially unreachable peers.
9. Informing the directory server of any changes in super-peer server hostnames.
10. Updating the group replication factor if required.

#### 4.2.2.3. Redundant Super-peers

The Pithos architecture states that redundant super-peers should be present in a group [1]. Redundant super-peers should be identical to the current super-peer. These super-peers can then immediately take responsibility for the group if the super-peer leaves. This form of redundancy is crucial in a high churn environment. One flaw in this requirement is that if both super-peer and redundant (back-up) super-peer leave the network, an additional mechanism is required to re-elect a new super-peer.

In Nomad, however, every peer in the group is a redundant super-peer. This means that every peer has an *inactive* super-peer server running alongside its *active* peer server. These inactive super-peer servers are purely used to ensure that if the current (active) super-peer leaves the group, a new super-peer is immediately selected and able to take control of the group.

This super-peer redundancy requires that every application instance contains both client-side and server-side logic for all services, i.e. every instance must contain logic to be either a peer or a super-peer. Additionally, all peer and super-peer group ledgers are required to be identical (storing identical data), in order to ensure that a consistent view of the group is always maintained regardless of super-peers leaving the network. Promoting a peer to a super-peer is therefore as simple as stopping the peer and group storage servers and activating the super-peer server.

#### 4.2.2.4. Directory Server

The Pithos architecture makes use of a directory server to store super-peer IP addresses, keep track of super-peer changes, facilitate new peers in joining the network, and assist peers with group migration, by providing them with relevant information.

Nomad's directory server implementation partially follows Pithos' directory server design requirements, with minor additions to improve in-group and out-of-group communication. Nomad's directory server is mainly responsible for facilitating peers joining the network, group migration and storing super-peer IDs. Additionally, the directory server is used for maintaining group membership, leader election and storing public peer and super-peer data. The above mentioned mechanisms are all discussed in subsequent sections.

Nomad's directory server responsibilities can be summarised as follows:

1. Facilitating network bootstrap and group join.

2. Facilitating group migration.
3. Maintaining group membership cache.
4. Maintaining leaders cache (list of group super-peers).
5. Storing all super-peer hostnames (ip:port).
6. Storing all group-storage-server hostnames.
7. Storing all peer-server hostnames.
8. Storing all overlay hostnames.
9. Facilitating leader (super-peer) election, described in section 4.3.3.

### 4.2.3. Peer Discovery

Nomad makes use of two separate peer discovery mechanisms, one for group storage and another for overlay storage. For group storage peer discovery, Nomad uses a hybrid discovery mechanism. Peers rely on the centralised directory or bootstrap server to discover super-peers that they can connect to. Similarly, super-peers rely on the directory server to discover peers that wish to join their group. When a peer connects to a group for the first time, it relies on the group's super-peer to inform it of other peers in the group. For overlay storage peer discovery, Nomad relies on the underlying DHT's peer discovery mechanisms.

#### 4.2.3.1. Storing Public Hostnames

Nomad's directory server is also used to store other peer information, like DHT, and group and super-peer hostnames. This is necessary, as peers are required to bootstrap to both group and overlay storage. Group storage bootstrap hostnames are always provided by the directory server. DHT bootstrap hostnames are provided by either the directory server or the peer's group cache, which is essentially a dynamic "group information" cache provided by the directory server.

When possible, peers will use their group cache to extract a DHT bootstrap hostname. However, since the first peer in every group is always a super-peer, no group cache is available. When no group cache is available, super-peers (or peers) will contact the directory server directly for a bootstrap DHT hostname.

Nomad's directory server stores *dht*, *group* and *leader* hostname information of all group peers. This gives peers the ability to request hostnames of peers that are not within their own group. Naturally, according to the Pithos architecture this is not a strict requirement, but it does promote a higher degree of developer freedom.

#### 4.2.3.2. Group Storage Discovery

On node start-up, peers automatically connect to the directory server, specified in the application configuration. After a connection is established, the peer broadcasts its hostname i.e. its IP address and port, to the directory server. After the peer successfully informs the directory server of its hostname, the directory server logic takes over in order to determine which group the peer should join.

After determining the group to join, the directory server responds to the peer's request with the representative super-peer hostname. The peer then sends a join request to the received super-peer hostname. If the super-peer accepts the peer, it responds to the request with a list of group objects and a *list of group peers*. This process partially describes Nomad's group join mechanism, which is described in more detail in section 4.4.1. This process can be broadly defined as Nomad's group-peer discovery process.

#### 4.2.3.3. Overlay Storage Discovery

Nomad's overlay storage component uses an existing structured overlay implementation, in the form of a DHT. In order to bootstrap to the network, overlay peers can either make use of cached peer hostnames from previous sessions, or, similar to group storage, make use of a well-known directory server. Unlike group storage's directory server, the overlay directory server can participate in the overlay network whilst maintaining a cache of active overlay peers [10].

## 4.3. Group Configuration

Nomad groups consist of a combination of Nomad's three peer types: (1) A global directory server to enable network bootstrapping, (2) a super-peer to allow group bootstrapping and perform maintenance tasks, and (3) multiple storage peers that participate in group and overlay storage. This section provides an overview of Nomad's group configuration, grouping mechanisms, leader election strategy and group formation.

### 4.3.1. Configurable Properties

Nomad enables users to configure various grouping properties. The main grouping properties that can be configured are *group size, grouping mechanism* and *migration*.

#### 4.3.1.1. Group Size

Since group storage is a fully connected implementation of a structured overlay, Nomad's performance relies primarily on group size. In general, Nomad's group storage provides

highly responsive storage operation. However, in some cases responsiveness is of higher importance than reliability and vice versa.

A trade-off exists between group size and network performance. Smaller group sizes improve responsiveness, since less overhead is required to maintain group consistency. However, smaller group sizes increase the frequency of group migration, which in turn decreases reliability. The opposite is also true. Larger group sizes increase reliability, since the frequency of group migration is reduced. However, since additional overhead is required to maintain group consistency, responsiveness is decreased. In order to give developers the ability to make logical trade-offs, group size is configurable.

#### 4.3.1.2. Grouping Mechanisms

Since Nomad uses group storage, a mechanism for logically grouping peers is required. Because Nomad has more than one grouping mechanism available, its configuration properties make it possible to select a grouping mechanism. Section 4.3.2 provides more information on Nomad's grouping mechanisms.

### 4.3.2. Grouping Mechanism

As mentioned in section 3.5.1, Pithos-based systems make use of group-based storage. Nomad therefore requires a grouping mechanism to logically group peers that are geographically close to one another in the virtual world for optimal group interactions and performance. Two logical grouping mechanisms can be used in Nomad: either *random grouping* or *Voronoi grouping*.

#### 4.3.2.1. Random Grouping

When Nomad's random grouping mechanism is used, peers are grouped sequentially. This means that a peer joining the network is directed towards the first sequential group that has an available peer position. Availability is determined by the *max peer* configuration property of the group super-peer. Super-peers therefore determine whether a peer can join their group or not. If no group has an available peer position, a new group is formed and the peer is promoted to a super-peer. Similarly, when all peers leave a group, the directory server removes the group from its cache to ensure no peers attempt to join a non-existent group.

#### 4.3.2.2. Voronoi Grouping

Nomad uses a *Voronoi grouping* [64] mechanism, to logically split the world into super-peer oriented sections. A Nomad Voronoi map consists of finitely many points, called *site points* which are representative of super-peer locations. Each site point has a corresponding

region, called a *Voronoi cell*, which is representative of a group's AoI. When Voronoi grouping is enabled, Nomad peers and super-peers construct their own Voronoi map, using site points from the directory server. A Voronoi map therefore segments the VE into a series of polygons, centred on super-peer locations.

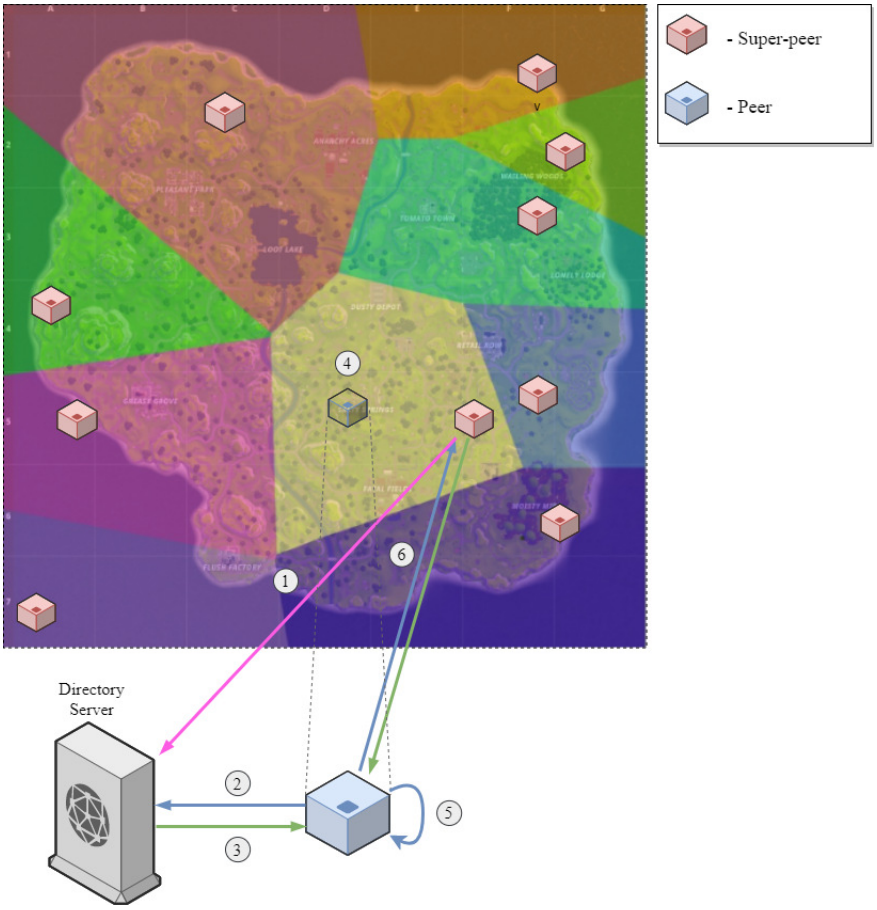


**Figure 4.2:** Nomad Voronoi grouping mechanism, visualised using an online Voronoi map generator [67], and an arbitrary *Fortnite* map [68]

Figure 4.2 illustrates the Voronoi grouping process.

1. When Voronoi grouping is enabled, Nomad super-peers store their hostname and VE location to the directory server. The directory server essentially maintains a cache of all Voronoi site points.

2. Each peer requesting to join the network connects to the directory server and requests all super-peer locations, i.e. Voronoi site points.

3. The directory server responds to the request with a list of super-peer IDs and Voronoi site points, which is used to construct the peer's Voronoi map.

4. After the Voronoi map has been generated, a peer uses its own location to determine which segment of the Voronoi map it occupies.

5. The segment site point, which contains the super-peer hostname, is then extracted and used for bootstrapping to the group's super-peer.

6. The peer then follows the usual join process.

### 4.3.3. Super-peer Selection Mechanism

In order to assign a super-peer to each group, a leader or super-peer selection mechanism is required, to ensure that super-peers are selected deterministically. Nomad uses its directory server to select super-peers, since it maintains a consistent cache of the entire network. The directory server uses a sequence-based super-peer selection algorithm to establish peer hierarchy. This means that at any given time, the "first" peer in the group is always the super-peer.

Since the directory server is used for super-peer selection, it can potentially become a performance bottleneck. The directory server implementation should therefore be vertically and horizontally scalable and maintain strong consistency between instances. Scaling the directory server vertically or horizontally will ensure that it maintains functionality as network size increases. Since horizontal scaling involves adding additional nodes, separate directory server instances should maintain strong consistency to ensure a consistent network cache. Nomad's directory server implementation, described in Section 5.1.3 and 5.3.8, was specifically selected to satisfy all these requirements.

Nomad's super-peer selection implementation is discussed in section 5.4.1.

### 4.3.4. Group Formation

In order to form groups, Nomad relies on its group configuration, grouping mechanism, and leader election strategy. Peers joining the network are directed towards the group that the internal grouping mechanism has determined it should join. Availability is determined by the *max peer* configuration property of the group super-peer. Super-peers therefore determine whether a peer can join their group or not. If no group has an available peer position, a new group is formed and the peer is promoted to a super-peer. Similarly, when all peers leave a group, the directory server removes the group from its cache to ensure no peers attempt to join a non-existent group. It is therefore clear that peers joining and leaving the network impact the number of groups within the Nomad network.

#### 4.3.4.1. Migration

Since Nomad is specifically designed for MMVEs where users are able to move around in a virtual world, a migration or hand-off mechanism is required to move peers between groups. Since not all MMVEs are concerned with a peer's location in a virtual world, it is

possible to completely disable group migration in Nomad's configuration properties. If Migration is disabled, no movement updates are possible for peers.

## 4.4. Group Join, Leave and Migration Mechanisms

Now that Nomad's network and group configuration are defined, the topic of network and group bootstrapping can be discussed. This section provides more information on Nomad's group join, leave and migration mechanisms.

### 4.4.1. Group Join Mechanism

Super-peers represent groups within the Nomad network, and act as gatekeepers, deciding which peers may join the network. When a super-peer is elected in the network, it shares its bootstrap hostname with the directory server.

On node start-up, peers automatically connect to the directory server, specified in the application configuration. After a connection is established with the directory server, the peer's client logic takes over in order to determine which group should be joined. If Voronoi grouping is enabled, the group to join is determined by the grouping mechanism discussed in Section 4.3.2.2. If no grouping algorithm is used, the client logic will select the first available group to join.

After determining the group to join, a request is sent to the directory server containing the super-peer ID. The directory server responds to the request with the super-peer hostname. The peer then sends a join request to the super-peer, which can either reject or accept the peer into its group. By default, peers will always determine beforehand if a super-peer contains an available position, if it does not, it will send a join request to one of the neighbouring groups. If no available group positions exist, the peer will be promoted to a super-peer and a new group will be created.

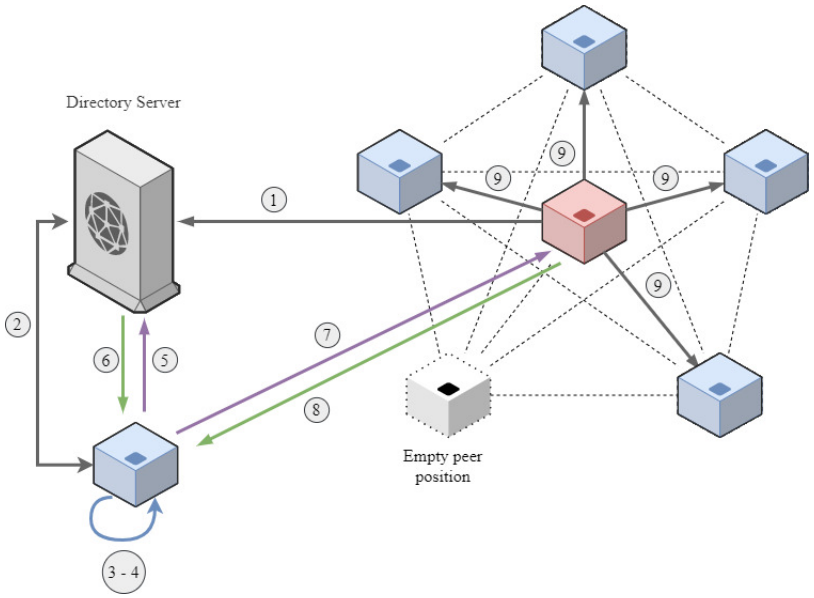


**Figure 4.3:** Nomad group join mechanisms, used to bootstrap to the Nomad network

Figure 4.3 illustrates the join procedure, which allows peers to join the Pithos network.

1. A super-peer shares its bootstrap hostname with the directory server. This can be used to join that super-peer's group. It is assumed that this node was previously elected as a super-peer (see section 5.4.1). The super-peer server hostname is stored to the directory server.

2. A new peer establishes a connection with the directory server, and sends a join request.

3. The peer's internal directory server client logic ensures the peer is added to the directory server's group cache and determines the group to join (results may vary depending on grouping mechanism used). The peer rules out groups that do not have any available peer positions.

4. The peer initialises its overlay storage component and bootstraps to the overlay network.

   (a) The peer requests a DHT hostname from one of its potential group members, which it uses to bootstrap to the overlay network.

   (b) If no other peers exist in the network, the peer becomes the initial bootstrap node.

5. The peer then requests the corresponding super-peer hostname from the directory server.

6. The directory server responds to the request with the hostname of the super-peer of the group the peer should join.

7. The peer sends a join request to the super-peer, containing peer information, such as server hostnames and its VE location.

8. The super-peer decides whether it wants to accept or reject the peer, based on the current group size and the peer's position in the VE.

   (a) If the super-peer accepts the peer, it provides the peer with a list of peers and objects in the group. The peer uses this information to populate its own group ledger.

   (b) If the super-peer rejects the peer, the peer will send a join request to a neighbouring super-peer.

   (c) If all super-peers reject the peer, no available positions exist and the peer is promoted to a super-peer.

9. (Assuming the peer was accepted) The super-peer notifies all peers within the network that a new peer was added.

### 4.4.2. Group Leave Mechanism

Nomad's leave mechanism is identical to that of Pithos, which is described in Section 3.6.2. The only addition, is that peer data stored to the directory is automatically removed from the directory server cache if the peer leaves the group. This includes the peer's group storage hostname, overlay storage hostname, peer hostname and redundant super-peer hostname.

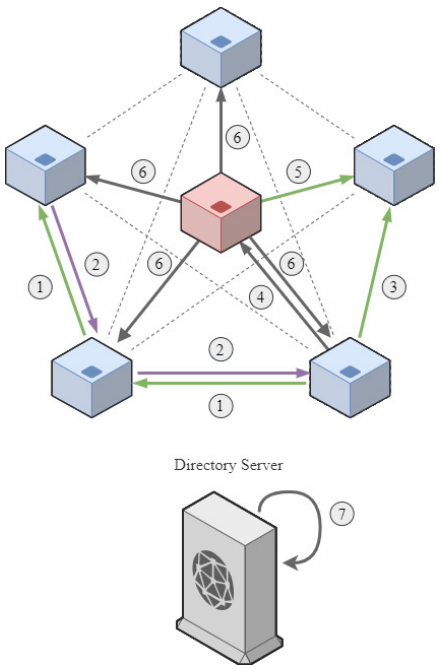


**Figure 4.4:** Nomad leave mechanism, used if a peer unexpectedly leaves the group

Figure 4.4 illustrates the group leave procedure, which ensures group consistency if a peer leaves the group unexpectedly.

1. A peer routinely sends a keep-alive ping request to another random peer within its group.
2. If a peer receives a keep-alive ping, it acknowledges the message with a pong.
3. If a peer does not acknowledge the keep-alive ping within a specific time window, the peer is considered to have left the network.
4. The original sender of the keep-alive ping notifies the super-peer that a peer is unreachable.
5. The super-peer sends a ping of its own to the peer in question, to verify that the peer has actually left the network.
6. If a peer does not acknowledge the super-peer ping within a specific time window, the super-peer informs all peers that the peer has left the group.
7. The leaving peer is removed from all group ledgers and ephemeral data, like the peer's group storage hostname, overlay storage hostname, peer hostname and redundant super-peer hostname, and is automatically removed from the directory server and group caches.

### 4.4.3. Super-peer Leave Mechanism

Similar to normal peers leaving the network, super-peers can also leave the network due to network churn. As peers are representative of a group, a new leader is required, in order to retain group functionality. As discussed in section 4.2.2.3, Nomad uses redundant super-peers to streamline the super-peer re-election process. Implementation details are provided in section 5.4.6.

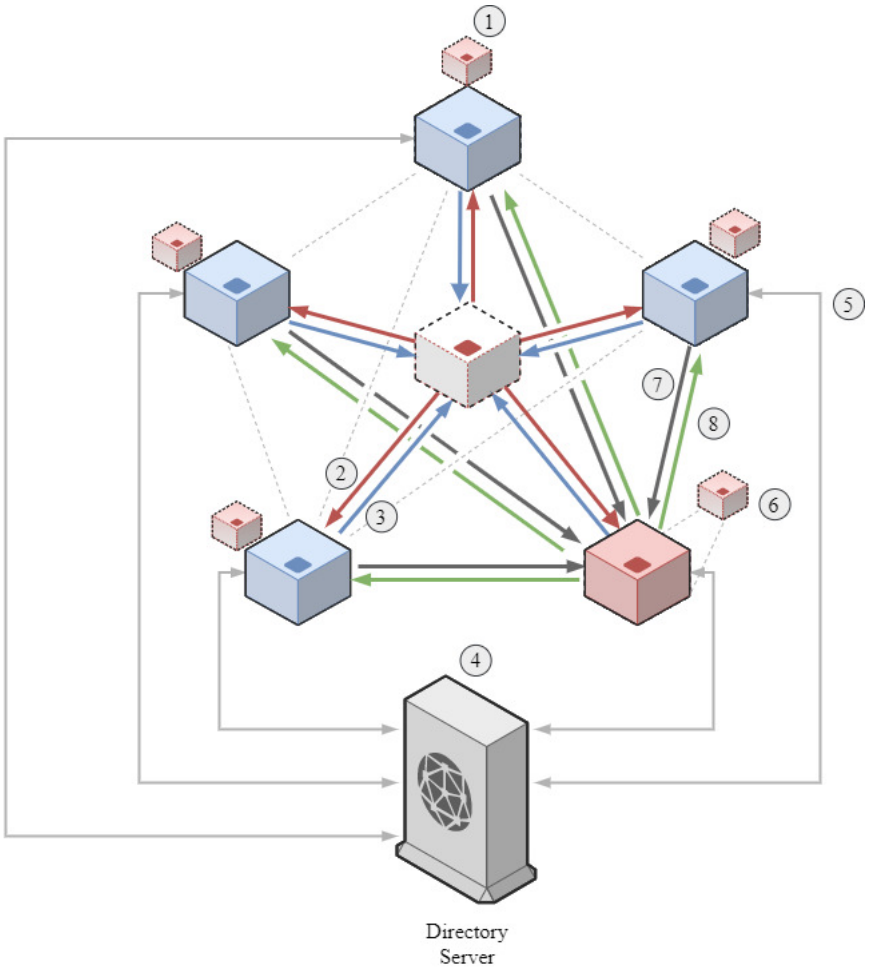


**Figure 4.5:** Illustration of Nomad's super-peer leave mechanism

Figure 4.5 illustrates Nomad's super-peer leave procedure and the process of re-electing a new super-peer.

1. Every peer in the group has a super-peer server running in the background. The super-peer mechanism is disabled and the redundant super-peer servers do not receive any traffic. The directory server caches all active and redundant super-peer hostnames per group, and uses these for leader selection. Each peer in the group also listens for a promotion message, which will indicate that it is the new super-peer.

2. If the super-peer ends its session and wishes to leave the group, it notifies all peers in the group that it is about to leave.

3. Every peer has to acknowledge the super-peer leave before the handover can be finalised. If a peer does not acknowledge the super-peer leave command, the request will be re-sent to the peer for a maximum of 5 times. If the peer still does not respond, it will no longer be able to participate in the group.

4. After the previous super-peer has left the network, the directory server automatically selects a new leader. Each peer contacts the directory server, requesting the id of

the new super-peer. Since Nomad's leader selection is facilitated by the directory server, peers are dependent on the directory server for new super-peer information.

5. The directory server responds with the id of the super-peer.

6. The new super-peer is notified by its internal listener that it has been selected as the new super-peer and immediately takes lead of the group. The peer stops all group related services and activates all super-peer related services.

7. Every other peer in the network will send a simplified join request to the new super-peer, to ensure that a good connection is formed with the new super-peer. The re-join strategy also ensures that all peers maintain a consistent view of group changes and objects that might have been added during the leader re-selection.

8. The new super-peer accepts all the previous group peers and sends its group ledger to all peers to ensure ledger consistency. All conflicts are overridden by the super-peer's ledger.

### 4.4.4. Group Migration Mechanism

As discussed in section 3.6.3, peers move around in a VE, which mean they can cross group AoI boundaries frequently. This requires an efficient migration mechanism that will not severely affect the user's experience. One of the weaknesses observed with the Pithos migration mechanism is that it relies heavily on the directory server in order to migrate peers to the correct geographical groups, which at some point will become a bottleneck. To solve this, Nomad delegates the group migration responsibilities to the super-peer of each group. This simplifies the required logic of the directory server to a simple cache mechanism.

Because Nomad's directory server is simply a caching mechanism for important group and peer information, no additional logic is required server-side to facilitate group migration. Migration logic is implement client-side, and is the responsibility of each super-peer in the network.

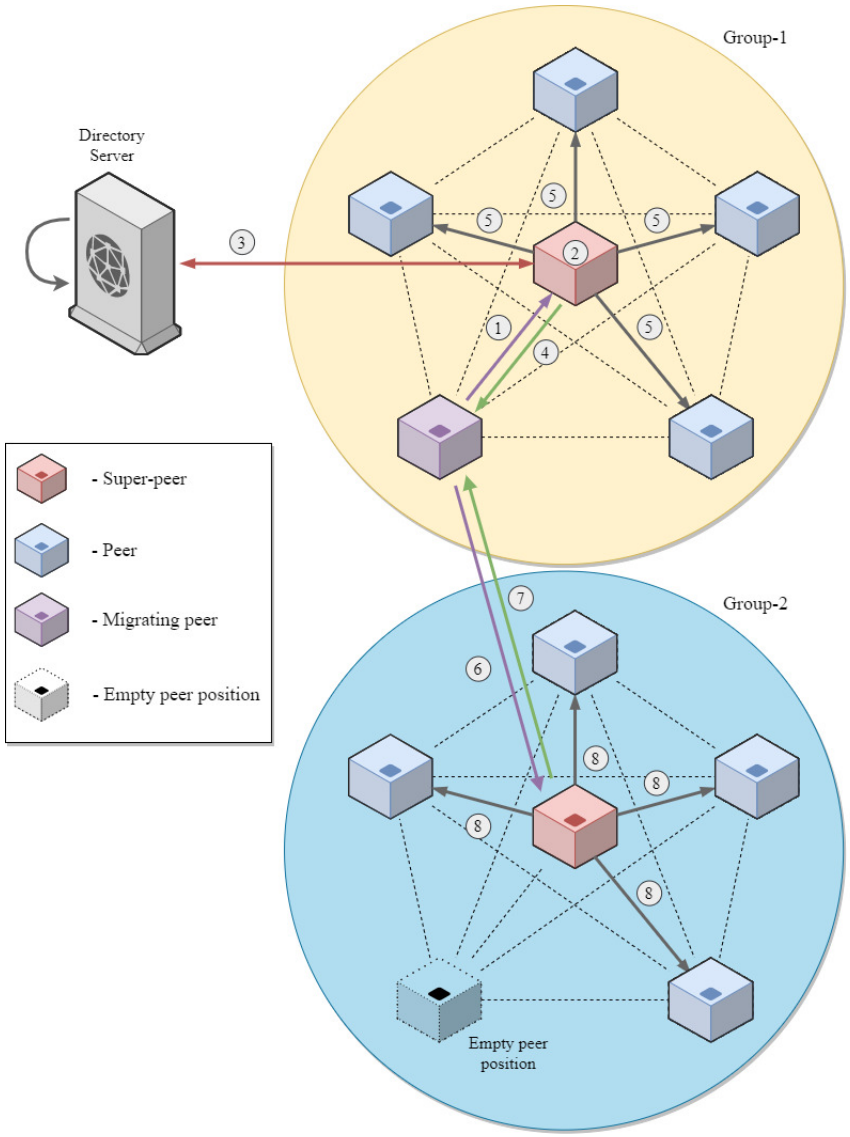


**Figure 4.6:** Nomad group migration mechanism.

Figure 4.6 illustrates Nomad's group migration procedure, which allows peers to migrate from one geographical group to another.

1. A peer routinely sends positional updates to its super-peer.
2. The super-peer server evaluates the peer's position and responds with the super-peer ID for the group within which the peer's position falls.
   (a) If the super-peer ID matches the peer's current group super-peer ID, the super-peer simply responds with an ACK. The peer is still in the correct group and does not need to be migrated.
   (b) If the super-peer ID does not match the peer's current group super-peer ID, the peer is no longer in the correct group. The peer needs to be migrated.

3. The super-peer's internal directory server client logic, which makes use of the grouping mechanism, determines which group the peer should be migrated to. If the peer cannot be migrated to a neighbouring group, the peer is promoted to a super-peer.

4. The super-peer responds with a new group name and super-peer hostname for the group which the peer should migrate to.

5. The peer sends a leave request to its current super-peer. Once the peer notifies the super-peer that it has left the group, the peer truncates its group ledger and local storage.

6. The super-peer notifies all peers in the group that the peer has left.

7. The peer sends a join request to the new super-peer. (The join request procedure is followed)

8. If the new super-peer accepts the peer, it provides the peer with a list of peers and objects in the group. The peer uses this information to populate its own group ledger.

9. (Assuming the peer was accepted) The new super-peer notifies all peers within the network that a new peer was added.

## 4.5. State Management and Persistence

Now that Nomad's group join, leave and migration mechanisms have been defined the topic of state consistency can be discussed. As discussed in Section 2.1, the state consistency architecture has two responsibilities, namely state management and state persistence.

Nomad uses various components to achieve state management and consistency within the network, namely group ledgers, local storage, group storage and overlay storage. The following subsections provide information on state persistence and state management in Nomad.

### 4.5.1. State Persistence

State persistence is responsible for persisting objects in long term storage. As in the Pithos architecture, Nomad achieves state persistence by making use of an existing structured overlay implementation in the form of a DHT.

#### 4.5.1.1. Overlay (DHT) Storage

According to the Pithos architecture, overlay storage is responsible for maintaining a backup object store, which improves reliability of object retrieval. In the Pithos evaluation

(see Tables 3.1 and 6.2), it was found that overlay storage is highly reliable and scalable. Nomad therefore follows Pithos' overlay storage component design. This design decision ensures that results from Nomad's evaluation are comparable to those of Pithos, since their underlying storage architectures are the same. Section 5.3.4 provides more information on Nomad's overlay storage component.

In Nomad, both peers and super-peers participate in the overlay storage network, in order to ensure an even higher degree of reliability. This is contrary to Pithos' architecture, where super-peers do not participate in any storage. Even though super-peers contribute their storage resources to the overlay network, they do not support any higher layer overlay requests.

A Nomad peer's overlay storage responsibilities can be summarised as follows:

1. Handling overlay store, retrieve and modify requests.
2. Maintaining an object store, with group objects.
3. Maintaining a consistent view of all peers within the overlay.
4. Maintaining a consistent view of all objects within the overlay.
5. Handling of overlay join and leave requests.
6. Routing of requests to the responsible peers.

## 4.5.2. State Management

State management is responsible for managing object state in the VE. As in the Pithos architecture, Nomad achieves responsive and reliable state management by making use of group ledgers, group storage and overlay storage. The following subsections describe group ledgers and group storage respectively, focusing on how they participate in state management.

### 4.5.2.1. Group Ledgers

Similar to the Pithos architecture, Nomad peers make use of an abstract data structure, called group ledgers, in order to keep track of all peers and objects within a group. Note that Nomad's implementation does vary from that of Pithos, discussed in section 3.7.5.

According to the Pithos architecture, group ledgers contribute to highly responsive group storage operations, by physically storing the object information within the ledger. All objects in Pithos are effectively stored in-memory. Pithos' group ledgers therefore have two responsibilities: in-memory storage and determining which peers store which objects.

Section 4.5.2.4 shows that in-memory storage may not be suitable for MMVEs since it requires large amounts of objects to be stored in memory.

Nomad peers make use of a separate local storage component to store objects, whereas the group ledger is exclusively used to keep track of peers and objects within the group. This separation of concerns simplifies Nomad's group ledger implementation. Therefore, instead of storing the entire object, the *object ledger* only stores object IDs and a *metadata object*, which represents the peer ID and a last-modified date. Similarly, *peer ledgers* only store peer IDs and a *metadata object*, which represents the object ID and a last-modified date. The last modified date contained in each entry's metadata is used to remove expired data from the group ledger. Nomad relies on more mature and efficient technologies for storage, and is only required to maintain IDs in the ledger. Section 5.3.6 provides more information on Nomad's group ledgers.

Figure 4.7 provides an illustration of Nomad's group ledger data structure.

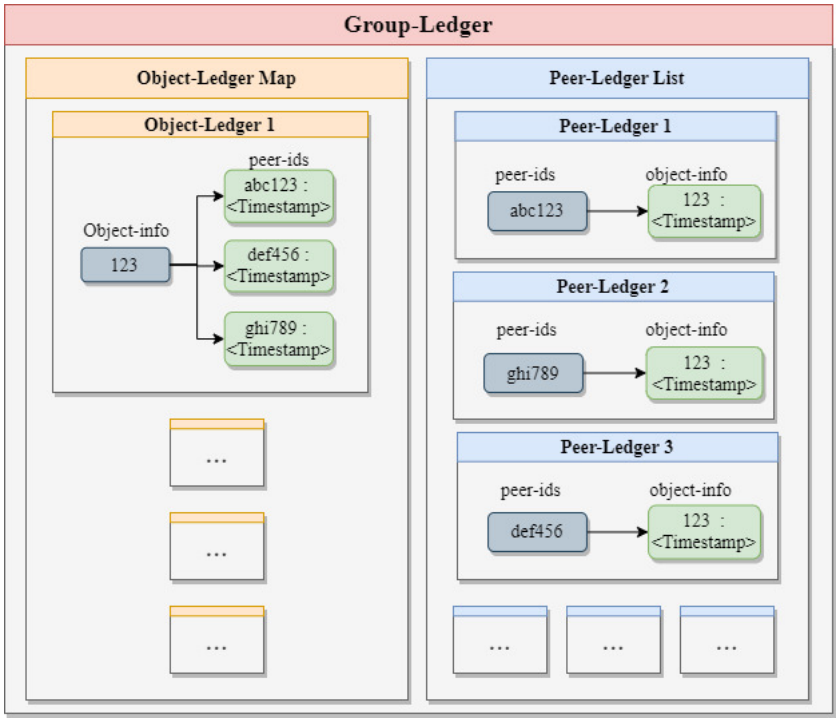


**Figure 4.7:** Nomad group ledger data structure

A group ledger's responsibilities can be summarised as follows:

1. Determining whether an object is stored in a group.
2. Determining whether an object is stored on a specific peer in the group.
3. Adding objects and peer references.
   (a) Adding new objects to its object and peer ledgers.
   (b) Adding new peers to its object and peer ledgers.

4. Removing objects and peer references.
   (a) Removing expired objects from its object and peer ledgers.
   (b) Removing peers that have left the group from its object and peer ledgers.

#### 4.5.2.2. Super-peer Group Ledgers

Similar to the Pithos architecture, super-peers in Nomad are responsible for maintaining group consistency and object repair. Super-peers make use of a group ledger to keep track of all peers and objects within the group. It should be noted that in contrast to Pithos, peer and super-peer group ledgers store exactly the same information, i.e. no discernment between the two ledgers can be made. Nomad therefore does not refer to super-peer group ledgers as "super-peer ledgers", but as normal "group ledgers".

This intentional combination of the super-peer and group ledger allows peers to easily and seamlessly become super-peers, since no additional ledger information is required at any point.

#### 4.5.2.3. Peer Storage

This section provides additional information on Nomad's primary storage components.

Nomad's peer-storage module can be seen as the top level storage module, which controls local, group and overlay storage. The peer storage module provides a basic interface for *retrieve, store* and *modify* operations, which are used by Nomad's storage API. Since only group storage peers are required to serve storage operations, when a peer is promoted to a super-peer, the peer storage component is disabled.

As will be discussed in Section 5.5, Nomad supports various storage and retrieval modes, to satisfy different storage requirements. For the purpose of this and subsequent sections, it is sufficient to note that Nomad supports parallel requests.

#### 4.5.2.4. Local Storage

In contrast to Nomad's design, Pithos' implementation ensures highly responsive group storage operations, by exclusively using in-memory storage. This is generally acceptable in decentralised storage networks. However, as section 2.5.3 states, P2P MMVEs have high storage requirements, and can potentially require storage of hundreds of thousands of objects. When storing large amounts of data in-memory, peers with lower available resources may become severely impacted. The requirements of local storage will vary per implementation, since requirements depend heavily on VE object density and group AoI.

Nomad therefore supports two high-level storage modes, namely *in-memory* and *disk-based* storage. In-memory storage is used when resource usage is not a concern, whereas disk-based storage is used when peers are expected to store an excessive amount of data.

For in-memory local storage, a lightweight relational database (RDB) is used. For disk-based local storage, an efficient key-value store is used. Section 5.3.2.1 provides more information on Nomad's local storage component.

### 4.5.3. Group Storage

According to the Pithos architecture, group storage is used to manage object state. In the Pithos evaluation, group storage was found to be highly responsive and reliable. Table 6.2 provides Pithos' evaluated group storage results. In Nomad, only storage peers participate in group storage, which is in line with Pithos' group storage component described in section 3.7.4.

The peer storage module is responsible for Nomad's group storage mechanism. Group storage is an $\mathcal{O}(1)$ implementation of a structured, fully connected overlay. As illustrated by Figure 4.8, all group peers' local storage instances are combined to form group storage. Group storage requests therefore only require a single hop to reach any group peer, which ensures high responsiveness and constant lookup times, given the requested object is contained within the group. Section 5.3.3 provides more information on Nomad's group storage component.

