PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

AUTONOMOUS RESOURCE MANAGEMENT FOR CLOUD-ASSISTED PEER-TO-PEER BASED SERVICES

Hanna Kavalionak

Advisor:

Prof. Alberto Montresor

Università degli Studi di Trento

December 2013

Abstract

Peer-to-Peer (P2P) and Cloud Computing are two of the latest trends in the Internet arena. They both could be labelled as large-scale distributed systems, yet their approach is completely different: based on completely decentralized protocols exploiting edge resources the former, focusing on huge data centres the latter. Several Internet startups have quickly reached stardom by exploiting cloud resources. Instead, P2P applications still lack a well-defined business model. Recently, companies like Spotify and Wuala have started to explore how the two worlds could be merged by exploiting (free) user resources whenever possible, aiming at reducing the cost of renting cloud resource.

However, although very promising, this model presents challenging issues, in particular about the autonomous regulation of the usage of P2P and cloud resources. Next-generation services need the possibility to guarantee a minimum level of service when peer resources are not sufficient, and to exploit as much P2P resources as possible when they are abundant. In this thesis, we answer the above research questions in the form of new algorithms and systems. We designed a family of mechanisms to self-regulate the amount of cloud resources when peer resources are not enough.

We applied and adapted these mechanisms to support different Internet applications, including storage, video streaming and online gaming. To support a replication service, we designed an algorithm that self-regulates the cloud resources used for storing replicas by orchestrating their provisioning. We presented CLIVE, a video streaming P2P framework that meet the real-time constraints on video delay by autonomously regulating the amount of cloud helpers upon need. We proposed an architecture to support large scale on-line games, where the load coming from the interaction of players is strategically migrated between P2P and cloud resources in an autonomous way. Finally, we proposed a solution to the NAT problem that employs cloud resources to allow a node behind it to be seen from outside.

Using extensive simulations, we showed that hybrid infrastructures can reduce the economical effort on the service providers, while offering a level of service comparable with centralized architectures. The results of this thesis

proved that the combination of Cloud Computing and P2P is one of the milestones for next generation distributed P2P-based architectures.

Keywords
[Peer-to-Peer; Cloud Computing; Distributed Systems; Self-regulation]

"To strive, to seek, to find, and not to yield."
"Ulysses" by Alfred, Lord Tennyson

To my family

Acknowledgements

This thesis would not have been possible without the help and support of many people around me, unfortunately only a small part of which I have space to acknowledge here.

First of all, I am deeply grateful to Professor Alberto Montresor for giving me the opportunity to work under his supervision, for encouraging my research and for allowing me to grow as a scientist. A great influence on my work was made by Alexei Ivanov, who helped me a lot in understanding what it takes for being a researcher. Thank you a lot for the philosophical discussions and support.

I would like to thank the people of Computer Systems Laboratory research group at SICS, Sweden. In particular a special thank you to Amir H. Payberah for the delightful and very inspiring work we have done together. Additional thank you to Fatemeh Rahimian and Amir H. Payberah for their incredible hospitality.

I would like to thank my friends and colleagues in Trento for one of the best period in my life. A particular thank you to Mikalai Tsytsarau and Yury Zhauniarovich for their patience and invaluable advices. I wish to thank much more people but unfortunately I cannot mention here all the names. Therefore, I thank all my friends for the great help and fruitful motivational "lunch-time" discussions.

My deepest thanks and love to the most important persons in my life. I thank my grandparents for the curiosity I feel about life and to my parents for their absolute support of all my endeavours. I would like to thank my sister Natallia for supporting and cheering me up during the hard times. Finally, I would like to thank to my dear Emanuele for his true love, great support, patience and deep understanding all this time. Writing this thesis without him would have been a million times harder.

Contents

1 Introduction **1**
1.1 Research objectives . . . 3
1.1.1 Massively multiplayer online games . . . 3
1.1.2 Persistent storage support . . . 5
1.1.3 Live Media Streaming . . . 6
1.1.4 NAT-aware peer-sampling . . . 7
1.2 General approach . . . 8
1.3 Main Contributions . . . 10
1.4 Thesis Organisation . . . 12
1.5 List of Publications . . . 12

2 Background **15**
2.1 Peer-to-peer . . . 15
2.2 Cloud Computing . . . 18
2.3 Autonomous systems . . . 19

3 Persistent storage support **23**
3.1 Problem statement . . . 23
3.2 The algorithm . . . 25
3.3 Evaluation . . . 29
3.4 Related Work . . . 32

4 Video Streaming **35**
4.1 Problem statement . . . 35
4.2 System architecture . . . 38
4.2.1 The baseline model . . . 38
4.2.2 The enhanced model . . . 39

4.3 System management . . . 41
4.3.1 The swarm size and upload slot distribution estimation . 42
4.3.2 Estimating the number of infected peers . . . 45
4.3.3 AHs management model . . . 48
4.3.4 Discussion on T_{lcw} . . . 49
4.4 Experiments . . . 51
4.4.1 Experimental setting . . . 52
4.4.2 The effect of T_{lcw} on system performance . . . 53
4.4.3 PH load in different settings . . . 54
4.4.4 Economic cost . . . 55
4.4.5 Accuracy evaluation . . . 59
4.5 Related work . . . 60
4.5.1 Content distribution . . . 60
4.5.2 Self-monitoring and self-configuration systems . . . 62

5 Virtual game environment 65
5.1 Architecture . . . 65
5.1.1 Distributed MMOG . . . 66
5.1.2 The proposed architecture . . . 67
5.1.3 State action manager . . . 67
5.1.4 Virtual nodes . . . 69
5.1.5 Replication and fault tolerance . . . 71
5.2 Problem statement . . . 72
5.2.1 Quality of service . . . 73
5.2.2 Problem statement . . . 76
5.3 Virtual node allocation . . . 76
5.3.1 Load prediction . . . 77
5.3.2 Virtual Nodes Assignment . . . 78
5.3.3 Migration . . . 81
5.4 Experimental Results . . . 82
5.4.1 Workload Definition . . . 82
5.4.2 Tuning the Virtual Nodes Dimension . . . 84
5.4.3 Tuning the Capacity Threshold . . . 88
5.4.4 Cost over the number of players . . . 89
5.4.5 Qos and Cost Trade-off . . . 92

5.4.6 Behaviour over different churn levels 93
5.5 Related work 95
5.5.1 Hybrid MMOG architectures 95
5.5.2 Hybrid P2P and cloud architectures 97

6 NAT-traversal systems 99
6.1 Problem statement 99
6.2 Nat-traversal system model 101
6.2.1 Modeling AF 102
6.2.2 The impact of LF 104
6.3 Nat-aware peer-sampling 106
6.3.1 Parents management 108
6.3.2 Parent Changing Policy 109
6.4 Related work 110

7 Conclusion 113

Bibliography 117

List of Tables

3.1 Parameters used in the evaluation 29
4.1 Slot distribution in freerider overlay. 52
5.1 Table of symbols 73

List of Figures

1.1 System regulation mechanisms. 9

2.1 Cloud Computing layers 18

3.1 Overlay size oscillation 26
3.2 Overlay size in case of network oscillates daily between 0 and 1024 peers. The single experiment lasted 3 days. 30
3.3 Overlay size in case of network oscillates daily between 0 and 1024 peers. Zooming over a single day. 31
3.4 Cloud in-degree for CLOUDCAST and our protocols in case of network oscillates daily between 0 and 1024 peers. 32
3.5 Deviation of the overlay size from the *sufficient* threshold under different levels of churn with variable epoch intervals. 33

4.1 The baseline model. 37
4.2 The enhanced model. 38
4.3 Live streaming time model. 46
4.4 Calculating the number of peers that is economically reasonable to serve with PH utilization instead of running an additional AH. 48
4.5 The percentage of the peers receiving 99% playback continuity with different values of T_{lcw} (measured in number of chunks). . . 53
4.6 Average playback delay across peers with different values of T_{lcw} (measured in number of chunks). 54
4.7 The cumulative PH load with different values of T_{lcw}. 54
4.8 The cumulative PH load with different values of churn rates($LCW =$ 40 chunks). 55
4.9 Number of AHs in different settings and scenarios. 56

4.10 PH load in different scenarios with dynamic changes of the number of AHs. . . . 57
4.11 The cumulative total cost for different setting and scenarios. . . 58
4.12 Avg. estimation error. . . . 60
4.13 The comparison between the real number of infected nodes and the estimated ones. . . . 61

5.1 Overall architecture 67
5.2 SAM architecture 68
5.3 Time management 77
5.4 Migration of a VN from the node A to node B 81
5.5 Objects and avatars placement in the virtual environment 84
5.6 Number of players over time (up) and correspondent load variation (down) 85
5.7 Average clients per entity per minute plotted in log-log 86
5.8 Probability density function of RTTs 87
5.9 95th percentile of MT with different amount of objects 87
5.10 Percentage of overloading over LF_{up} for different eps 88
5.11 Total simulation cost over LF_{up} with different eps 89
5.12 Cost per minute over time considering workload 1 90
5.13 Cost per minute over time considering workload 2 90
5.14 Total cost with workload 1 91
5.15 Total cost with workload 2 92
5.16 QoS over time with different QoS thresholds 93
5.17 Cost per minute with different QoS thresholds 94
5.18 QoS over time with different churn levels 94
5.19 Cost over time with different churn levels 95

6.1 AF of a child node vs. number of its parents for different system configurations and AS failure rates. 103
6.2 AF derivative for different number of parents in layout without AS and when all the parents belong to the same AS. $\alpha = 0.1$ and $r = 0.1$. 104
6.3 The LF impact for different ratio between public and private nodes depends on the average number of parents per child. . . . 106

6.4 Maximum number of parent nodes in a child descriptor for maximum allowed LF 107
6.5 Redundant and Critical thresholds for $\gamma = 0.05$ and $LF_{max} = 1\%$ 107

List of Algorithms

3.1 Resource management algorithm 28
3.2 idleTime and recovery procedures 28
4.1 Estimating the swarm size, upload slot distribution, and T_{lcw} average. 43
4.2 Estimating the swarm size, upload slot distribution, and T_{lcw} average. 44
4.3 Lower bound for the diffusion tree size. 47
5.1 Server's load estimation 78
5.2 Virtual Nodes Selection 79
5.3 Destination Selection 80
6.1 Algorithm executed by child nodes 108

Chapter 1

Introduction

Cloud computing represents an important technology to compete in the worldwide information technology market. According to the U.S. Government's National Institute of Standards and Technology [1], cloud computing is *'a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services)'*. Usually cloud services are transparent to the final user and require minimal interactions with the cloud provider. By giving the illusion of infinite computing resources, and more importantly, by eliminating upfront commitments, small- and medium-sized enterprises can play the same game as web behemoths such as Microsoft, Google and Amazon.

Cloud Computing promises exceptional levels of service quality (QoS), but all it comes with a cost, possibly requiring significant financial investments. In addition, despite all the advertising, cloud platforms can suffer from major outages, as recent data loss incidents have shown [2]. Hence, enterprises must apply and considers mechanisms to protect their data in the cloud, to guarantee smooth and performant service delivery. Moreover, by relying only on a cloud-based architecture, a user fully depends on the economical policy of the cloud service provider.

The rise of cloud computing has progressively dimmed the interest in another Internet trend of the first decade of this century: the peer-to-peer (P2P) paradigm. P2P systems are composed by nodes (i.e. peers) that simultaneously play the role of both clients and servers. In this paradigm, tasks are managed in

[1] http://csrc.nist.gov

[2] http://www.businessinsider.com/amazon-lost-data-2011-4

a distributed fashion without any point of centralization. The lack of centralization provides scalability, while exploitation of user resources reduces the service cost. However, P2P architectures are sensitive to users behaviour. Seasons, and even the time of the day influence the amount of peers in the network. In case the number of peers is low, a P2P-based service can suffer from reduced performances and the whole service can even be unavailable in case of extremely small amount of users. Hence, P2P-based services have limitations concerning availability and reliability.

P2P held similar promises with respect to cloud computing, but with relevant differences. While P2P remains a valid solution for cost-free services, the superior QoS capabilities of the cloud makes it more suitable for those who want to create novel web businesses and cannot afford to lose clients due to the best-effort philosophy of P2P. One question arises above all others: is it possible to join the forces of both P2P and cloud technologies and eliminate their weaknesses? Our works tries to answer positively to this question.

Our research belongs to a "middle ground" that combines the benefits of both paradigms, offering highly-available services based on the cloud, while lowering the economic cost by exploiting peers whenever possible. As an example, let us consider a large application running on a cloud, such as a real-time online game. In such application, the current state of the game is stored on a collection of machines rented from the cloud. As the number of players increases, it would be possible to transfer part of the game state to the machines of the player themselves, by exploiting a P2P approach. This would reduce the economical costs of maintaining a cloud infrastructure, yet providing a stable level of service when peer resources are insufficient or entirely absent.