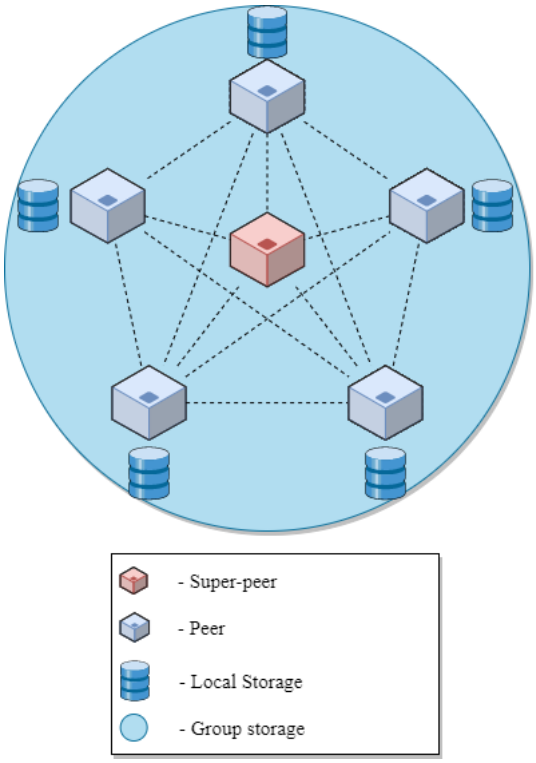


**Figure 4.8:** Nomad group storage architecture

A Nomad peer's group storage responsibilities can be summarised as follows:

1. Handling group store, retrieve and modify requests.

2. Maintaining an object store, with a subset of group objects.
3. Maintaining the peer's group ledger.
   (a) Maintaining a consistent view of all peers within the group.
   (b) Maintaining a consistent view of all objects within the group.
4. Maintaining group membership on behalf of the peer.
5. Monitoring the availability of peers in the group with scheduled network pings.
6. Notifying peers and the super-peer of new objects.
7. Handling repair requests from the super-peer.

#### 4.5.3.1. Overlay (DHT) Storage

Section 4.5.1.1 provides information on Nomad's overlay storage component. Since not all requests can be guaranteed to be served by group storage, the overlay storage is required to serve out-of-group requests. This means that for out-of-group requests, the state of an object is determined solely by the overlay storage. DHTs typically have built-in quorum mechanisms.

## 4.6. Redundancy

Now that it is clear how Nomad ensures state persistence and manages object state, the topic of redundancy in Nomad can be discussed. Nomad primarily makes use of object replication and repair mechanisms to provide a level of redundancy in the storage network. The following subsections provide an overview of Nomad's replication mechanisms and how replicas are maintained using a repair mechanism.

### 4.6.1. Replication Mechanism

Replication is used in Nomad to ensure object availability and to allow the use of a quorum mechanism to verify objects. Nomad supports a configurable replication factor. For every object stored in the Nomad network, $\mathcal{R}$ object replicas are created. Replicas are randomly distributed amongst all the peers within the group in order to promote fairness amongst peers in terms of resource usage. In Nomad, the replication factor can be configured on a per peer basis. When a peer connects to a group, it will automatically inherit the replication factor set by the super-peer and its local replication factor will be overridden.

### 4.6.2. Repair Mechanism

As Nomad is intended to be used in a high churn environment, peers leaving the network will lead to object replicas being destroyed. Nomad therefore requires a repair mechanism to ensure $\mathcal{R}$ object replicas are always available. Nomad's repair mechanism is identical to Pithos' repair mechanism, as it too consists of *periodic (scheduled) repair* and *leave repair*.

#### 4.6.2.1. Periodic Repair

During scheduled repair, a super-peer periodically checks that $\mathcal{R}$ object replicas are stored in the group, by checking its group ledger. If an object is found to have insufficient replicas, a repair request is sent to peers that do not already store the object. Scheduled repair is useful, as peers might not leave the group gracefully, which means that leave repair is never executed.

#### 4.6.2.2. Leave Repair

Leave repair is initiated when a peer leaves the group, and object replicas are destroyed. During leave repair, super-peers pre-emptively create new replicas of all objects stored by the peer leaving the group. Using leave repair ensures that $\mathcal{R}$ object replicas are always available if peers leave the group gracefully. The repair mechanism can therefore be seen as an extension of the leave mechanism.

Figure 3.7 illustrates the object repair procedure which ensures $\mathcal{R}$ object replicas are always available.

## 4.7. Object State Consistency

Previous sections have shown how Nomad distributes object replicas and maintains them. Next, the topic of object state consistency can be discussed. Nomad primarily makes use of its group ledgers to ensure group consistency, but object state can vary between peers. This means that a mechanism to ensure object state consistency in the network is required. Nomad makes used of a quorum mechanism to achieve object state consistency.

### 4.7.1. Quorum Mechanism

Nomad's quorum mechanism is similar to Pithos' quorum mechanism described in section 3.9.1. Nomad's mechanism can be described by the simple quorum formula, $(\mathcal{R}/2) + 1$, where $\mathcal{R}$ is the replication factor. The mechanism is implemented within the group storage module, and when active, Nomad executes multiple object retrievals from unique peers. The quorum mechanism compares objects and verifies that the object replicas are

consistent. The object value that occurs the most amongst all responses is considered to be the true object value.

### 4.7.2. Resolving Object State Conflicts

Due to network disturbances and delays, it can happen that responses from two peers storing identical object replicas have contradicting states. This is called an object state conflict. In order to resolve such conflicts, Nomad's quorum mechanism selects the object state that is provided by the majority of peers from group and overlay storage. If no majority can be reached, the retrieval request will fail. Since overlay storage also makes use of replication, similar quorum mechanisms are used to resolve state conflicts. Since Nomad merely implements an existing DHT, the details of overlay storage conflict resolution will not be discussed.

## 4.8. Satisfying Requirements for P2P MMVE Storage

Now that all of Nomad's components and mechanisms have been introduced, we can look at how Nomad satisfies the requirements of P2P MMVE storage. Section 3.11 described how the underlying Pithos architecture satisfies the requirements of P2P MMVE storage. By following Pithos' design, Nomad inherently satisfies the requirements for P2P MMVE storage. This section builds on Section 3.11, by describing the adjustments made to the Pithos architecture for Nomad's design.

### 4.8.1. Responsiveness

Nomad achieves responsiveness by splitting the network into fully connected groups of peers.

### 4.8.2. Group Ledger

Nomad's group ledger does not store any object values, but merely acts as a ledger that is used to locate objects. Additionally, to ensure that failed storage requests are not added to the group ledger, object IDs are only added once local storage confirms that the object was stored successfully.

### 4.8.3. Local Storage

Instead of storing objects in a group ledger, Nomad relies on more advanced storage implementations, like relational databases and key-value stores, which ensures optimal lookup times for stored resources. Additionally, Nomad's local storage component supports in-memory storage to reduce lookup time of stored resources even further.

#### 4.8.3.1. Grouping Mechanism

Nomad supports two grouping mechanisms, namely random and Voronoi grouping, which divide the VE into geographical groups. This ensures that the majority of requests within a geographical group can be served from within the group.

### 4.8.4. Reliability

Reliability in Nomad, as in Pithos, is defined as the ratio between the total number of successful responses and the total number of requests.

$$reliability = \frac{successful\ responses}{total\ requests} \tag{4.1}$$

Reliability is achieved in Nomad by making use of the various concepts and mechanisms implemented in the Pithos architecture. Section 3.11.2 describes how the underlying Pithos architecture satisfies its reliability requirements.

#### 4.8.4.1. Directory Server

Nomad caches all group information, such as peer, group storage and DHT hostnames, within the directory server. This ensures that important group information is always available to peers within the network. Group caches are made available to peers, who can use cached hostnames to form direct connections with other group peers and to bootstrap to the DHT. If a peer's group cache is unavailable, direct calls to the directory server can still be used to access important group information.

#### 4.8.4.2. Local storage

To ensure peers are not overloaded by the number of objects in the group, Nomad makes use of a separate local storage component in the form of an RDB or key-value store. The local storage component supports both in-memory and disk-based storage operations to serve different VE use cases. Separating ledger and storage logic simplifies the required logic of the ledger and local storage modules.

#### 4.8.4.3. Retry mechanisms

As an additional reliability layer, Nomad uses retry mechanisms to ensure that failed directory server and peer requests are retried. This ensures that requests to the directory server or another peer do not fail due to temporary network disturbances. Nomad's retry mechanism improves the overall stability and robustness of the storage network.

### 4.8.5. Security

Two mechanisms are used in Nomad to ensure security: object replication and a quorum mechanism. The Pithos architecture additionally makes use of a certification mechanism to ensure peers are identifiable and to sign object modifications.

For the purpose of this work, a certification mechanism was not included. Since Pithos' certification mechanism does not add excessive computational overhead, it was determined to be non-essential for measuring the performance.

### 4.8.6. Fairness

Distributed computing is one of the key requirements of the P2P MMVE architecture. It is therefore required that load be fairly distributed across the system or in Nomad's case, the group.

#### 4.8.6.1. Directory Server

To ensure fairness, peers and super-peers are not selected on any discriminating factors, but based on the order of peers joining a group. Nomad's directory server implements a simple sequence-based leader selection algorithm, which ensures that the first peer in the group becomes the super-peer.

#### 4.8.6.2. Group Ledger

Nomad's group ledgers are identical for peers and super-peers. Consistency is ensured in group ledgers by requiring peers to inform all group participants when objects have been added to its local storage. This change to the group ledgers ensures that Nomad peers can seamlessly become super-peers when required.

### 4.8.7. Scalability

As previously mentioned, in fully connected networks the amount of messages required per request scales quadratically - $\mathcal{O}(N^2)$. To improve scalability, whilst still benefitting from the advantages of fully connected networks, Pithos groups peers in the VE. Some adjustments were made to Pithos' directory server with the aim of improving scalability and reducing the probability of a bottleneck.

### 4.8.8. Directory Server

Nomad moves its migration logic to super-peers, instead of relying on the directory server to migrate peers. This simple change reduces the required directory server logic and the

likelihood of a bottleneck. Nomad's directory server is however still a concern to the systems scalability.

## 4.9. Nomad Storage API

Section 3.2 introduced the Pithos architecture's use cases. The interface consists of four operations: **store, retrieve, modify** and **remove**. In order to satisfy these use cases, Nomad provides a simple REST API with the following endpoints:

- Storage: /storage/put
- Retrieval: /storage/get/{objectId}
- Modification: /storage/update

Similar to Pithos' API, no explicit removal endpoint is required, since object removal is handled internally based on object TTLs. Section 5.5 provides a detailed overview of Nomad's storage, retrieval and modification procedures.

## 4.10. Conclusion

This chapter introduced Nomad, the first real-world implementation of the Pithos architecture. The key modules and mechanisms of Nomad were introduced and discussed in light of the adjustments made to the underlying Pithos architecture.

The most noticeable adjustments made to the architecture are the following:

1. Separated peer maintenance and storage logic.
2. Separated the group ledger's storage and ledger responsibilities. Nomad stores objects in a local object store and implements a simple group ledger for keeping track of object and peer IDs.
3. Added grouping mechanisms i.e. Voronoi or random grouping.
4. Added a super-peer selection using the directory server.
5. Added directory server caching mechanisms for peer and super-peer locations and critical group and overlay information.
6. Moved group migration logic from the directory server to super-peers.
7. Added retry mechanisms to directory server and peer requests to increase network reliability and robustness.

The next chapter will discuss Nomad's implementation in terms of frameworks and technologies used.

# Chapter 5

# Nomad Implementation

The previous chapter presented a high level design of Nomad, a distributed storage network based on the Pithos architecture. Additionally, it provided an overview of Nomad's key modules and mechanisms and how the designed system satisfies the key requirements of P2P MMVE storage.

This chapter provides an overview of the various tools and technologies used for Nomad's implementation. Furthermore, this chapter provides information on how these are used to satisfy the primary use cases of a P2P MMVE storage network, namely **store, retrieve, modify** and **remove**.
Nomad's repository information is provided in Appendix H.

## 5.1. Nomad Technical Stack: Frameworks and Libraries

In this section, the various frameworks and tools used Nomad's implementation are introduced. An in-depth explanation of these technologies is believed to be beyond the scope of this work, and will therefore not be provided.

### 5.1.1. Implementation Language

Nomad was designed to be modular and extensible, in order to encourage collaboration. To ensure the longevity and maintainability of the project, a popular programming language which follows an object orientated approach was required. A myriad of suitable languages exist, with Java [69], C++, Golang [70] and Kotlin [71] being some examples. From the considered languages, the authors chose Java as the implementation language, due to their substantial experience and fundamental understanding of the language.

Java is a powerful programming language, with comparable performance to C++, without having to manage memory allocation and deallocation. Additionally, Java is an established language, and one of the most popular programming languages today [72]. Powerful Java frameworks and libraries exist, which improve the developer experience tremendously by reducing the need for tedious boilerplate code.

#### 5.1.1.1. Spring Boot

The Nomad application is implemented using the Spring Boot framework. Spring Boot [73] is a powerful Java framework that makes creating production-grade stand-alone web applications easy by allowing autoconfiguration and reducing boilerplate code. Nomad uses the Spring Boot framework for:

- Creating its web application.
- Code generation.
- Simplifying build configuration.
- Creating APIs.
- Reducing boilerplate code.

#### 5.1.1.2. Apache MapStruct

When dealing with network communication, object mapping is required to transform transportable objects into domain objects. To simplify object mapping in Nomad, Apache MapStruct was used. MapStruct is a mature Java annotation processor used for generating type-safe bean mapping classes [74].

Resources stored in the Nomad network are represented as Java objects. Various object mappings are required to translate a resource, namely

- gRPC transportable $\longleftrightarrow$ internal object
- data access object (DAO) $\longleftrightarrow$ storage layer object

### 5.1.2. gRPC

In Nomad, peers communicate using remote procedure calls (RPCs). The RPC protocol uses an existing transport protocol such as Transmission Control Protocol/Internet Protocol (TCP/IP) or User Datagram Protocol (UDP) to transport messages. The RPC protocol uses the C/S model to establish connections between machines. This allows applications to work with remote procedures as if the procedures were local. Nomad uses *gRPC* as its peer-to-peer communication framework.

gRPC is a modern RPC framework that can be used as both interface definition language (IDL) and underlying message interchange format. By default, gRPC uses protocol buffers (protobuf) to describe the structure of serialised data. This ensures small, simple payloads that enable efficient communication. Protobuf is primarily used to serialise structured data to into a compact binary wire format that enables high speed network

communication. Listing 5.1 provides an example of Nomad's "GameObject" protobuf description.

A gRPC *service* is defined by specifying its methods and models in a well-known protobuf format, in the form of *.proto* files. A gRPC *server* implements the defined service interface and runs a server that can handle client calls. A gRPC *client* also implements the service interface with stub methods, which enables calls to be made to a corresponding gRPC server [75].

```
{
  message GameObjectGrpc {
    string id = 1;
    int64 creationTime = 2;
    int64 ttl = 3;
    bytes value = 4;
    int64 lastModified = 5;
  }
}
```

**Listing 5.1:** Example of Nomad's GameObject protobuf message

### 5.1.3. Apache ZooKeeper

In order to allow peers to bootstrap to the network, facilitate peer migration and maintain strongly consistent caches, a scalable, performant directory server is required.

Nomad uses Apache ZooKeeper as a directory server for peer bootstrapping, super-peer (leader) election and keeping track of group membership. ZooKeeper is a scalable and easy to use centralized service that provides a simple API for maintaining important configuration information. This allows for distributed synchronisation and grouping services [76].

To implement the ZooKeeper client logic in Nomad, Apache Curator is used. Curator is a Java client library for ZooKeeper that provides a simplified API framework, utilities and common use cases in the form of *recipes*.

### 5.1.4. H2

When running in in-memory storage mode, Nomad requires a highly performant, efficient and reliable storage implementation. When Nomad uses in-memory storage H2 is used for local storage. H2 is a highly responsive relational database written in Java designed for efficient, secure and robust storage [77]. H2's main features include:

- Simple to integrate in Java applications,

- Supports many different platforms,
- More robust than native applications,
- High performant execution of user defined triggers.

A comprehensive list of features is provided by [78].

### 5.1.5. RocksDB

When running in disk-based storage mode, Nomad uses RocksDB for its local storage implementation. RocksDB is a highly responsive key-value store written in C++, designed for efficient, low latency and robust storage [79]. RocksDB's main features include:

- High performance,
- Adaptability,
- Optimized for fast storage,
- Supports basic and advanced database operations.

A comprehensive list of features is provided by [80].

### 5.1.6. TomP2P

Nomad uses TomP2P for overlay storage. TomP2P is a P2P Java library that provides a highly scalable and efficient Distributed Hash Table (DHT) implementation for distributed systems. TomP2P is a structured overlay implementation that uses XOR-based iterative routing, similar to Kademlia. TomP2P's underlying communication framework, Netty, uses Java non-blocking input/output (NIO) to handle concurrent connections [81].

TomP2P provides a simple API, which supports standard DHT operations, *get* and *put*, to interact with the DHT. The API also provides support for extended DHT operations like *putIfAbsent*, *add*, *send* and *digest*. Other features include [81]:

- Bloom filters for efficient object retrieval,
- Direct and indirect replication,
- Support for both blocking and non-blocking requests,
- Support for secure signature-based data validation,
- Port-forwarding detection.

### 5.1.7. Tektosyne

Nomad uses the Tektosyne Java library for Voronoi map generation. The Tektosyne library provides algorithms for computational geometry and graph-based pathfinding, mathematical utilities and specialised collections [82].

## 5.2. Nomad Application

Nomad is a Java application, packaged into a self-contained JAR. No discernment between peer and super-peer applications can be made. Every application instance contains both client-side and server-side logic for all services and mechanisms. Client-side logic refers to the logic required to generate requests, allowing peers to request resources from other peers. Server-side logic refers to logic required to serve requests, allowing peers to serve requests from other peers. The active services and mechanisms depend on the node's system role, determined at runtime. Figure 5.1 provides a high level UML diagram of the Nomad application.

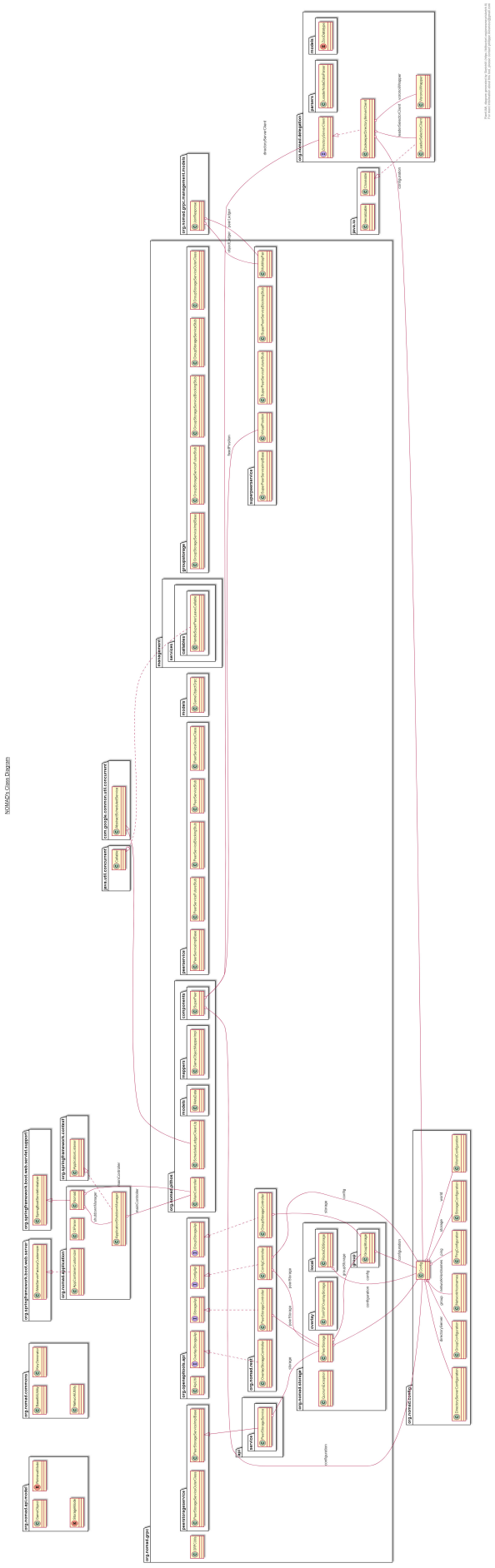

**Figure 5.1:** Nomad implementation UML diagram (best viewed electronically)

Section B.2 provides a description of Nomad's package structure and the purpose of each individual module.

### 5.2.1. Nomad Modules and Services

The application consists of three gRPC servers: a **super-peer**, **peer** and **group storage** server and their accompanying clients, **super-peer**, **peer** and **group storage** clients.

Apart from gRPC services, the application consist of four main internal components, namely **local storage**, **overlay storage** and a **group ledger**. Depending on the node's system role, i.e. peer or super-peer, certain system components are disabled. Figure 5.2 illustrates the services and components that are active during runtime for each peer role.

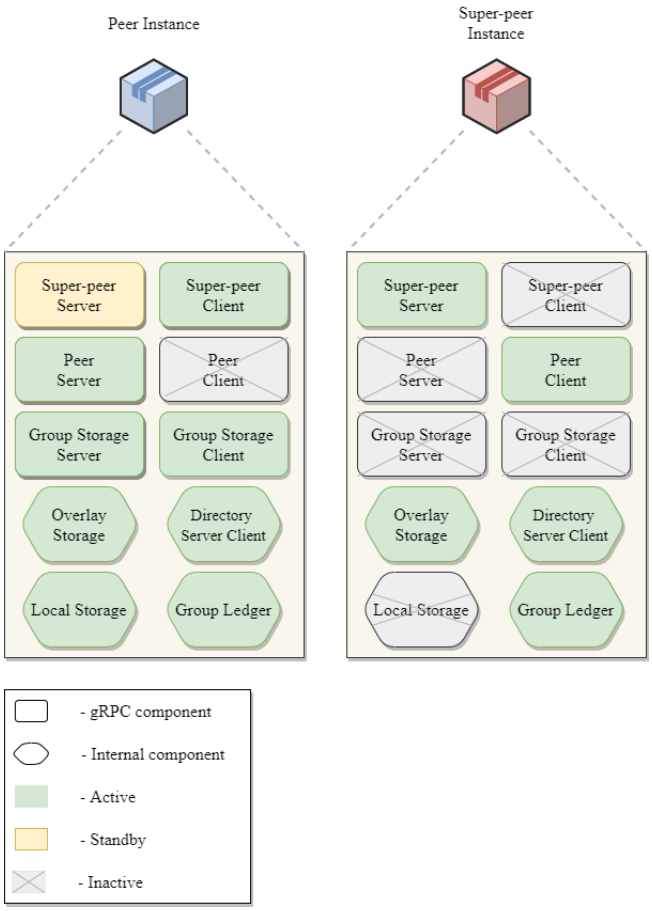


**Figure 5.2:** A simplified view of a Nomad application instance and its various internal and external (gRPC) components

## 5.3. Satisfying Key Modules

In the previous chapter, the key modules, mechanisms of Nomad were set out, by providing an overview of adjustments made to each module in order to satisfy the Pithos architecture requirements. This section provides more detail on each component's implementation in

terms of frameworks and tools used.

### 5.3.1. Peer

Nomad peers are mainly responsible for group and overlay storage, but also contain client logic for joining and leaving the network, group migration, maintaining group consistency, and maintaining group ledgers. The super-peer module is discussed in Section 5.3.5.

Nomad's design separates storage and maintenance responsibilities by implementing two separate modules, namely a peer and a peer-storage module. This separation of concerns simplifies the required logic significantly.

#### 5.3.1.1. Peer Module

Nomad implements the peer module by defining internal and external responsibilities. A storage peer defines a gRPC service, namely the **peer service**, that is responsible for handling all maintenance requests. The service interface is implemented by the **peer server**, which is responsible for handling all client requests issued by **peer clients**. In simple terms, a peer's *peer server* handles all requests from the super-peer's *peer client*. To ensure consistency within the group, a super-peer creates a peer client for every peer within its group. Figure 5.3 illustrates how the peer service is implemented within Nomad.

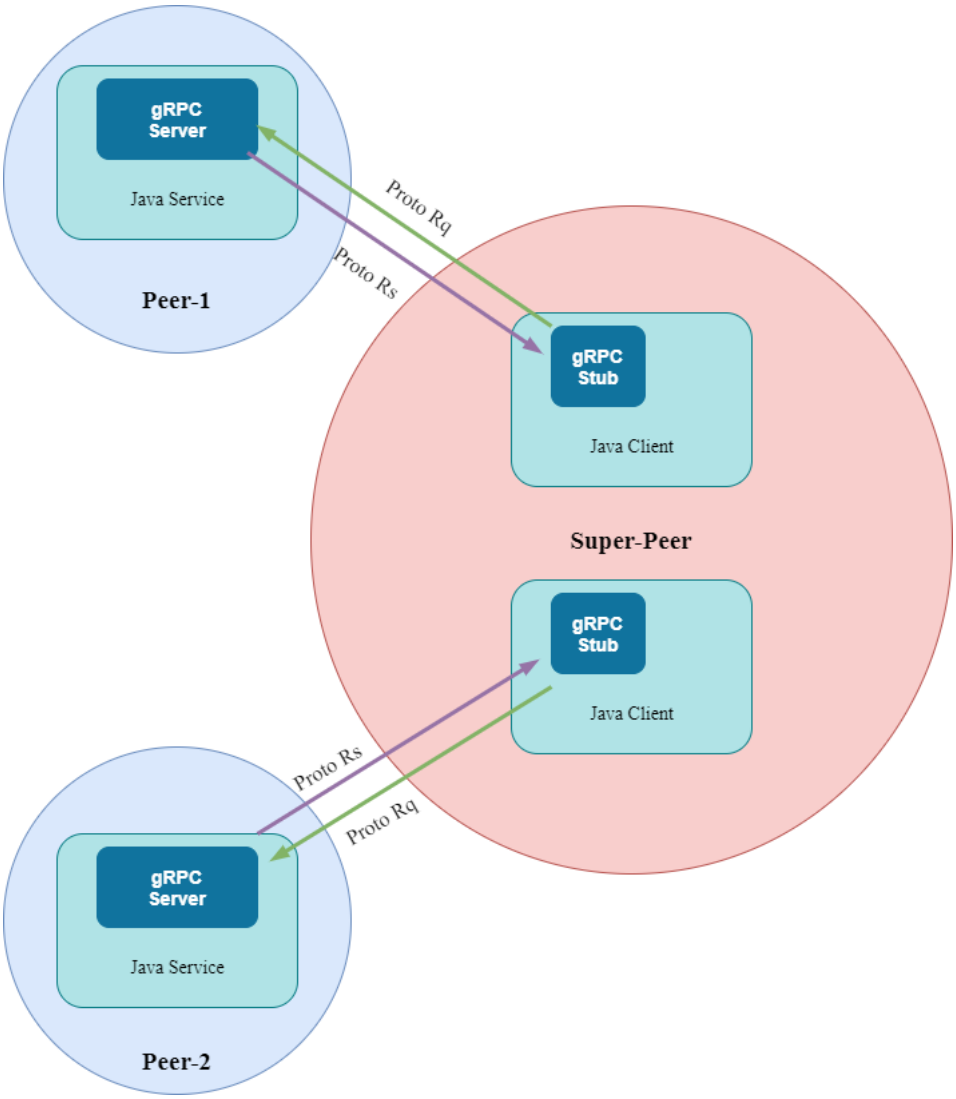


**Figure 5.3:** Illustration of the peer service implementation within Nomad (Rq = request, Rs = response)

A peer server's responsibilities can be summarised as follows:

1. Handling add peer requests from the super-peer's peer client.
2. Handling remove peer requests from the super-peer's peer client.
3. Handling super-peer leave requests from the super-peer's peer client.
4. Handling repair requests from the super-peer's peer client.

Figures C.4, C.5 and C.6 provide Unified Modeling Language (UML) diagrams of a peer's gRPC service, server and client modules.

The peer module is additionally responsible for maintaining and facilitating internal mechanisms. The peer's secondary responsibilities can be summarised as follows:

1. Informing the directory server of any changes in group or peer server hostnames.

2. Participating in group joins, leaves and migrations.
3. Participating in group migrations.
4. Executing group keep-alive pings.
5. Updating the replication factor if required.
6. Elevating the peer to a super-peer if required.

## 5.3.2. Peer Storage Module

Nomad's peer-storage module can be seen as the top level storage module that provides a basic interface for *retrieve, store* and *modify* storage operations. The peer storage module maintains local, group and overlay storage modules.

### 5.3.2.1. Local Storage Module

To satisfy the need for responsive storage without restricting peers with limited resources from participating in the network, Nomad defines a *LocalStorage* interface, which is used for internal (local) object storage. This interface allows for extensibility and interchangeability of the local object store.

Nomad supports two high-level storage modes, namely *in-memory* and *disk-based* storage (4.5.2.4). For in-memory storage, Nomad uses H2 (5.1.4), a highly responsive relational database. In-memory storage is used when resource usage is not a concern. H2 supports disk-based storage; however, Nomad opts for more a efficient alternative. For efficient disk-based storage, Nomad uses RocksDB (5.1.5), a persistent key-value store, specifically designed for applications with the need for fast storage. Disk-based storage is used when peers are expected to store an excessive amount of VE objects.

## 5.3.3. Group Storage Module

Nomad's group storage component acts as an abstraction layer that connects all peers' local storage components, effectively creating a fully connected storage component.

Nomad defines a separate gRPC service for handling group storage operations. This separation of concerns simplifies the peer module by splitting the logic into two sub-modules for group maintenance and storage operations respectively. A storage peer defines a gRPC service, namely the **group storage service**, that is responsible for group storage. The service interface is implemented by the **group storage server**, which is responsible for handling all client requests issued by **group storage clients**. In simple terms, a peer's *group storage server* handles all requests from another peer's *group storage client*. Every group storage peer creates a client for every group storage server. This ensures a single

hop network architecture within each group. Figure C.4 provides UML diagram of the group storage server.

#### 5.3.3.1. Group Storage Logic

The group storage server is responsible for handling all group storage requests. The group storage server's responsibilities can be summarised as follows:

1. Handling all group *store, retrieve,* and *modify* requests from peer storage clients.
   (a) Replicating objects within the group.
   (b) Retrieving objects from its own or another peer's local storage, i.e. group storage.
2. Maintaining the peer's group ledger, by handling all add object requests from peer storage clients.
   (a) Maintaining a consistent view of all peers within the group.
   (b) Maintaining a consistent view of all objects within the group.
3. Keeping track of requests.

Figure 5.4 illustrates how the group storage service is implemented within Nomad.

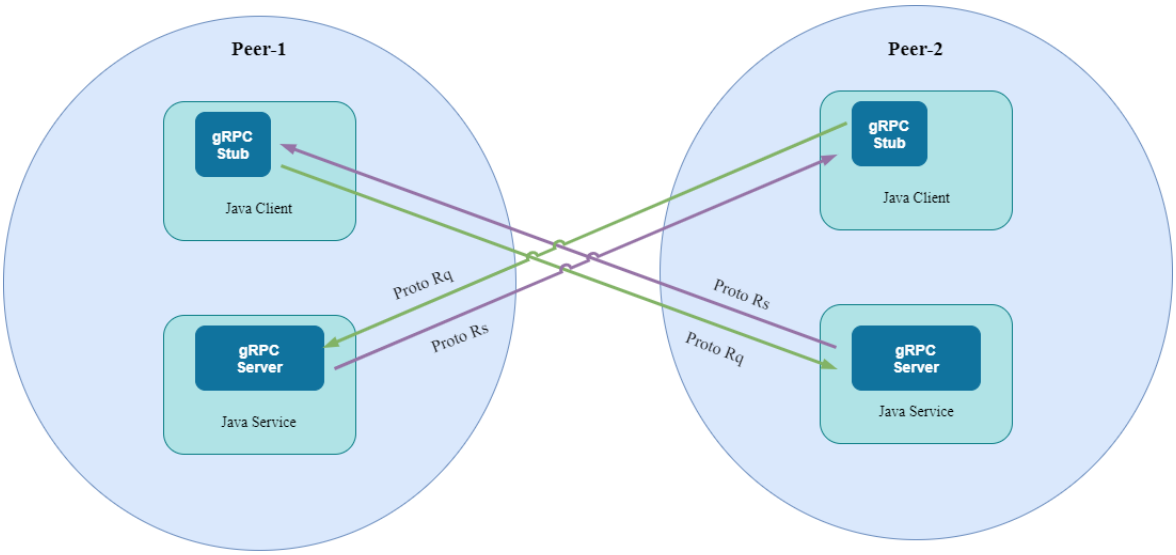


**Figure 5.4:** Illustration of the group storage service within Nomad (Rq = request, Rs = response)

### 5.3.4. Overlay (DHT) Storage

Overlay storage is responsible for state persistence and maintaining a backup object store, which assists in resolving storage conflicts, and serves out-of-group object requests. To

satisfy the need for reliable overlay storage, Nomad defines a *DHTOverlayStorage* interface, which allows for extensibility and interchangeability of overlay storage implementation. A Nomad peer's overlay storage responsibilities can be summarised as follows:

1. Handling overlay store, retrieve and modify requests.
2. Maintaining an object store, with all group objects.
3. Maintaining a consistent view of all peers within the overlay.
4. Maintaining a consistent view of all objects within the overlay.
5. Handling of overlay join and leave requests.
6. Routing of requests to the responsible peers.

#### 5.3.4.1. Overlay Implementation Challenges

One of Nomad's main design challenges was finding a suitable DHT implementation. Many DHT systems exist [9], but not many are designed to work with Java applications. This limited the available choices. A suitable DHT implementation was selected based on the following criteria:

- Actively maintained,
- Compatible with Java applications,
- Scalable and robust implementation,
- Usable interface for overlay bootstrapping, storage operations and maintenance operations,
- Configurable,
- Active developer community.

Three potential java implementations were identified, namely TomP2P [81], OpenChord [83] and FreePastry [84]. Table 5.1 provides a summary of the potential DHT implementations and their relevance according to the evaluation criteria.

| Criteria | TomP2P | OpenChord | FreePastry |
|---|---|---|---|
| Java compatible | ✓ | ✓ | ✓ |
| Scalable an robust | ✓ | ✗ | ✗ |
| Usable interface | ✓ | ✓ | ✗ |
| Configurable | ✓ | ✗ | ✗ |
| Active community | ✗ | ✗ | ✗ |
| Well maintained | ✓ | ✗ | ✗ |

**Table 5.1:** Summary of potential DHT implementations

#### 5.3.4.2. TomP2P

TomP2P is an open-source Java library that provides a highly scalable and efficient DHT implementation for distributed systems. TomP2P uses XOR-based iterative routing, similar to Kademlia. The TomP2P java implementation does not have an active developer community, as the team's attention has shifted to a *Golang* implementation [85]. The repository [81], however, is being maintained by a few developers, who also offer support. The latest release of TomP2P at the time of writing was published in 2020, which was considerably better than the alternative DHTs. A PoC application was used to determine TomP2P's validity as an overlay storage option for Nomad. The PoC determined that TomP2P provides an excellent interface for bootstrapping and storage, is easy to configure, is highly responsive with response times under 10 ms, and is scalable. It was also found that TomP2P can handle considerable load without dropping in performance or reliability. Under high network churn the responsiveness of TomP2P did drop considerably.

#### 5.3.4.3. OpenChord

A Chord DHT implementation was used as overlay storage for the Pithos simulation. It was therefore high priority to use Chord to satisfy Nomad's overlay storage requirement. OpenChord is an open-source Java implementation of Chord, known to be highly reliable and responsive, and was evaluated in the form of a proof-of-concept (PoC) application. The OpenChord implementation was last updated in 2016 [83] and has no active developer community, which means that the application required some additional code changes to be incorporated into a modern Java application. After the required implementation changes were made, the implementation was packaged as a Maven dependency and incorporated into an early version of the Nomad implementation.

OpenChord provides a usable interface for bootstrapping and storage, but lacks configurability and does not perform as expected under network churn, due to critical bootstrapping issues. Scaling the network proved to cause considerable performance issues. Under high load it was also found that the OpenChord implementation becomes completely unresponsive, and nodes become overloaded very easily. The PoC determined that OpenChord could be suitable for stable, small environments; however, for use as MMVE storage it is not suitable.

#### 5.3.4.4. FreePastry

FreePastry is an open-source Pastry implementation, known to be highly reliable and responsive. However, the FreePastry implementation was last updated in 2009 [84] and has no active developer community. This implementation is not considered to be well maintained and was therefore not selected for this study.

#### 5.3.4.5. Conclusion

After investigation, TomP2P was determined to be the only suitable overlay storage implementation for Nomad. TomP2P satisfies the requirement of highly scalable, responsive overlay storage.

### 5.3.5. Super-peer

Nomad's super-peers are representative of administrative nodes in the network and participate in overlay storage. Super-peers in the Nomad network are mainly responsible for facilitating group join, leave and migration operations, maintaining object replicas, and maintaining a consistent view of peers and objects in the network. Nomad's super-peers participate in overlay storage as storage nodes, since it does not provide an interface for querying the overlay storage. By design, a super-peer's resource usage is fairly low, which means that super-peers have resource capacity to contribute to the overlay storage network. This alteration to the Pithos design improved the overall reliability of the overlay network.

#### 5.3.5.1. Super-peer Module

Nomad implements the super-peer module by defining internal and external responsibilities. A super-peer defines a gRPC service, namely the **super-peer service**, which is responsible for initiating all maintenance tasks and facilitating group bootstraps. The service interface is implemented by the **super-peer server**, which is responsible for handling all client requests issued by **super-peer clients**. In simple terms, a super-peer's *super-peer server* handles all requests from the peer's *super-peer client*. Figure 5.5 illustrates how the super-peer service is implemented within Nomad.