In this context, we tackle the issue of the resource management between the cloud and the peers. In other words, the system should use available P2P resources and, at the same, time rely on the cloud whenever peer resources are not enough to support the service. The core idea of our approach is to manage the resources by leveraging the self-organization properties of P2P structured and unstructured overlay networks. The whole set of self-* properties of these overlays yields to improved scalability, robustness and performance of systems and tolerance to network *churn*.

1.1 Research objectives

In this thesis, we apply cloud-assisted P2P computing model to improve the performance of multiple application scenarios, such as massively multiplayer online games [46], persistent storage support [47] and live media streaming systems [75]. A common trait of general P2P based application is that peers are required to communicate with each other over the Internet directly. However, as a large fraction of peers in the Internet are behind Network Address Translation (NAT) gateways and firewall systems, it is highly important to design systems that overtake this limitation and therefore be applicable in real contexts. To this end, we also demonstrate the benefits that the hybrid approach can bring in the field of NAT-aware protocols (Chapter 6). Each of these applications has its own particular issues in P2P/Cloud Computing combination exploitation.

1.1.1 Massively multiplayer online games

In the last years, on-line gaming entertainment has acquired an increasing popularity among both industrial and academic researchers. This attention is justified by the economic growth of the field, in particular regarding *massively multiplayer online games*(MMOG). The market of MMOGs has been evaluated around 5 billion dollars in 2010, with 20 million users worldwide[3].

MMOGs are large-scale distributed applications that allow large communities of users to share a real-time *virtual environment* (VE). MMOGs operators provide the necessary hardware infrastructure to support such communities, obtaining their profit from the fees users pay periodically to enjoy the game. Regardless of their business model, operators' profit is strictly linked to the number of users who participate in the MMOG; in fact, the more popular is a game, the more attractive it becomes for new users. For this reason, offering an acceptable level of service while assuring the infrastructure to sustain a large number of users is a core goal for MMOG operators.

Currently, most MMOGs rely on a client/server architecture. This centralized approach enables a straightforward management of the main functionalities of the virtual environment, such as user identification, state management, synchronization between players and billing. However, with larger and larger

[3] `http://www.mmodata.net`, August 2012

numbers of concurrent users, centralized architectures are hitting their scalability limits, especially in terms of economical return for the operators.

Indeed, server clusters have to be bought and operated to withstand service peaks, also balancing computational and electrical power constraints. A cluster-based centralized architecture concentrates all communication bandwidth at one data center, requiring the static provisioning of large bandwidth capability. This may lead to *over-provisioning*, which leaves the MMOG operators with unused resources when the load on the platform is not at its peak.

On-demand resources provisioning may alleviate the aforementioned scalability and hardware ownership problems [67, 68]. The possibility of renting machines lifts the MMOG operators from the burden of buying and maintaining hardware, and offers the illusion of infinity resource availability, allowing (potentially) unlimited scalability. Also, the pay-per-use model enables to follow the daily/weekly/seasonal access patterns of MMOGs.

However, the exploitation of cloud computing presents several issues. The recruiting and releasing of machines must be carefully orchestrated in order to cope with start-up times of on-demand resources and to avoid incurring on unnecessary expenses due to unused servers. Further, besides server time, bandwidth consumption may represents a major expense when operating a MMOG. Thus, even if an infrastructure based entirely on on-demand resources is feasible, the exploitation of user-provided resources may further reduce the server load and increase the profit margin for the MMOG operator.

These aspects have been deeply investigated in the research community in the last decade. Mechanisms to integrate user-provided resource in an MMOG infrastructure naturally evolved from the peer-to-peer (P2P) paradigm. However, P2P-based infrastructures require additional mechanisms to suit the requirements of a MMOG. When a peer leaves the system, its data must be transferred somewhere else; given that the disconnection may be abrupt, replication mechanism must guarantee that data will not be lost. The lack of a central authority makes security enforcement more difficult. Moreover, user machines typically have heterogeneous constraints on computational, storage and communication capabilities, making them complex to be exploited.

1.1.2 Persistent storage support

In recent years, P2P architectures became a popular basis for distributed storage applications. In these applications replicated data is distributed among the peers. Nevertheless, if the peer that maintains a particular piece of data fails or goes off-line, the data may be lost or become temporarily unavailable until the peer returns on-line. The specific problems of P2P networks, i.e., churn rate and unreliable user resources raise an important issue, which is how to maintain a certain level of reliability in P2P-based storages.

To improve availability and durability, the usual straightforward approach is to replicate data several times in the network. Increased replication allows to decrease the probability the data to be lost or damaged due to the network churn or users behaviour. At the same time, replication causes additional issues, such as replica synchronization and resource look-up. An higher number of data replicas increases the amount of update messages exchange between replica holders. Hence, there is a trade-off: the more replicas, the higher the traffic overhead. Furthermore, a P2P system may have a limited amount of storage available, so increasing the number of replicas of an object decreases the number of replicas of another one. Finally, the replication cannot solve the issue of small P2P overlays. When the number of peers in the P2P network is small the whole network can be partitioned into separate overlays and the data replicated between peers of one overlay can be not accessible by peers of another overlay.

The solution for the problems mentioned above can be to store a replica of each piece of data in a server. For example, the role of such server can be played by the cloud [106, 63]. Cloud storage servers can guarantee the service to be reliable while P2P resources are not available, but the cost grows proportionally to the size of storage space used. Moreover, while the aforementioned works demonstrate the benefits of hybrid P2P/Cloud computing approach, there is still space to improve. Indeed, the cloud is used all the time, even when there are enough P2P resources to support the service itself. In fact, the use of the cloud can be reduced, therefore decreasing the cost of the service, with an effective resource management inside the P2P overlay.

As a consequence, one of the problems we address in our work is the regulation of storage resources between cloud and P2P technologies. The problem of an effective resource management is a complex problem, composed with differ-

ent components, namely optimization of a system architecture combined with an effective resource monitoring and provisioning. Moreover, as P2P overlays are usually highly dynamic and affected by uncontrollable users behaviour, the resource management mechanism should be resilient to the rapid changes of the overlay state.

1.1.3 Live Media Streaming

Peer-to-peer (P2P) live streaming is becoming an increasingly popular technology, with a large number of academic [27, 28, 72, 74, 78] and commercial [56, 80, 87] products being designed and deployed.

In such systems, one of the main challenges is to provide a good QoS in spite of the dynamic behavior of the network. For live streaming, QoS means *playback continuity* and short *playback delay*. To obtain high playback continuity, or smooth media playback, nodes should receive chunks of the stream with respect to certain timing constraints; otherwise, either the quality of the playback is reduced or its continuity is disrupted. Likewise, to have a small playback delay, nodes should receive chunks of the media that are close in time to the most recent part of the media delivered by the provider.

There is a trade-off between these two properties: it is possible to increase the playback continuity by adopting larger stream buffers, but at the expense of increasing the delay. On the other hand, reducing playback delay requires that no bottlenecks are present in either the upload bandwidth capabilities of the media source and the aggregated upload bandwidth of all nodes in the *swarm*, i.e., the nodes forming the P2P streaming overlay [95].

Increasing the bandwidth at the media source is not always an option, and even when possible, bottlenecks in the swarm have proven to be much more disruptive [53]. One approach to solve this issue is adding auxiliary *helpers* to accelerate the content propagation. A helper could be an *active* computational node that participates in the streaming protocol, or a *passive* storage service that just provides content on demand. The helpers increase the total upload bandwidth available in the system, thus, potentially reducing the playback delay. Both types of helpers could be rented on demand from an IaaS (Infrastructure as a Service) cloud provider, e.g., Amazon AWS [4].

[4] "Amazon Web Services" http://http://aws.amazon.com/

Considering the capacity and the cost of helpers, the problem consists in selecting the right type of helpers (passive vs. active), and provisioning their number with respect to the dynamic behavior of the users. If few helpers are present, it could be impossible to achieve the desired level of QoS. On the other hand, renting helpers is costly, and their number should be optimized.

This P2P-cloud hybrid approach has already been pursued by a number of P2P content distribution systems [95, 63]. However, adapting the cloud-assisted approach to P2P live streaming is still an open issue. Live streaming differs from content distribution for its soft real-time constraints and a higher dynamism in the network, as the users may be zapping between several channels and start or stop to watch a video at anytime [88, 89].

Thus, the problem to be solved is how to guarantee the desired QoS for live media streaming in the highly dynamic cloud assisted distributed network. The particular interest is an optimization of total economic cost caused by cloud computing.

1.1.4 NAT-aware peer-sampling

One important service for distributed system support is the Peer Sampling Service (PSS) [75, 47]. PSSs periodically provide nodes with a uniform sampling of the live nodes in the network, i.e., partial view. This allows to keep the P2P network updated and connected. Moreover, PSSs increases the resilience of the overlay to the network churn.

A relevant limitation in the usage of PSSs is that in the Internet, a large fraction of nodes are behind NATs gateways and firewall systems. Hence, such nodes cannot establish direct connections to each other, and they become underrepresented in partial views, and traditional PSS become biased [49].

Several works employs Peer-to-Peer (P2P) principles to provide an effective PSS in presence of NAT. Nodes can be divided into two types: *private* and *public* [21]. Private nodes are behind a NAT or a firewall system and cannot be reached via a direct communication. Conversely, public nodes can directly access the network. In this context, they define *parent* nodes as public nodes through which the *child* nodes (private ones) are reachable from outside the NAT.

However, state-of-art solutions do not tackle the following two relevant is-

sues:

- *Public nodes overloading*. Since most of the shuffling is done by public nodes, the ratio between public and private nodes in the system is an important parameters for the protocol. In case the ratio is low, existing public nodes may become overloaded by the increasing intensity of shuffling requests from private nodes.

- *Reliability of the service*. In case of small networks, the original protocol cannot provide an effective PSS, for instance when the amount of public nodes is small or all the available nodes are private. In this case some private nodes can become isolated from the network.

These limitations can significantly decrease the reliability of the distributed applications relying on the PSS service. Hence we explore this direction by resolving the problem aforementioned.

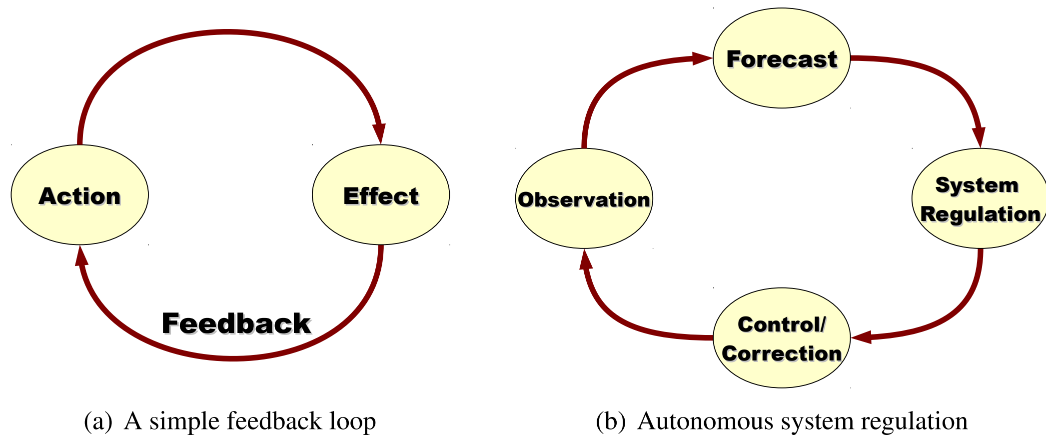
1.2 General approach

The peer-to-peer and cloud computing paradigms have both advantages and disadvantages, in terms of availability, economical effort, durability and reliability.

Our solution tries to combine the strengths of peer-to-peer with cloud computing, reducing (as far as possible) their weak points. The goal is to provide, within a specific application domain, an adaptive system that can either manually or programmatically control the trade-off between QoS and the economical effort.

In general, we often recurred to a *feedback loop* (Figure 1.1(a)) approach to manage the resources. This approach is not new in the field of autonomous systems. Nevertheless, we adopted and applied it to our issues (Figure 1.1(b)). We generally decomposed the resource regulation loop into several steps:

- *Observation.* In order to tune the resource exploitation it is important to know the up-to-date state of the target system, for example the amount of available P2P resources or the current churn rate. Hence, the system periodically runs monitoring protocols and in distributed or centralized way collects the required data. Distributed system monitoring provides each



(a) A simple feedback loop (b) Autonomous system regulation

Figure 1.1: System regulation mechanisms.

node with an aggregated knowledge about the system [76]. In case of centralized monitoring the knowledge is accumulated in the central point, for example specially rented server.

- *Forecast.* Based on the previously and current collected data the system builds a prediction function for the parameter of interest: as an example, churn rate or upload bandwidth requirement. The prediction function plays an important role in the system regulation, since it prevents the system to eventually arrive in critical states by giving the possibility of reacting in advance.