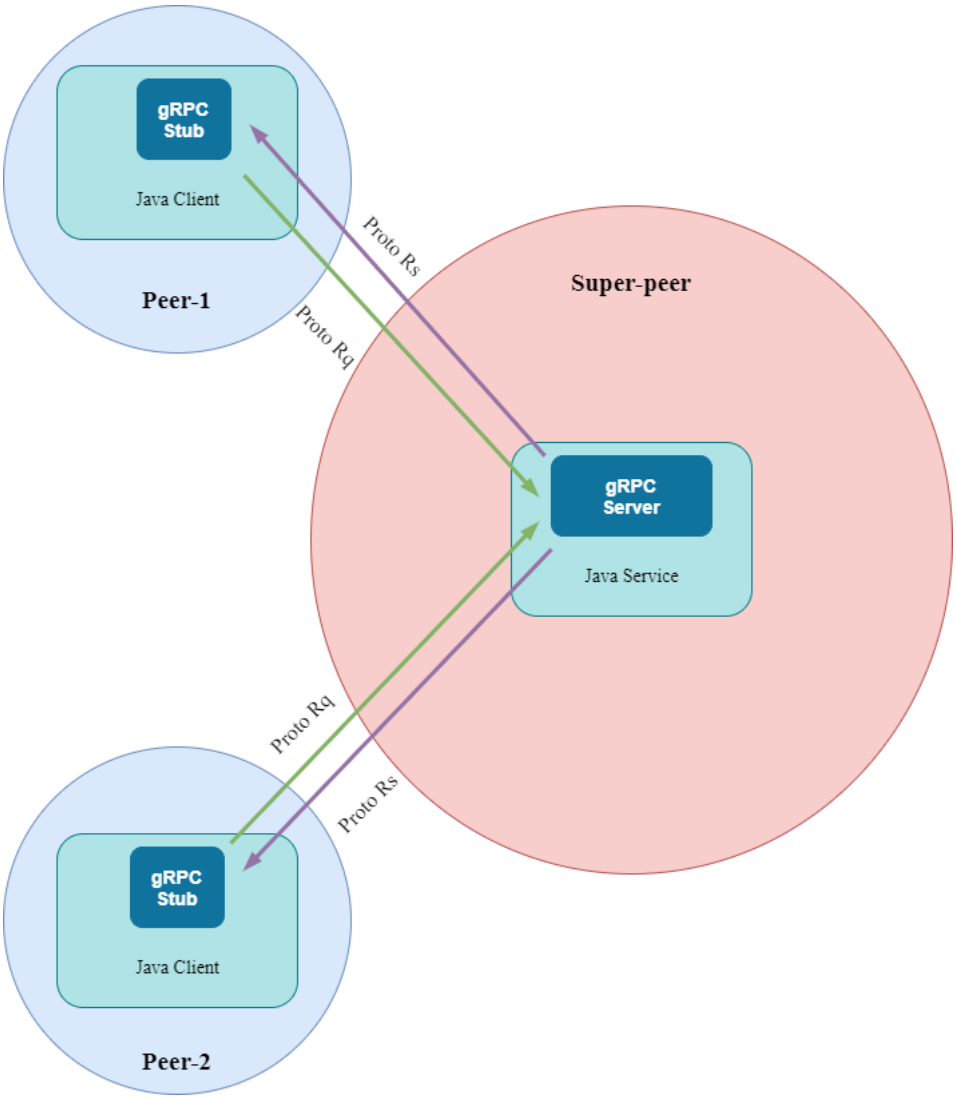


**Figure 5.5:** Illustration of the super-peer service implementation within Nomad (Rq = request, Rs = response)

The *super-peer server* is responsible for handling all peer communication with the purpose of group maintenance and bootstrapping. Separate logic exists for maintaining all internal super-peer mechanisms. A super-peer's responsibilities can be summarised as follows:

1. Handling all group leave and join requests.
2. Facilitating group migration.
3. Ensuring group consistency.
4. Initiating object repair.
5. Maintaining its own super-peer ledger.
6. Keeping track of peer positions in the VE.

7. Executing final ping requests to potentially unreachable peers.

Figures C.4, C.5 and C.6 provide Unified Modeling Language (UML) diagrams of a super-peer's gRPC server, service and client modules. The super-peer module is additionally responsible for maintaining and facilitating internal mechanisms. The super-peer's secondary responsibilities can be summarised as follows:

1. Informing the directory server of any changes in super-peer server hostnames.
2. Updating the group replication factor if required.

#### 5.3.5.2. Redundant Super-peers

Within the Nomad storage network, every peer is also a redundant super-peer (4.2.2.3). As described in section 5.2, the Nomad application is self-contained - there is no difference between the peer and super-peer executables. Every application instance contains both client-side and server-side logic for all services. And since group peers and super-peers have identical group ledgers, storing identical data, promoting a peer to a super-peer is just a matter of starting and stopping the correct gRPC servers.

Section 4.4.3 describes the super-peer leave procedure, where the purpose of redundant super-peers is made clear.

### 5.3.6. Group Ledgers

Nomad makes use of an abstract data structure, namely group ledgers, in order to keep track of all peers and objects within a group. Sections 3.7.5 and 4.5.2.1 describe the anatomy of group ledgers and their usage in more detail. Nomad defines a *GenericGroupLedger* interface as an abstraction layer. This interface allows the group ledger implementation to be interchangeable.

In the original Pithos design, the group ledger physically stores the object information within the ledger effectively, all objects are stored in memory. As Java does not provide fine grain memory management and an alternative to C++'s reference *pointers*, it is required that object storage and ledger logic be separated. Nomad therefore makes use of separate storage component, namely local storage (4.5.2.4), to store objects, whereas the group ledger is exclusively used to keep track of peers and objects within the group.

This separation of concerns simplifies Nomad's group ledger implementation. Instead of storing the entire object in the object ledger, only the ID and last-modified date are stored. Similarly, peer ledgers only store peer and object IDs. Nomad relies on more mature and efficient technologies for storage, and is only required to maintain IDs in the ledger. As objects are not physically stored within the ledger, Nomad's group ledger also makes use of the object's TTL. This ensures that expired objects are removed from both

local storage and the group ledger. Figure C.2 provides a UML diagram of the group ledger module.

### 5.3.7. Group Ledger Storage Procedure

The means by which objects are stored to the group ledger is illustrated by figure 5.6. The steps discussed below form part of the Nomad *store* procedure, which is explained in more detail in Section 5.5.2.

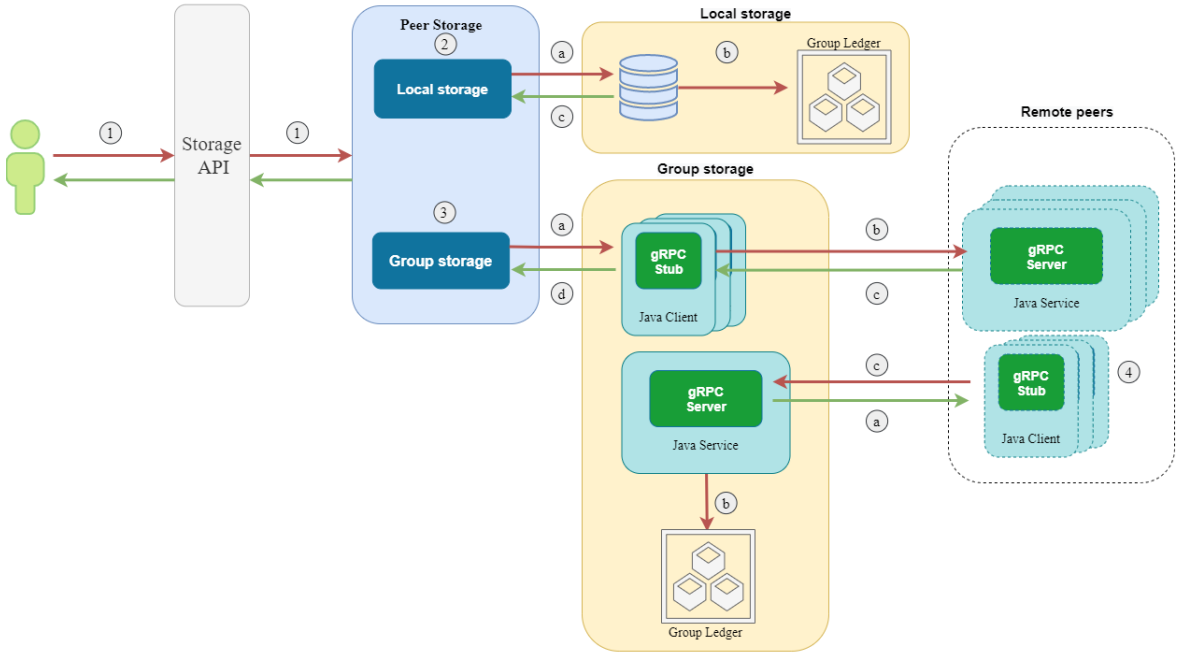


**Figure 5.6:** Illustration of how object references are added to the group ledger

1. The higher layer VE logic uses Nomad's peer storage API to execute an *add* operation. The request is relayed to the **Peerstorage** class, where a storage operation is executed.

2. The object is stored to local storage. The peer adds the object to its authoritative object store.

   (a) The storage operation is relayed to the peer's local storage module (H2/RocksDB).

   (b) Only if the operation succeeds, does the peer add the object ID, coupled with a TTL, to its group ledger.

   (c) The local storage module responds with success.

3. Group storage is used to notify peers of the new ledger entry.

   (a) The peer's group storage module is responsible for maintaining ledger consistency within the group. The group storage component maintains multiple group

storage gRPC clients, a client per peer in the group. The new ledger entry notification is therefore relayed to the peer's group storage module.

(b) The client logic triggers an *addObjectReference* request on every peer in the group, by providing a request payload containing the ID of the stored object and peer ID the object was stored to. Each peer receiving the request, adds an identical entry to its own group ledger.

(c) Peers respond to the *addObjectReference* request with a simple success or failure message, depending on the execution result.

4. The peer can also receive an *addObjectReference* from a remote peer, which notifies it of objects stored on other peers.

   (a) The peer extracts the object and peer ID from the request payload and adds an entry to its own group ledger.

   (b) The peer responds with a success or failure message, depending on the execution result.

### 5.3.8. Directory Server

To satisfy the requirement of a centralised directory server, Nomad defines a *DirectoryServerClient* interface. This interface allows the directory server implementation to be interchangeable.

Nomad uses Apache ZooKeeper (5.1.3) as its directory server. The directory server module therefore only requires client side logic to fulfil Nomad's directory server requirements. As a result, a standalone ZooKeeper instance is required to ensure system functionality. The implemented client-side logic enables peers to interact with the ZooKeeper server namespace.

The ZooKeeper namespace is somewhat like a standard filesystem, with the exception that each file can also be a directory. Each ZooKeeper node (znode) in the namespace can be associated with data, as well as with child nodes. The Nomad ZooKeeper namespace consists of both permanent and ephemeral znodes. Permanent znodes maintain state even after the last znode has left the namespace, whereas ephemeral nodes will automatically be removed if the associated znode leaves the namespace. As high network churn is expected in MMVEs, ephemeral nodes are used to represent peer components like groups, peers and super-peers. Figure C.1 provides an illustration of Nomad's ZooKeeper namespace hierarchy.

For the sake of convenience, the Nomad repository [86] provides a ZooKeeper Docker [6] container, which can either be run on a local Docker instance or deployed to a local Kubernetes [7] cluster.

## 5.4. Satisfying Key Mechanisms

In the previous chapter, the key mechanisms of Nomad were discussed, by providing an overview of adjustments made to each mechanism in order to satisfy the Pithos architecture requirements. This section provides more detail on each mechanism's implementation.

### 5.4.1. Super-peer Selection Mechanism

Nomad's ZooKeeper directory server provides a built-in leader selection mechanism. The mechanism uses a simple sequential selection, where the smallest znode sequence number becomes the leader. Nomad makes use of this mechanism for its super-peer selection. In Nomad, all peers in the group therefore have equal right to be the group's super-peer and therefore all participate in leader election.

Nomad's super-peer election root path can be described as */election*. Every peer that joins the group creates a path under the root znode, such as */election/group-number/peer-id_sequence-number* (the sequence number being the order in which the group is joined). The child node with the lowest sequential number is selected by ZooKeeper as the leader. Figure C.1 provides an illustration of Nomad's ZooKeeper namespace hierarchy.

#### 5.4.1.1. Super-peer Selection Logic

The group super-peer selection process can be summarised as follows:

1. A peer stores its hostname to the super-peer selection path, creating a child znode at */election/group-number/peer-id_sequence-number*.

2. Once a peer is selected as the leader (super-peer), it is notified by an internal cache listener that it has acquired group leadership.

3. After the peer is notified of its promotion, it shuts down its local storage and group storage components.

4. The peer informs other group peers that it is no longer a storage peer. (This is usually not required as the super-peer selection mechanism is only used for new groups.)

5. The peer then stores its hostname to the */leaders* znode, which is publicly available within the Nomad network. The */leaders* znode is used to publish each group's leader node, which is used during group joins and migration.

### 5.4.2. Grouping Mechanism

Nomad's ZooKeeper directory server has out-of-the-box support for group membership and leader election. Groups are represented by a node; each group member is represented by an ephemeral node. Nomad makes use of ZooKeeper's group membership mechanism to maintain group consistency and easily extract peer information about any other peer in the group. Two group znodes exist, namely */cache* and */groups*, each used for a different purposes.

#### 5.4.2.1. Nomad groups vs ZooKeeper Groups

An important distinction should be noted between ZooKeeper group membership and Nomad group membership. Nomad's group membership refers to a fully connected group of peers and a super-peer, which together form group storage. ZooKeeper's group membership is usually identical to the peers within a Nomad group, but it does not refer to the actual Nomad group. Instead, it refers to the group cache that is created in the ZooKeeper directory server under the */cache* znode. ZooKeeper group membership is purely used to facilitate better peer-to-peer communication and simplify bootstrapping.

#### 5.4.2.2. ZooKeeper Grouping Mechanism

The "/cache" znode is used to maintain group membership cache, allowing group members to easily detect changes in the cache and extract peer information. Group members create znodes with paths such as */cache/peer-id*, which contain all the important data of that group peer, namely,

1. Group storage server hostname,
2. Peer server hostname (if applicable),
3. Super-peer server hostname (if applicable),
4. DHT server hostname.

The "/groups" znode is used to determine which groups have available positions and to extract peer information from other groups if needed. Group members create znodes with paths such as */groups/group-number/peer-id*, which contain only the group storage server hostname of that group peer. Nomad can also be configured to make use of Voronoi grouping [64].

#### 5.4.2.3. Voronoi Grouping Mechanism

When Voronoi grouping is enabled, Nomad peers construct a Voronoi map, which segments the VE into a series of polygons based on super-peer locations.

A Nomad Voronoi map consists of finitely many points, called *site points* which are representative super-peer locations. When a new super-peer is selected, it stores its super-peer hostname and VE location (site point) to the directory server. Each peer and super-peer maintains a local cache of all super-peer site points, which it retrieves from the directory server on start-up, or when a new super-peer joins the network. Each site point has a corresponding region, called a *Voronoi cell*, which is representative of a group's AoI. To construct a Voronoi map, Nomad uses the Tektosyne library 5.1.7.

To simplify Nomad's Voronoi grouping logic, once a super-peer is selected and it sends its VE location to the directory server, it can no longer receive any positional updates. This means that super-peer positions are static in Nomad's current implementation. Figure 4.2 illustrates the Voronoi grouping process.

### 5.4.3. Storing Public Hostnames

Section 4.2.3.1 discussed the directory server's responsibility to store the public hostname information of various peer data, like *dht*, *group* and *leader* hostnames. Nomad's namespace therefore contains */dht*, */groups* and */leaders* znodes, which publish the hostname data of all group peers. This gives peers the ability to request hostnames of peers that are not within their own group. This is not a strict requirement according to the Pithos architecture, but it does promote a higher degree of developer freedom.

### 5.4.4. Group Join Mechanism

In this section, Nomad's group join mechanism, which was introduced in section 4.4.1, is discussed with regard to its implementation.

#### 5.4.4.1. Group Join Logic

After a connection is established with the ZooKeeper directory server, the peer's ZooKeeper client logic takes over in order to determine which group should be joined. If Voronoi grouping is enabled, the group to join is determined by the grouping mechanism discussed in section 4.3.2.2. If no grouping algorithm is used, the client logic will select an arbitrary group with an available peer position to join.

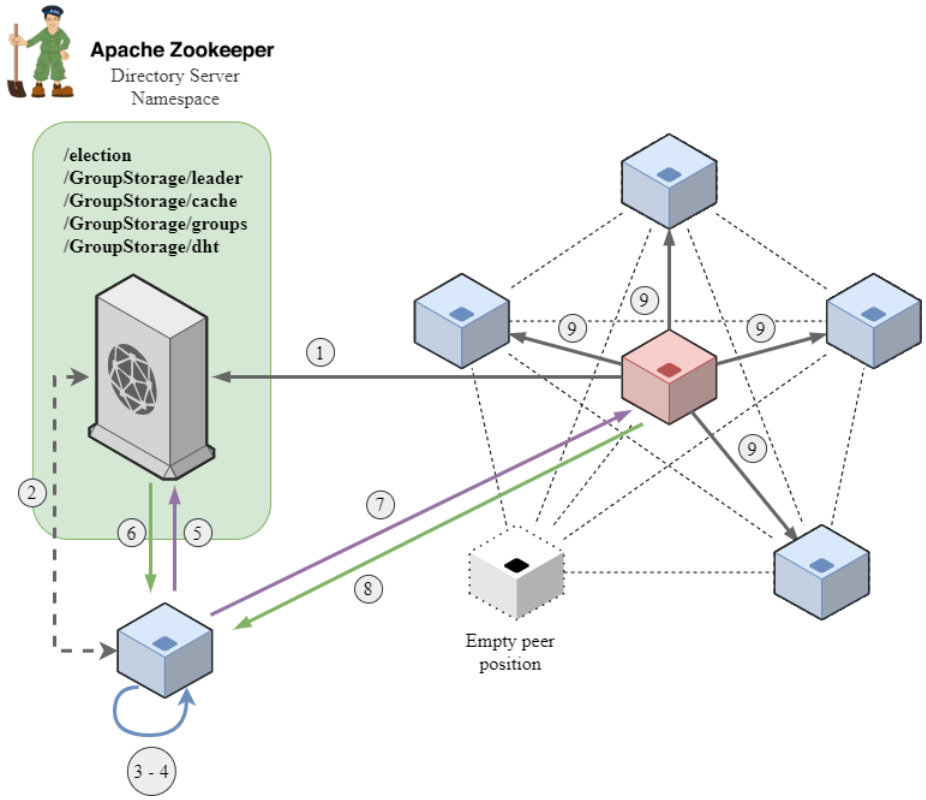


**Figure 5.7:** Nomad group join mechanisms, used to bootstrap to the Nomad network

Figure 5.7 illustrates the join procedure which allows peers to join the Nomad network.

1. A super-peer shares its bootstrap hostname with the directory server. This can be used to join that super-peer's group. It is assumed that this node was previously elected as a super-peer (see section 5.4.1). The super-peer server hostname is stored under the */leaders* znode. Additionally, if Voronoi grouping is used, the super-peer will also share its location with the directory server, which will allow peers and super-peers to construct a Voronoi map.

2. A new peer establishes a connection with the directory server, and sends a join request.

3. The peer's internal ZooKeeper client logic ensures the peer is added to the directory server's group cache and determines the group to join (results may vary dependent on the grouping mechanism used). The peer will always try to ensure that the super-peer has an available peer position before determining the group.

4. The peer initialises its overlay storage component and bootstraps to the overlay network.

(a) The peer requests a DHT hostname from one of its potential group members, using the */dht* znode. This overlay hostname is used to bootstrap to the overlay network.

(b) If no other peers exist in the network, the peer becomes the initial bootstrap node.

5. The peer then requests the corresponding super-peer hostname from the directory server.

6. The directory server responds to the request with the hostname of the super-peer of the group the peer should join. The peer's hostname information is also stored to child nodes of the */dht*, */groups* and */cache* znodes.

7. The peer sends a join request to the super-peer, containing peer information, such as server hostnames and its VE location.

8. The super-peer decides whether it wants to accept or reject the peer requesting to join the group, based on the current group size and the peer's VE location.

   (a) If the super-peer accepts the peer, it provides the peer with a list of peers and objects in the group. The peer uses this information to populate its own group ledger.

   (b) If the super-peer rejects the peer, the peer will send a join request to a neighbouring super-peer.

   (c) If all super-peers reject the peer, no available positions exist and the peer is promoted to a super-peer.

9. (Assuming the peer was accepted) The super-peer notifies all peers within the network that a new peer was added.

### 5.4.5. Group Leave Mechanism

In this section, Nomad's group leave mechanism, introduced in Section 4.4.2, is discussed with regard to its implementation.

#### 5.4.5.1. Group Leave Logic

As peer information is stored on ephemeral znodes under the */election*, */leaders*, */groups*, */dht* and */cache* paths, when the peer disconnects from the ZooKeeper directory server, the data is removed after its grace period expires.

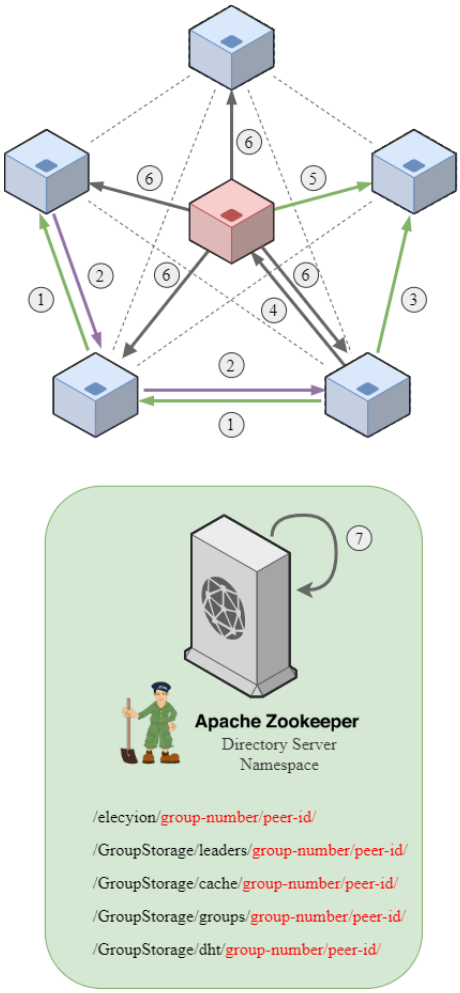


**Figure 5.8:** Illustration of Nomad's leave mechanism

Figure 5.8 illustrates the group leave procedure, which ensures group consistency if a peer leaves the group unexpectedly.

1. A peer routinely sends a keep-alive ping request to another random peer.
2. If a peer receives a keep-alive ping, it acknowledges the message with a pong.
3. If a peer does not acknowledge the keep-alive ping within a specific time window, the peer is considered to have left the network.
4. The original sender of the keep-alive ping notifies the super-peer that a peer is unreachable.
5. The super-peer sends a ping of its own to the peer in question, to verify that the peer has actually left the network.
6. If a peer does not acknowledge the super-peer ping within a specific time window, the super-peer informs all peers that the peer has left the group.

7. Ephemeral znodes created by the peer under the */election*, */leaders*, */groups*, */dht* and */cache* paths, are automatically removed by ZooKeeper.

### 5.4.6. Super-peer Leave Mechanism

Section 4.4.3 discussed Nomad's super-peer leave mechanism. Figure 5.9 illustrates this procedure and the process of re-electing a new super-peer. When a super-peer leaves the network, all peers are informed and are required to acknowledge the leave. The directory server elects a peer as the new super-peer. This peer is promoted to a super-peer, which then receives a simplified join request from all active peers within the group. The steps are exactly the same as in Section 4.4.3.

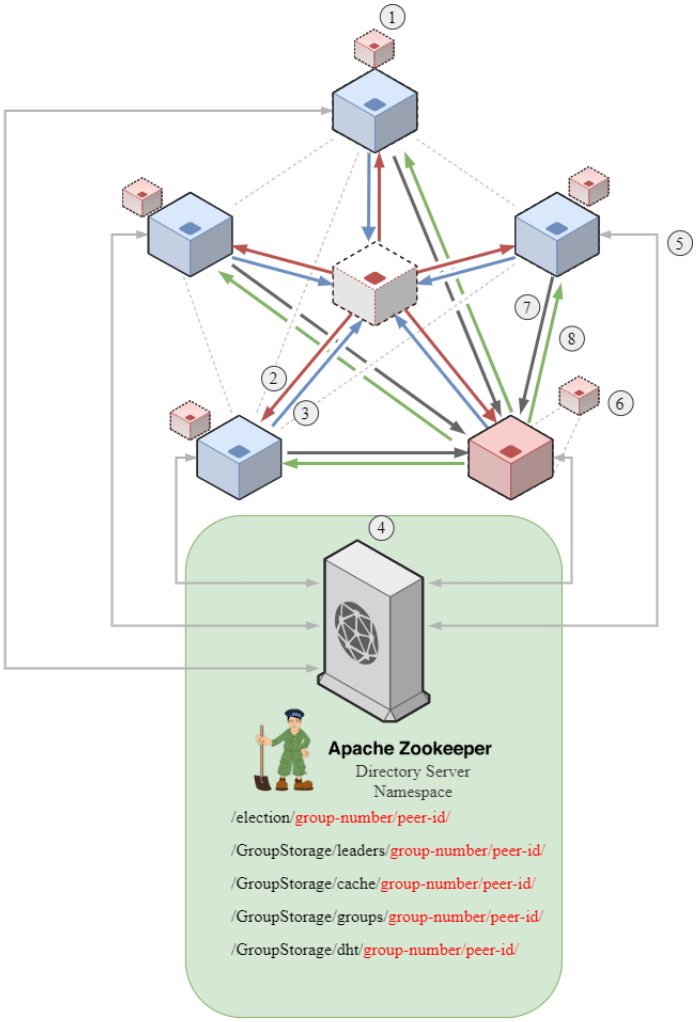


**Figure 5.9:** Illustration of Nomad's super-peer leave mechanism

### 5.4.7. Group Migration Mechanism

In this section, Nomad's group migration mechanism, introduced in section 4.4.4, is discussed with regard to its implementation.

#### 5.4.7.1. Group Migration Logic

As discussed in section 4.4.4, peers move around in a VE, which means they can cross group AoI boundaries frequently. As mentioned in section 5.4.2.3, when Voronoi grouping is used, super-peers store their VE positions to the directory server. This allows all peers and super-peers to maintain a consistent local cache of all super-peer positions, which are used as site points for generating a Voronoi map. In Nomad, at least two super-peers are required for the Voronoi map to be generated. This is dictated by the Tektosyne library's Voronoi map generation.

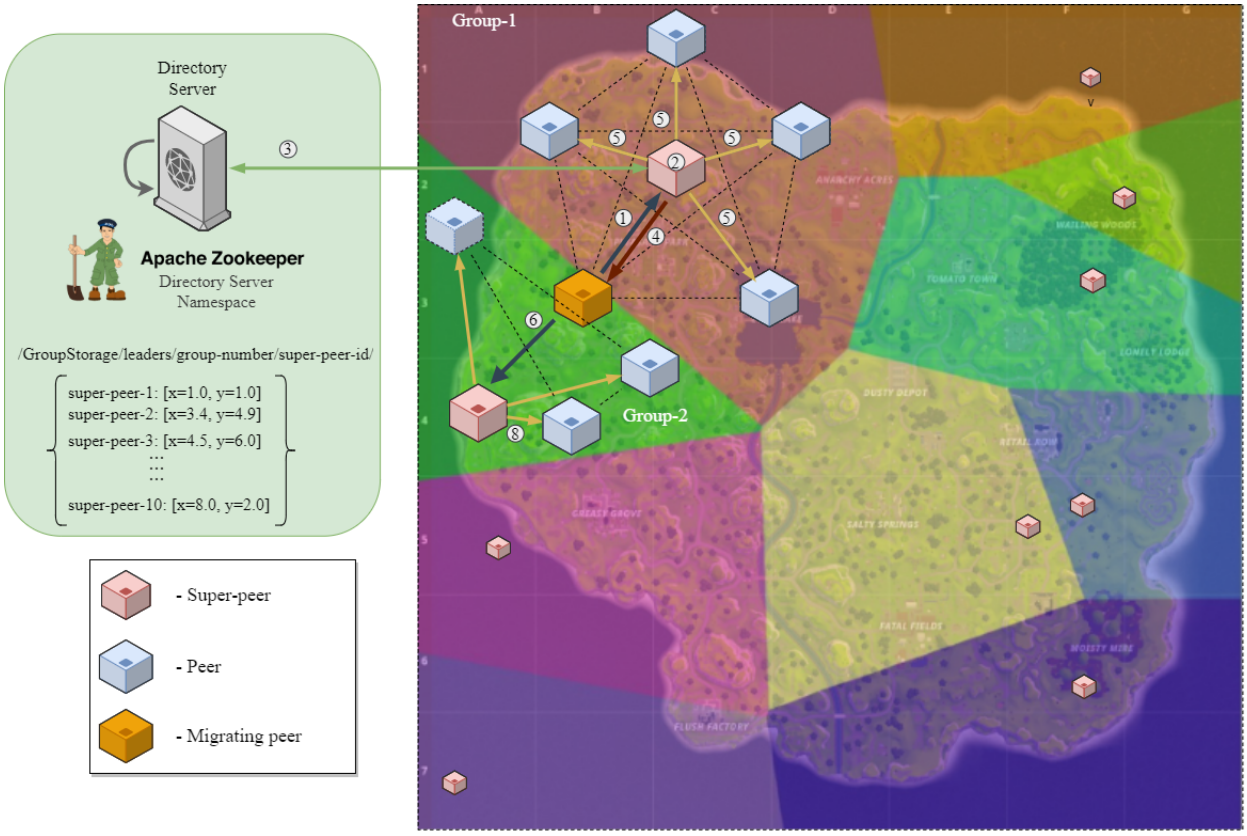


**Figure 5.10:** Illustration of Nomad's group migration mechanism, when Voronoi grouping is used

Figure 5.10 illustrates Nomad's group migration procedure, that allows peers to migrate from one geographical group to another.

1. A peer routinely sends positional updates to its super-peer.
2. The super-peer server evaluates the peer's position by checking where the new position falls within its local copy of the Voronoi map. The Voronoi map is constructed by using data from the super-peer cache, which contains the IDs and positions of all super-peers, and these are used as Voronoi site points.

(a) If the peer is still within the group's AoI, the super-peer simply responds with an ACK. The peer is still in the correct group and does not need to be migrated.

(b) If the peer is not within the AoI, the peer needs to be migrated to a new group.

3. The super-peer evaluates its local Voronoi map and super-peer position cache, and determines within which Voronoi group the peer's position falls. Using the group information that is extracted from the Voronoi map, the peer can cross-reference its local super-peer position cache to find the super-peer of the new group.

4. The super-peer responds with a new group name and super-peer hostname, which the peer should migrate to.

5. The peer sends a leave request to its current super-peer. Once the peer notifies the super-peer that it has left the group, the peer truncates its group ledger and local storage.

6. The super-peer notifies all peers in the group that the peer has left. Ephemeral znodes created by the peer under the */election*, */leaders*, */groups*, */dht* and */cache* paths are automatically removed by ZooKeeper.

7. The peer sends a join request to the new super-peer. (The join request procedure is followed)

8. If the new super-peer accepts the peer, it provides the peer with a list of peers and objects in the group. The peer uses this information to populate its own group ledger. If the peer cannot be migrated to a neighbouring group, like if the group is full, the peer will be promoted to a super-peer.

9. (Assuming the peer was accepted) The new super-peer notifies all peers within the network that a new peer was added.

## 5.5. Satisfying P2P MMVE Storage Use Cases

All Pithos modules and mechanisms have now been redesigned for a real-world Java implementation. As described in Section 3.2, the Pithos architecture, and by extension Nomad, is designed to satisfy P2P MMVE storage use cases, whilst adhering to the requirements of such an environment. This section provides the required detail to describe how Nomad satisfies the implementation use cases of object **storage, retrieval, modification and updates**. As in Section 3.2, all storage requests are assumed to originate from some higher layer, like a VE or game logic.

### 5.5.1. General

As discussed in section B.2.7, Nomad provides a REST API to satisfy all use cases. This means user traffic is generated by HTTP requests.

### 5.5.2. Store

In Section 3.10.1, Pithos' object storage procedure was described. This section provides an overview of how Nomad employs the designed procedure.

Nomad has two storage modes, namely *fast* and *safe* storage. In fast storage mode, Nomad's peer-storage module sends a storage request to multiple group storage peers and to overlay storage. The first successful response is propagated to the higher layer. Only if all responses fail is the storage operation deemed to have failed. This improves the responsiveness of storage commands, as some peers may be geographically closer to the requesting peer. In fast storage mode, Nomad's group storage service favours group peers with low latency connections. Fast storage is much more responsive than safe storage and no less reliable, however, it does lack security in the presence of malicious peers.
In safe storage mode, Nomad's peer-storage module sends a storage command to multiple group storage peers and to overlay storage. Only after all responses are received from both group and overlay storage is a decision made whether the request was a success or failure. If the majority of responses are successful, the response is determined as successful. Safe storage is coupled with Nomad's quorum mechanism, which is used to verify the number of successful storage responses. Safe storage is much less responsive compared to fast storage, but in the presence of malicious peers, safe storage is more secure.

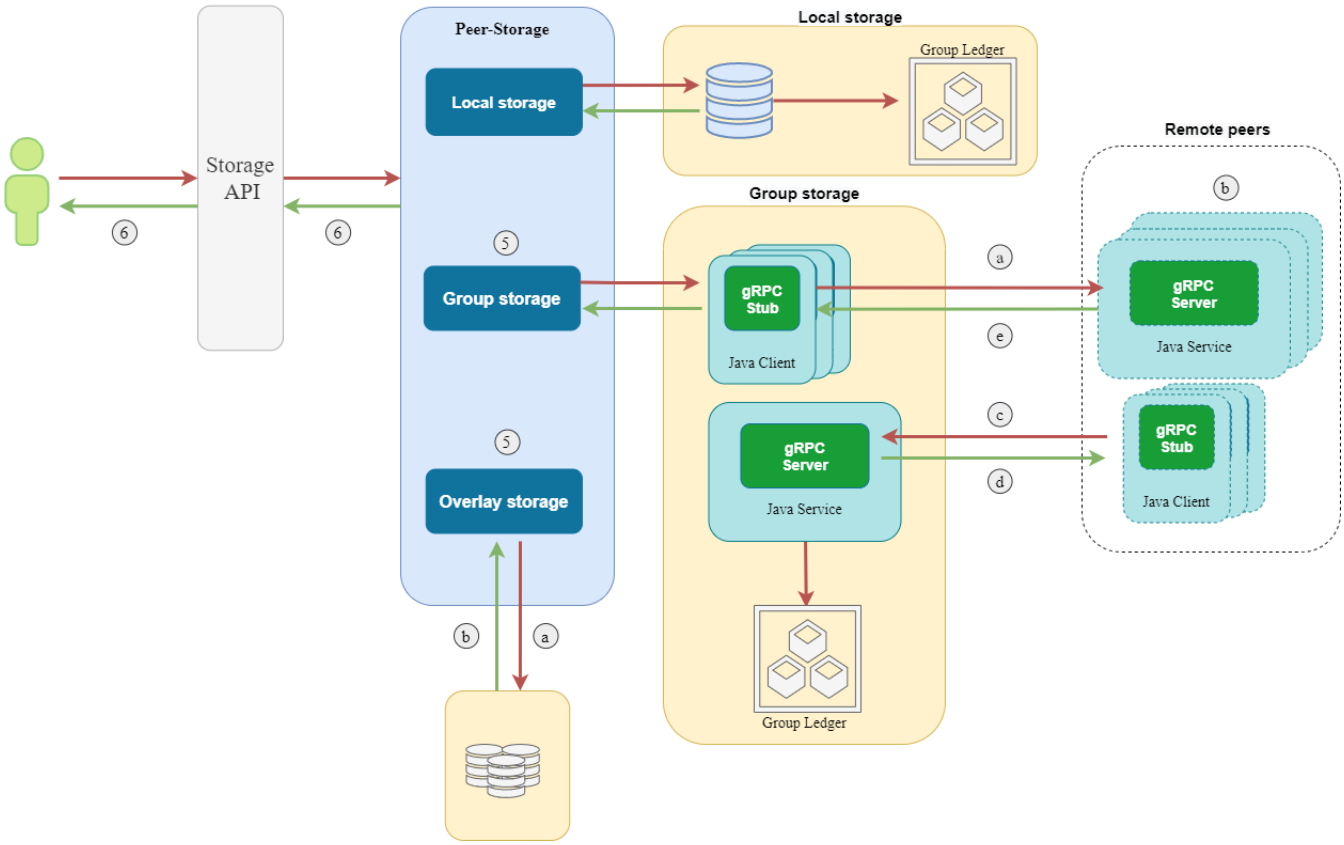


**Figure 5.11:** Nomad object storage procedure

Figure 5.11 illustrates Nomad's storage procedure. The illustration constitutes an extension of the steps described in section 5.3.7. Steps 1 to 4 are the same as for the add-to-ledger procedure, after which follows:

5. Nomad propagates the storage request to multiple peers within the group for replication purposes. Requests are made in parallel to ensure maximum efficiency.