- *System Regulation.* Based on the predicted values the system is optimized in order to satisfy the required QoS while trying to keep the economical costs at the minimum. For example, if available P2P resources are expected to be not enough to meet the QoS requirements, cloud resources are included in the system in advance.

- *Control/Correction.* Periodically the predicted values are compared with real ones. In case the difference between predicted and currently observed data is more than a maximum allowed error, the system corrects the prediction function according on the new observed data.

1.3 Main Contributions

This thesis presents research on the methods and mechanisms of enabling and achieving self-managements for internet services built on cloud-based P2P computing. We answered these problems in the form of new algorithms and architectures, aiming different application contexts.

Massively multiplayer online games. We designed a MMOG architecture where an operator can decide to have an infrastructure more reliable and responsive (for example for particularly interactive MMOGs) or to reduce the economical effort and provide a less powerful infrastructure, perhaps suitable for less interactive MMOGs [46]. In other words, we let the operator to choose on how to make profit, either by offering a more controllable service, or by saving on the cost of infrastructure or in a point in the middle between the two.

Our infrastructure for the management of MMOG objects is based on a Distributed Hash Table [92]. This choice allowed us to lively migrate objects among cloud- or user-provided resources with few impact on the QoS. We provided a model for the QoS and the economical cost of the MMOG on top of the integrated architecture. In addition, we developed an orchestration algorithm that, according to the preferences defined by the operator, manages the migration of the objects. Finally, we tuned and evaluated the architecture through extensive simulations. The results showed that slightly decreasing the quality of service yield to cost reductions up to 60%.

Persistent storage support. We applied the concepts of cloud-assisted P2P systems and autonomous resource regulation to persistent storage support [47]. We propose a protocol based on the distributed resource provisioning, where each node maintains a resource provisioning function built on the degree of data replication in the P2P nodes.

The regulation system considers three replicas thresholds: Critical, Sufficient and Redundant. Critical threshold indicates high risk of data loss. In case the Critical state is expected by the provisioning function, the data is replicated into the cloud holder in advance. The Sufficient threshold shows the regular level of replication that corresponds to desired QoS. The replication degree between Critical and Sufficient requires the system to replicate data into available

P2P holders. Finally, the Redundant threshold allows to reduce the replication redundancy and avoid network traffic overhead. The degree of replication is expected to stay in range between Sufficient and Redundant thresholds. The thresholds are numerically defined by the type of the application and by the desired QoS.

Live media streaming. We designed and evaluated CLIVE [75], a novel cloud-assisted P2P live streaming system that guarantees a predefined QoS level by dynamically renting helpers from a cloud infrastructure. We modelled the problem as an optimization one, where the constraints are given by the desired QoS level, while the objective function is to minimize the total economic cost incurred in renting resources from the cloud. We provide an approximate, on-line solution that is: (i) adaptive to dynamic networks and (ii) decentralized.

CLIVE extends existing *mesh-pull* P2P overlay networks for video streaming [28, 72, 110], in which each node in the swarm periodically sends its data availability to other nodes, which in turn pull the required chunks of video from the neighbours that have them. The swarm is paired with the CLIVE *manager* (CM), which participates with other nodes in a gossip-based aggregation protocol [40, 64] to find out the current state of the swarm. Using the collected information in the aggregation protocol, the CM computes the number of active helpers required to guarantee the desired QoS. CLIVE includes also a passive helper, whose task is to provide a last resort for nodes that have not been able to obtain their video chunks through the swarm.

To demonstrate the feasibility of CLIVE, we performed extensive simulations and evaluated our system using large-scale experiments under dynamic realistic settings. We show that we can save up to 45% of the cost by choosing the right number of active helpers compared to only using a passive helper to guarantee the predefined QoS.

Nat-aware peer-sampling. We designed a novel approach that can be applied to resolve the NAT problem for real network P2P applications. The proposed mechanism successfully combines P2P and cloud computing technologies for an effective NAT-aware peer-sampling.

As well as the authors of Croupier protocol [21], our approach considers two types of entities: child and parent nodes. A child node is connected with nodes

on the other side of the NAT via parent nodes. The role of the parent nodes can be played by P2P or cloud entities. To keep the connection open the child periodically sends a keep-alive message to the parent. On one side, the more parent nodes a child node has, the lower the risk that all of them at the same time are off-line. On the other hand, a high number of parents increases the costs of network support due to increasing traffic of keep-alive messages.

In this context we investigate the trade-off between availability of child nodes and alive messages traffic. We developed a model that allows the system to autonomously switch between P2P and cloud parent nodes based on the upload bandwidth overhead and node availability.

1.4 Thesis Organisation

The rest of the thesis is organised as follows. In Chapter 2 we present the required background for this thesis project. The rest of the thesis deals with the problem of self-management for dynamic hybrid distributed systems. We present an autonomous regulation mechanism applied for persistent storage support in Chapter 3. In Chapter 4 we present CLIVE, a novel cloud-assisted P2P live streaming system, while Chapter 5 presents a cloud-assisted MMOG. Chapter 6 discusses the possibilities of cloud assisted NAT-aware peer-sampling. In Chapter 7 we conclude the thesis.

1.5 List of Publications

The list of papers published in this work are:

- H. Kavalionak and A Montresor. P2P and cloud: A marriage of convenience for replica management. In *Proc. of IWSOS'12*, pages 60–71. Springer, 2012
- A.H. Payberah, H. Kavalionak, V. Kumaresan, A. Montresor, and S. Haridi. CLive: Cloud-Assisted P2P Live Streaming. In *Proc. of the 12th Conf. on Peer-to-Peer Computing (P2P'12)*, 2012
- H. Kavalionak, E. Carlini, L. Ricci, A. Montresor, and M. Coppola. Integrating peer-to-peer and cloud computing for massively multiuser online games. *Peer-to-Peer Networking and Application*, pages 1–19, 2013

- A.H. Payberah, H. Kavalionak, A. Montresor, J. Dowling, and S. Haridi. Lightweight gossip-based distribution estimation. In *Proc. of the 15th IEEE International Conference on Communications (ICC'13)*. IEEE, June 2013
- A.H. Payberah, H. Kavalionak, A. Montresor, and S. Haridi. CLive: Hybrid P2P-Cloud Live Streaming System. *IEEE Transactions on Parallel and Distributed Systems*, 2013. Submitted 01-10-2013

Chapter 2

Background

The content of this chapter is divided into two parts. The first part (Sections 2.1 and 2.2) presents an overview on the main topics considered in this thesis, like P2P and Cloud Computing. The second part (Section 2.3) provides an overview on the main issues related to the field of the autonomous resource management for the services based on the distributed architecture.