   (a) The request is propagated to a subset of the group peers, depending on the replication factor ($\mathcal{R}$).

   (b) Each peer receiving the request ensures that it does not already store the object, by checking its group ledger. If the object is not stored on the peer, the peer will initiate its own storage request.

   (c) Once the object is stored successfully, an *addObjectReference* request is made to all group peers.

   (d) Peers add the object reference to their local group ledger and respond with an *ACK* message.

   (e) The peer informs the original requesting peer that the object replica has been stored successfully.

6. Nomad simultaneously propagates the storage request to its overlay storage component. As each peer participates in both group and overlay storage, overlay storage is represented by a single entity in figure 3.8. In reality, the storage request is propagated to another peer in the network.

   (a) The storage request is made to the overlay storage component.

   (b) The overlay storage component completes the storage request and responds with success or failure.

7. Nomad responds to the higher layer:

   (a) **Fast storage:** Nomad responds with the successful result from either overlay or group storage.

   (b) **Safe storage:** Nomad monitors responses from multiple peers in group storage and overlay storage. If the majority of responses are successful $[(\mathcal{R}/2) + 1]$, the response is deemed successful.

### 5.5.3. Retrieve

In section 3.10.2, Pithos' object retrieval procedure was described. This section provides an overview of how Nomad employs the designed procedure.

Nomad has three retrieval modes, namely *fast*, *safe* and *parallel* retrieval. Fast retrieval mode is similar to fast storage mode. In fast retrieval mode, Pithos sends a retrieval command to a single group storage peer and to overlay storage. In order to select a group peer to send the request to, the originating peer queries its group ledger and filters out all peers not storing the requested object. From the filtered peers, a random peer is selected and is sent the retrieval request. The first successful response object that it receives, either from group or overlay storage, is then propagated to the higher layer. Fast retrieval is generally very responsive and requires minimum bandwidth, but it is more susceptible to malicious peers and object manipulation.

In safe retrieval mode, Nomad's peer storage component sends a retrieval command to multiple group storage peers and to overlay storage. After a sufficient number of response objects are received, a quorum mechanism (3.9.1) is used to determine the original object. If the majority of responses are successful, the response is deemed successful. Safe storage is resilient against malicious peers, however similarly to safe storage, it is less responsive compared to fast and parallel retrieval.

In parallel retrieval mode, Pithos sends a retrieval command to multiple group storage peers, and to overlay storage. Similar to fast storage, parallel retrieval uses the first successful response to serve the higher layer. Parallel retrieval is generally more responsive and reliable than fast storage, but is more susceptible to malicious peers than safe retrieval.

As shown in Section 3.10.2, using multiple peers to serve retrieval requests improves responsiveness and reliability against object manipulation and network churn. Figure 5.12 illustrates Nomad's retrieval procedure.

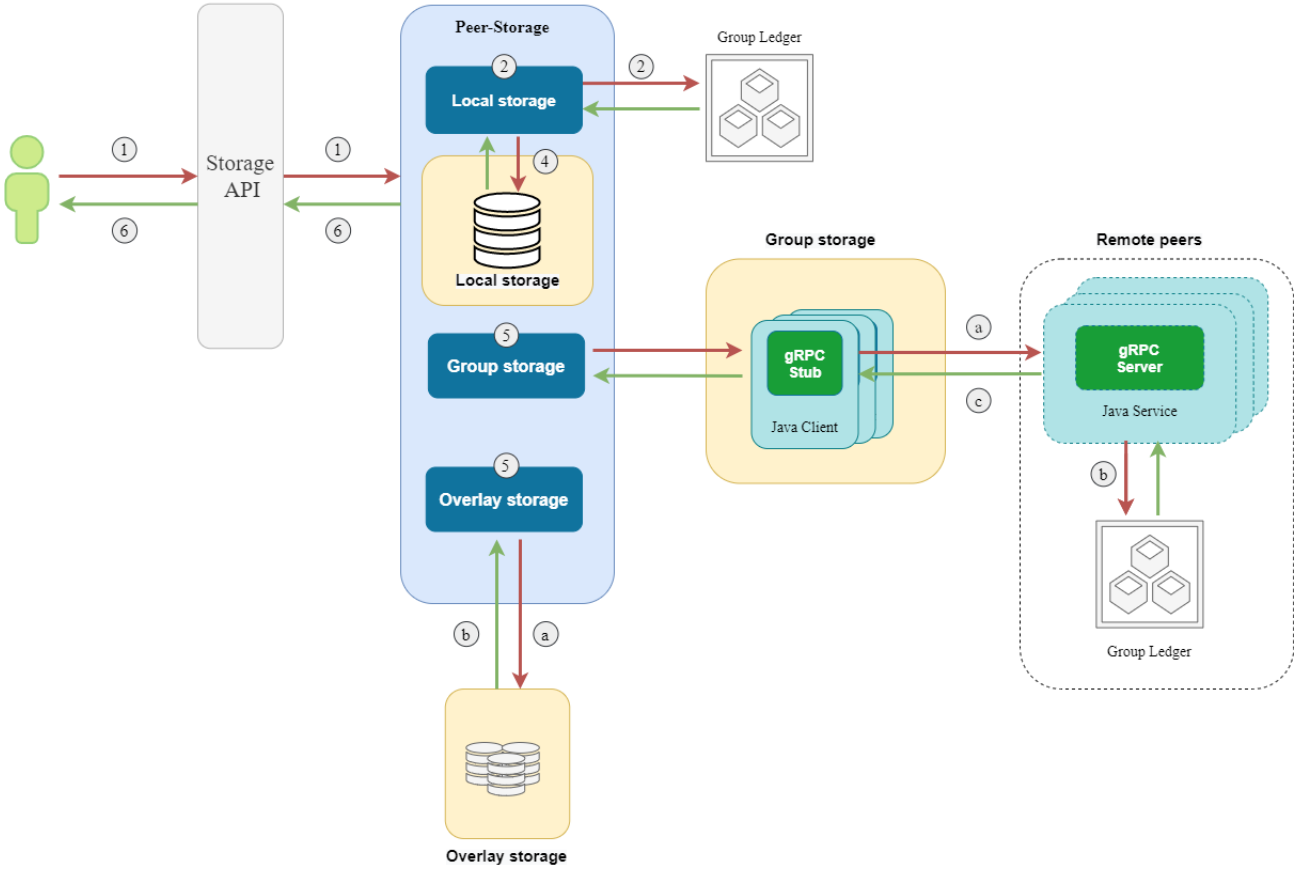


**Figure 5.12:** Nomad object retrieval procedure

1. A retrieval request is sent from the higher layer to the Nomad storage interface.
2. Nomad first determines whether the object is hosted within the group by checking its group ledger.
3. If the object is stored locally, it is retrieved from local storage and used to determine the appropriate response.
4. If the object is stored within the group, Nomad propagates the retrieval request to both group storage and overlay storage peers.
   (a) Nomad only sends a retrieval request to peers that, according to its group ledger, are storing the required object.
   (b) Peers that receive the retrieval request, verify that they are storing the requested object, by using their own group ledger. If the peer does store the object, it is retrieved from the peer's local storage.

(c) The retrieved object is returned to the originating peer, along with the appropriate response message.

5. Nomad simultaneously propagates the storage request to its overlay storage component. If the object is not stored within the group, a retrieval request is only sent to overlay storage. This is known as an out-of-group request.

   (a) The retrieval request is made to the overlay storage component.

   (b) The overlay storage component completes the retrieval request and responds with the resulting object.

6. Nomad responds to the higher layer:

   (a) **Fast retrieval:** Nomad responds with the resulting object from either local, overlay or group storage, whichever successful response it receives first.

   (b) **Safe retrieval:** Nomad monitors responses from local, group storage and overlay storage. A quorum mechanism $[(\mathcal{R}/2) + 1]$ determines if the request was successful and which object should be sent to the higher layer.

   (c) **Parallel retrieval:** Nomad responds with the resulting object from either local, overlay or group storage, whichever succeeds first.

### 5.5.4. Modify

Within Nomad, modify requests are treated similarly to store requests in safe mode. In contrast to modification in Pithos, Nomad in its current state requires the entire object payload to be provided. As an additional safety measure, object IDs are immutable. Similar to all other storage operations, modify requests are received from the higher layer.

The peer receiving the modify request from the higher layer verifies that the object is stored in the group. If the object is not stored within the group, the object is only modified in overlay storage. If the object is stored within, the group the object is modified within group storage and overlay storage. Only peers storing the object will receive an object update request. Only if all object modifications succeed is the modify request deemed successful.

In its current state, Nomad does not make use of a certification mechanism. Object changes are therefore not monitored.

### 5.5.5. Delete

The Pithos architecture does not explicitly support object removal. Similarly, Nomad's storage API does not explicitly support removal requests. Instead it relies on an object's TTL. When an object is created in Nomad, it is assigned a TTL that dictates the object's

"expiry" time. When an object's TTL period expires, the object is removed from group storage, overlay storage and the group ledger. Object TTL is directly supported in TomP2P, which means that objects are automatically removed from overlay storage after TTLs expire. As group storage is effectively a collection of every group peer's local storage, it is only required to manage object expiry on a local level. In order to ensure objects are removed from a peers local storage after their TTLs expire, a scheduled maintenance task periodically checks the local storage for expired TTLs. Objects that have expired TTLs are removed from local storage. In addition, peers use a similar scheduled maintenance task to remove expired objects from their group ledger.

Section 3.10.4 argues that object deletion in a distributed system can be fallible, especially if connectivity issues cause peers to be unavailable. This may lead to storage inconsistency within the group. Object TTL is a reliable way to ensure that objects are consistently removed from the system.

## 5.6. Conclusion

This chapter linked the conceptual design of Nomad to its real-world implementation. Nomad's various modules and mechanisms were discussed by providing information on the technologies used and the motivations for the design decisions. This chapter also demonstrated how Nomad satisfies Pithos' key use cases. Implementation design and UML diagrams can be found in Appendix C.

# Chapter 6

# Nomad Evaluation

Previous chapters, discussed the design and implementation of the Nomad decentralised storage network. This chapter focusses on the evaluation of Nomad and the suitability of the implementation for storage in P2P MMVE systems.

This chapter serves a dual purpose, firstly to verify the simulated Pithos results and secondly to verify that Nomad meets the storage requirements introduced in Section 2.3. To achieve this, a number of test scenarios were created to measure Nomad's responsiveness, reliability, fairness and security.

Nomad's group and overlay storage components are tested individually, which allows for direct comparison to those of Pithos.

## 6.1. Pithos Baseline Performance

In order to compare Nomad's implementation to that of Pithos, the Pithos results are used as a baseline. This section provides an overview of Pithos performance in a simulated environment for different storage criteria, namely responsiveness, reliability and scalability. These results were extracted from Engelbrecht and Gilmore [31] and are used with permission from the authors. For the purpose of establishing baseline performance, the results of Pithos' reliability, responsiveness and scalability will be highlighted. As supplementary information, Section D.1 shows Pithos' load balancing results.

### 6.1.1. Responsiveness and Reliability

This section provides an brief summary of Pithos' simulation results for overlay and group storage. Table 6.2 presents the responsiveness, reliability and overhead for various overlay implementations and using Pithos in different storage/retrieval modes.
Table 6.1 describes the experimental setup [31].

| Test Configuration | |
|---|---|
| Network Size | 2500 peers, 100 super-peers |
| Simulation Length | 10,000 s |
| Request Rate | 0.2 Requests Per Second (RPS) |
| No. Requests | 5 mil storage & retrieval requests, 600,000 generated objects |
| Object Size | 1024 bytes |
| Overlay | Chord |
| $\mathcal{R}$ | 6 |
| Repair | None |
| Request Description | Exclusively in-group requests (No out-of-group requests) |
| Simulation Framework | OverSim |

**Table 6.1:** Pithos responsiveness and reliability experimental setup [31]

| Entity | Reliability (%) | | Responsiveness (s) | | Bandwidth (Bps) | |
|---|---|---|---|---|---|---|
| | Store | Retrieve | Store | Retrieve | in | out |
| Chord (high) | 98.31 | 93.65 | 1.217 | 1.745 | 2175 | 2189 |
| Chord (medium) | 96.90 | 93.20 | 1.214 | 1.582 | 1183 | 1197 |
| Chord (low) | 79.08 | 62.16 | 1.245 | 2.071 | 301 | 314 |
| Pastry | 98.97 | 94.90 | 0.625 | 1.182 | 1979 | 2088 |
| Kademlia | 45.53 | 35.13 | 0.908 | 4.604 | 512 | 498 |
| Fast Pithos (fast retrieval) | 100.00 | 99.7 | 0.0665 | 0.192 | 1370 (187) | 1380 (183) |
| Fast Pithos (parallel retrieval) | 100.00 | 99.98 | 0.0665 | 0.085 | 1967 (784) | 1932 (735) |
| Safe Pithos (fast retrieval) | 97.05 | 99.77 | 1.554 | 0.189 | 1366 (183) | 1377 (180) |
| Safe Pithos (parallel retrieval) | 97.05 | 99.98 | 1.554 | 0.086 | 1981 (798) | 1991 (794) |

**Table 6.2:** Pithos performance for different overlay implementations as well as storage an retrieval modes [31]

#### 6.1.1.1. Overlay Storage

For the Pithos evaluation, various DHTs were tested. Table 6.2 provides an overview of the measured results for different DHTs. Out of all evaluated DHTs, Chord achieved the highest reliability, with a store reliability of 98.31% and a retrieval reliability of 93.65%. Kademlia proved to be the least reliable DHT, with a store reliability of 45.53% and a retrieval reliability of 35.13%. In terms of responsiveness, all DHTs were seen to have relatively high response times. Out of all evaluated DHTs, Pastry was found to have the best response times, at the cost of an additional bandwidth requirement. Kademlia proved to have the highest response times for object retrieval.

#### 6.1.1.2. Pithos (Overall) Storage

Since Pithos provides multiple storage and retrieval modes, it was evaluated for its responsiveness and reliability in different storage/retrieval modes. Pithos was found to be highly reliable for both fast and parallel retrieval modes, with reliabilities of 99.70% and 99.98% respectively. Parallel retrieval was found to be slightly more responsive than fast retrieval, since requests are sent to multiple peers, with retrieval response times of 8.4 ms versus 192 ms.

Pithos in safe storage mode provides a greater degree of security, by ensuring the majority of requests succeed before it determines success or failure. This comes at the cost of storage response times of 1.55 seconds. However, since safe storage ensures a lower number of false positives in the group ledger, responsiveness is slightly improved for both fast and parallel retrieval.

### 6.1.2. Scalability

This section provides an brief summary of Pithos' scalability for overlay, overall and group storage. Table 6.4 presents Pithos' scalability in terms of its overlay, group and overall scalability for different network sizes.
Table 6.3 describes the experimental setup [31].

| Test Configuration | |
|---|---|
| Network Size | (1) 2500 peers, 100 super-peers<br>(2) 10,000 peers, 400 super-peers. |
| Simulation Length | 10,000 s |
| Request Rate | 0.2 Requests Per Second (RPS) |
| No. Requests | 15.8 mil storage & retrieval requests.<br>2.4 mil generated objects |
| Object Size | 1024 bytes |
| Overlay | Chord (Medium) |
| $\mathcal{R}$ | 6 |
| Repair | None |
| Request Description | Exclusively in-group requests (No out-of-group requests) |
| Simulation Framework | OverSim |

**Table 6.3:** Pithos scalability experimental setup [31]

| No. of peers | Module | Reliability (%) | Responsiveness | Bandwidth (Bps) | |
|---|---|---|---|---|---|
| | | | mean (var.) (s) | in | out |
| 2600 | Pithos | 99.7 | 0.192 (0.181) | 1370 | 1380 |
| 2600 | Group | 97.75 | 0.134 (0.0629) | 187 | 183 |
| 2600 | Overlay | 91.4 | 1.76 (0.824) | 1183 | 1197 |
| 10400 | Pithos | 99.71 | 0.191 (0.194) | 1647 | 1657 |
| 10400 | Group | 98.19 | 0.134 (0.0674) | 180 | 177 |
| 10400 | Overlay | 90.06 | 1.96 (1.005) | 1467 | 1480 |

**Table 6.4:** Pithos performance for different network sizes [31]

To show that Pithos scales for different request rates, table 6.6 provides an overview of performance for different network sizes and request rates. Table 6.5 describes the experimental setup [31].

| Test Configuration | |
|---|---|
| Network Size | (1) 250 peers, 10 super-peers<br>(2) 100 peers, 40 super-peers.<br>(3) 2500 peers, 100 super-peers |
| Simulation Length | 10,000 s |
| Request Rate | 0.2, 4, 10 Requests Per Second (RPS) |
| No. Requests | 15.8 mil storage & retrieval requests.<br>2.4 mil generated objects |
| Object Size | 1024 bytes |
| Overlay | Chord (Medium) |
| $\mathcal{R}$ | 6 |
| Repair | None |
| Request Description | Exclusively in-group requests (No out-of-group requests) |
| Simulation Framework | OverSim |

**Table 6.5:** Pithos scalability with variable request rates experiment setup [31]

| No. peers | RPS per peer | Module | Reliability (%) | | Responsiveness (s) | | Bandwidth | (Bps) |
|---|---|---|---|---|---|---|---|---|
| | | | Store | Retrieve | Store | Retrieve | in | out |
| 260 | 0.2 | Pithos | 100 | 99.78 | 0.064 | 0.175 | 967 | 968 |
| 260 | 0.2 | Group | 99.84 | 98.03 | 0.064 | 0.131 | 176 | 164 |
| 260 | 0.2 | Overlay | 97.42 | 92.77 | 0.888 | 1.42 | 791 | 804 |
| 260 | 10 | Pithos | 100 | 99.79 | 0.065 | 0.169 | 19142 | 18819 |
| 260 | 10 | Group | 99.99 | 98.5 | 0.064 | 0.134 | 8966 | 7940 |
| 260 | 10 | Overlay | 97.52 | 92.33 | 0.911 | 1.462 | 10176 | 10879 |
| 1040 | 0.2 | Pithos | 100 | 99.75 | 0.066 | 0.184 | 1185 | 1184 |
| 1040 | 0.2 | Group | 99.87 | 98.2 | 0.065 | 0.135 | 177 | 166 |
| 1040 | 0.2 | Overlay | 97.09 | 91.49 | 1.113 | 1.676 | 1008 | 1018 |
| 1040 | 10 | Pithos | 100 | 99.77 | 0.065 | 0.171 | 20685 | 20351 |
| 1040 | 10 | Group | 99.85 | 98.43 | 0.064 | 0.131 | 9189 | 8159 |
| 1040 | 10 | Overlay | 97.33 | 91.78 | 1.089 | 1.622 | 11496 | 12192 |
| 2600 | 0.2 | Pithos | 100 | 99.71 | 0.065 | 0.192 | 1359 | 1362 |
| 2600 | 0.2 | Group | 99.88 | 97.98 | 0.064 | 0.134 | 177 | 166 |
| 2600 | 0.2 | Overlay | 95.93 | 91.05 | 1.217 | 1.781 | 1182 | 1196 |
| 2600 | 4 | Pithos | 100 | 99.75 | 0.065 | 0.182 | 9145 | 9023 |
| 2600 | 4 | Group | 99.9 | 98.3 | 0.065 | 0.134 | 3672 | 3273 |
| 2600 | 4 | Overlay | 97.11 | 91.16 | 1.213 | 1.774 | 5473 | 5750 |

**Table 6.6:** Pithos performance for different network sizes and request rates [31]

#### 6.1.2.1. Group Storage

From both experiments, it is evident that Pithos' group storage component scales well. Group storage maintains its reliability of above 98% and response times of around 6 ms and shows no increase in bandwidth for all network sizes. Different request rates did not affect group storage responsiveness or reliability; however, as expected, bandwidth usage increased with higher request rates.

#### 6.1.2.2. Overlay Storage

The results from both experiments, show that Pithos' overlay storage component does not perform well. It was found that for increased network sizes, overlay storage performance decreased in both responsiveness and reliably. Different request rates seemingly had no effect on the overlays performance.

#### 6.1.2.3. Pithos (Overall) Storage

From both experiments, it can be concluded that Pithos is scalable. For object retrieval at different network sizes and request rates, Pithos maintains a reliability of 100% and response times of around 6 ms. For object storage, Pithos maintains a reliability of above 99.70% and response times of below 200 ms. A slight increase in bandwidth usage was seen, which was caused by the overlay storage.

## 6.2. Experimental Setup

Now that Pithos' baseline performance has been established, Nomad can be evaluated and compared accordingly. This section provides an overview of the experimental infrastructure used to evaluate Nomad.

### 6.2.1. Experimental Infrastructure

A local area network (LAN) consisting of four physical machines (nodes) and a network switch was. Using multiple physical hosts provides more realistic network conditions compared to those of a simulated environment, whilst still ensuring low latency connections. All four nodes were physically (Ethernet) connected to a network switch to ensure local latencies of under 2 ms. Each node hosted multiple **Nomad** peers.

Although Nomad was specifically designed to be future proof, and potentially deployed to the cloud, additional development is required to solve NAT traversal complications when using cloud instances. This was deemed future work and it was therefore decided that a physical network should be used to evaluate Nomad.

#### 6.2.1.1. Network Description

| Network Switch | |
|---|---|
| Model | TP-Link Archer C2300 |
| LAN Bandwidth | 1 Gigabit (1000 Mbps) |

| Node 1 | |
|---|---|
| Operating System | Windows 10 |
| CPU | AMD Ryzen 5 3600X (6-cores @ 4.0 GHz) |
| Memory (RAM) | 32 GB |

| Node 2 | |
|---|---|
| Operating System | Windows 10 |
| CPU | AMD Ryzen 5 4500U (6-cores @ 4.0 GHz) |
| Memory (RAM) | 16 GB |

| Node 3 | |
|---|---|
| Operating System | OS X |
| CPU | Intel Core i7 (6-cores @ 2.6 GHz) |
| Memory (RAM) | 16 GB |

| Node 4 | |
|---|---|
| Operating System | Ubuntu 16.04 LTS |
| CPU | Intel Core i7 (4-cores @ 2.0 GHz) |
| Memory (RAM) | 16 GB |

### 6.2.2. Directory Server

Nomad requires a ZooKeeper instance for its directory server functionality. To satisfy this requirement a ZooKeeper Docker container was deployed to a single node Kubernetes (K8s) cluster on *node-4*. The K8s cluster was created and managed using *kubeadm* [87].

### 6.2.3. Local Storage Mode

To ensure that Nomad's evaluation results would be comparable to those measured during Pithos' simulations, it was decided to use Nomad's local storage component in *in-memory* storage mode. This would ensure optimal responsiveness for database queries. As previously stated, using in-memory storage does increase resource usage and therefore limits the amount of instances that can run simultaneously on a single machine. Super-peers consume on average **350 MB** of memory, whereas storage peers using an in-memory database can consume up to **1.5 GB** of memory, depending on object lifetimes and session length.

### 6.2.4. Generating Load

Artillery is a powerful and easy-to-use performance testing toolkit. The toolkit can be used to execute test scenarios against systems, in order to generate a controlled load on the system. Performance tests are intended to verify that a system is working as expected under specific user-controlled test scenarios and load.

No higher layer application implementing Nomad exists as of yet. A standalone higher layer application was therefore required to generate storage requests on Nomad peers. To

satisfy this requirement, Artillery [88] was used to generate HTTP requests to each peer's storage API.

For Nomad's evaluation, Artillery performance tests were executed from *node-3*. Test scenarios that were tested include, *store, retrieve* and *modify*. Variable request rates were used during load tests.

### 6.2.5. Test Objects

Artillery was used to generate random payload values. Below is a code snippet of Nomad's API *GameObject*, which is used as both request and response object payload. The *value* field is used to store a base64 encoded version of the VE object.

For the Pithos simulation, an object size of **1024 bytes** was used [1]. The same was used for the Nomad evaluation, in order to ensure results are comparable.

In general, an object TTL is set on a per-request basis. However, for evaluation purposes and taking node resource limitation into account, a static TTL of 600 seconds was used per object.

```
{
``id": "123",
"creationTime": 1628060903,
"lastModified": 1628060903,
"ttl": 600,
"value": "/9j/4AAQSkZJR..."
}
```

**Listing 6.1:** Nomad's API Request\Response object

### 6.2.6. Measurement of Metrics

The key metrics that were used to evaluate the system's performance were: reliability, responsiveness (latency) and bandwidth usage.

#### 6.2.6.1. Reliability

The reliability of Nomad's HTTP requests was measured as the ratio of successful responses received, to the total number of requests sent. This is expressed by the formula:

$$reliability = \frac{successful\ responses}{total\ requests}$$

#### 6.2.6.2. Responsiveness

Responsiveness was measured as the round-trip latency of a request in milliseconds or seconds. Round-trip latency (RTL) or round-trip delay (RTD) in this context, refers to the time it takes a request to be sent plus the time it takes for a response to that request to be received. The RTL also includes the processing time of the peer receiving the request. For all response times measured, the 95th and 99th percentiles were used.

#### 6.2.6.3. Load Balancing

Nomad's load balancing was evaluated by increasing and decreasing group sizes whilst maintaining a stable load. This differs from how load balancing was evaluated for Pithos, where object distribution was tested. With Nomad, request distribution was tested instead. For this reason, the load balancing results will not be compared.

#### 6.2.6.4. Bandwidth Usage

Bandwidth was measured in bytes per second (Bps) and kilobytes per second (KBps), referring to the amount of data transferred. As Nomad's storage layer consists of two separate decentralised storage mechanisms, a distinction could be made between the bandwidth usage of group storage and overlay storage. Group storage traffic primarily consists of TCP requests, whereas overlay storage traffic consists of a combination of TCP and UDP requests. The measured bandwidth therefore included both TCP and UDP traffic. Measured packets generally consisted of a message payload, described by Section 6.2.5, and network overhead such as a preamble, destination and source Media Access Control (MAC) addresses, protocol type, and frame check sequences [89].

Wireshark [90], a popular network protocol analyser, was used to measure Nomad's incoming and outgoing bandwidth. All network packets were captured and evaluated on Node 1. To measure bandwidth usage, one peer from each Nomad group was selected and TCP and UDP traffic on their group storage and overlay storage ports were measured. A top level traffic filter can be described by the following Wireshark filter query:

```
(ip.addr eq <node1> and ip.addr eq <node2>) or
(ip.addr eq <node2> and <node1>) or
(ip.addr eq <node1> and ip.addr eq <node3>) or
(ip.addr eq <node3> and <node1>) or
(ip.addr eq localhost and ip.addr eq localhost)
```

**Listing 6.2:** Wireshark filter query

Bandwidth was measure using Wireshark's I/O graphs, illustrated in Figure 6.1. The figure shows results for an example experiment with a Nomad network of 24 peers and 1 super-peer, using safe retrieval mode, each peer receiving an arbitrary load. The graph

illustrates the group storage bandwidth usage for 7 Nomad group peers with an interval of 1 second and a Simple Moving Average (SMA) period of 20. To establish each group's overlay and group storage bandwidth usage, additional graph queries were used:

```
Group Storage:
tcp.port == <group-port>
Overlay Storage:
tcp.port == <overlay-port> || udp.port == <overlay-port>
```

**Listing 6.3:** Wireshark I/O graph query

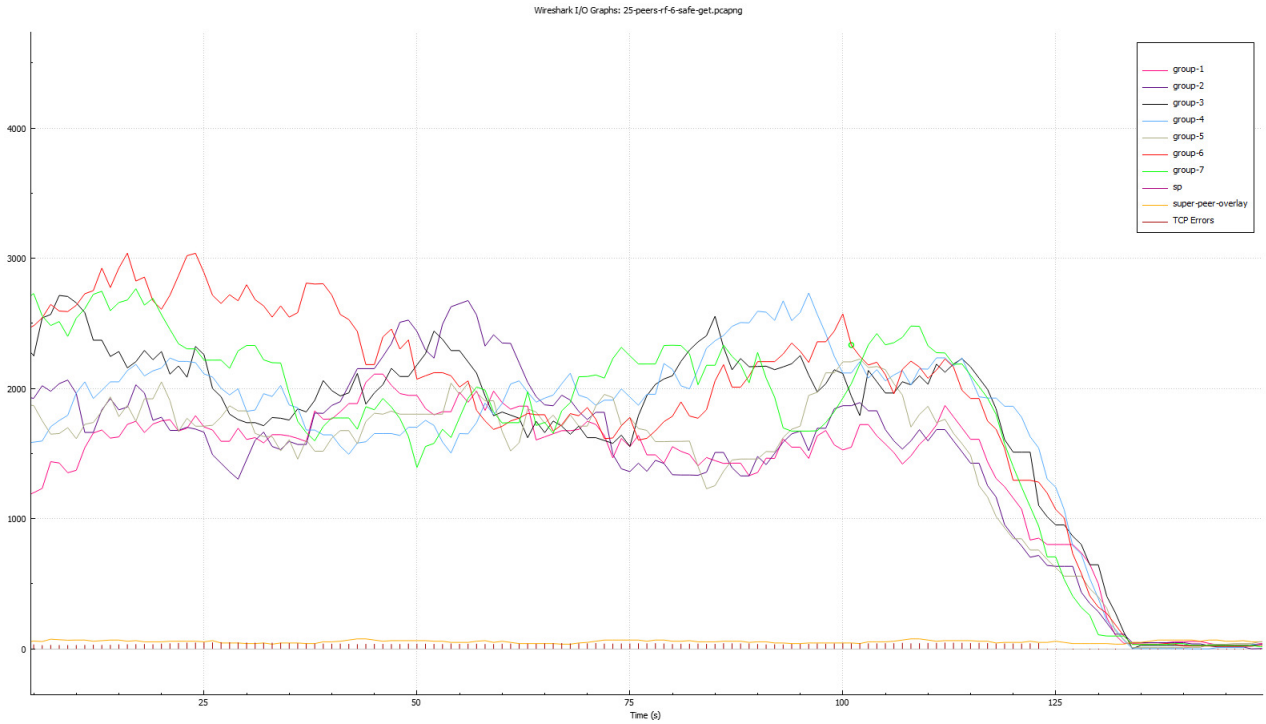


**Figure 6.1:** Example Wireshark I/O graph

## 6.3. Group Storage: Responsiveness, Reliability and Load Balancing

In order to evaluate Nomad's group storage component in isolation, a test API was implemented, namely the *GroupStorageController*. This API provided an isolated interface for direct access to Nomad's group storage. When this API endpoint is used, local storage becomes inaccessible, which means that all responses have to be extracted from another peer in the group. The purpose of this test was to assess the responsiveness, reliability and load balancing of Nomad's group storage. The impact of various application configurations on group storage's responsiveness and reliability was also measured. Variables that impact responsiveness and reliability include **group size, replication factor**, and **storage and**

**retrieval mode**.

### 6.3.1. Experimental Setup

Table 6.7 describes the experimental setup used during group storage evaluation.

| Test Configuration | |
|---|---|
| Group Size | 5 to 25 (4 to 24 peers, 1 super-peer) |
| Simulation Length | 1800 to 3600 s |
| Request Rate | 100 Requests Per Second (RPS) |
| No. Requests | 150,000 requests (avg.) |
| Object Size | 1024 bytes |
| Overlay | TomP2P (Kademlia) |
| $\mathcal{R}$ | 2 to 6 replicas |
| Storage Modes | Fast, Safe |
| Retrieval Modes | Fast, Parallel, Safe |
| Scheduled Repair | Disabled |
| Leave Repair | Enabled |
| Group Migration | Disabled |
| Voronoi Grouping | Disabled |
| Request Description | Exclusively in-group requests (No out-of-group requests) |

**Table 6.7:** Group storage experimental setup

#### 6.3.1.1. Group Size

To measure the effect of group size on performance, group sizes were varied from 5 to 25 peers per group. For these various network sizes, the response time and success rate were measured for object storage and retrieval.

#### 6.3.1.2. Iteration Length

Each experiment was run for 1800 to 3600 seconds, generating on average 150,000 storage and retrieval requests. During storage tests, an average of 150,000 objects were generated.

#### 6.3.1.3. Request Rate

A load test of 100 Requests Per Second (RPS) was executed on each iteration, with a 60 second initialisation phase, which was increased incrementally from 10 to 100 RPS. The request rate was kept stable at 100 RPS for the rest of the experiment.

### 6.3.2. Retrieval Results

This section provides performance the results of group storage retrieval operations for various group sizes and replication factors.

#### 6.3.2.1. The Effect of Group Size on Retrieval

The following subsection describes the effect of group size on retrieval responsiveness and reliability. During this experiment, a replication factor of 6[1] was used, which ensured that the original object, including a 5 replicas of it, would be stored in the network.

| Retrieval mode | Group size | Responsiveness (ms) | Reliability (%) |
|---|---|---|---|
| Fast | 5 | 10.90 | 100.00 |
| | 8 | 13.97 | 100.00 |
| | 15 | 14.56 | 100.00 |
| | 25 | 14.91 | 100.00 |
| Parallel | 5 | 26.87 | 100.00 |
| | 8 | 20.88 | 100.00 |
| | 15 | 16.61 | 100.00 |
| | 25 | 14.84 | 100.00 |
| Safe | 5 | 28.36 | 100.00 |
| | 8 | 19.23 | 100.00 |
| | 15 | 21.89 | 100.00 |
| | 25 | 17.54 | 100.00 |

**Table 6.8:** The effect of group size on retrieval response time and reliability

Table 6.8 provides an overview of Nomad's responsiveness for different group sizes and retrieval modes. Since the responsiveness and reliability experiments were tested under a stable load, group storage load balancing was inherently tested. The overall trend shows that a larger group size improves load balancing within the group, and therefore leads to improved response times.

*Fast* retrieval was marginally impacted by group size. Retrieval times increase from 10.90 ms to 14.91 ms for group sizes 5 to 25 peers. The overall trend of higher response times can be attributed to variable peer latencies, since some peers provide better response times than others. Figure 6.2 illustrates fast retrieval response times for various group sizes.

*Parallel* retrieval was also found to be marginally impacted by the group size. Since parallel retrieval sends multiple group requests to random peers, more responsive peers

[1]When the replication factor is higher than the group size, as many replicas as available peers are stored.

are favoured. The overall load is also better distributed within the group as the group increases. Under a stable load, response times therefore decrease from 26.87 ms to 14.84 as the group size increases from 5 to 25 peers, attributed to better load balancing and favouring of more responsive peers. Figure 6.3 illustrates parallel retrieval response times for various group sizes. *Safe* retrieval follows a similar trend: as the group size increases from 5 to 25 peers, response times steadily decrease from 28.36 ms to 17.54 ms. Figure 6.4 illustrates safe retrieval response times for various group sizes.

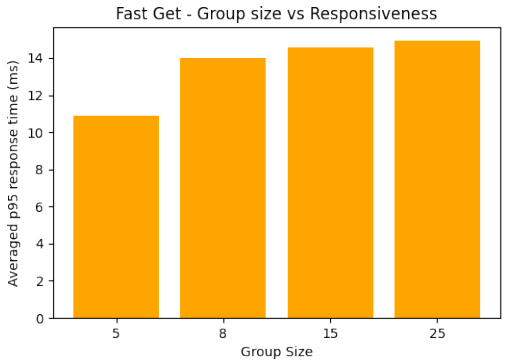


**Figure 6.2:** Fast Retrieval - response time versus group size

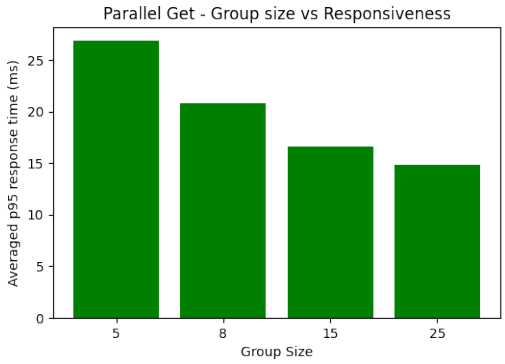


**Figure 6.3:** Parallel Retrieval - response time versus group size

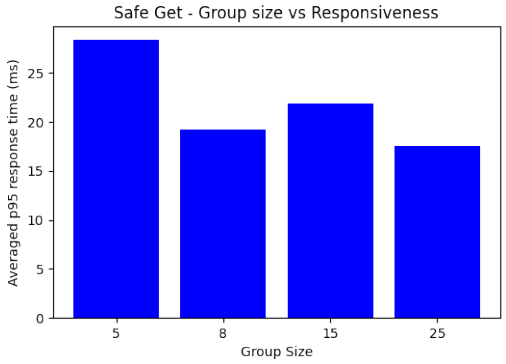


**Figure 6.4:** Safe Retrieval - response time versus group size

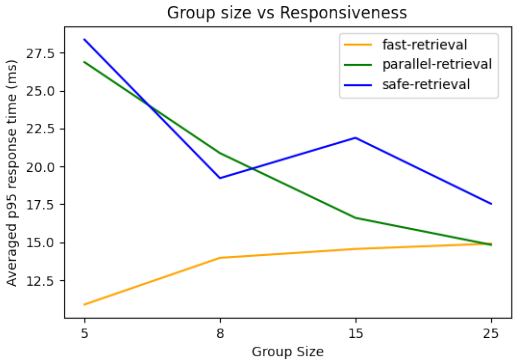


**Figure 6.5:** Retrieval responsiveness for different storage modes versus group size

Figure 6.5 illustrates the overall group size versus responsiveness trends per retrieval mode. Although parallel retrieval was expected to consistently outperform fast retrieval, the result show that only as group size reaches 25 peers parallel retrieval provides better response times. This can be partially explained by Nomad's use of Java thread pools and resource limitations. Since multiple instances ran on each machine thread pools were limited to two threads per instance to avoid system overloading.