2.1 Peer-to-peer

Generally speaking, a P2P network can be modelled like a collection of peers that collaborate with each other in order to realize and participate to a service at the same time. This collaboration often results in peers playing the roles of both clients and servers of an Internet application. In other words, a peer consumes resources of the other peers when acting as a client, while provide its own resources to other peers when acting as a server.

By reducing the load on centralized servers, P2P-based solutions present several attractive advantages. First, P2P techniques are inherently scalable – the available resources grow with the number of users. Second, upon peer failures, P2P networks are able to self-repair and reorganize, hence providing robustness to the infrastructure. Third, network traffic and computation is distributed among the users involved, making difficult the creation of bottlenecks. Nevertheless, peers are not reliable, as they may join and leave the system at will. This phenomenon of peer perturbation is known as network *churn*.

P2P networks usually implement an abstract *overlay topology* over existing physical networks. Nodes in a network overlay are connected through logical

links. Each link corresponds to a path in the underlying physical network. To join the network, a peer must connect to another peer already participating to the network, so to acquire knowledge about others peers populating the network. P2P overlays can be classified in structured and unstructured, according on how the nodes in the network overlay are linked to each other.

Unstructured overlays. In unstructured overlays, link between nodes are established arbitrarily. One of the most notable example of unstructured P2P networks is based on *epidemic* or *gossip* protocols [99, 100]. The popularity of these protocols is due to their ability to diffuse information in large-scale distributed systems even when large amount of nodes continuously join and leave the system. Such networks have proven to be highly scalable, robust and resistant to failures.

The idea behind epidemic protocols takes inspiration from both the biological process of epidemic spreading of viruses, and the social process of spreading rumors [24]. At the beginning, a single entity (peer) is infected. At each communication round, each infected peer tries and infects another peer at random, roughly doubling the number of infected peers at each round. The process ends when all peers have been infected, i.e. when the information is spread.

One of the fundamental challenges of epidemic protocols is how to exploit local peer's knowledge to acquire global knowledge on the entire network. Ideally, each node should be aware of all peers in the system, and select randomly among them. But in large, dynamic networks that is unrealistic: the cost of updating all peers whenever a new peer joins or leaves is simply too high.

In order to keep the information about the network nodes up-to-date *peer sampling services* (PSSs) are widely used in the fields as information dissemination [63], aggregation [40] and overlay topology management [41]. PSSs provide each node with information about a random sample of the entire population of nodes in the network. In other words, each node maintains a partial view of the network and using only local data keeps the information about entire network up-to-date [42, 99].

The protocol execution of a PSS service is divided into cycles. In each cycle a node selects one node from its partial network view and asks this node to exchange subsets of their partial view. This process is also known as the *active thread* of the node. Together with the active thread, each node executes the

passive one, that is responsible for processing the shuffling requests coming from other network nodes. During the shuffling both of the nodes exchange a random subset of the network peers' descriptors from their local views.

Structured overlays. A second category of overlays, called structured overlays, is based on combining a specific geometrical structure with appropriate routing and maintenance mechanisms. The routing support is key-based, meaning that object identifiers are mapped to the peer identifier address space. Hence, an object request is routed to the nearest peer in the peer address space.

A specific type of such overlays is called Distributed Hash Table (DHT). The identifiers of a DHT are computed using a consistent hash function and each peer is responsible for a portion of the addresses space defined by the hash table. To maintain the overlay and to route requests, each node maintains a list of neighbors, the so called *routing table*. Examples of DHT overlays are Pastry [81], Chord [92], Kademlia [60].

One of the critical problems in P2P networks is the problem of network churn: peers leave the overlay arbitrarily and do not participate in the overlay for a predictable time. Hence the peers with routing table that contains the links to the joining/leaving peers need to be updated. In fact, in presence of leaving peers the latency of message exchange in the network increases and can lead to timeouts. Higher churn rate may lead to network partitioning [9].

Therefore, maintaining the routing tables in spite of churn is one of the issues in DHTs. Overlay maintenance require effective routing tables update according to the current overlay state and churn rate. Nevertheless, both the look-up query and overlay maintenance traffic compete for the underlying network resources, i.e., upload bandwidth.

The maintenance of the overlay requires the traffic to be proportional to the churn rate of the network. Hence, high levels of churn rate may cause excessive overhead and even possibly result in broken networks [8]. Therefore, the key feature in structured overlay maintenance is in the development of efficient maintenance algorithms that are able to deal with high network churn rate. For example, Kademlia [60] maintains a routing table where to each of the entry corresponds to a set of neighbours. This set of neighbours is ordered according to the last neighbour update, i.e. the recently connected neighbours are at the top of the list. Hence, when the current peer in the routing table leaves the net-

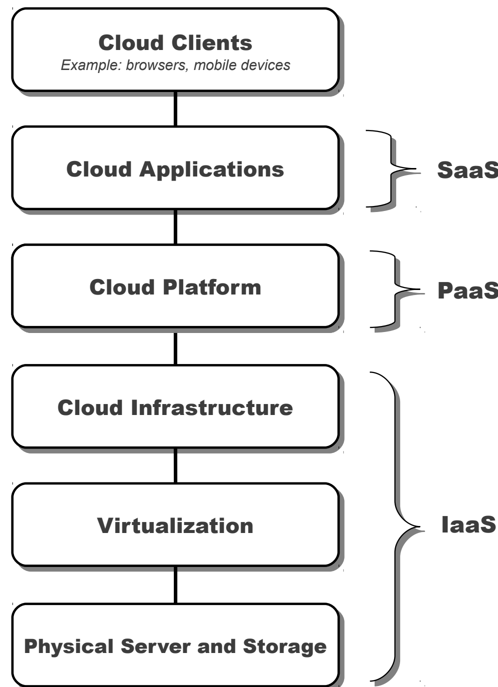


Figure 2.1: Cloud Computing layers

work the newer peer from the corresponding list takes its place. This "back up" list maintenance allows to reduce the effect of high network churn.

One more issue of DHT is to deal with dynamic data [12]. When a piece of data changes, its position in the address space may change as well. This often implies data to be exchanged among nodes. When the rate of changing of data is high, it may generates high network traffic.

2.2 Cloud Computing

Cloud computing has become a significant technology trend that has reshaped information technology processes and the marketplace, due to the ability of providing dynamically scalable and virtualized resources. By considering resources as commodity, cloud computing provides unquestionable advantages for users, including high availability, scalability, and pay-per-use models.