In terms of overall reliability, when using six object replicas, all three retrieval modes proved to be highly reliable, with 100% reliability regardless of group size. This is an indication that reliability is dependent on the replication factor.

#### 6.3.2.2. The Effect of Replication Factor on Retrieval

In order to evaluate the effect of the number of replicas versus group size for object retrieval, a static group size of 25 peers was used.

| Retrieval Mode | $\mathcal{R}$ | Responsiveness (ms) | Reliability (%) |
|---|---|---|---|
| Fast | 2 | 13.09 | 99.87 |
| | 4 | 5.65 | 100.00 |
| | 6 | 14.91 | 100.00 |
| Parallel | 2 | 7.90 | 99.92 |
| | 4 | 6.29 | 100.00 |
| | 6 | 14.84 | 100.00 |
| Safe | 2 | 8.75 | 99.96 |
| | 4 | 9.05 | 100.00 |
| | 6 | 17.54 | 100.00 |

**Table 6.9:** The effect of replication factor on retrieval response time and reliability

Table 6.9 provides an overview of Nomad's responsiveness using a group size of 25 and various replication factors ($\mathcal{R}$) and retrieval modes. The overall trend shows that for a higher replication factor, the reliability increases at the cost of higher response times, due to an increase in group storage traffic.

For *parallel* and *safe* retrieval, the replication factor dictates how many group retrieval requests are made. Since both retrieval modes make use of parallel requests, a higher $\mathcal{R}$ improves reliability, from 99.92% and 99.96% respectively to 100%. However, more group requests lead to higher load on the group, which increases response times. Parallel retrieval response time increases from 7.90 ms to 14.84 ms. Similarly safe retrieval response times increase from 8.75 ms to 17.54 ms.

*Fast* retrieval's reliability increases from 99.87% to 100%, since more replicas are available. Interestingly, fast retrieval response time also increases from 13.09 ms to 14.91 ms with a higher $\mathcal{R}$. Since fast retrieval only sends one retrieval request, it would be expected that the response time would remain stable. The increase in response times can be attributed to varying peer latencies, or an increase in load on peers storing requested objects, which in turn leads to some peers receiving a higher load than others. This indicates that fast storage might not be as load-balanced as the other retrieval modes.

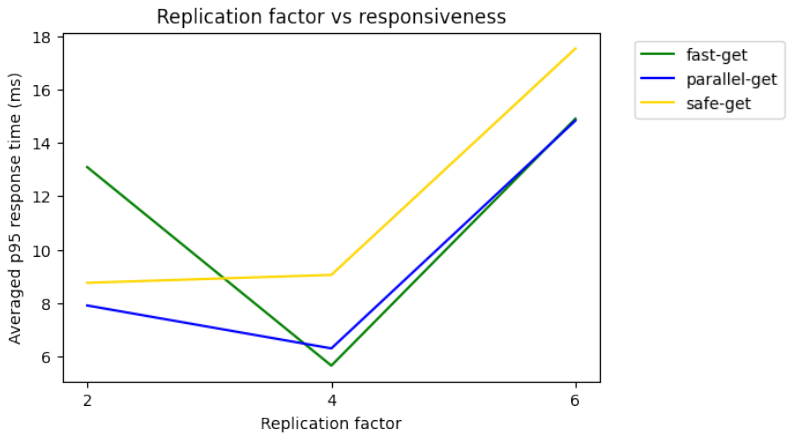


**Figure 6.6:** Effect of replication factor on response time for (a) fast retrieval, (b) parallel retrieval and (c) safe retrieval

Figure 6.6 illustrates the response times for the different retrieval modes, using various replication factors. From the figure, it can be concluded that parallel retrieval performs marginally better than both safe and fast retrieval. An interesting observation is made when $\mathcal{R} = 4$, where response times for fast and parallel retrieval seem to improve and then increase again when $\mathcal{R} = 6$. For parallel retrieval, this may be attributed to a limited number of threads available for each instance. However, more investigation is needed to verify this. Tables E.1 in appendix E provides a broad overview of replication factor on retrieval response time and reliability for various group sizes.

### 6.3.3. Storage Results

In order to evaluate the effect of the replication factor on object retrieval responsiveness and reliability, a static group size of 25 peers was used.

| Storage mode | Group size | Responsiveness (ms) | Reliability (%) |
|---|---|---|---|
| Fast | 5 | 23.46 | 100.00 |
| | 8 | 29.61 | 100.00 |
| | 15 | 67.85 | 100.00 |
| | 25 | 66.73 | 100.00 |
| Safe | 5 | 31.65 | 100.00 |
| | 8 | 66.60 | 100.00 |
| | 15 | 127.91 | 100.00 |
| | 25 | 232.00 | 100.00 |

**Table 6.10:** The effect of group size on storage response time and reliability

Table 6.10 provides an overview of Nomad's responsiveness for different group sizes and storage modes. Since peers need to inform all group participants of new objects, the

amount of required messages increases quadratically ($N^2$) as group size increases. Both fast and safe storage modes are therefore severely impacted by increases in group size.

For *fast* storage, as shown by figure 6.7, overall response time increases from 23.46 ms to 66.73 ms for group sizes 5 to 25 peers. For *safe* storage, as shown by figure 6.8, overall response time increases from 31.65 ms to 232 ms for group sizes 5 to 25 peers. In general, fast storage provides better response times than safe storage, regardless of group size. In terms of reliability, both storage modes provide a reliability of 100% when $\mathcal{R} = 6$. From Figure 6.8 it is clear that response time increases as group size increases.

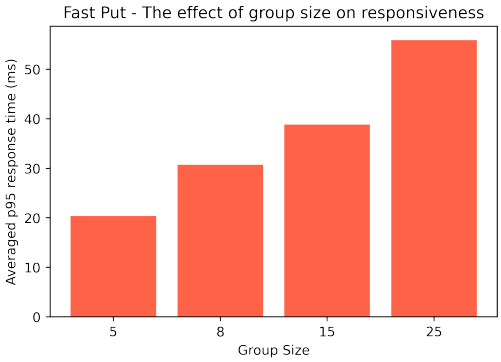


**Figure 6.7:** Fast storage - response time versus group size

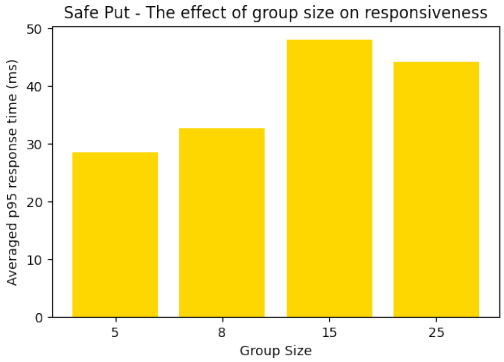


**Figure 6.8:** Safe storage - response time versus group size


**Figure 6.9:** Storage responsiveness for different storage modes versus group size

#### 6.3.3.1. The Effect of Replication Factor on Storage

In order to evaluate the effect of the replication factor on object storage responsiveness and reliability, a static group size of 25 peers was used.

| Storage Mode | $\mathcal{R}$ | Responsiveness (ms) | Reliability (%) |
|---|---|---|---|
| Fast | 2 | 40.48 | 100.00 |
| | 4 | 67.53 | 100.00 |
| | 6 | 66.73 | 100.00 |
| Safe | 2 | 32.55 | 99.99 |
| | 4 | 52.89 | 100.00 |
| | 6 | 232.00 | 100.00 |

**Table 6.11:** The effect of replication factor on storage response time and reliability

Table 6.11 provides an overview of Nomad's responsiveness for different group sizes, replication factors ($\mathcal{R}$) and storage modes. The overall trend shows that for a higher $\mathcal{R}$, the reliability increases at the cost of higher response times, due to an increase in group storage traffic.

*Fast* storage is only marginally affected by the replication factor, since it only reports the first successful response, disregarding any other storage responses. A reliability factor of 100% may be misleading, since some replicas could have failed. In terms of responsiveness, as the replication factor is increased from $\mathcal{R} = 2$ to $\mathcal{R} = 6$, response times increase from 40.48 ms to 66.73 ms.

*Safe* storage, however, is severely impacted by the replication factor, since it is required to wait for all storage requests to complete before it can report success or failure. As the replication factor is increased from $\mathcal{R} = 2$ to $\mathcal{R} = 6$, the reliability increases from 99.99% to 100%, which means no "replica" storage requests fail. Yet the increase in $\mathcal{R}$ causes an increase in response times, from 32.55 ms to 232 ms.

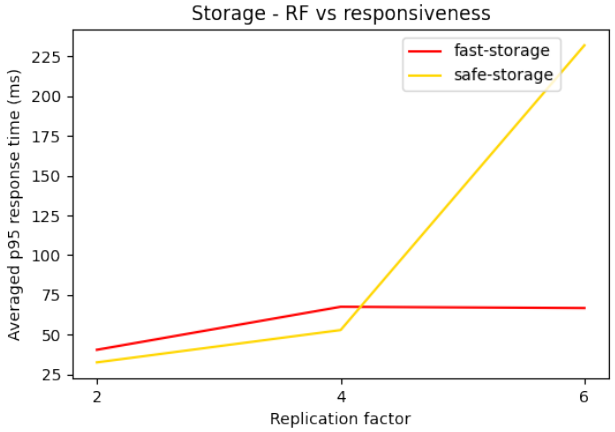


**Figure 6.10:** Storage responsiveness versus replication factor

Figure 6.10 illustrates storage response times versus $\mathcal{R}$. Overall, Fast storage provides lower response times than safe storage. An interesting observation is made when $\mathcal{R} = 2$ is increased to $\mathcal{R} = 4$, at which point safe storage provides similar response times to fast

storage. This is attributed to differences in thread pool size for the two storage modes. Safe storage has access to three threads per call, whereas fast storage only has access to two threads per call. Effectively, safe storage provides a higher degree of parallelism due to its stricter success criteria. Table E.2 in Appendix E shows the effect of replication factor on storage response time and reliability for various group sizes.

### 6.3.4. Group Storage Bandwidth Requirements

In order to compare Nomad's group storage component to that of Pithos', bandwidth requirements have to be taken into account. This section provides an overview of group storage bandwidth requirements. It should be noted that all bandwidth results are rough averages, measured using Wireshark I/O graphs, with an interval of 1 second and an SMA period of 20, in order to visualise bytes per second.

#### 6.3.4.1. Bandwidth Usage

The following experiments were performed on a single group, varying its group size from 5 to 25 peers. The bandwidth requirements for two request rates were measured, namely 0.2 RPS and 10 RPS per peer.

| Bandwidth - 0.2 RPS per peer | | | | |
|---|---|---|---|---|
| G.size | $\mathcal{R} \rightarrow$ | 2 | 3 | 6 |
| | Operation | Bandwidth (Bps) | | |
| 5 | fast-put | 550 | 900 | 4200 |
| | safe-put | 550 | 925 | 4250 |
| | fast-get | 550 | 500 | 550 |
| | parallel-get | 800 | 1100 | 1075 |
| | safe-get | 800 | 1100 | 1500 |
| 10 | fast-put | 750 | 1600 | 2250 |
| | safe-put | 750 | 1600 | 3650 |
| | fast-get | 550 | 550 | 550 |
| | parallel-get | 800 | 1250 | 1250 |
| | safe-get | 800 | 1250 | 1600 |
| 15 | fast-put | 1350 | 2900 | 4250 |
| | safe-put | 1350 | 2900 | 7500 |
| | fast-get | 550 | 550 | 550 |
| | parallel-get | 800 | 1250 | 1250 |
| | safe-get | 800 | 1250 | 2100 |
| 25 | fast-put | 2500 | 7200 | 7500 |
| | safe-put | 2500 | 7200 | 12,400 |
| | fast-get | 550 | 650 | 550 |
| | parallel-get | 800 | 1350 | 1250 |
| | safe-get | 800 | 1350 | 2000 |

**Table 6.12:** Summary of how storage mode, group size and replication factor impact Nomad's group storage bandwidth usage

| Bandwidth - 10 RPS per peer | | |
|---|---|---|
| G.size | $\mathcal{R} \rightarrow$ | 6 |
| | Operation | Bandwidth (Bps) |
| 5 | fast-put | 46,000 |
| | safe-put | 46,500 |
| | fast-get | 17,500 |
| | parallel-get | 47,000 |
| | safe-get | 45,000 |
| 10 | fast-put | 79,500 |
| | safe-put | 79,500 |
| | fast-get | 15,000 |
| | parallel-get | 45,000 |
| | safe-get | 44,000 |
| 15 | fast-put | 141,000 |
| | safe-put | 139,000 |
| | fast-get | 16,000 |
| | parallel-get | 51,500 |
| | safe-get | 49,000 |
| 25 | fast-put | 290,000 |
| | safe-put | 32,0000 |
| | fast-get | 22,000 |
| | parallel-get | 65,000 |
| | safe-get | 65,000 |

**Table 6.13:** Summary of how storage mode and group size impact Nomad's group storage bandwidth usage using a replication factor of 6

Tables 6.12 and 6.13 provide an overview of Nomad's bandwidth requirements for different replication factors, group sizes and request rates. Table 6.12 proves that both group size and replication factor impact the bandwidth usage. Figures 6.11, 6.12, 6.13 and 6.14 show that as the group size increases, the bandwidth requirements increase. This is attributed to the additional messages required to replicate objects and inform peers of a new objects.

Fast retrieval is far more bandwidth efficient than both safe and parallel retrieval, since only one request is sent to a peer containing the requested object. The additional bandwidth requirement of parallel and safe storage does, however, contribute to the security of retrieval requests, as multiple responses can be compared.

Fast and safe storage were found to have very similar bandwidth footprints. In contrast to fast retrieval, fast storage sends multiple storage requests, determined by the replication

factor. This means that its bandwidth requirement is similar to that of safe storage, but since it only waits for the first successful response, its bandwidth requirement for inbound traffic is slightly lower.

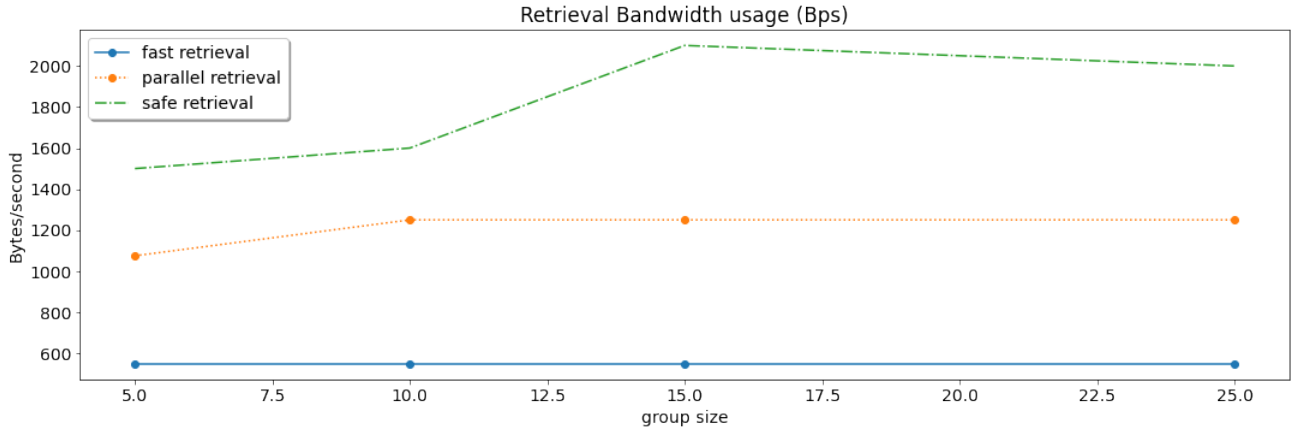


**Figure 6.11:** Nomad group storage retrieval bandwidth usage for 0.2 RPS per peer, $\mathcal{R} = 6$

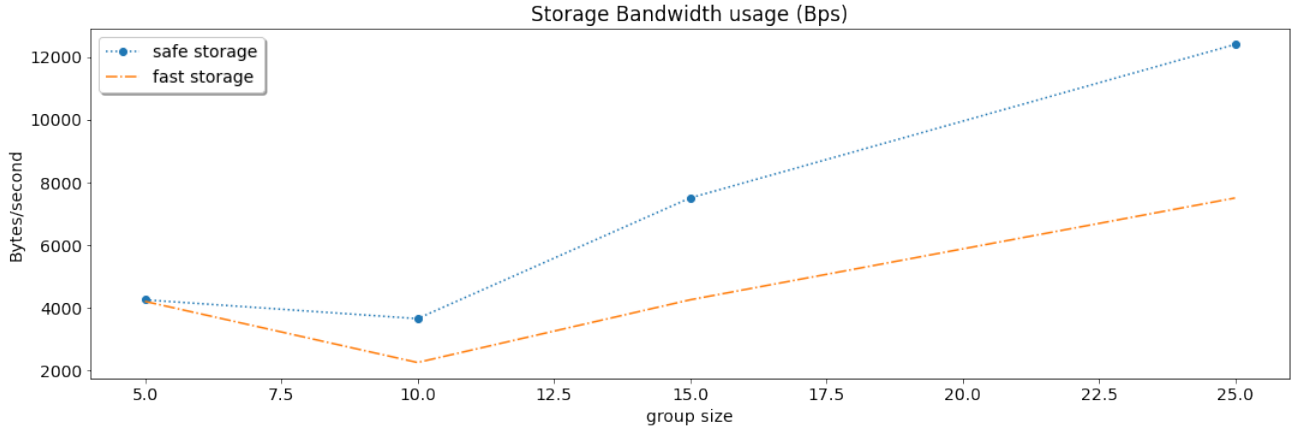


**Figure 6.12:** Nomad group storage store bandwidth usage for 0.2 RPS per peer, $\mathcal{R} = 6$

Table 6.13 shows that increasing the request rate by a factor of 50 has a significant impact on Nomad's group storage bandwidth requirement. The bandwidth usage is expected to increase by a similar factor; however, the measured bandwidth usage increases only by a factor of 30. This can be attributed to inaccurate bandwidth measurements, since Wireshark was only used to provide an estimate of the bandwidth usage results, and not for fine grain precision. Section 6.2.6.4 provides more information on how bandwidth was measured.

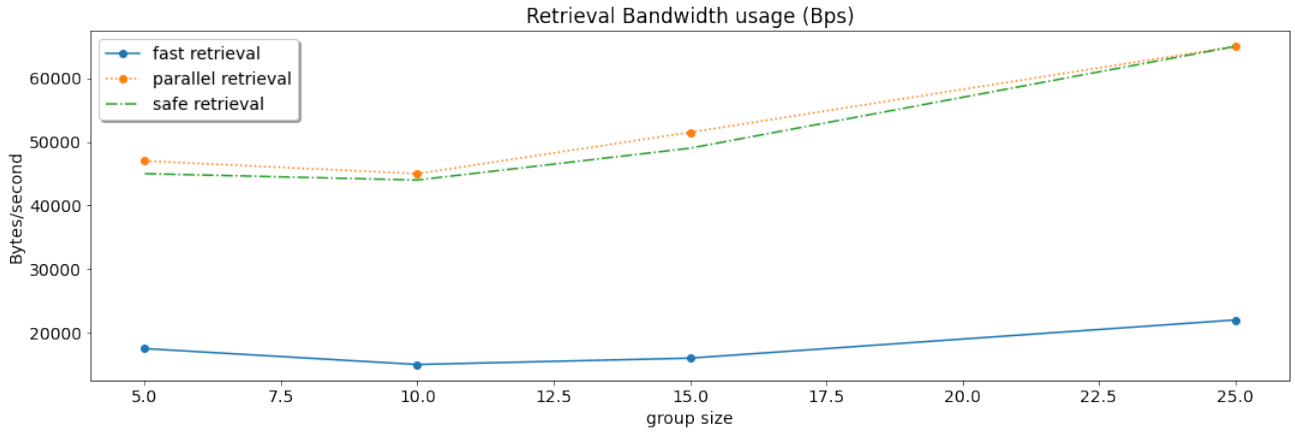


**Figure 6.13:** Nomad group storage retrieval bandwidth usage - 10 RPS per peer, $\mathcal{R} = 6$

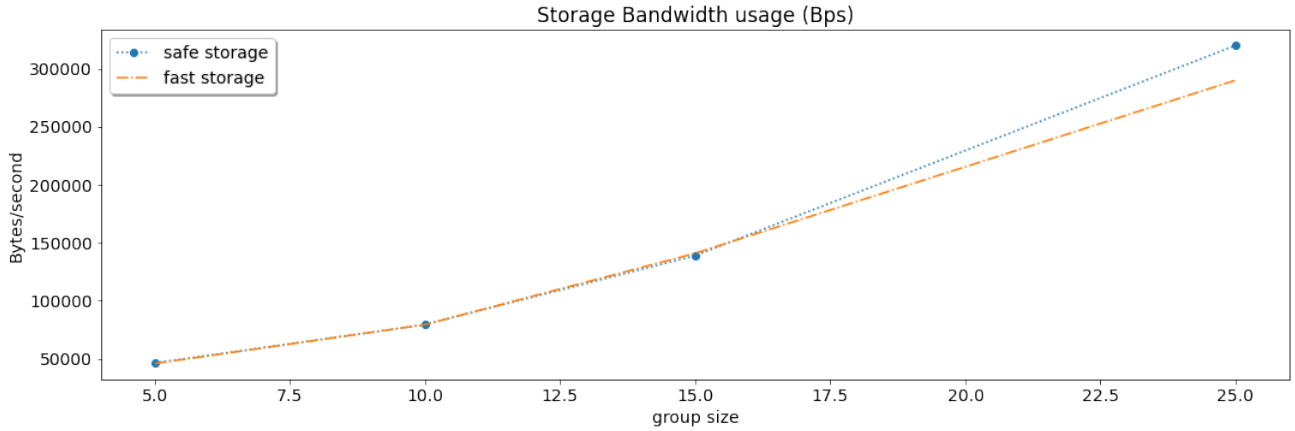


**Figure 6.14:** Nomad group storage store bandwidth usage - 10 RPS per peer, $\mathcal{R} = 6$

### 6.3.5. Conclusion

Nomad's group storage component was found to be responsive and reliable in for storage and retrieval operations under stable network conditions. Object retrieval in parallel and fast mode were found to be the most responsive, and just as reliable as safe storage. Fast storage has lower bandwidth requirements, but lacks the security and reliability that parallel and safe retrieval provide.

Object storage in fast mode was found to be more responsive and just as reliable as safe storage. Fast storage has similar bandwidth requirements to safe storage, as it sends multiple storage requests to peers (only reporting the first success). Even though it is less responsive and requires more bandwidth, safe storage provides better reliability under network churn and is more resilient to object manipulation.

## 6.4. Overlay Storage Evaluation

In order to evaluate Nomad's overlay storage component in isolation, a test API was implemented, namely the *OverlayStorageController*. This API provided an isolated interface for direct access to Nomad's overlay storage functionality.

### 6.4.1. Experimental Setup

The overlay storage test setup is described below.

| Test Configuration | |
|---|---|
| Network Size | 5 to 50 |
| Simulation Length | 1800 s |
| Request Rate | 100 Requests Per Second (RPS) |
| No. Requests | 150,000 requests (avg.) |
| Object Size | 1024 bytes |
| Overlay | TomP2P (Kademlia) |
| Scheduled Repair | Disabled |
| Leave Repair | Enabled |
| Group Migration | Disabled |
| Voronoi Grouping | Disabled |
| Request desc. | Exclusively out-of-group requests |

**Table 6.14:** Summary of overlay storage evaluation variables

#### 6.4.1.1. Size of the P2P Network

In order to test the overlay component's scalability, the network size was varied from 5 to 50 overlay peers, 50 being the maximum number of Nomad instances that could be created. For the various network sizes, the response time and success rate were measured for overlay storage operations.

#### 6.4.1.2. Iteration Length

Each experiment lasted between 1800 to 3600 seconds, generating on average 150,000 storage & retrieval requests. During storage tests, an average of 150,000 objects were generated. Up to three test iterations were executed for each scenario.

#### 6.4.1.3. Request Rate

A load test of 100 RPS was executed on each iteration, with a 60 second warm-up phase, which increased incrementally from 10 to 100 RPS. The request rate was kept stable at

100 RPS for the rest of the experiment.

## 6.4.2. Storage and Retrieval Results

The following subsection provides an overview of Nomad's overlay storage responsiveness and reliability.

### 6.4.2.1. Storage

Table 6.15 provides an overview of Nomad's overlay storage responsiveness and reliability for different network sizes. The experiment iterations were executed in the absence of network churn. Object storage, referred to as *storage operations*, was measured to have a *95th* percentile under 25 ms and a *99th* percentile under 35 ms, which is highly performant for DHT storage. In terms of reliability, TomP2P's *storage* operations were shown to be reliable, as no overlay request failed during any test iterations. Figure 6.15 provides an illustrated view of overlay storage *store* responsiveness and reliability as the number of nodes increases. When no network churn is present, TomP2P storage commands are seen to be performant, even more so than group storage operations.

| No. nodes | Min (ms) | Max (ms) | Median (ms) | p95 (ms) | p99 (ms) | Reliability (%) |
|---|---|---|---|---|---|---|
| 5 | 2.42 | 38.67 | 4.11 | 7.671 | 18.36 | 100.00 |
| 10 | 3.62 | 33.51 | 4.66 | 5.87 | 10.58 | 100.00 |
| 15 | 3.74 | 34.18 | 4.76 | 8.45 | 13.53 | 100.00 |
| 25 | 4.67 | 46.81 | 8.77 | 22.88 | 34.26 | 100.00 |
| 35 | 4.12 | 30.97 | 6.9 | 15.54 | 20.32 | 100.00 |
| 50 | 2.70 | 36.31 | 5.52 | 9.30 | 19.30 | 100.00 |

**Table 6.15:** Evaluation of TomP2P storage (put) responsiveness and reliability

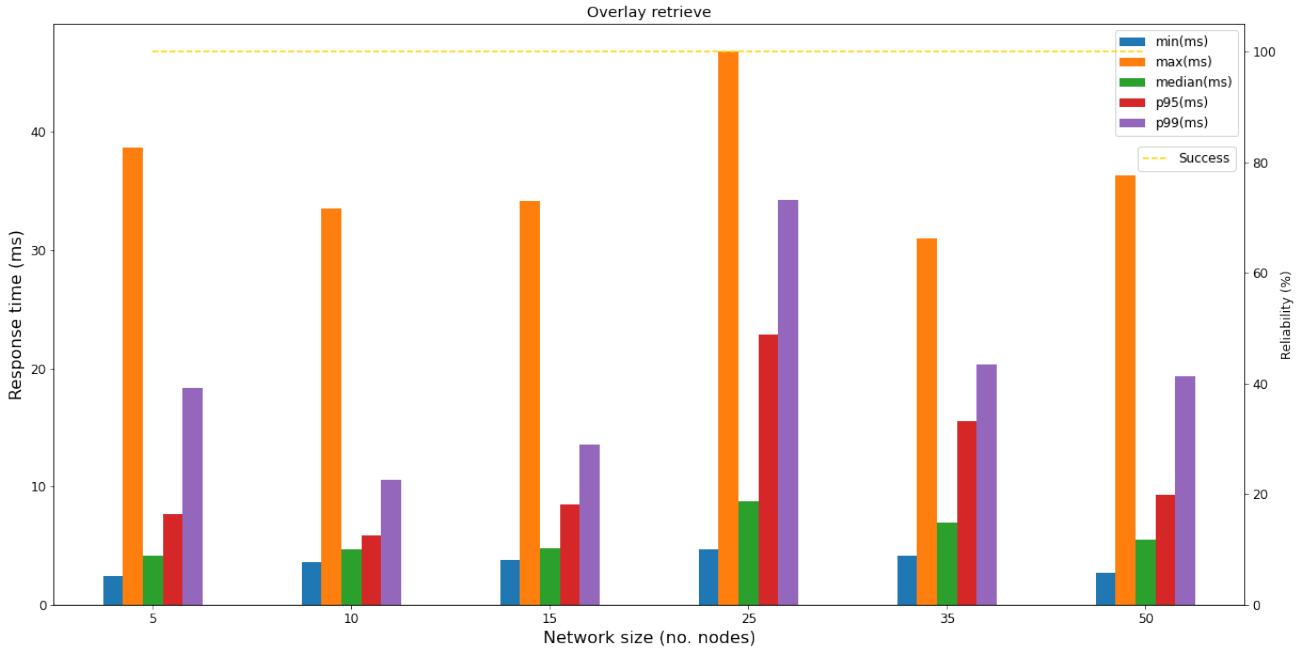


**Figure 6.15:** Evaluation of TomP2P storage (put) responsiveness and reliability as the network size increases

### 6.4.2.2. Retrieval

Table 6.16 provides an overview of Nomad's overlay retrieval responsiveness and reliability for different network sizes in the absence of network churn. Object retrieval, referred to as *retrieval operations*, was measured to have a *95th* percentile under 15 ms and a *99th* percentile under 45 ms. However, in terms of reliability, TomP2P's *retrieval* operations proved to be less reliable than *storage* operations. Figure 6.16 shows overlay storage *retrieval* responsiveness and reliability per network size. It is clear that as the size of the network increases, the probability of successful object retrieval decreases. The overall latency is seen to increase linearly until the network consists of more than 25 nodes, at which point response times drop and reliability stabilises at around 41%.

It can be concluded that Nomad's overlay is not reliable for object retrievals. As P2P MMVEs are known to have thousands of nodes, a success rate of 41.37% for 50 nodes is not sufficient. This suboptimal rate will result in unreliable out-of-group requests and will severely affect the reliability of state persistence.

| No. nodes | Min (ms) | Max (ms) | Median (ms) | p95 (ms) | p99 (ms) | Reliability (%) |
|---|---|---|---|---|---|---|
| 5 | 5.92 | 31.06 | 9.59 | 12.92 | 20.94 | 100.00 |
| 10 | 5.22 | 66.17 | 6.14 | 8.04 | 26.93 | 77.43 |
| 15 | 5.23 | 78.90 | 6.77 | 11.51 | 40.42 | 63.57 |
| 25 | 5.25 | 35.49 | 9.58 | 13.75 | 22.41 | 41.13 |
| 35 | 5.92 | 31.06 | 9.58 | 12.92 | 20.94 | 41.04 |
| 50 | 5.68 | 35.56 | 9.63 | 14.03 | 23.50 | 41.37 |

**Table 6.16:** Evaluation of TomP2P retrieval (get) responsiveness and reliability

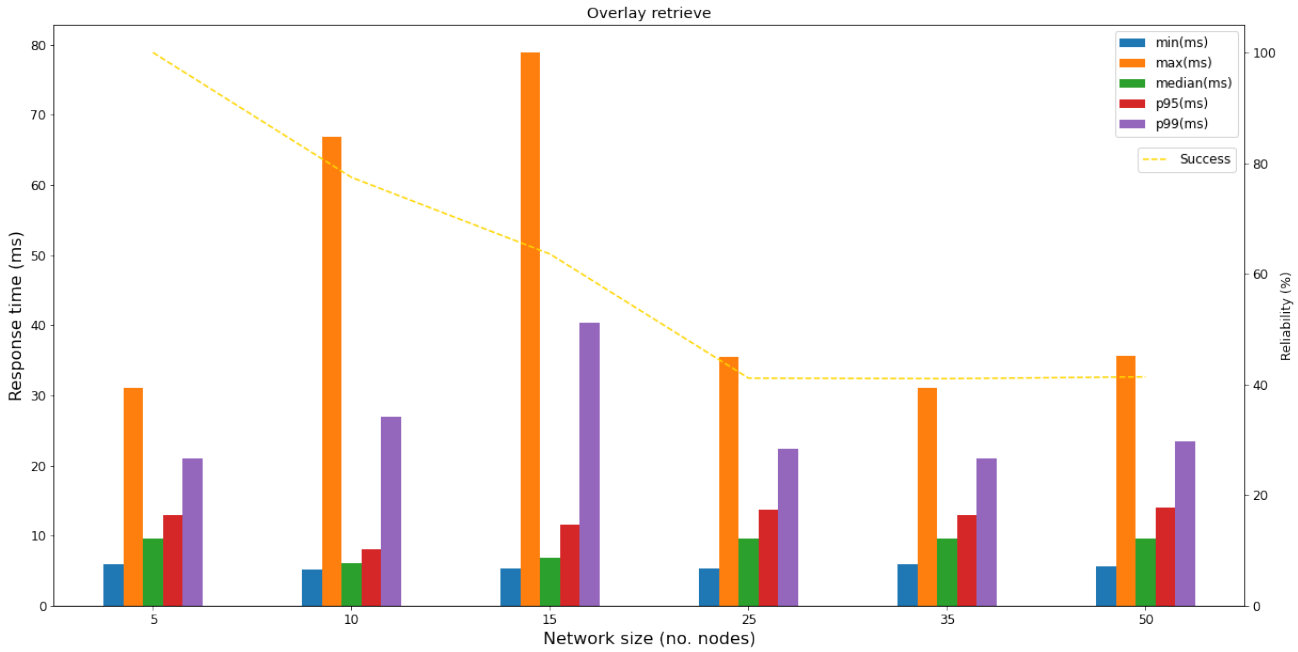


**Figure 6.16:** Evaluation of TomP2P retrieval (get) responsiveness and reliability as network size increases

### 6.4.3. Storage and Retrieval Results with Network Churn

To evaluate Nomad's overlay storage performance under network churn, the same configuration was used as in table 6.14. At the beginning of each test iteration, 50 Nomad peers were created, then at 1 to 2 minute intervals, a peer instance was either added or removed. For peer removal, random peers were selected and shut down, either gracefully or forcefully.

| Operation | Min (ms) | Max (ms) | Median (ms) | p95 (ms) | p99 (ms) | Reliability (%) |
|---|---|---|---|---|---|---|
| Store | 8.33 | 1704.05 | 76.86 | 1630.80 | 1701.25 | 99.99 |
| Retrieve | 15.08 | 3080.60 | 86.80 | 2949.74 | 3044.37 | 47.40 |

**Table 6.17:** Evaluation of TomP2P retrieval (get) operation's responsiveness and reliability as network size increases under heavy network churn

Table 6.17 shows TomP2P's responsiveness and reliability under network churn. TomP2P's performance was seen to decrease drastically under network churn. Object storage response times were measured to have a *95th* percentile of 1.6 seconds and a *99th* percentile of 1.7 seconds. Object retrieval response times were measured to have a *95th* percentile of 2.9 seconds and a *99th* percentile of 3.0 seconds.

Reliability for *put* operations proved to be reliable, although decreasing slightly to 99.99% due to nodes that received a storage request leaving the network. Reliability for *get* operations under network churn improved slightly from 41.37% to 47.40% under churn. This can be attributed to TomP2P's use of dynamic replication factor, which increases under network churn in an attempt to improve reliability, but at the cost of responsiveness. This should be investigated in more detail, since TomP2P lacks documentation.

## 6.4.4. Bandwidth Requirements

In order to compare Nomad's overlay performance to that of Pithos, bandwidth requirements have to be taken into account. The following section provides an overview of TomP2P's bandwidth usage for two request rates, namely 0.2 RPS and 10 RPS per peer, and network sizes 5 to 50. The overlay's replication factor is not taken into consideration, since it is not configurable. A replication factor of 1 is assumed.

### 6.4.4.1. Bandwidth Usage

Table 6.18 shows Nomad's overlay bandwidth usage for a request rate of 0.2 RPS with no network churn. Figures 6.17 and 6.18 illustrate a linear relationship between bandwidth and network size. It can be seen that regardless of request rate, as the size of the network increases, the bandwidth requirements increase. This is mainly due to the additional routing required within the overlay network, caused by request reroutes and internal maintenance mechanisms.

| Bandwidth - RPS 0.2 per peer | | |
|---|---|---|
| No. nodes | Operation | Bandwidth (Bps) |
| 5 | store | 1175 |
| | retrieve | 1075 |
| 10 | store | 1450 |
| | retrieve | 1280 |
| 15 | store | 2050 |
| | retrieve | 1850 |
| 25 | store | 2800 |
| | retrieve | 2500 |
| 35 | store | 3250 |
| | retrieve | 2800 |
| 50 | store | 4500 |
| | retrieve | 4000 |

**Table 6.18:** Overlay storage observed bandwidth usage - 0.2 RPS per peer, $\mathcal{R} = 6$

| Bandwidth - RPS 10 per peer | | |
|---|---|---|
| No. nodes | Operation | Total BW (Bps) |
| 5 | store | 34,500 |
| | retrieve | 21,000 |
| 10 | store | 59,500 |
| | retrieve | 51,400 |
| 15 | store | 80,500 |
| | retrieve | 60,000 |
| 25 | store | 170,000 |
| | retrieve | 115,000 |
| 35 | store | 153,500 |
| | retrieve | 116,000 |
| 50 | store | 210,000 |
| | retrieve | 202,000 |

**Table 6.19:** Overlay storage observed bandwidth usage - 10 RPS per peer, $\mathcal{R} = 6$

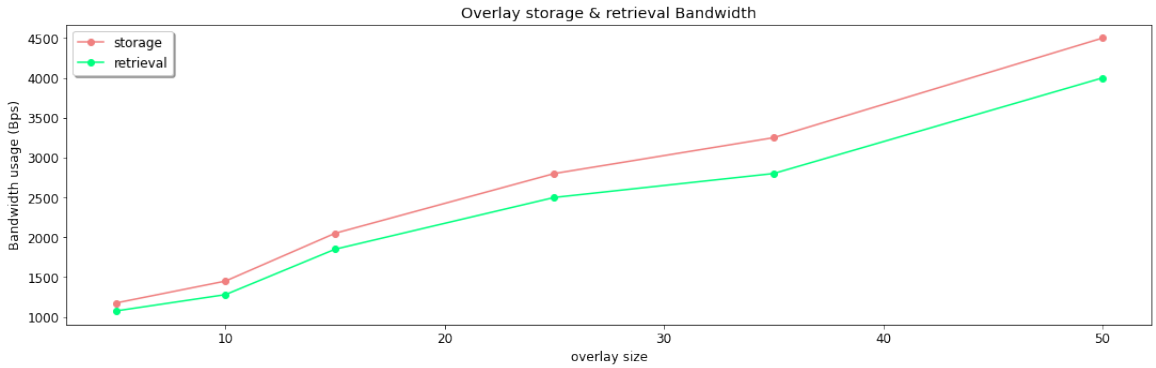


**Figure 6.17:** Overlay storage observed bandwidth usage - 0.2 RPS per peer

Table 6.19 shows Nomad's overlay bandwidth usage when the request rate is increased by a factor of 50. TomP2P bandwidth requirement increases by an average factor of 41, which is lower than expected. This can be attributed to inaccurate bandwidth measurements, since Wireshark was only used to provide an estimate of the bandwidth usage results, and not for fine grain precision.

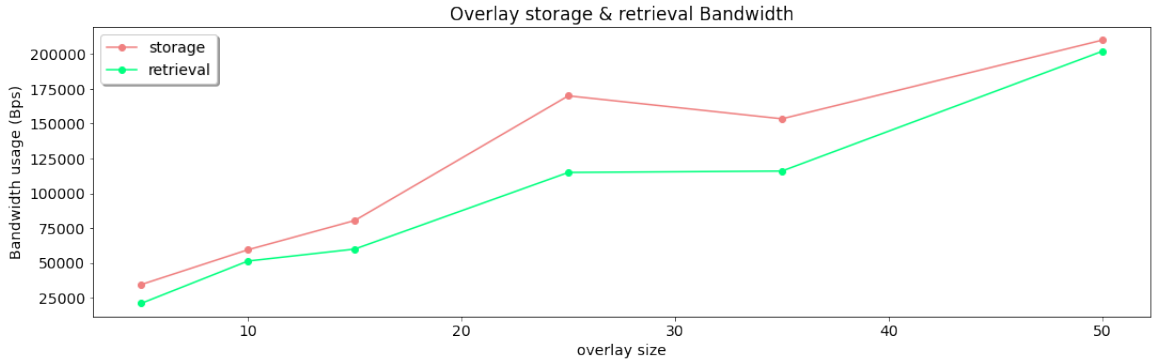


**Figure 6.18:** Overlay storage observed bandwidth usage - 10 RPS per peer

### 6.4.5. Conclusion

Nomad's overlay storage component, TomP2P, was found to be responsive and reliable in performing overlay operations under stable network conditions. Object storage was found to be particularly responsive and reliable. Object retrieval was also seen to be responsive, but not as reliable as the size of the network increased, with a reliability of 41.37% for 50 peers.

When network churn was introduced, the overlay became less reliable and far less responsive. Responsiveness of storage and retrieval operations increased to 1.6 and 2.9 seconds respectively. Retrieval success rate was seen to improve under network churn, with a reliability of 47.40%, which may be explained by internal mechanisms that ensure remapping and replication of objects within TomP2P. However, this needs to be investigated.

Compared to group storage, TomP2P has a large bandwidth requirement, requiring up to 4500 Bps to function under low load, and 202,000 Bps for high load environments. The measured results for storage requests were seen to have a slightly higher bandwidth requirement, compared to retrieval requests.

TomP2P's poor retrieval responsiveness and reliability will cause issues within Nomad. An inefficient overlay will lead to unreliable out-of-group requests and will cause overall state persistence to be unreliable, since the overlay is also used as a persistence layer. Nomad's overlay storage was therefore found to be one of the system's weaknesses and requires further investigation and development.

## 6.5. System Evaluation

Nomad's overall responsiveness, reliability and security were evaluated with respect to four scenarios, (1) stable network conditions, (2) in the presence of network churn, (3) in the presence of malicious peers and lastly (4) with group migration and Voronoi grouping enabled.

In order to generate load and evaluate system functionality, requests were made to Nomad's client storage API. As mentioned in Sections 5.3.2 and B.2.7, the *PeerStorage-Controller* provides the client interface for performing storage operations on the Nomad decentralised storage network.

### 6.5.1. Performance Under Stable Network Conditions

This section provides an overview of Nomad's baseline performance metrics, measured under stable network conditions.

| Test Configuration | |
|---|---|
| Group Size | 5 (4 peers, 1 super-peer) |
| Network Size | 50 (40 peers, 10 super-peer) |
| Simulation Length | 3600 s |
| Request Rate | 100 Requests Per Second (RPS) |
| No. Requests | 350,000 requests (avg.) |
| No. Generated Objects | 330,000 (avg.) |
| Object Size | 1024 bytes |
| Overlay | TomP2P (Kademlia) |
| $\mathcal{R}$ | 3 replicas |
| Storage Modes | Fast, Safe |
| Retrieval Modes | Fast, Parallel, Safe |
| Modify Modes | Safe |
| Scheduled Repair | Disabled |
| Leave Repair | Enabled |
| Group Migration | Disabled |
| Voronoi Grouping | Disabled |
| Request Description | Exclusively in-group requests |
| Network Churn | On |

**Table 6.20:** Summary of group storage evaluation variables for stable network experiment

#### 6.5.1.1. Size of the P2P Network

To evaluate Nomad's overall performance, the test infrastructure discussed in section 6.2.1 was used to run as many Nomad instances as possible. With the available resources, 50 Nomad instances could be created.

#### 6.5.1.2. Iteration Length

10 test iterations were executed in order to ensure deterministic results. Each test iteration was run between 1800 to 3600 seconds, generating on average 350,000 storage and retrieval requests. During storage tests, an average of 330,000 objects were generated.

#### 6.5.1.3. Request Rate

A load test of 100 RPS distributed over the entire network was executed in each iteration, using a 60 second warm-up phase, increased incrementally from 10 to 100 RPS. The request rate was kept stable at 100 RPS for the rest of the experiment.

#### 6.5.1.4. Storage, Retrieval and Modification Results

| Mode | Min (ms) | Max (ms) | Median (ms) | p95 (ms) | p99 (ms) | Reliability (%) |
|---|---|---|---|---|---|---|
| Fast Retrieval | 1.41 | 359.88 | 4.65 | 7.00 | 29.34 | 100.00 |
| Parallel Retrieval | 1.23 | 473.00 | 3.83 | 6.94 | 177.11 | 99.89 |
| Safe Retrieval | 1.22 | 817.77 | 4.03 | 18.81 | 123.23 | 99.99 |
| Fast Storage | 6.00 | 667.50 | 11.50 | 26.50 | 81.94 | 100.00 |
| Safe Storage | 11.50 | 980.60 | 18.40 | 36.70 | 111.98 | 99.99 |
| Safe Modify | 8.84 | 720.71 | 13.74 | 69.57 | 218.67 | 100.00 |

**Table 6.21:** System retrieval results

Table 6.21 shows Nomad's overall responsiveness for each storage, retrieval and modify mode. For the purpose of this discussion, the 95th percentile of response times is used. *Parallel retrieval* proved to be the most responsive, with an average response time of 6.94 ms, with *fast retrieval* achieving similar response times of 7.00 ms. *Safe retrieval* produced the highest response times, since it has to wait for all requests to complete. Fast retrieval proved to be slightly more reliable than safe and parallel retrieval under stable conditions, although it is expected to be less reliable under churn.

*Fast storage* proved to be slightly more responsive than *safe storage*. With response times under 30 ms compared to 40 ms, safe storage (99.99%) is inherently more reliable, regardless of fast storage reporting 100% reliability.

Object modification was not evaluated during Pithos' simulation, as it was deemed future work. As part of Nomad's evaluation, modification response times and reliability were measured, in order to provide metrics for Pithos modification operations. Since modify and safe storage operations are nearly identical in implementation, the two operations could be compared. *Safe modify* proved to have higher response times compared to safe storage, with a response time of 69.57 ms. However, modification operations proved to be more reliable, with a success rate of 100%.

### 6.5.2. Long Running Test Results

As part of Nomad's evaluation, long running performance tests, spanning 7740 seconds and generating more than 790,000 requests, were executed. The same application configuration mentioned in table 6.20 was used. Table 6.22 provides a summary of the updated experiment configuration.

| Test Configuration | |
|---|---|
| Simulation Length | 7740 s |
| No. Requests | 790,000 requests (avg.) |
| No. Generated Objects | 400,000 (avg.) |
| Network Churn | Off |

**Table 6.22:** Summary of group storage evaluation variables used for long running tests

#### 6.5.2.1. Storage, Retrieval and Modify Results

Previous experiments focussed on single storage operations, as only one operation type was executed for the entirety of the experiment. In contrast, the higher layer application (Artillery) was now made to generate random store, retrieve or update requests to peers, in order to simulate normal network traffic. A request rate of 100 RPS was used.

| Operation | Min (ms) | Max (ms) | Median (ms) | p95 (ms) | p99 (ms) | Reliability (%) |
|---|---|---|---|---|---|---|
| Fast Retrieval | 2.90 | 30.84 | 6.62 | 14.77 | 26.65 | 100.00 |
| Parallel Retrieval | 3.09 | 31.38 | 5.09 | 12.74 | 26.57 | 100.00 |
| Safe Retrieval | 3.07 | 44.65 | 5.01 | 17.39 | 36.83 | 100.00 |
| Safe Storage | 7.10 | 980.6 | 18.4 | 36.7 | 111.98 | 100.00 |
| Fast Storage | 7.10 | 57.03 | 11.64 | 24.1 | 38.21 | 100.00 |
| Safe Modify | 8.56 | 61.90 | 13.09 | 30.02 | 42.72 | 100.00 |

**Table 6.23:** Long running system evaluation results, using a request rate of 100 RPS distributed over the network of 50 peers

Table 6.23 shows all operation response times and reliability. Nomad was able to maintain its reliability and responsiveness during long running load tests. Compared to the results seen in Table 6.21, response times were slightly higher, but also slightly more reliable.

### 6.5.3. Performance Under Network Churn

The following subsection provides an overview of Nomad's overall responsiveness and reliability in the presence of network churn. The same application configuration as mentioned in Table 6.20 was used during experiment iterations.

To evaluate Nomad's performance under network churn, at the beginning of each test iteration, 50 Nomad peers were created, then at 1 to 2 minute intervals, an instance was either added or removed. For peer removal, random instances were selected and shut down, either gracefully or forcefully. Both peers and super-peers were affected by network churn, in order to test both peer and super-peer leave scenarios.

#### 6.5.3.1. Storage, Retrieval and Modify Results

| Operation | Min (ms) | Max (ms) | Median (ms) | p95 (ms) | p99 (ms) | Reliability (%) |
|---|---|---|---|---|---|---|
| Fast Retrieval | 3.43 | 42.47 | 4.36 | 17.01 | 42.25 | 100.00 |
| Parallel Retrieval | 3.74 | 75.68 | 5.08 | 19.28 | 74.59 | 100.00 |
| Safe Retrieval | 3.79 | 415.53 | 5.1 | 254.05 | 414.03 | 100.00 |
| Fast Storage | 8.08 | 93.41 | 10.14 | 27.09 | 92.38 | 100.00 |
| Safe Storage | 223.69 | 1329.79 | 673.99 | 1231.37 | 1328.32 | 83.30 |
| Safe Modify | 9.54 | 1426.29 | 28.174 | 736.60 | 1225.5 | 99.29 |

**Table 6.24:** System retrieval results under network churn

Table 6.24 shows Nomad's storage and retrieval results under network churn. For the purpose of this discussion, the 95th percentile of response times was used. In terms of object retrieval, fast and parallel retrieval proved to be responsive, both producing response times of under 20 ms, with safe retrieval providing response times of over 250 ms. All retrieval modes were shown to provide a reliability of 100%.

In terms of object storage, fast storage performed well, with a response time of under 30 ms and good reliability. Safe storage experienced a drop in performance due to network churn, with response times over 1.2 seconds and reliability falling to 83.30%. Similarly, modify requests proved to have reduced reliability of 99.29% and much higher response times of 736.60 ms.

The drop in performance for safe retrieval, safe modify and safe storage was primarily caused by Nomad's overlay component, which has proved to be unreliable and unresponsive under high network churn (6.4.3).

### 6.5.4. Security

This section provides an overview of Nomad's security against malicious peers.

#### 6.5.4.1. Experimental Setup

To evaluate Nomad's performance in the presence of malicious peers, the **malicious** test property was introduced. This property allows for the introduction of malicious peers to

the system, without any intrusive code.

```
node:
  # Used for testing only
  malicious: false
```

**Listing 6.4:** Application configuration to enable malicious peers

When the malicious property is set to *true* on a peer, the peer will maliciously alter objects before responding to group storage requests. Object alterations are executed right after extracting the object from the database by replacing object values with a new random 1024 byte payload and a static, recognisable malicious ID.

In order to measure all occurrences of successful object manipulation, i.e. every time an object that was manipulated is returned to the higher layer, a special HTTP response code (**418**) was used to inform the higher layer that an altered object was returned. The static malicious ID was used to identify altered objects.

To measure the effect of malicious peers on retrieval security, malicious peers were incrementally added to the system, starting with 0% malicious peers, until all peers in the network were malicious. In reality, a malicious peer count of above 20% would be unrealistic.

Table 6.25 provides a summary of the important configuration settings. The rest of the configuration settings were identical to those seen in Table 6.20.

| Test Configuration | |
|---|---|
| Group Size | 5 (4 peers, 1 super-peer) |
| Network Size | 50 (40 peers, 10 super-peer) |
| Simulation Length | 3600 s |
| Request Rate | 100 Requests Per Second (RPS) |
| No. Requests | 350,000 requests (avg.) |
| $\mathcal{R}$ | 4 replicas |
| Retrieval Modes | Fast, Parallel, Safe, Overlay |
| Network Churn | Off |

**Table 6.25:** Summary of important Nomad security evaluation variables

#### 6.5.4.2. Security Results

Table 6.26 shows each retrieval mode's security, versus the number of malicious peers in the network. *Security* in this context refers to the probability that a maliciously altered object is returned to the higher layer. As expected, *fast* retrieval's security is proportional to the percentage of malicious peers in the network, since it implements no object validation

mechanisms. *Parallel* retrieval, which also does not implement any object validation, however provides slightly better security, as a result of parallel requests. *Safe* retrieval provides the best security as a result of using a quorum mechanism, which requires each response to have an identical payload, consisting of an ID, creation time, modification time, TTL and value.

| Malicious Peers (%) | Retrieval Mode | Security (%) |
|---|---|---|
| 0 | fast | 100.00 |
| | parallel | 100.00 |
| | safe | 100.00 |
| 10 | fast | 90.00 |
| | parallel | 95.00 |
| | safe | 99.00 |
| 25 | fast | 77.00 |
| | parallel | 79.00 |
| | safe | 81.00 |
| 50 | fast | 45.00 |
| | parallel | 50.00 |
| | safe | 52.00 |
| 75 | fast | 21.00 |
| | parallel | 23.00 |
| | safe | 26.00 |
| 100 | fast | 0.00 |
| | parallel | 0.00 |
| | safe | 0.00 |

**Table 6.26:** Nomad's security against malicious peers for different retrieval modes

It should be noted when using a group size of 5 and replication factor of 4, the quorum mechanism requires $(4/2)+1 = 2$ replicas. Each group consists of 4 peers and 1 super-peer, which does not participate in group storage. This means that if more than 2 peers in the group are malicious, Nomad's quorum mechanism will not be able to provide secure retrieval.

### 6.5.5. Voronoi Grouping and Group Migration Enabled

This section provides Nomad's performance results within a virtual environment, under network churn, where peers receive positional updates and migrate from one group to another. In order to induce a load on the system, the same higher layer application, Artillery, was used to generate storage and retrieval requests.

#### 6.5.5.1. Experimental Setup

In order to enable group migration and Voronoi grouping, two application configuration settings were used,

```
node:
  group:
    migration: true
    voronoiGrouping: true
```

**Listing 6.5:** Application configuration to enable Voronoi grouping and migration

Table 6.27 contains the various configuration settings (test variables) used during the system evaluation test iterations.

| Test Configuration | |
|---|---|
| Group Size | 25 (24 peers, 1 super-peer) |
| Network Size | 50 (40 peers, 10 super-peer) |
| Simulation Length | 3600 s |
| Request Rate | 100 Requests Per Second (RPS) |
| No. Requests | 350,000 requests (avg.) |
| No. Generated Objects | 330,000 (avg.) |
| Object Size | 1024 bytes |
| Overlay | TomP2P (Kademlia) |
| $\mathcal{R}$ | 6 replicas |
| Storage Modes | Fast |
| Retrieval Modes | Fast |
| Scheduled Repair | Disabled |
| Leave Repair | Enabled |
| Group Migration | Enabled |
| Voronoi Grouping | Enabled |
| Request Description | in-group & out-of-group |
| Network Churn | On |

**Table 6.27:** Summary of group storage evaluation variables for Voronoi grouping experiment

#### 6.5.5.2. The Test Virtual Environment

For the purpose of this experiment, the geography of the map is not important. The application merely requires $x$ and $y$ limits. The map limits ensure that all player movements are restricted to the virtual plane. During the experiment, $x$ and $y$ limits of 10.0 were used for the virtual world. Figure 6.19 illustrates the virtual world, segmented into Nomad

groups, using a Voronoi grouping mechanism. The image was generated by super-imposing the Voronoi map generated during one of the test iteration onto the fortnite map [68]. The visualisation of the Voronoi map was generated using an online tool [67]. Black dots represent super-peer positions, which act as Voronoi site points.

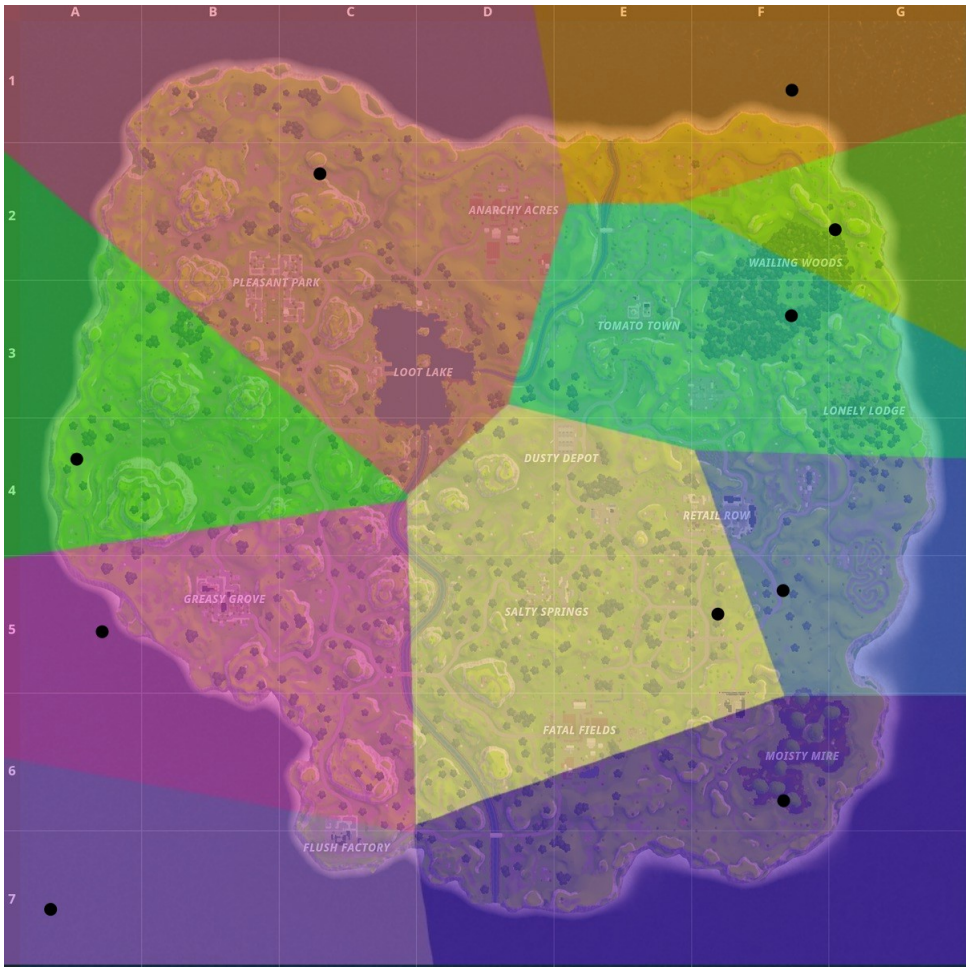


**Figure 6.19:** Example of Voronoi divided map and super-peer positions

#### 6.5.5.3. Player Movement Model

In order to simulate realistic player movement within the test virtual world, a player movement model was required. We made use of recent work by Moll et al. [91], who present extrapolated player movement patterns and network traffic, based on analysis of a popular battle royal game, *Fortnite* [92].

The FortniteTraces [93] dataset was used as the player movement model for each Nomad storage peer. Voronoi-grouping split the virtual world into geographical segments. This facilitated group migration of a peers, enabling their movement across multiple Voronoi segments within the virtual world. Movement traces were contained within the $x$ and $y$ map limits mentioned earlier.

Each Nomad instance was started with a movement file provided by the FortniteTraces dataset. The movement files provided every peer with positional updates , i.e. x, y

coordinates, every 10 seconds, which represented the peer's movement within the virtual world.

#### 6.5.5.4. Network Churn

At the beginning of each test iteration, 50 Nomad peers were created, then at 1 to 2 minute intervals, an instance was either added or removed, representing a game session ending or starting. For peer removal, random instances were selected and shut down, either gracefully or forcefully. Both peers and super-peers were affected by network churn, in order to test both peer and super-peer leave scenarios. The network size was kept above 45 peers.

#### 6.5.5.5. Iteration Length

Two test iterations were performed. Each ran for 3680 seconds, generating on average 350,000 storage & retrieval requests. During storage tests, an average of 330,000 objects were generated.

#### 6.5.5.6. Request Rate

A load test of 100 RPS distributed over the entire network was executed in each iteration, using a 60 seconds warm-up phase, increased incrementally from 10 to 100 RPS. The request rate was kept stable at 100 RPS for the rest of the experiment. A differentiating factor of this experiment was that object requests were made to random peers in random groups, therefore generating both *in-group* and *out-of-group* requests.

### 6.5.6. Results

| Operation | Min (ms) | Max (ms) | Median (ms) | p95 (ms) | p99 (ms) | Reliability |
|---|---|---|---|---|---|---|
| Fast Storage | 4.75 | 950.38 | 10.038 | 215.25 | 356.68 | 100.00 % |
| Fast Retrieval | 6.17 | 246.60 | 13.8439 | 204.48 | 246.60 | 59.00 % |

**Table 6.28:** Performance results for Nomad in fast storage and retrieval mode under network churn, using Voronoi grouping, with $\mathcal{R} = 6$

Table 6.28 shows the measured results for *fast* storage and retrieval under heavy network churn with Voronoi grouping and group migration enabled. For the purpose of this discussion, the 95th percentile of response times is used. Object storage and retrieval in fast mode provided response times of 215 ms and 205 ms respectively.

Comparing the results to response times produced by the churn tests from section 6.5.3, response times are much higher and less reliable. The drop in performance can be

explained by the ratio of in-group versus out-of-group requests. In previous experiments, group objects were ensured to be present within a group, simulating normal usage of the Nomad system. During this experiment, no such guarantees could be made, as peers could move freely between groups. This resulted in more out-of-group calls than normally expected within the system. Further investigation showed that up to 70% of requests may have been out-of-group.

The 59% overall reliability for object retrieval is not surprising, since overlay object retrieval has a reliability of between 41.37% to 47.40% under network churn. Fast object storage proved to be perfectly reliable under network churn with group migration enabled.

## 6.6. Bandwidth Requirements

In order to compare Nomad's overall performance to that of Pithos, bandwidth requirements have to be taken into account. Nomad's bandwidth usage was measured for two request rates, namely 0.2 RPS and 10 RPS per peer, with a network size of 50. Table 6.29 provides the application configuration used for testing.

| Test Configuration | |
|---|---|
| Group Size | 5 (24 peers, 1 super-peer) |
| Network Size | 50 (40 peers, 10 super-peer) |
| Simulation Length | 3600 s |
| Object Size | 1024 bytes |
| Overlay | TomP2P (Kademlia) |
| $\mathcal{R}$ | 6 replicas |
| Storage Modes | Fast, Safe |
| Retrieval Modes | Fast, Parallel, Safe |
| Modify Modes | Safe |
| Group Migration | Disabled |
| Voronoi Grouping | Disabled |
| Request Description | in-group |
| Network Churn | Off |

**Table 6.29:** Summary of group storage evaluation variables

### 6.6.1. Total Bandwidth Usage

Tables 6.30 and 6.31 show Nomad's overall bandwidth usage for a load of 0.2 RPS and 10 RPS per peer. When comparing these bandwidth measurements to those in tables 6.12, 6.13, 6.18 and 6.19, these results are similar.

At 0.2 RPS, a slightly lower bandwidth usage was seen for all operations compared to group bandwidth testing, but at 10 RPS the results once again look similar. This may mean that at low request rates, the combination of group storage and overlay storage slightly decreases group storage bandwidth usage, since some requests are handled by the overlay. However, since bandwidth measurements using Wireshark are less accurate, more investigation is required.

| Bandwidth - RPS 0.2 per peer | | | | |
|---|---|---|---|---|
| No. peers | Mode | Group BW (Bps) | Overlay BW (Bps) | Avg total (Bps) |
| 50 | Fast Storage | 2000 | 4400 | 6400 |
| | Safe Storage | 2000 | 4500 | 6500 |
| | Fast Retrieval | 500 | 4000 | 4500 |
| | Parallel Retrieval | 1250 | 4000 | 5250 |
| | Safe Retrieval | 2000 | 4000 | 6000 |
| | Safe Modify | 1600 | 4500 | 6100 |

**Table 6.30:** Nomad bandwidth requirements for 50 peers with a group size of 5, under a load of 0.2 RPS per peer

| Bandwidth - RPS 10 per peer | | | | |
|---|---|---|---|---|
| No. peers | Mode | Group BW (KBps) | Overlay BW (KBps) | Avg total (KBps) |
| 50 | Fast Storage | 30 | 210 | 240 |
| | Safe Storage | 47 | 210 | 257 |
| | Fast Retrieval | 17.5 | 202 | 219.5 |
| | Parallel Retreival | 45 | 202 | 249 |
| | Safe Retrieval | 47 | 202 | 247 |
| | Safe Modify | 47 | 170 | 217 |

**Table 6.31:** Nomad bandwidth requirements for 50 peers with a group size of 5, under a load of 10 RPS per peer

In Table 6.30, it can be seen that the majority of Nomad's bandwidth usage is attributed to its overlay storage component, which on average uses almost 2.45 times more bandwidth than group storage. Figure 6.20 illustrates Nomad's bandwidth usage per storage component under a 0.2 RPS load per peer.

Nomad's storage operations have similar bandwidth footprints under low system load, with an overall bandwidth requirement of around 6.5 KBps. Retrieval bandwidth requirements are heavily dependent on the retrieval mode. Nomad in fast retrieval mode does not require a large amount of bandwidth, with a total requirement of 4.5 KBps. In parallel and safe retrieval mode, Nomad's bandwidth requirement increases to 5.25 KBps and 6 KBps respectively, due to a higher number of group requests. Modification requests

use slightly less bandwidth than safe retrieval, with a measured bandwidth usage of 6.1 KBps.

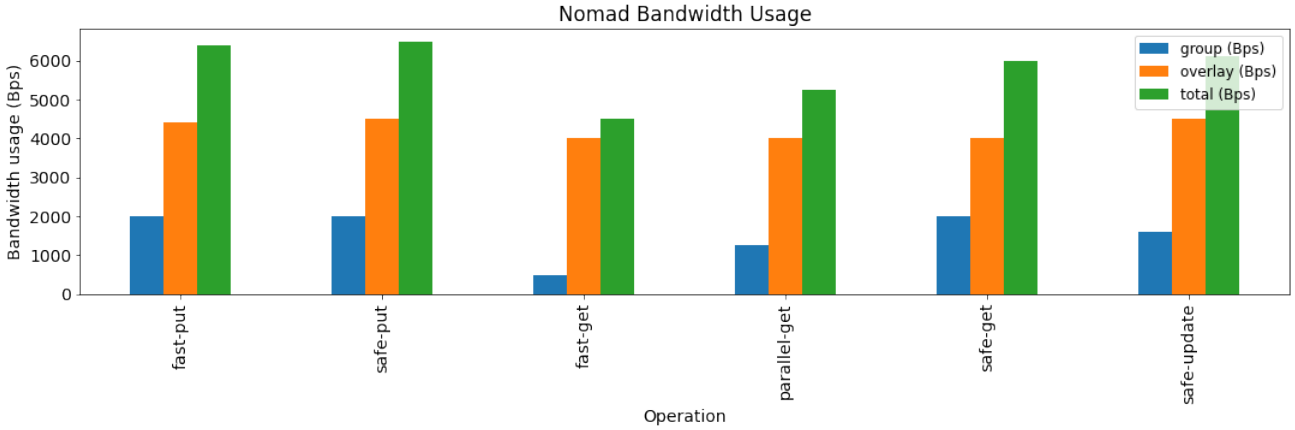


**Figure 6.20:** Overview of bandwidth requirements per storage and retrieval mode under a load of 0.2 RPS

Increasing the request rate by a factor of 50 increases the bandwidth usage by an average factor of 41 - as previously mentioned, these bandwidth measurements are merely used to provide an estimate and not for high accuracy. From Table 6.31 the majority of Nomad's bandwidth usage is attributed to the overlay storage component, which on average uses 7 times more bandwidth than group storage. This increase in the bandwidth usage to request rate ratio is mainly caused by the discrepancy between these rates for group and for overlay. Figure 6.21 illustrates Nomad's bandwidth usage per storage component under a load of 10 RPS per peer.

Storage operations have similar bandwidth footprints under high system load. Nomad in fast storage mode requires slightly less bandwidth than when it is in safe storage mode, with an overall bandwidth requirement of 240 KBps compared to 257 KBps. Nomad in fast retrieval mode does not require a large amount of bandwidth, with a total requirement of 219.5 KBps. In parallel and safe retrieval mode, Nomad's bandwidth requirement increases due to a higher number of group requests. Parallel retrieval requires a total of 245 KBps, whereas safe retrieval requires 249 KBps.

Object modification requests required the lowest amount of bandwidth amongst all storage operations under high load, with a total bandwidth requirement of 217 KBps. This is likely due to the overlay storage being more bandwidth efficient during modify requests. This observation was only made at high request rates, and might be due to adaptive routing within TomP2P. More investigation is required.

Figure 6.21 provides a visualisation of the data in Table 6.31.

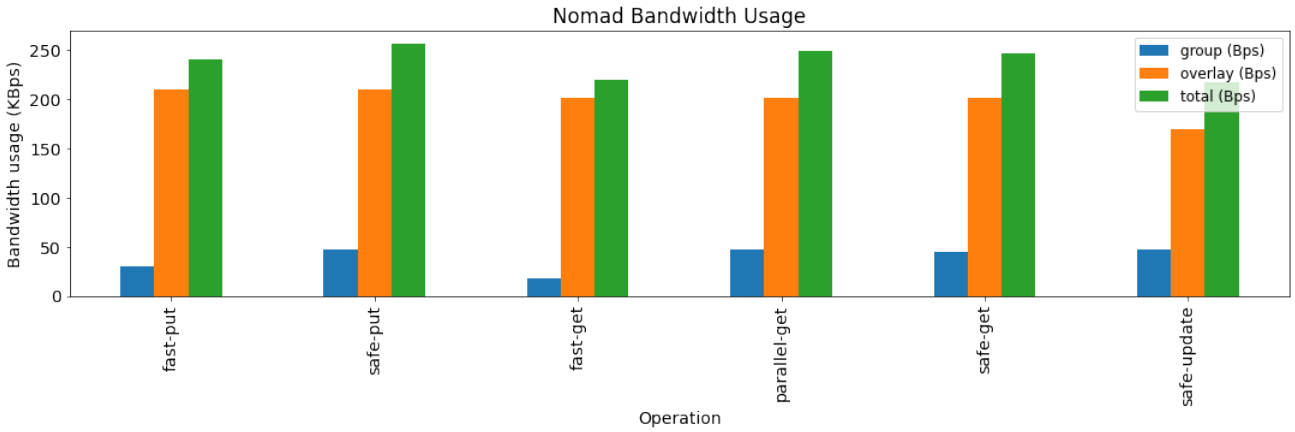


**Figure 6.21:** Overview of bandwidth requirements per storage and retrieval mode under a load of 10 RPS per peer

#### 6.6.1.1. Conclusion

Nomad's overall bandwidth requirement is high. For a network of 50 peers receiving a load of 10 RPS per peer, Nomad requires up to 250 KBps for storing 1024 bytes. Nomad's overlay storage component appears to be bandwidth inefficient and is the biggest cause of Nomad's large bandwidth requirement.

Fast retrieval is far more bandwidth efficient than both safe and parallel retrieval, since only one request is sent to a peer containing the requested object. The additional bandwidth requirement of parallel and safe storage does, however, bring with it to the security and responsiveness of retrieval requests.

Fast and safe storage were found to have very similar bandwidth footprints. In contrast to fast retrieval, fast storage sends multiple storage requests, determined by the replication factor. This means that its bandwidth requirement is similar to that of safe storage, although as it only waits for the first successful response, its bandwidth requirement for inbound traffic is slightly lower.

Modify requests were found to be the least bandwidth intensive under high load, presumably because of more efficient overlay routing, since safe modify and safe storage requests have identical group storage bandwidth requirements.

## 6.7. Nomad versus Pithos

Previous sections in this chapter discussed the various experiments performed on Nomad, in order to evaluate the system's reliability, responsiveness and security. This section provides a comparison between the Pithos simulation results and the implemented system Nomad. The 95th percentile of both systems' evaluation results was used to make comparisons.

It is important to note that Pithos was evaluated in a simulated environment, which generally provides an optimistic view in terms of performance and reliability. As mentioned

before, this study provides verification of the Pithos results, in order to determine whether the Pithos architecture is in fact suitable for P2P MMVE storage.

In order to provide a detailed comparison between the Pithos simulation and the Nomad implementation, each storage component will now be compared individually before comparing system performance. Table 6.32 lists the system configurations used for comparisons.

| Param. | Nomad Configuration | Pithos Configuration |
|---|---|---|
| Group Size | 25 (24 peers, 1 super-peer) | 25 (24 peers, 1 super-peer) |
| Network Size | 50 (40 peers, 10 super-peer) | 2,600 (2500 peers, 100 super-peer) |
| Simulation Length | 3600 s | 10,000 s |
| No. Requests | 350,000 (avg.) | 5,000,000 (avg.) |
| No. Generated Objects | 330,000 (avg.) | 600,000 (avg.) |
| Object Size | 1024 bytes | 1024 bytes |
| Overlay | TomP2P (Kademlia) | Chord (medium) |
| $\mathcal{R}$ | 6 replicas | 6 replicas |
| Network Churn | On | On |

**Table 6.32:** Summary of group storage evaluation variables for Nomad vs Pithos

## 6.7.1. Group Storage Comparison

The group storage component is the most important storage component of the Pithos architecture, since the majority of requests are expected to be served from within the group.

In order to reliably compare both systems' group storage component, identical configuration parameters were used. Group sizes were limited to 25 peers per group and a replication factor of 6 was used. Both systems were tested under network churn. Since group storage performance relies mostly on group size and replication factor, the overall size of the network was not considered important.

It should be noted that Pithos peer connections were limited to a channel width of 10 Mbps, whereas Nomad peer connections were limited to 1000 Mbps. This bandwidth limitation on Pithos peer connections should be taken into account when making comparisons.

The following subsections provide a comparison of both systems' storage and retrieval performance.

### 6.7.1.1. Storage

Table 6.33 provides a comparison between Nomad's and Pithos' storage responsiveness for different storage modes.

| System | Mode | Reliability (%) | Responsiveness (ms) |
|---|---|---|---|
| Pithos | Safe Storage | 97.05 | 1554.00 |
| Nomad | | 83.30 | 1231.00 |
| Pithos | Fast Storage | 100 | 6.65 |
| Nomad | | 100 | 27.09 |

**Table 6.33:** Group storage - storage operations comparison

From Nomad's evaluation results, it was found that when safe storage is used, a reliability of 83.3% is achieved. Compared to Pithos' safe storage evaluation, with a reliability of 97.05%, Nomad is less reliable when using safe storage. This is explained by the observations made in section 6.4, regarding Nomad's unreliable overlay storage.

Fast storage will report success even if the majority of requests fail, effectively trading reliability for responsiveness. For both Nomad and Pithos, a reliability of 100% is achieved.

In terms of responsiveness, Nomad using fast storage provides a responsiveness of under 28 ms. In comparison, Pithos in fast storage mode is more responsive than Nomad, with response times under 7 ms. In safe storage mode, however, Nomad is more responsive than Pithos, with a response time of 1.2 s compared to Pithos' 1.5 s.