Cloud Computing technology can be viewed as a collection of services, which can be considered to be built on top of each other (Figure 2.1):

- *Software-as-a-Service (SaaS)*. Saas allows users to access applications re-

motely. Users typically use thin clients, whereas all computations are executed on the server (e.g Google docs, GMail).

- *Platform-as-a-Service (PaaS)*. Provides virtual machines with already included operating systems, tools and frameworks required for a particular application. Clients control the applications that run on the virtual hosting environment (possibly they having some control over the environment) but has no access to control the operating system, hardware or network infrastructure (e.g. Amazon Elastic MapReduce).

- *Infrastructure-as-a-service (IaaS)*. Provides full access to virtual machines. Virtual machines are usually characterized with guaranteed processing power and reserved bandwidth for storage and Internet access. Clients can operate directly on operative system, storage, deployed application and sometimes firewall. Nevertheless, the customer cannot control the underlying cloud infrastructure. The most notable example in this category is Amazon EC2.

One of the key aspects of cloud computing is the high exploitation of virtualization technologies. This allows to share resources among different applications optimizing the server utilization. In the context of cloud computing this means that a single hardware resource can be shared among different clients with different operating systems and configurations.

Trust, security and privacy are also currently hot topics in cloud-related research. The lack of control over the data and the globally distributed infrastructure can lead to potential data loss or data leaks to third parties, for example in case of security holes in remote servers. Also, untrustworthy clients may in principle have access to unlimited cloud computational resources, exposing the infrastructure to attacks like denial of service. For further details about research issues related to Cloud Computing, see [85].

2.3 Autonomous systems

The combination of *Self-monitoring* and *self-configuration* mechanisms are essential to manage large, complex and dynamic systems in an effective way. The self-monitoring function detects the current states of system components, while

the self-configuration function adapts system configuration according to the received information. Both mechanisms are often implemented based on the P2P communication model, due to its flexibility and scalability.

Self-monitoring Self-monitoring allows the system to have a view on its current utilization. One of the popular approaches for self-monitoring is the exploitation of P2P networks to achieve decentralized aggregation of monitoring data.

Decentralized aggregation in large-scale distributed systems has been well-studied in the past. A popular technique is the hierarchical approach [97, 105, 32, 45], where nodes are organized in tree-like structures, and each node in the tree monitors its children. Hierarchical approaches provide high accuracy results with minimum time complexity, but are extremely vulnerable to churn. An alternative approach is based on gossip protocols [40, 82, 34, 66, 98, 37], where information about nodes is exchanged between randomly selected partners and aggregated to produce local estimates.

In the field of gossip-based distribution estimation, Haridasan and van Renesse recently proposed EQUIDEPTH [34]. In this protocol, each node initially divides the set of potential values into fixed-size bins, and over the course of the execution, bins are merged and split based on the number of received values in each bin. In EQUIDEPTH, nodes send the entire current distribution estimate in each message exchange.

Following this research line, Sacha et al. proposed ADAM2 [82], an algorithm that provides an estimation of the cumulative distribution function (CDF) of a given attribute across the population of nodes. The proposed approach allows nodes to compute their own accuracy and to tune the trade-off between communication overhead and estimation accuracy.

Alternative gossip-based averaging techniques to overcome message loss, network churn and topology changes have been proposed by Eyal et al. [25] and Jesus et al. [43]. In LIMOSENSE [25], each node maintains a pair of values, e.g., a weight and an estimation, which is continuously updated during node communication. Jesus et al. [43] propose a technique where each node uses its current set of neighbours and maintains a dynamic mapping of value flows across the neighbours.

Self-configuration Self-configuration is the process that autonomously configures components and protocols according to specified target goals, e.g., reliability and availability. To self-tune according to on-going state, the system can use an external component that controls the system either via control loops [48] or in a decentralized way [5, 41, 47]. A relevant example of a self-configuration mechanism is T-MAN [41], an overlay topology management that uses a ranking function exploited locally by each peer to choose its neighbours.

Chapter 3

Persistent storage support

In order to strike a balance between service reliability and economical costs, we propose a self-regulation mechanism that focuses on replica management in cloud-based, peer-assisted applications. Our target is to maintain a given level of peer replicas in spite of churn, providing a reliable service through the cloud when the number of available peers is too low.

3.1 Problem statement

We consider a system composed by a collection of peer nodes that communicate with each other through reliable message exchanges. The system is subject to churn, i.e. node may join and leave at any time. Byzantine failures are not considered, meaning that peers follow their protocol and message integrity is not at risk. Peers form a random overlay network based on peer sampling service [42].

Beside peer nodes, an highly-available cloud entity is a part of the system. The cloud can be joined or removed from the overlay according to the current system state. The cloud can be accessed by other nodes to retrieve and write data, but cannot autonomously initiate the communication. The cloud is pay-per-use: storing and retrieving data with a cloud is connected with monetary costs.

The problem to be solved is the maintenance of an appropriate redundancy level in replicated storage. Consider a system where a collection of data objects are replicated in a collection of peers. A data object is *available* if at least one replica can be accessed at any time, and is *durable* if it can survive failures.

If potentially all peers replicating an object fail or go offline, or are temporary unreachable, the data may become unavailable or even be definitely lost. A common approach to increase durability and availability is to increase the number of replicas of the same data. Nevertheless, the replicas of the same data should be synchronized among each other and retrieved in case of failure. Hence, there is a trade-off: the higher amount of replicas, the higher the network overhead. Furthermore, a peer-to-peer system may have a limited amount of storage available, so increasing the number of replicas of an object can decrease the number of replicas of another one.

An alternative approach is to store a replica of each piece of data in the cloud [106]. This method improves reliability and scalability. Nevertheless, the economical cost grows proportionally to the size of storage space used.

In order to autonomously support the target level of availability and durability and reduce the economical costs we exploit a hybrid approach. The replicas of the same data object are organized into an overlay. To guarantee reliability (durability and availability), the data object must be replicated a predefined number of times. In case P2P resources are not enough to guarantee the given level of reliability, the cloud resources are exploited, potentially for a limited amount of time until P2P resources are sufficient again.

The main goal is thus to guarantee that the overlay maintains a given size, including the cloud in the overlay only when threats of data loss do exist.

In order to realize such system, the following issues have to be addressed:

- To reach the maximum economical effectiveness of the network services, we have to effectively provide mechanisms to automatically tune the amount of cloud resources to be used. The system has to self-regulate the rate of cloud usage according to the level of data reliability (overlay size) and service costs.