#### 6.7.1.2. Retrieval

Table 6.34 provides a comparison between Nomad's and Pithos' storage responsiveness for different retrieval modes. As object retrieval reliability relies on the storage mode used to store objects, test objects were stored using safe storage, to ensure that objects were stored successfully.

| System | Mode | Reliability (%) | Responsiveness (ms) |
|---|---|---|---|
| Pithos | Safe Retrieval | 99.77 | 800.00 |
| Nomad | | 100 | 254.05 |
| Pithos | Fast Retrieval | 99.70 | 189.00 |
| Nomad | | 100 | 17.01 |
| Pithos | Parallel Retrieval | 99.98 | 85.90 |
| Nomad | | 100 | 19.28 |

**Table 6.34:** Group storage - retrieval operations comparison

Nomad generally achieves a higher degree of retrieval reliability than Pithos due to the design alteration to the group ledger. In Pithos, object references are added to the group ledger regardless of whether the operation succeeded or not. Nomad's group ledger only stores object and peer ID references. Additionally, IDs are only added to the ledger if the object was successfully stored in local storage.

Nomad in fast retrieval mode is more reliable than Pithos, with a reliability of 100% versus Pithos' 99.70% reliability. Nomad in parallel retrieval mode is more reliable, with a reliability of 100% versus Pithos' 99.98%. Similarly, in safe mode, Nomad is more reliable than Pithos, with a reliability of 100% compared to Pithos' 99.77%.

The RTL of Nomad's retrieval requests is also lower than those of Pithos. In safe mode, Pithos is far less responsive than Nomad, with response times of over 800 ms compared to Nomad's 254.05 ms. Nomad is also much more responsive than Pithos in fast and parallel retrieval mode, with response times of 17.01 ms and 18.29 ms respectively, compared to Pithos' 189.00 ms and 75.90 ms response times.

Pithos' lower retrieval results may be explained by the 10 Mbps bandwidth limitation that was imposed on Pithos peers, but at low retrieval rates, the bandwidth limitation should not severely limit performance.

#### 6.7.1.3. Conclusion

It is possible to conclude from the results provided in tables, 6.33 and 6.34 that group storage is reliable and responsive. In general, Pithos simulation provided higher responsiveness and reliability when storing objects, although this may be due to Nomad's poor overlay storage performance. Both systems have high response times of above 1 second when using safe storage, since in safe mode, both systems wait for all responses to complete before sending a response to the higher layer.

For object retrieval, Nomad was found to be slightly more reliable and responsive than Pithos. The adjustments that were made to various modules and mechanisms explain the performance and reliability improvements that Nomad achieved over Pithos. Another cause of Pithos' reduced performance could be the imposed bandwidth limitations, although at low request rates, performance should not be affected.

### 6.7.2. Overlay Storage Comparison

As seen during Nomad's evaluation, the overlay storage component impacts the performance and reliability of the system. It is therefore important to compare the overlay storage results of the two systems. Whilst both systems use a DHT for overlay storage, the actual DHT implementations that are used differ. Pithos makes use of Chord, whilst Nomad makes use of a Kademlia based DHT, TomP2P. Section 5.3.4.1 contains more detail on why a different DHT was used.

During Pithos' evaluation, different DHTs were compared, Kademlia being one of them [1]. It was found that Kademlia was unreliable and unresponsive. In order to keep the results comparable, Table 6.35 includes the results for Pithos' performance using both Chord and Kademlia.

| System | DHT | Storage Reliability (%) | Retrieval Reliability (%) | Store (s) | Retrieve (s) | BW (KBps) |
|---|---|---|---|---|---|---|
| Pithos | Chord (medium) | 96.90 | 93.20 | 1.21 | 1.58 | 4.36 |
| Pithos | Kademlia | 45.53 | 35.13 | 0.90 | 4.60 | 0.61 |
| Nomad | Kademlia | 100 | 47.40 | 1.70 | 3.04 | 8.50 |

**Table 6.35:** Overlay Storage comparison: Nomad Kademlia DHT vs Pithos Chord DHT

From Table 6.35, it is clear that Pithos' DHT implementation of choice, Chord, is far more reliable than Nomad's Kademlia implementation. In terms of object storage, with a reliability of 96.90% and response times of under 1.21 seconds, Chord is more responsive, if slightly less reliable than Nomad's Kademlia. For object retrieval, Chord is vastly more reliable and responsive than Kademlia, with a reliability of 93.20% and response times of under 1.58 seconds. In general, Pithos' overlay component provides better response times and reliability, and uses less bandwidth than Nomad's overlay component.

Nomad was also compared to Pithos using a Kademlia DHT, but it should be noted that Pithos' bandwidth limitations may have limited its performance. From Table 6.35, it is clear that when Pithos uses a Kademlia based DHT, Nomad's implementation is more reliable and responsive. For storage requests, Nomad is more reliable and slightly less responsive, with a reliability of 100% and response times of 1.70 seconds. In terms of object retrieval, both implementations perform poorly, with Nomad providing slightly higher reliability of 47.40% and lower response times of 3.04 seconds. Pithos' Kademlia implementation does, however, require less bandwidth than Nomad's implementation.

#### 6.7.2.1. Conclusion

We can conclude from the results provided in Table, 6.35 that overlay storage has the potential to improve the reliability of the overall system, but lacks responsiveness. In general, Pithos simulation provided higher responsiveness and reliability for overlay storage operations. As Nomad uses a Kademlia DHT implementation, overall storage performance is impacted negatively.

Since no suitable Chord implementation currently exists, it is recommended that in future a Chord implementation be built for Nomad. This will enable Nomad to achieve similar overlay performance to Pithos.

### 6.7.3. Overall Comparison

As a final validation, the overall performance of Nomad and Pithos was compared. Both systems were evaluated using the system configuration parameters listed in table 6.32.

For Pithos' evaluation, the results of a network size of 2600 peers were simulated in OverSim, with the experiment lasting 10,000 seconds. Such a large network size was achievable due to the simulated nature of the experimental setup. For Nomad's evaluation, resource restrictions only allowed up to 50 peers to be evaluated simultaneously, with experiment iterations lasting up to 7740 seconds.

Both systems were evaluated under network churn and requests were made for local group objects, therefore no out-of-group requests were performed. The 95th percentile of all measurements was used.

#### 6.7.3.1. Responsiveness and Reliability

Table 6.36 shows a comparison between Nomad and Pithos in terms of overall responsiveness and reliability per component.

Overall, Nomad was found to be slightly more reliable and responsive when retrieving objects from group storage compared to Pithos. When using parallel retrieval, Nomad provides response times of under 20 ms and 100% reliability, which is a better than Pithos' 60 ms response times and 99.98% reliability. The same observation was made for fast retrieval.

In terms of object storage, Nomad was found to be more responsive, but not as reliable as Pithos. Using safe storage, Nomad provides response times of 1.2 seconds and reliability of 83.30%, whereas Pithos provides slightly slower response times of 1.554 seconds, but better reliability at 97.05%. This is due to Nomad's inefficient overlay storage component. Section 6.7.1 provides a more detailed comparison of the two group storage components.

When comparing the two systems' overlay components, Pithos' clearly has a better overlay implementation. Pithos' overlay was found to be more reliable and more responsive under network churn. Chord provides much better retrieval reliability of 93.20%, compared to TomP2P's 47.40%, whilst simultaneously providing better responsiveness. Section 6.7.2 provides a more detailed comparison of the two overlay storage components.

Nomad's bandwidth requirement is much higher than that of Pithos. Overall bandwidth usage is calculated as the sum of retrieval, storage and overlay bandwidth usage. On average, Nomad's bandwidth usage is almost 7 times that used by Pithos. This is caused by Nomad's higher bandwidth usage for both group and overlay storage. Additionally, Nomad's overlay storage was found to be extremely bandwidth inefficient.

| System | Entity | Reliability (%) | | Responsiveness (s) | | Bandwidth |
|---|---|---|---|---|---|---|
| | | store | retreive | store | retreive | (Bps) |
| Pithos (safe store) | Chord (med) | 96.9 | 93.20 | 1.214 | 1.582 | 2,380 |
| | Kademlia | 45.53 | 35.13 | 0.908 | 4.604 | 1,010 |
| | Group Storage (fast retrieve) | 97.05 | 99.77 | 1.554 | 0.189 | 2,750 (363) |
| | Group Storage (parallel rertrieve) | 97.05 | 99.98 | 1.554 | 0.0859 | 3,972 (1,592) |
| Pithos (fast store) | Group Storage (fast retrieve) | 100 | 99.70 | 0.0665 | 0.192 | 2,743 (370) |
| | Group Storage (parallel rertrieve) | 100 | 99.98 | 0.0665 | 0.0846 | 3,899 (1,519) |
| Nomad (safe store) | Kademlia | 100 | 47.40 | 1.630 | 2.949 | 8,500 |
| | Group Storage (fast retrieve) | 83.30 | 100 | 1.231 | 0.017 | 21,450 (12,950) |
| | Group Storage (parallel rertrieve) | 83.30 | 100 | 1.231 | 0.0192 | 22,150 (13,650) |
| | Group Storage (safe retrieve) | 83.30 | 100 | 1.231 | 0.254 | 22,900 (14,400) |
| Nomad (fast store) | Group Storage (fast retrieve) | 100 | 100 | 0.0271 | 0.017* | 16,550 (8,050) |
| | Group Storage (parallel rertrieve) | 100 | 100 | 0.0271 | 0.0192* | 17,250 (8,750) |
| | Group Storage (safe retrieve) | 100 | 100 | 0.0271 | 0.254* | 18,00 (9,500) |

**Table 6.36:** Pithos versus Nomad performance for different overlay implementations and storage and retrieval modes [31] (Retrieval operations marked with * were tested using safe storage)

Table 6.37 provides a summary of the average performance measurements of Nomad and Pithos respectively.

| Implementation | Module | Reliability (%) | Responsiveness (ms) | Bandwidth (KBps) |
|---|---|---|---|---|
| Pithos | Overall | 99.70 | 192.00 | 2.75 |
| Nomad | | 100 | 30.00 | 12.00 |
| Pithos | Group | 97.75 | 134 | 0.370 |
| Nomad | | 100 | 27.09 | 7.5 |
| Pithos | Overlay | 91.40 | 1.760 | 2.38 |
| Nomad | | 47.40 | 2.22 | 8.50 |

**Table 6.37:** Pithos versus Nomad average responsiveness and reliability

#### 6.7.3.2. Security

Table 6.38 provides a comparison between Nomad and Pithos in terms of security against malicious peers. As section 6.5.4 has already discussed Nomad's security results in detail, this section will merely provide a comparison between the results for Nomad and Pithos.

| Malicious Peers (%) | Retrieval Mode | Pithos Security (%) | Nomad Security (%) |
|---|---|---|---|
| 10 | Fast | 90.00 | 90.00 |
| | Safe | 99.00 | 99.00 |
| | Parallel | - | 95.00 |
| 25 | Fast | 78.00 | 77.00 |
| | Safe | 95.00 | 81.00 |
| | Parallel | - | 79.00 |
| 50 | Fast | 55.00 | 45.00 |
| | Safe | 65.00 | 52.00 |
| | Parallel | - | 50.00 |
| 75 | Fast | 28.00 | 21.00 |
| | Safe | 28.00 | 26.00 |
| | Parallel | - | 23.00 |

**Table 6.38:** Overview of Nomad vs Pithos security against malicious peers for different retrieval modes (Safe Retrieval use 4 compares for Quorum)

Table 6.38 shows that Nomad's reliability follows a similar trend to that of Pithos, with safe storage being the most reliable of the retrieval methods. Nomad's safe storage implementation does, however, decay faster in terms of security than that of Pithos. Nomad's lower security is due to the fact that it does not implement a certification mechanism, which Pithos does. The development of a certification mechanism for Nomad is deemed future work.

## 6.8. Conclusion

This chapter reported on detailed experiments on Nomad's group and overlay storage components. Nomad's group storage component was found to be responsive, reliable and scalable, with the option of security. In comparison to the Pithos simulation, Nomad's group storage component proved to be slightly more reliable and responsive.

It was found that overlay storage impacts the overall reliability and responsiveness of the system. In general, the Pithos simulation provided higher responsiveness and reliability for overlay storage operations. Nomad's use of a Kademlia DHT impacted the overall storage performance negatively. The overlay storage component is considered to be one of Nomad's weaknesses and is recommended to be replaced by a different DHT in future.

From the reviewed results, we can conclude that Nomad is an accurate implementation of the Pithos architecture. The results provided by various experiments prove that Nomad is a reliable, responsive, scalable and secure decentralised storage network. Although scalability and load balancing were not explicitly tested, Nomad has thus far proven to be scalable, as group sizes can be controlled to cater for system requirements. In Terms of fairness, it can be concluded that Nomad is fair (load-balancing), since objects and requests are randomly distributed amongst all peers within a group and the overlay.

The next chapter provides a conclusion with regard to this project's primary objectives, a project summary, and future recommendations.

# Chapter 7

# Conclusion and Recommendations

The objective of this work was to design, implement and evaluate a peer-to-peer (P2P) storage network, for state management and persistence (SMP) of a P2P MMVE.

This chapter presents a summary of the work done in this project and highlights its contributions. Recommendations are also made for future work which will allow Nomad to be used in internet facing systems.

## 7.1. Conclusion

Section 2.3 set out the key storage requirements of P2P MMVE systems as responsiveness, reliability, security, fairness, and scalability. In light of the evaluation of Nomad described in Chapter 6, Pithos' simulated results can now be verified in terms of these storage requirements.

### 7.1.1. Responsiveness

Nomad achieves responsiveness by splitting the network into fully connected groups of peers. The underlying architecture uses a distance-based group-based storage mechanism to store, retrieve, distribute and replicate objects. The assumption is made that peers are more likely to request objects from their own group, which helps Nomad to achieve highly responsive storage and retrieval for in-group requests.

Nomad's evaluation confirmed that group storage is extremely responsive in performing storage and retrieval operations under stable network conditions or under network churn. In terms of storage and retrieval modes, object retrieval and storage in fast mode were found to be the most responsive and to require the lowest amount of bandwidth compared to safe or parallel storage. Fast operations do, however, lack the security and reliability provided by parallel and safe operations.

Creating object replicas proved to be costly in terms of resource usage, although it does improve the responsiveness of storage and retrieval requests as some peers may be geographically closer than others, leading to lower RTLs. In parallel retrieval and fast storage mode, Nomad sends identical requests to multiple peers and only awaits the first successful response, thereby taking advantage of the distribute nature of object replicas.

In terms of responsiveness, Nomad's evaluation produced similar values to those measured during Pithos' simulation. It can therefore be concluded that Pithos' measured response times are accurate and that Pithos satisfies the P2P MMVE storage requirement of responsiveness.

### 7.1.2. Reliability

Reliability in Nomad is defined as the ratio between the number of successful responses and the total number of requests, expressed as:

$$reliability = \frac{successful\ responses}{total\ requests}$$

Reliability is achieved in Nomad by making use of various modules and mechanisms. An overlay storage component in the form of a DHT increases reliability, as it is able to serve any in-group or out-of-group retrieval request. Object replication ensures that sufficient replicas exist in case of storage peers leaving the network. Replicas are maintained by object repair mechanisms, which ensures a set number of object replicas are always available.

Nomad's evaluation confirmed that the overlay storage does impact the overall performance of the network. During Nomad's evaluation, a Kademlia based DHT was used as overlay component, whereas during the Pithos simulation a Chord based DHT was used. Kademlia proved to be far less reliable and responsive compared to Chord, and led to a lower overall reliability. This proves that overlay storage does impact the overall performance of the network, and that it is required to ensure system reliability under network churn and group migration.

Nomad's overlay component was far less responsive under network churn compared to group storage. Object storage requests were reliable, but retrieval requests far less so, and also less responsive. Nomad can be described as less reliable than the Pithos simulation, as out-of-group requests are likely to fail in Nomad. As such, Nomad's overlay storage is one of the system's weaknesses and requires further investigation.

The use of object replicas improved overall reliability, as replication provides high availability for objects in the presence of group migration and network churn. Nomad's scheduled and leave-repair mechanisms proved to consistently maintain object replicas, but had a significant impact on performance.

Nomad's evaluation provided similar results to those of the Pithos simulation. It can therefore be concluded that Pithos satisfies the P2P MMVE storage requirement of reliability.

### 7.1.3. Security

Security is achieved in Nomad by making use of object replication and a quorum mechanism. Nomad also supports *safe* reads and writes, which inherently increases reliability and security.

Object replication diminishes the impact of malicious peers in an MMVE. As multiple peers store object replicas, a maliciously altered version of the object is easily identifiable. Multiple object replicas also enable the use of quorum mechanisms on parallel responses.

When parallel requests are made, depending on the retrieval mode, a quorum mechanism can be used to verify object state. This way, a peer can decide to select only the most frequently occurring version of an object. This effectively diminishes the impact of malicious peers in the system and improves security. Nomad's quorum mechanism can be described by the quorum formula $(\mathcal{R}/2) + 1$, where $\mathcal{R}$ is the replication factor.

During Nomad's evaluation, it was proven that different retrieval modes have varying degrees of success in identifying maliciously altered objects. Safe storage is the most secure retrieval mode, as multiple retrieval responses are compared before a decision is made on the integrity of an object. Fast and parallel retrieval are less secure, as the first response is always sent to the higher layer.

Pithos implements a certificate authority (CA) and a certification mechanism, in order to assign IDs and certificates to peers, and to sign object changes. Nomad does not make use of this certification mechanism and therefore, in its current state does not support Secure Sockets Layer (SSL) communication between peers or signing of object changes. These certification mechanisms are considered future.

Nomad's implementation lacks the security features of the Pithos architecture, but does provide core security functionality for verifying object integrity. Nomad's evaluation proved that Pithos' results are accurate and that Pithos satisfies the P2P MMVE storage requirement of security.

### 7.1.4. Fairness

To ensure fairness (load-balancing), peers and super-peers are not selected on any discriminating factors but purely based on their location and time of joining the group. Load is distributed evenly across all peers in the network by making use of randomly distributing group storage requests. Group storage and overlay storage are inherently considered to be fair.

The group storage mechanism distributes object replicas in a uniformly random fashion, ensuring that no one peer is favoured for storage purposes. Peers use their own copy of the peer ledger to choose random peers in the group when (a) requesting objects or (b) storing object replicas.

For overlay storage, a DHT is used. DHTs map objects and peers to the same identifier

space. Objects are stored on peers with the closest ID match. This way, if IDs are assigned in a uniformly random fashion, storage load is inherently distributed in a uniformly random fashion.

During Nomad's evaluation, fairness was not directly measured, but observation of resource usage confirmed that all peers in a group had similar memory footprints. From Nomad's evaluation, it can be confirmed that objects were evenly distributed amongst peers. This means that Pithos' measured object distribution results can be considered accurate and that Pithos satisfies the P2P MMVE storage requirement of fairness.

### 7.1.5. Scalability

In Nomad scalability, is achieved by making use of modules that are scalable in themselves. Nomad uses either a size-based or Voronoi-based grouping mechanism, which splits the network into fully connected groups. Groups have a fixed size limit, as fully connected networks scale quadratically in terms of messages required per transaction. Group size is therefore the limiting scaling factor. For its overlay storage component, Nomad makes use of a DHT, which is by design scalable.

One possible system bottleneck is the directory server, since it facilitates network bootstrapping, and provides peers with a local group cache which is used for various purposes. For Nomad's directory server, a scalable and highly consistent distributed coordination system called ZooKeeper is used. ZooKeeper can be scaled vertically and horizontally, and is already provided by many cloud providers as a fully configured product.

Nomad's evaluation proved that as the number of nodes in the network increased, performance was not severely affected. Since Nomad could only be evaluated with up to 50 peers, it is recommended that a scalability evaluation be executed using a higher number of nodes.

## 7.2. Summary of Work

In recent years, there has been much development surrounding distributed systems. In this project, the focus was massively multi-user virtual environments, which can be defined as VEs where thousands of users can simultaneously interact. In P2P MMVE systems, an individual peer contributes the required resources to the network to host itself. Since the peer acts as both a server and a client, the function of the consistency architecture also becomes distributed amongst peers.

Research by Gilmore and Engelbrecht [1] investigated the state of MMVEs with a focus on P2P MMVEs. After designing a generic state consistency model for both C/S and P2P models and performing a review of SMP architectures, the authors identified the key challenges of P2P MMVEs, mentioned in subsection 1.4.4. In particular it was

found that research regarding *state consistency* in P2P MMVEs was generally lacking. This observation led to the design and implementation of a novel SMP architecture, called Pithos, which was partially implemented and evaluated through simulation.

This project presented the development and evaluation of a Pithos based decentralised storage network, created specifically for P2P MMVEs, that could be used to verify Pithos' simulated results. A standalone Java based application, called Nomad, was created using various frameworks and technologies, in order to satisfy the component and use case requirements of the Pithos architecture. The implemented application was deployed to multiple machines and evaluated under Normal network conditions.

### 7.2.1. Nomad Design and Implementation

The use case of a Pithos based system, is that of a generic storage system, supporting object storage, retrieval, modification and removal. Nomad implements all Pithos' required modules and mechanisms, in order to satisfy the basic system use cases of a storage network, whilst still adhering to the P2P MMVE storage requirements of reliability, responsiveness, scalability, security and fairness.

Various alterations and adjustments were made to the original architecture, in order to satisfy the maintainability requirements of a real-world implementation, whilst dealing with language specific restrictions.

### 7.2.2. Nomad Evaluation

The responsiveness, reliability and security of Nomad were evaluated, taking bandwidth usage into account. Nomad's performance was compared to Pithos' simulated performance so as to verify the integrity of both the architecture and the simulation.

Results provided by various experiments proved that Nomad is a reliable, responsive, scalable and secure decentralised storage network. Although scalability and fairness were not explicitly tested, Nomad has thus far proven to be scalable, as group sizes can be controlled to cater system requirements. In terms of fairness, it can be concluded that as objects are randomly distributed amongst all peers, Nomad is indeed fair.

The Nomad implementation also verified that Pithos' simulated results are accurate and that a Pithos is a suitable storage architecture for P2P MMVEs.

## 7.3. Recommendations for Future Work

1. *Implement a fully functional certification mechanism* - Nomad currently does not make use of any certification mechanism, which means that peer validation and tracking of object changes are not possible. A certification mechanism should be implemented in order to allow for additional security in Nomad.

2. *Replace Nomad's overlay storage component* - Nomad currently makes use of TomP2P, a Kademlia based DHT, as overlay storage. During Nomad's evaluation, the overlay storage component was found to be extremely unresponsive under network churn, and generally unreliable. It is therefore recommended to either replace Nomad's overlay component with a more suitable DHT implementation, or to implement a Chord based DHT from scratch.

3. *Research alternatives for directory server implementation* - Nomad currently makes use of ZooKeeper, for its directory server. In its current state, the directory server is completely functional and capable of providing the required logic. The longevity of ZooKeeper is, however, somewhat of a concern. Projects like *etcd* [94] and *consul* [95] are very promising alternatives that are widely used in the industry and have the prospect of longevity.

4. *Improve safe storage quorum mechanism* - Nomad's current quorum mechanism relies heavily on Java POJO (Plain Old Java Object) comparison, which is robust, but prone to human error, i.e. if the underlying models are changed without using a framework to generate *GameObjects*. It is recommended to implement a more secure and robust quorum mechanism for Nomad, in order to ensure the correct function of safe retrieval and updates.

5. *Improve fast storage response times* - In order to make fast storage completely independent of group size, additional software improvements are required. In Nomad's current state, a peer will notify all other group peers of an object added before reporting success. This can be improved by returning storage results *before* notifying peers of new objects. This means that once the object is stored locally, a successful storage result is returned.

6. *Allow super-peer movements* - To simplify Nomad's Voronoi grouping logic, once a super-peer is selected and it sends its VE location to the directory server, it can no longer receive any positional updates. This means that super-peer positions are static in Nomad's current implementation. Additional logic is required to allow super-peers to move within the VE. This means that groups will have a velocity and that all peers and super-peers need to recalculate their Voronoi maps regularly to ensure a consistent view of the VE. Calculating the Voronoi map too frequently causes excessive system load, especially for large VEs. It is therefore recommended to recalculate the Voronoi map at a reasonable interval.

7. *Solving NAT challenges* - Docker containers and Kubernetes deploy scripts already exist for Nomad, but in order for Nomad to be used in the cloud, it is required to first solve the problem of NAT traversal. NAT traversal is a problem, due to Nomad's

use of random ports for its services, since multiple Nomad instances are executed on a single machine. If static ports are used, the problem will become significantly easier to solve.

# Bibliography


[1] J. S. Gilmore and H. A. Engelbrecht, "A State Management and Persistence Architecture for Peer-to-Peer Massively Multi-user Virtual Environments," Ph.D. dissertation, Stellenbosch University, South Africa, 2013.

[2] Blizzard Entertainment Inc. "World of Warcraft". Visited on 2021-08-19. [Online]. Available: https://worldofwarcraft.com/en-gb/

[3] A. Yahyavi and B. Kemme, "Peer-to-Peer Architectures for Massively Multiplayer Online Games: A Survey," *ACM Computing Surveys, vol. 46, no. 1*, pp. 39:9–39:10, 2013.

[4] IGN.com. (2012) "World of Warcraft Reaches 12 Million Subscribers". Visited on 2021-06-29. [Online]. Available: https://www.ign.com/articles/2010/10/07/world-of-warcraft-reaches-12-million-subscribers

[5] Statista.com. (2016) "World of Warcraft estimated subscribers from 2015 to 2030". Visited on 2021-06-29. [Online]. Available: https://www.statista.com/statistics/276601/number-of-world-of-warcraft-subscribers-by-quarter/

[6] Docker Inc. "Docker overview". Vistied on 2021-07-03. [Online]. Available: https://docs.docker.com/get-started/overview/

[7] Cloud Native Computing Foundation. "What is kubernetes?". Vistied on 2021-07-03. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[8] Napster. "Napster". Visited on 2021-07-01. [Online]. Available: http://www.napster.com

[9] S. El-Ansary and S. Haridi, "An Overview of Structured P2P Overlay Networks," *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks*, pp. 3–19, 2005.

[10] E. Jörg, R. Schollmeier, S. Zöls, and G. Kunzmann, "Structured P2P Networks in Mobile and Fixed Environments," p. 3, 2021.

[11] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski, "Scale-up x Scale-out: A Case Study using Nutch/Lucene," 2007, pp. 1–8.

[12] K. Aberer, L. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth, "The essence of P2P: a reference architecture for overlay networks," in *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, 2005, pp. 11–20.

[13] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, vol. 2003, 2003.

[14] P. Maymounkov and D. Eres, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric," in *Lecture Notes in Computer Science, vol. 2429*, vol. 2429, 2002.

[15] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for Internet applications," *IEEE Transactions on Networking*, vol. 11, 2003.

[16] M. Castro, M. Costa, and A. Rowstron, "Debunking Some Myths About Structured and Unstructured Overlays," pp. 85–87, 2005.

[17] F. Bordignon and G. Tolosa, "Gnutella: Distributed System for Information Storage and Searching Model Description," 2001.

[18] L. Fan, P. Trinder, and H. Taylor, "Design issues for Peer-to-Peer Massively Multiplayer Online Games," *IJAMC*, vol. 4, pp. 108–125, 2010.

[19] S. Benford and L. Fahlén, "A Spatial Model of Interaction in Large Virtual Environments," in *Proceedings of the Third European Conference on Computer-Supported Cooperative Work*, 1993.

[20] G. Morgan, F. Lu, and K. Storey, "Interest management middleware for networked games," in *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, 2005, pp. 57–64.

[21] A. Yu and S. Vuong, "MOPAR: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games," 2005, pp. 99–104.

[22] N. Matsumoto, Y. Kawahara, H. Morikawa, and T. Aoyama, "A scalable and low delay communication scheme for networked virtual environments," in *GLOBECON 2004: IEEE Global Telecommunications Conference Workshops*, 2004, pp. 529–535.

[23] S. Deering and D. Cheriton, "Multicast Routing in Datagram Internetwork and Extended LANs," *ACM Trans. Comput. Syst.*, vol. 8, pp. 85–110, 1990.

[24] T. Iimura, H. Hazeyama, and Y. Kadobayashi, "Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games," 2004, pp. 116–120.

[25] A. Bharambe, J. Pang, and S. Seshan, "Colyseus: A Distributed Architecture for Online Multiplayer Games," pp. 155–158, 2007.

[26] P. Trinder and H. Taylor, "MAMBO: Membership-Aware Multicast with Bushiness Optimisation," 2008.

[27] J. Gilmore and H. Engelbrecht, "A Survey of State Persistence in Peer-to-Peer Massively Multiplayer Online Games," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, pp. 819–825, 2012.

[28] A. Normoyle, G. Guerrero, and S. Jörg, "Player Perception of Delays and Jitter in Character Responsiveness," *Proceedings of the ACM Symposium on Applied Perception, SAP 2014*, pp. 1–10, 2014.

[29] M. Claypool and K. Claypool, "Latency and player actions in online games," *Commun. ACM*, vol. 49, pp. 40–45, 11 2006.

[30] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2009.

[31] H. A. Engelbrecht and J. S. Gilmore, "Pithos: Distributed Storage for Massive Multi-user Virtual Environments," *ACM Trans. Multimedia Comput. Commun.*, vol. 13, no. 3, pp. 31:1–31:33, 2017.

[32] Skynet Labs. "Sia Vision". Visited on 2021-07-04. [Online]. Available: https://sia.tech/about

[33] D. Vorick and L. Champine, "Sia: Simple Decentralized Storage," pp. 1–7, 2014.

[34] M. Riley and I. Richardson. "An introduction to Reed-Solomon codes: principles, architecture and implementation". Visited on 2021-07-04. [Online]. Available: https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html

[35] J. Callas, J. Walker, K. Kohno, M. Bellare, D. Whiting, B. Schneier, S. Lucks, and N. Ferguson, "The Skein Hash Function Family," 2009.

[36] Skynet Labs. "Skynet Overview". Visited on 2021-07-04. [Online]. Available: https://support.siasky.net/

[37] Figure Studios. "SkyGameSDK Repository". Vistied on 2021-07-04. [Online]. Available: https://github.com/figurestudios/SkyGameSDK

[38] Skynet Labs. "SkyGameSDK Demo". Visited on 2021-07-04. [Online]. Available: https://skynect4.hns.siasky.net/#/55612418230948740

[39] ——. "Skynet Basics". Visited on 2021-07-04. [Online]. Available: https://support.siasky.net/getting-started/skynet-basics#skynet-is-built-on-top-of-sia

[40] D. Vorick and L. Champine, "Storj: A Decentralized Cloud Storage Network Framework," 2018.

[41] N. Ahmad, L. Wei, and M. Jabbar, "Advanced Encryption Standard with Galois Counter Mode using Field Programmable Gate Array." *Journal of Physics: Conference Series*, vol. 1019, 2018.

[42] Storj Labs Inc. "Product Overview". Visited on 2021-07-04. [Online]. Available: https://docs.storj.io/dcs/storage/considerations

[43] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System," 2014.

[44] D. Vorick and L. Champine, "Filecoin: A Decentralized Storage Network," 2017.

[45] Protocol Labs. "Network performance". Visited on 2021-07-05. [Online]. Available: https://docs.filecoin.io/about-filecoin/network-performance/#financial-transfers

[46] ——. "What is Filecoin". Visited on 2021-07-05. [Online]. Available: https://docs.filecoin.io/about-filecoin/what-is-filecoin/

[47] R. Sunitha, "Impact of Digital Humanities And Literary Study In Electronic Era," *International Journal of Computer Trends and Technology*, vol. 68, pp. 22–24, 2020.

[48] Gartner, "Gartner Says Worldwide IaaS Public Cloud Services Market Grew 40.7in 2020," *Gartner Newsroom*, visited on 2021-07-06. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-06-28-gartner-says-worldwide-iaas-public-cloud-services-market-grew-40-7-percent-in-2020

[49] Amazon Web Services. "Amazon GameLift". Vistied on 2021-07-07. [Online]. Available: https://aws.amazon.com/gamelift/

[50] Google. "Game Servers documentation". Vistied on 2021-07-07. [Online]. Available: https://cloud.google.com/game-servers/docs

[51] Microsoft. "Azure for Gaming". Vistied on 2021-07-07. [Online]. Available: https://docs.microsoft.com/en-us/gaming/azure/

[52] AWS. AWS Storage Services Overview. Visited on 2021-07-06. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/aws-storage-services-overview/welcome.html

[53] G. DeCandia, D. Hastorun, M. Jampni, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," *Amazon.com*, pp. 205–209.

[54] Game Developer. (2009) "GDC Austin: An Inside Look At The Universe Of Warcraft". Visited on 2021-09-26. [Online]. Available: https://www.gamasutra.com/php-bin/news_index.php?story=25307

[55] R. Miller. "WoW's Back End: 10 Data Centers, 75000 Cores". Visited on 2021-09-26. [Online]. Available: https://www.datacenterknowledge.com/archives/2009/11/25/wows-back-end-10-data-centers-75000-cores

[56] IFIXIT. "WoW Server Blade Teardown". Visited on 2021-09-26. [Online]. Available: https://www.ifixit.com/Teardown/WoW+Server+Blade+Teardown/9389

[57] ActivePlayer.io. "World of Warcraft Live Player Count and Statistics". Visited on 2021-08-26. [Online]. Available: https://activeplayer.io/world-of-warcraft/

[58] G. Moore, "Cramming More Components Onto Integrated Circuits," *Proceedings of the IEEE*, vol. 86, 1998.

[59] Amazon Web Services. "AWS pricing calculator". Vistied on 2021-08-26. [Online]. Available: https://calculator.aws/#/

[60] HostBarrel. "WoW Private Server Hosting". Visited on 2021-09-26. [Online]. Available: https://www.hostbarrel.net/wow-private-server-hosting-compare.php

[61] Quora. "How much does a single World of Warcraft server cost to operate each month?". Visited on 2021-09-27. [Online]. Available: https://qr.ae/pG9b0E

[62] I. Baumgart, B. Heep, and S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework," in *IEEE Global Internet Symposium*, 2007, pp. 79–84.

[63] G. Ekaputra, C. Lim, and I. E. Kho, "Minecraft: A Game as an Education and Scientific Learning Tool," 2013, pp. 237–238.

[64] S.-Y. Hu, S.-C. Chang, and J.-R. Jiang, "Voronoi State Management for Peer-to-Peer Massively Multiplayer Online Games," 2008, pp. 1134–1138.

[65] Cambridge Dictionary. "Definition of Nomad". Vistied on 2021-08-27. [Online]. Available: https://dictionary.cambridge.org/dictionary/english/nomad?q=Nomad

[66] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, 1980.

[67] A. Beutel. "Interactive Voronoi Diagram Generator with WebGL". Vistied on 2021-04-12. [Online]. Available: http://alexbeutel.com/webgl/Voronoi.html

[68] Fortnite Wiki. “Map Gallery”. Vistied on 2021-04-12. [Online]. Available: https://fortnite.fandom.com/wiki/Map_Gallery

[69] J. Gosling, “Java: an Overview,” pp. 1–2, 1995.

[70] Golang.org. “GO”. Vistied on 2021-07-25. [Online]. Available: https://golang.org/

[71] JetBrains. “Kotlin”. Vistied on 2021-07-25. [Online]. Available: https://kotlinlang.org/

[72] Stack Overflow. “Programming, Scripting, and Markup Languages”. Vistied on 2021-07-22. [Online]. Available: https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages

[73] I. VMware. “Spring Boot”. Vistied on 2021-07-22. [Online]. Available: https://spring.io/projects/spring-boot

[74] The Apache Software Foundation. “MapStruct”. Vistied on 2021-07-22. [Online]. Available: https://mapstruct.org/documentation/stable/reference/html/

[75] gRPC Authors. “Introduction to gRPC”. Vistied on 2021-07-22. [Online]. Available: https://grpc.io/

[76] The Apache Software Foundation. “ZooKeeper”. Vistied on 2021-07-22. [Online]. Available: https://zookeeper.apache.org/doc/r3.7.0/zookeeperOver.html

[77] H2. “History and Roadmap”. Vistied on 2021-07-24. [Online]. Available: https://www.h2database.com/html/history.html

[78] ——. “Features”. Vistied on 2021-07-24. [Online]. Available: https://www.h2database.com/html/features.html

[79] Facebook. “RocksDB: A Persistent Key-Value Store for Flash and RAM Storage”. Vistied on 2021-07-04. [Online]. Available: https://github.com/facebook/rocksdb

[80] ——. “RocksDB Wiki”. Vistied on 2021-07-04. [Online]. Available: https://github.com/facebook/rocksdb/wiki

[81] T. Bocek. “TomP2P”. Vistied on 2021-07-22. [Online]. Available: https://github.com/tomp2p/TomP2P

[82] K. Repository. “Tektosyne”. Vistied on 2021-07-22. [Online]. Available: https://github.com/kynosarges/tektosyne

[83] SKKU/UNIST Data Intensive Computing Lab. “OpenChord Repository”. Visited on 2021-07-25. [Online]. Available: https://github.com/DICL/OpenChord

[84] freepastry.org. “FreePastry”. Visited on 2021-07-25. [Online]. Available: http://www.freepastry.org/FreePastry/

[85] S. Lohanathan, D. Dietler, and F. König. “P2P-Golang”. Vistied on 2021-07-25. [Online]. Available: https://gitlab.com/p2p-library-in-golang

[86] I. T. de Villiers. “Nomad Repository”. Vistied on 2021-07-04. [Online]. Available: https://gitlab.com/iggydv12/nomad

[87] Cloud Native Computing Foundation. “Creating a cluster with kubeadm”. Vistied on 2021-09-04. [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/

[88] Artillery.io. “Why Artillery?”. Vistied on 2021-08-04. [Online]. Available: https://artillery.io/docs/guides/overview/why-artillery.html

[89] G. Combs. “Ethernet (IEEE 802.3)”. Vistied on 2021-08-08. [Online]. Available: https://gitlab.com/wireshark/wireshark/-/wikis/Ethernet

[90] ——. “What is wireshark?”. Vistied on 2021-08-08. [Online]. Available: https://www.wireshark.org/docs/wsug_html_chunked/ChapterIntroduction.html#ChIntroWhatIs

[91] P. Moll, M. Lux, S. Theuermann, and H. Hellwagner, “A Network Traffic and Player Movement Model to Improve Networking for Competitive Online Games,” 2018, pp. 1–6.

[92] Epic Games. “Fortnite”. Vistied on 2021-08-07. [Online]. Available: https://www.epicgames.com/fortnite/en-US/home

[93] P. Moll, M. Lux, S. Theuermann, and H. Hellwagner. “FortniteTraces”. Vistied on 2021-04-12. [Online]. Available: https://github.com/phylib/FortniteTraces

[94] Cloud Native. “What is etcd?”. Vistied on 2021-09-03. [Online]. Available: https://etcd.io/

[95] HashiCorp. “Service Discovery and Health Checking”. Vistied on 2021-09-03. [Online]. Available: https://www.consul.io/use-cases/service-discovery-and-health-checking

[96] The Apache Software Foundation. “What is Maven?”. Vistied on 2021-07-22. [Online]. Available: https://maven.apache.org/what-is-maven.html

# Appendix A

# Structured Overlays

This appendix provides supplementary reading on structured overlays.

## A.1. Key-based Routing API for Structured Overlays

The KBR API is a three tiered API, designed in an effort to define a common interface for structured peer-to-peer overlays. This API provides the necessary guidelines for how nodes can interact with resources in the network. The KBR abstraction layers are categorised into the following tiers [13]:

- **tier-0**: The key-based routing layer
- **tier-1**: The higher abstraction layer
- **tier-2**: The application layer

Figure A.1 illustrates the relationship between each abstraction tier. Tier-0 provides an API with the fundamental building blocks used to implement tier-1 abstractions or tier-2 applications. Tier-2 applications can use multiple abstractions.

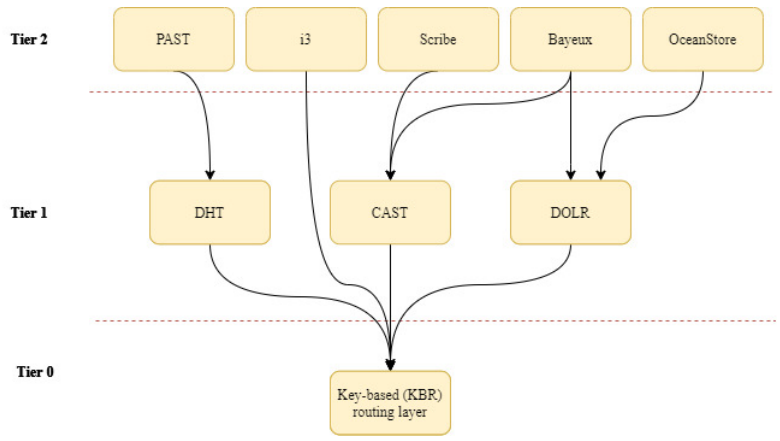


**Figure A.1:** KBR tiered API structure, based on Dabek et al. [13]

## A.2. Summary of Tier-0 KBR API

Figure A.1 provides a summary of the tier-0 KBR API.

| Interface method | Description |
|---|---|
| void route(K key, M msg, nodehandle hint) | Delivers a message, $M$, to key $K$'s root, using an optional hint |
| void forward(K key, M msg, nodehandle nextHopNode) | Informs the application that a message $M$ with key $K$ is being forwarded to nextHopNode |
| void deliver(K key, M msg) | A convenience method, executed on the root for key $K$ upon arrival of message $M$ |
| nodehandle[] localLookup(K key, int num, boolean safe) | Produces a list of possible next hops on route to key $K$, within *num* hops. The safe keyword guarantees that the number of faulty nodes in the result is no higher than the number of faulty nodes in the overlay |
| nodehandle[] neighbourSet(int num) | Produces a list of neighbours with list size *num* |
| nodehandle[] replicaSet(K key, int maxRank) | Produces a list of nodes (rank $\leq$ *maxRank*) where replicas of object with key $K$, can be stored |
| void update(nodehandle n, boolean joined) | Informs the application that node $n$ has joined or left the neighbour group |
| boolean range(nodehandle n, rank r, K iKey, K rKey) | Determines whether a node $n$ contains keys within the ranges [*iKey*; *rKey*] |

**Table A.1:** KBR API for structured overlays [13]

As a validation of the KBR API, Table A.2 illustrates how the KBR API can be implemented to satisfy the DHT API requirements.

| DHT | KPR implementation |
|---|---|
| put(key, value) | route(key,[PUT,value], R)) |
| remove(key) | route(key, [PUT,value,S], NULL) |
| get(key) | route(key, [GET,S], NULL) |

**Table A.2:** KBR API implementation for DHTs [13]

# Appendix B

# Nomad Implementation Details

This appendix provides supplementary details on Nomad's implementation.

## B.1. Nomad Build Tools

For dependency management and Java archive (JAR) compilation, Nomad uses Maven [96]. Maven is a Java build and management tool. Its primary objectives are to allow a developer to comprehend the full development state. Nomad uses Maven for;

- Simplifying the build process.
- Providing a uniform build system.
- Managing dependencies.

## B.2. Package Structure

The Nomad code repository consists of nine packages, namely:

- Application
- Commons
- Config
- Delegation
- gRPC
- Pithos
- Rest
- Storage

### B.2.1. Application

The "application" package consists of the main application module, which is responsible for setting the required system properties and starting the initial main controller application. The **Nomad** class is effectively the entry point of the application.

### B.2.2. Commons

The "commons" package consists of three utility classes, namely the *Base64Utility*, *KeyGenerator* and most importantly the *NetworkUtility* class.

The *Base64Utility* and *KeyGenerator* classes are not mission-critical, but they do provide useful functionality for encoding and decoding base64 IDs and converting images to byte arrays. The *NetworkUtility* class contains the key network utilities that allow the Nomad application to determine a peer's IP, check if specific ports are available, assign random ports, bind to network interfaces, and performing TCP pings to other peers within the network.

### B.2.3. Config

Two of Spring Boot's many advantages are externalised application configuration and easy access to properties defined in properties files. Nomad's "config" package makes use of Spring Boot's *@ConfigurationProperties* annotation, which provides automatic binding between properties defined in configuration files and Plain Old Java Objects (POJOs). This automatic configuration binding streamlines access to application properties.

Configuration is split into five different classes in Nomad, namely *WorldConfiguration, StorageConfiguration, NetworkHostnames, GroupConfiguration* and *DirectoryServerConfiguration* which are all accessible through the **Config** class. The underlying configuration file contains various configurable properties, and aims to make Nomad open and flexible for developers.

### B.2.4. Delegation

Nomad's "delegation" package contains the *directory server client, Voronoi grouping* and *leader election* logic. The **DirectoryServerClient** interface provides various interface functions that are used throughout Nomad's codebase. The use of the factory pattern ensures a high degree of flexibility by allowing developers to replace underlying implementations seamlessly.

### B.2.5. gRPC

Nomad peers request services from one another by using remote procedure calls (RPC). Nomad's key modules and mechanisms make use of various gRPC services in order to satisfy design requirements. The "grpc" package contains logic for Nomad's various gRPC components, namely *Peer, Super-peer* and *GroupStorage* gRPC server, service and client logic.

Like many RPC systems, gRPC is based around defining a service, by specifying interface methods that can be called remotely. **server** hosts a single gRPC **service**, which

is responsible for implementing all service.

### B.2.6. Pithos

The "pithos" package contains the implementation of all key peer components introduced by the Pithos architecture, namely the *peer, super-peer* and *group ledger* modules. Additionally, the package contains the **MainController** class, which is used to determine a nodes role, by establishing connections to the directory server and overlay, and starting the various peer and super-peer services.

### B.2.7. Rest

Each Nomad peer exposes a representational state transfer (REST) API for client use. A RESTful API simply makes use of Hypertext Transfer Protocol (HTTP) requests to access system resources. The "rest" package contains the main controller class for the API intended to be used by developers - the **PeerStorageController**.

Three additional controller classes exist, namely **ConfigController, GroupStorageController** and **overlayStorageController**, which are used for evaluation purposes only.

### B.2.8. Storage

The "storage" package contains the storage components used in Nomad, namely *local, group* and *overlay* storage. The **PeerStorage** module is used as the unifier of all Nomad's storage components. The PeerStorage class contains logic for initialising and maintaining storage components as well as interacting with VE resources.

# Appendix C

# Nomad System Overview and UML Diagrams

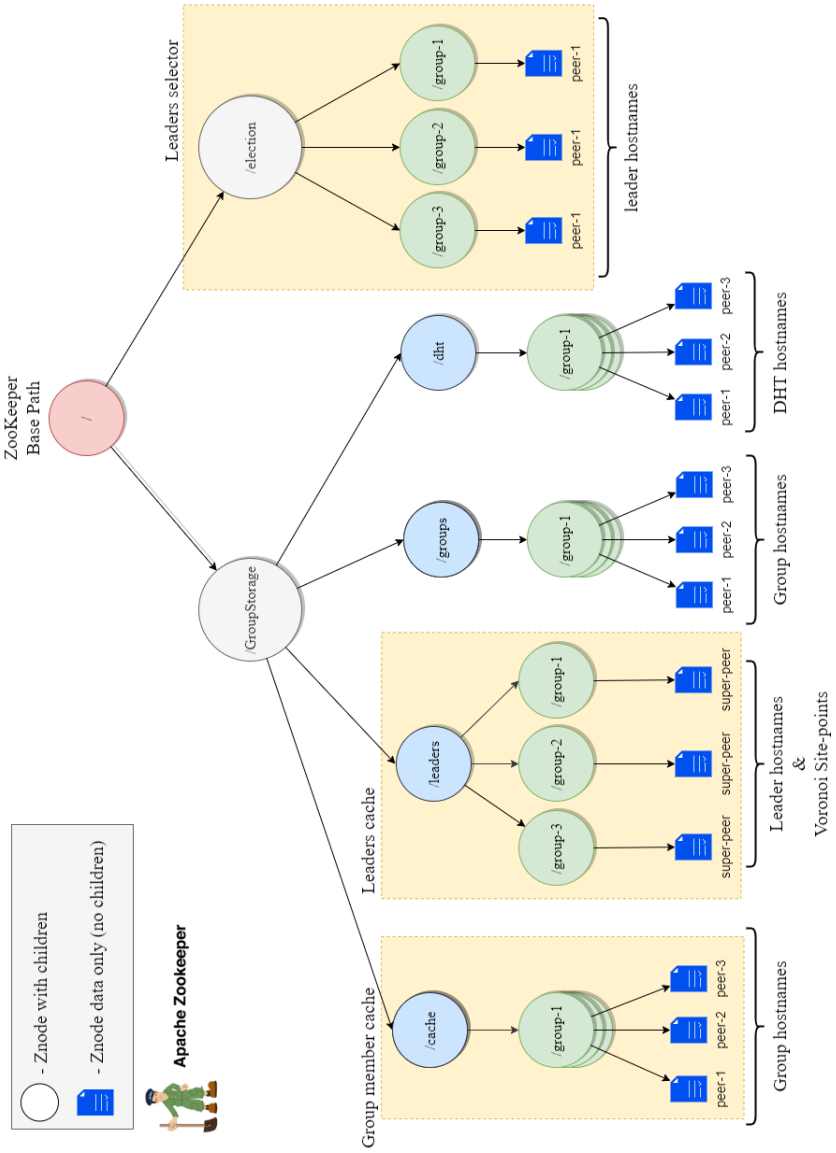


**Figure C.1:** Nomad ZooKeeper namespace, a directory server namespace used to cache peer, super-peer and DHT hostnames. Additionally if Voronoi grouping is enabled, the directory server caches super-peer location (Voronoi site points)

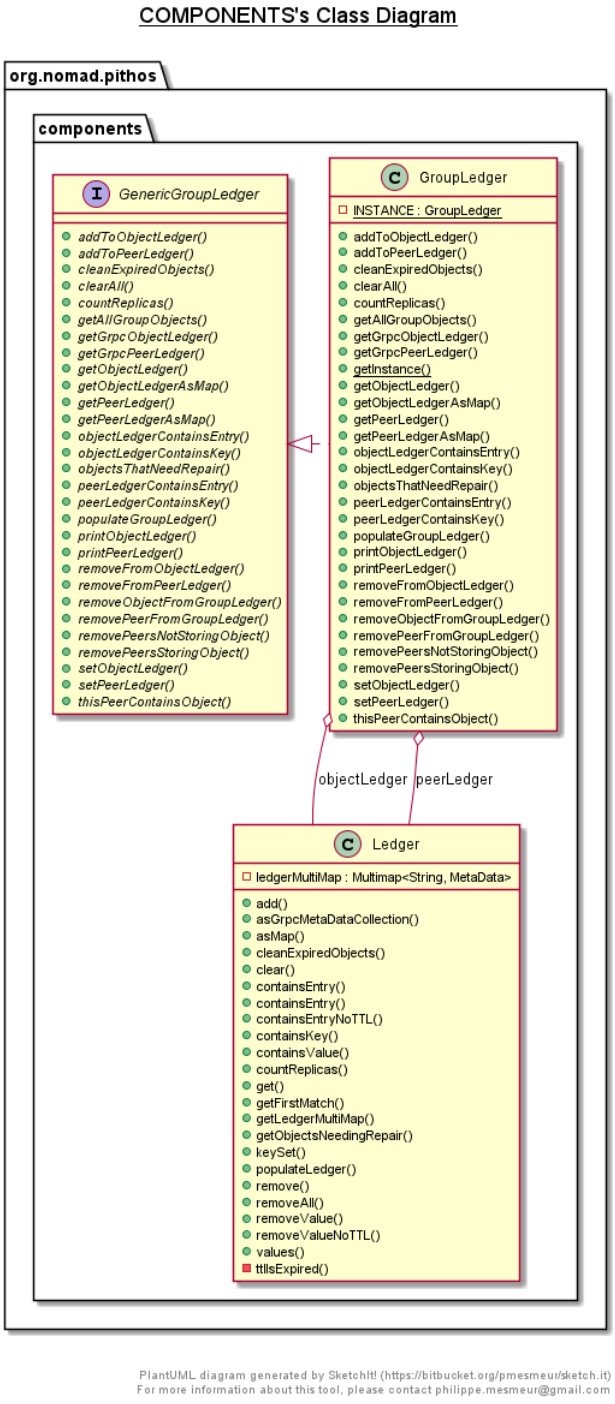


**Figure C.2:** Nomad Ledger implementation UML diagram (best viewed electronically)

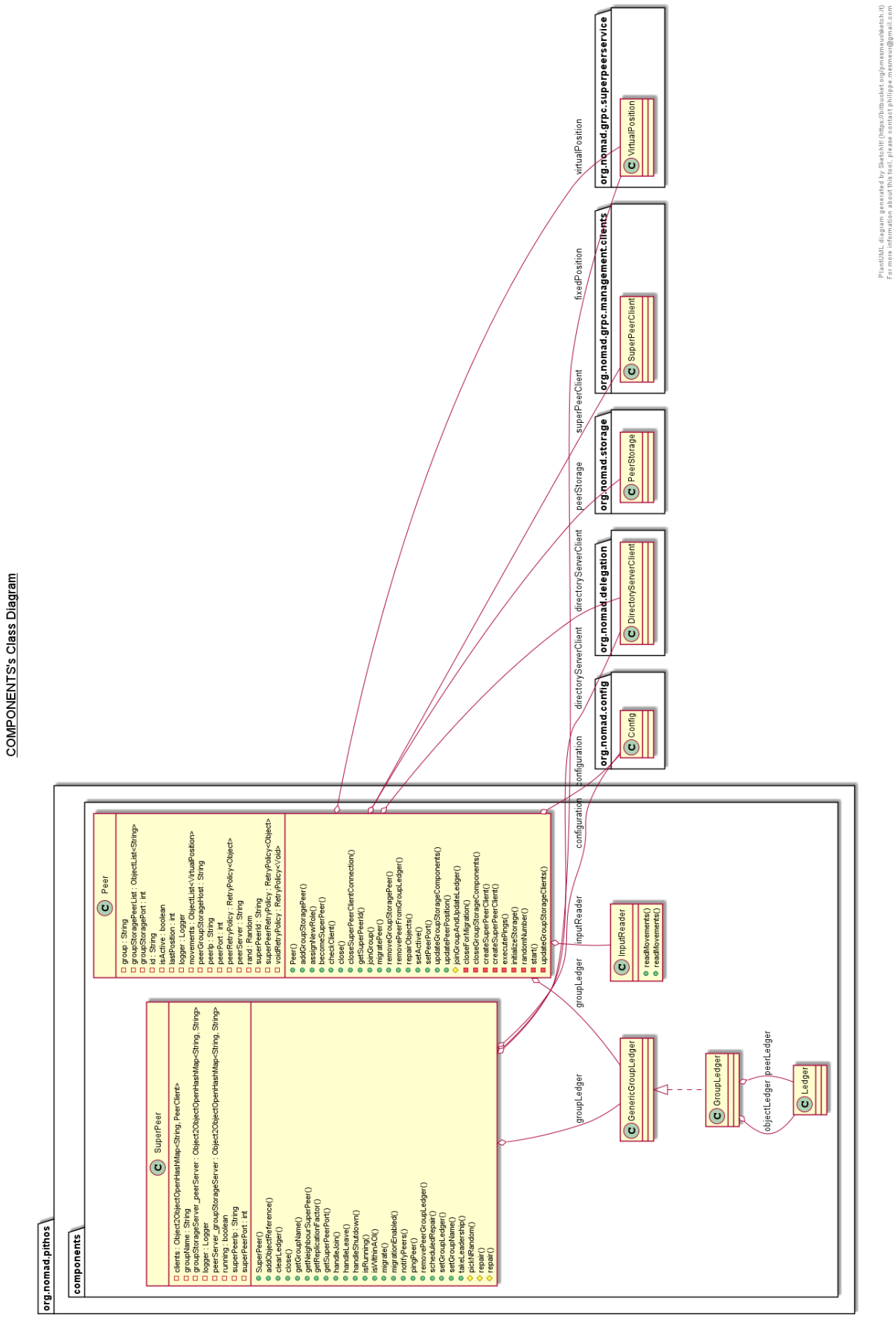


**Figure C.3:** Nomad ledger implementation UML diagram (best viewed electronically)

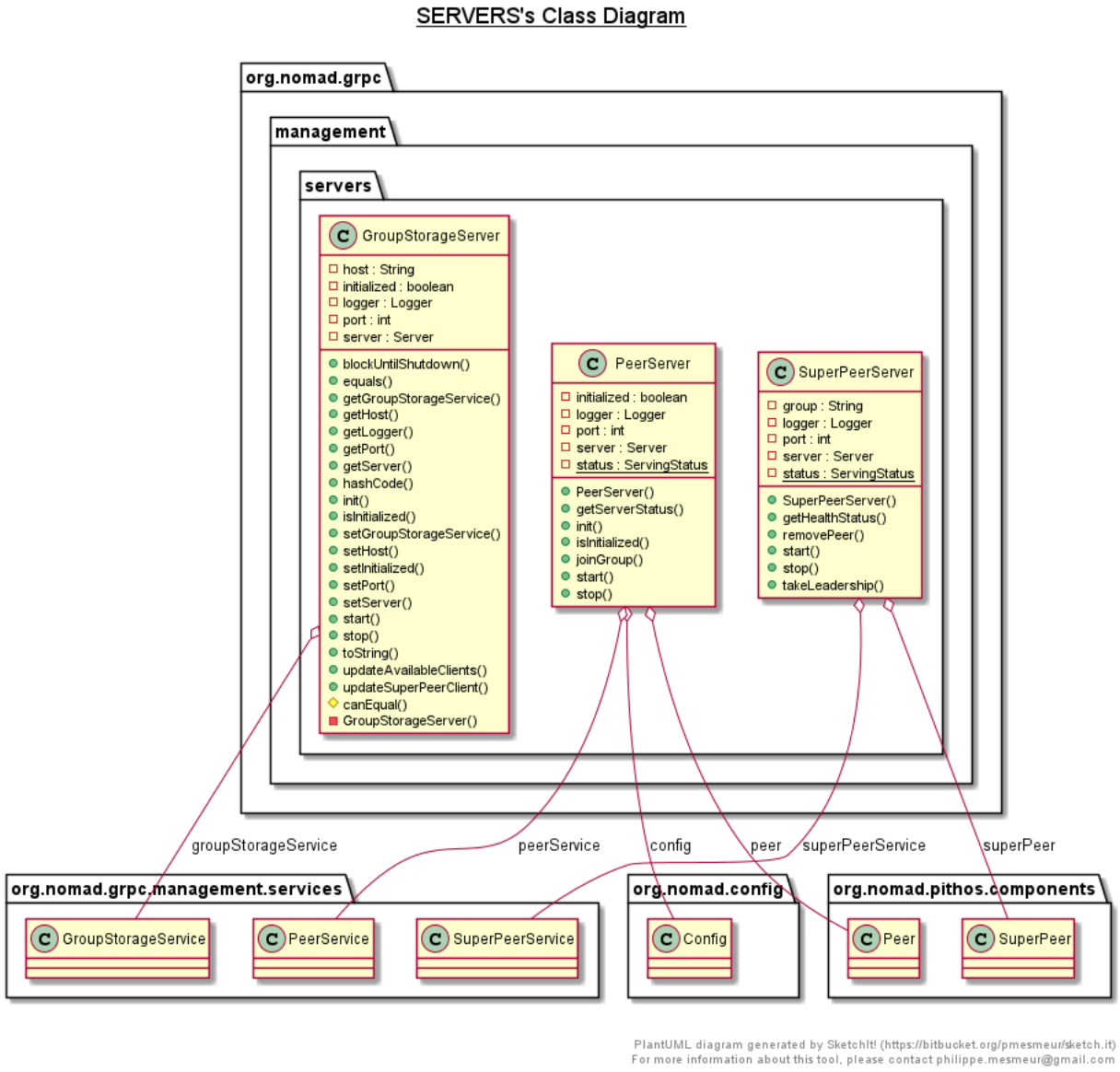


**Figure C.4:** Nomad gRPC servers UML diagram (best viewed electronically)

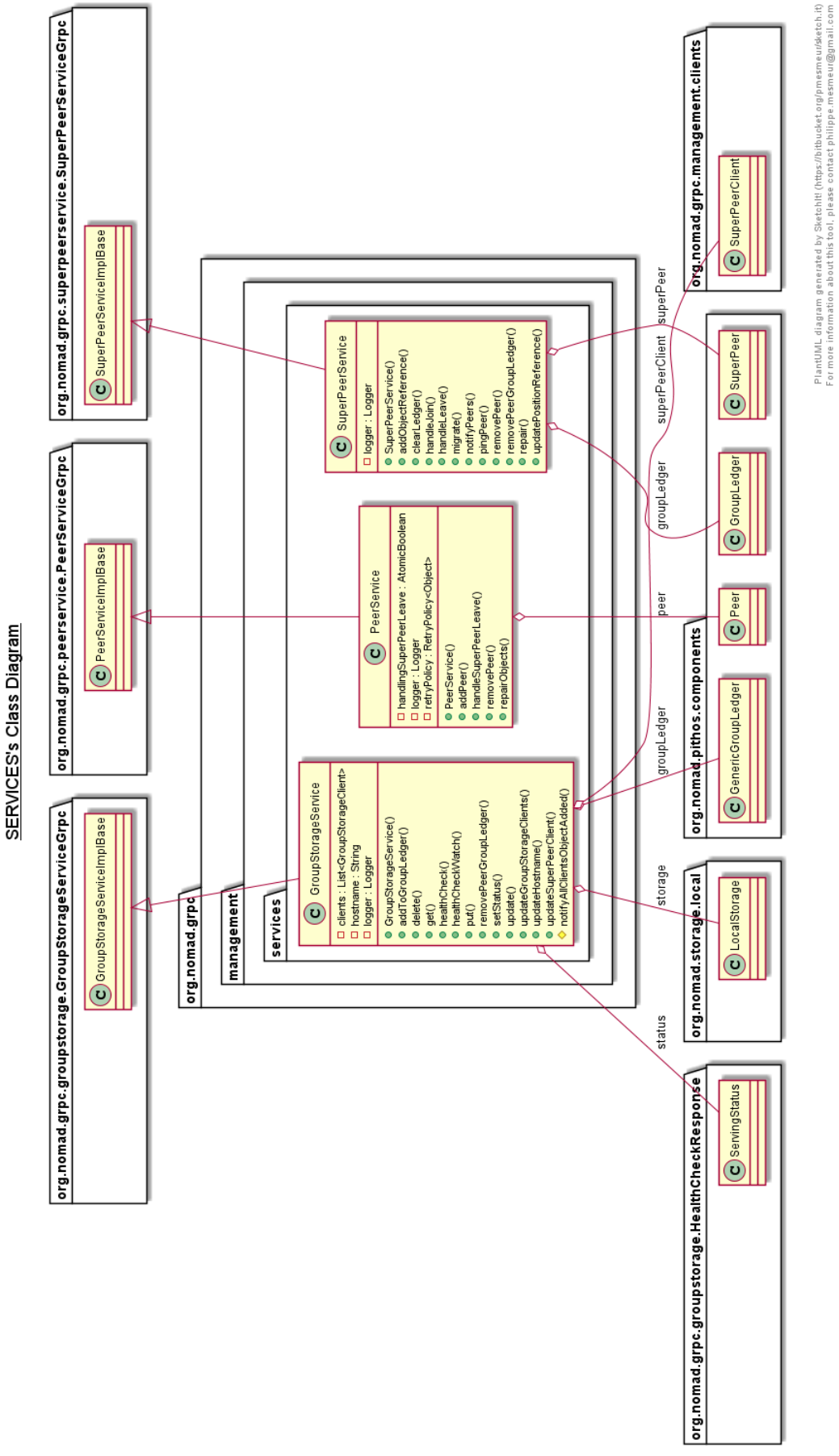


**Figure C.5:** Nomad gRPC services UML diagram (best viewed electronically)

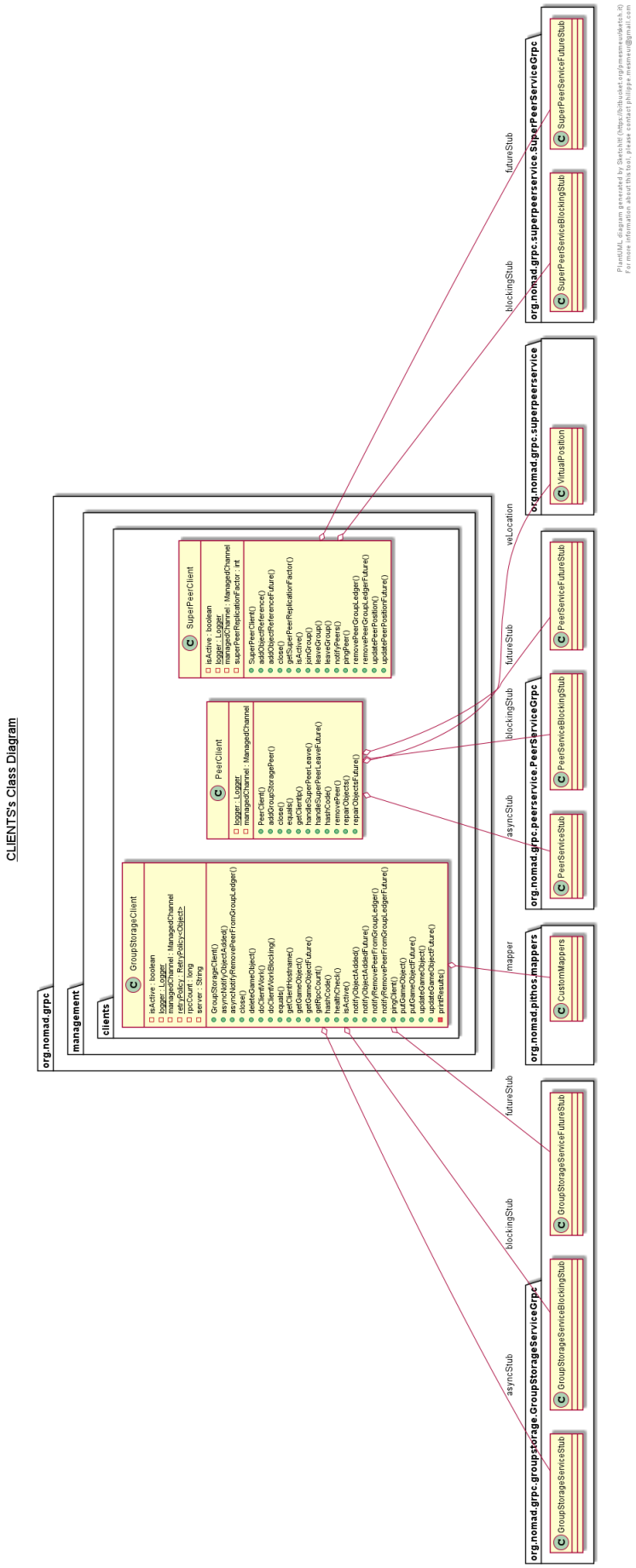


**Figure C.6:** Nomad gRPC clients UML diagram (best viewed electronically)

# Appendix D

# Pithos Evaluation Results

## D.1. Fairness

This section provides a brief summary of Pithos' fairness for overlay and overall storage. Pithos' fairness (load balancing) was tested using an identical experiment setup to that in Section 6.1, with the addition that peers generate objects for 100 seconds instead of 20 seconds. This allows for a higher number of objects to be stored within the network [31]. Groups were tested in fast storage and retrieval mode.

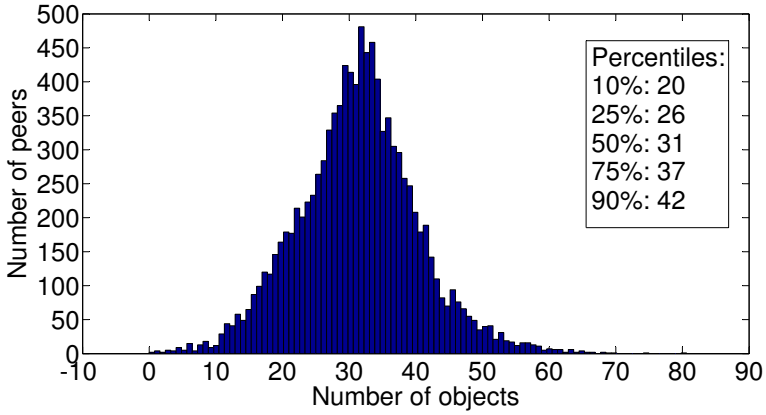


**Figure D.1:** Pithos fairness - object distribution over peers during peer lifetime [31]

Figure D.1, from work by Engelbrecht and Gilmore [31], illustrates object distribution (or load-balancing) amongst Pithos peers. On average, it was found that each peer stores at least 31.2 objects, with a standard deviation of 9 objects. From these results, it can be concluded that Pithos distributes objects amongst peers in a fair manner.

# Appendix E

# Nomad Evaluation

## E.1. Group Storage: Supplementary Storage & Retrieval Results

Tables E.1 shows Nomad's group retrieval results in terms of group size and replication factor. Figures E.1, E.2 and E.3 illustrate the emerging trends as group size and the replication factor increase.

| Group size | | 5 | 8 | 15 | 25 | Reliability(%) |
|---|---|---|---|---|---|---|
| Retrieval Mode | $\mathcal{R}$ | Responsiveness (ms) | | | | |
| Fast | 2 | 8.27 | 16.65 | 16.19 | 13.09 | 99.48 |
| | 4 | 9.65 | 7.63 | 19.97 | 5.65 | 99.83 |
| | 6 | 10.90 | 13.97 | 14.56 | 14.91 | 100.00 |
| Parallel | 2 | 16.34 | 17.33 | 18.71 | 7.90 | 99.60 |
| | 4 | 11.02 | 7.88 | 21.79 | 6.29 | 99.93 |
| | 6 | 26.87 | 20.88 | 16.61 | 14.84 | 100.00 |
| Safe | 2 | 17.01 | 18.38 | 25.85 | 8.75 | 99.78 |
| | 4 | 18.62 | 9.52 | 24.71 | 9.05 | 99.93 |
| | 6 | 28.36 | 19.23 | 21.89 | 17.54 | 100.00 |

**Table E.1:** The effect replication factor on retrieval response time for different group sizes

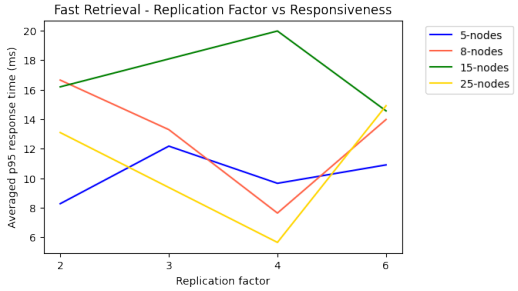


**Figure E.1:** Fast retrieve - response versus replication factor and group size

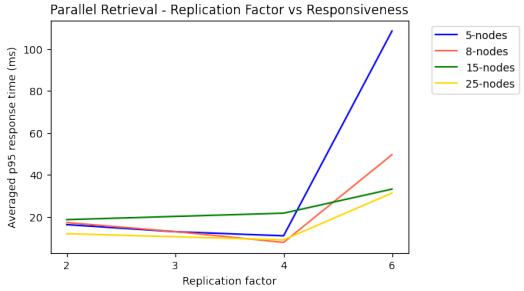


**Figure E.2:** Parallel retrieve - response versus replication factor and group size

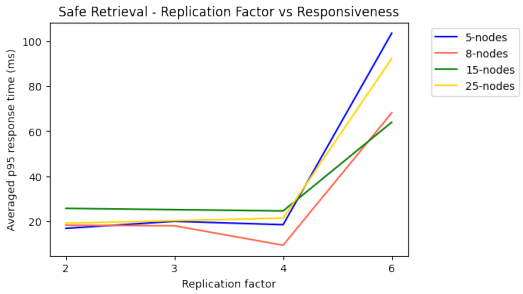


**Figure E.3:** Safe retrieve - response versus replication factor and group size

Table E.2 shows Nomad's group storage results in terms of group size and replication factor. Figures E.4 and E.5 illustrate the emerging trends as group size and the replication factor increase.

| Group size | | 5 | 8 | 15 | 25 | Reliability (%) |
|---|---|---|---|---|---|---|
| Storage Mode | $\mathcal{R}$ | Responsiveness (ms) | | | | |
| Fast | 2 | 20.35 | 30.71 | 38.81 | 40.48 | 99.99 |
| | 4 | 16.13 | 27.65 | 78.74 | 67.53 | 99.99 |
| | 6 | 23.46 | 29.61 | 67.85 | 66.73 | 100 |
| Safe | 2 | 28.54 | 32.65 | 47.97 | 32.55 | 100.00 |
| | 4 | 24.48 | 53.39 | 55.14 | 52.89 | 100.00 |
| | 6 | 31.65 | 66.60 | 127.91 | 232.00 | 100 |

**Table E.2:** The effect replication factor on storage response time for different group sizes

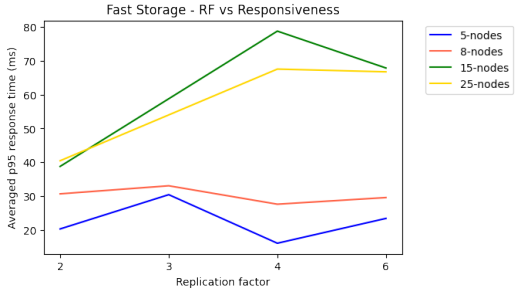


**Figure E.4:** Fast store - response versus replication factor and group size

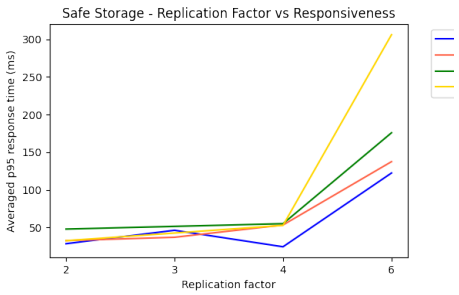


**Figure E.5:** Safe store - response versus replication factor and group size

# Appendix F

# AWS DynamoDB P2P techniques

| Problem | Technique | Advantage |
|---|---|---|
| Partitioning | Consistent hashing | Incremental scalability |
| High-availability (HA) | Vector clocks with reconciliation during reads | version size is decoupled from update rates |
| Disaster recovery (temporary failures) | Sloppy quorum and hinted handoff | Provides HA and durability when replicas on failed nodes become inaccessible |
| Disaster recovery (permanent failures) | Merkle trees | Synchronises divergent data |
| Membership | Gossip based membership protocol and failure detection | Preserves system symmetry and reduces the need for central membership and liveness information |

**Table F.1:** Summary of peer-to-peer techniques used in DynamoDB [53]

# Appendix G

# Project Complications

Apart from technical challenges, two external challenges were faced during the implementation period.

## G.1. Covid-19 Pandemic

The implementation was evaluated on as many resources as possible, which due to the covid-19 pandemic, was extremely limited. Like millions of others around the world, the bulk of this work was completed remotely, with whatever resources were available.

## G.2. Working Abroad

The author of this work lives in the Netherlands and therefore did not have access to the equipment at the university where this study was undertaken.

# Appendix H

# Nomad Repository

The Nomad implementation repository is openly available on gitlab.com. The repository provides a useful wiki, demo videos and a README with instructions on how to run the Nomad application locally. View the repository at https://gitlab.com/iggydv12/nomad.