- To regulate the cloud usage, each peer in the overlay has to know an updated view of overlay size. Hence, one of the key aspects that has to be considered is the network overhead caused by system monitoring.

- Churn rate may vary due to different reasons such as time of day, day of the week, period of the year, etc. and average lifetime of nodes can be even shorter than one minute. Hence, the size of the overlay can change

enormously. Data reliability must be supported even in the presence of one single replica (the replica stored on the cloud), as well as with hundreds of replicas (when the cloud is not required any more and should be removed to reduce monetary costs).

3.2 The algorithm

In order to provide a network topology, in our work we make use of a number of well-known protocols like CLOUDCAST [63] and CYCLON [99] for overlay maintenance and an aggregation protocol [40] to monitor the overlay size.

We consider the following three thresholds for the overlay size: (1) *redundant* R, (2) *sufficient* S and (3) *critical* C, with $R > S > C$. The computation of the actual threshold values is application-dependent; we provide here only a few suggestions how to select reasonable parameters, and we focus instead on the mechanisms for overlay size regulation, general enough to be applied to a wide range of applications.

- The *critical* threshold C should be the sum of (i) the minimum number of replicas that are enough for successful data recovering and (ii) the amount of nodes that will leave during the cloud backup replication phase. The minimum number of replicas is determined by the chosen redundancy schema, such as erasure coding or data replication.

- The *sufficient* threshold S can be computed as the sum of (i) *critical* threshold and (ii) the amount of nodes that are expected between two successive recovery phases, meaning that these values depends also on the expected churn rate.

- The *redundant* threshold R is an important parameter that allows to optimize the network resources utilization. The *redundant* threshold depends on the overlay membership dynamics (i.e. peers continuously join and leave the network) and on the application type. For example, backup applications are more sensitive to replica redundancy than content delivery applications. In a typical backup application, users save their data in the network and R is determined according to the adopted redundancy schema and overlay membership dynamics.

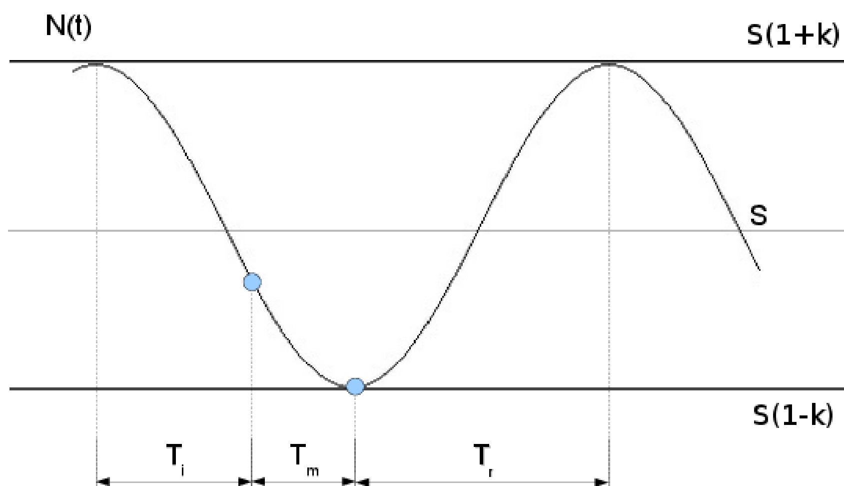


Figure 3.1: Overlay size oscillation

On the other hand, in content delivery applications, popular data is replicated in a high (possibly *huge*) amount of nodes of the network. In this case the *redundant* threshold is not relevant, since peers keep pieces of data based on their popularity. Nevertheless, the *critical* and *sufficient* thresholds are important in order to keep data available during the oscillation of network population.

In general, the size of the overlay is expected to stay in the range $[S, R]$. When the current size is larger than R, some peers could be safely removed; when it is smaller than S, some peers (if available) should be added. When it is smaller than C, the system is in a dangerous condition and part of the service should be provided by the cloud.

Figure 3.1 illustrates the behaviour of our system in the presence of churn. The size $N(t)$ oscillates around threshold S, with $k\%$ deviation. To withstand the oscillation, a system recovery process is executed periodically, composed of *monitoring* phase with duration T_m, a recovery phase with duration T_r and *idle* phase of duration T_i. The length of the monitoring phase depends on the particular protocol used to monitor the system, while the idle phase is autonomously regulated to reduce useless monitoring and hence overhead.

Monitoring is based on the work of Jelasity et al. [40]. The authors proposed a gossip protocol to compute the overlay size, that is periodically restarted. The execution of the protocol is divided into epochs, during which a fixed number

of gossip rounds are performed. The number of rounds determines the accuracy of the measure. At the end of each epoch, each peer obtains an estimate of the size as measured at the beginning of the epoch.

We have tuned the aggregation protocol in order to fit our system, with a particular emphasis on reducing network overhead. In our version, each aggregation epoch is composed by monitoring and idle periods. The monitoring period corresponds to the aggregation rounds in the original protocol [40], whereas in idle periods the aggregation protocol is suspended. Our approach allows to reduce network overhead by reducing the amount of useless aggregation rounds.

All peers follow the Algorithms 3.1 and 3.2. The system repeats a sequence of actions forever. The monitoring phase is started by the call to getSize(), which after T_m time units returns the size of the system at the *beginning* of the monitoring phase. Such value is stored in variable N_b. Given that during the execution of the monitoring phase further nodes may have left the system, the expected size of the system at the *end* of the monitoring phase is computed and stored in variable N_e. Such value is obtained by computing the failure rate f_r as the ratio of the number of nodes lost during the idle phase and the length of such phase.

According to the size of the overlay, a peer decides to either start the recovering process, delete/create cloud from the overlay or do nothing. If the system size $N_e \geq S$, the peer removes the (now) useless links to the cloud from its neighbour list; in other words, the system autonomously switch the communication protocol from CLOUDCAST to CYCLON. Furthermore, if the system size is larger than threshold R, the excess peers are removed in a decentralized way: each peer independently decide to leave the system with probability $p = (N_e - R)/N_e$, leading to an expected number of peers leaving the system equal to $N_e - R$.

When the number of peers is between critical and sufficient ($C < N_e < S$) a peer invokes function recovery() to promote additional peers to the overlay in a distributed fashion. Each peer in the overlay has to promote only a fraction of additional peers N_a, computed as the ratio of the number of peers needed to obtain the desired $S(1 + k)$ nodes and the current overlay size N_e. The ratio is rounded to the upper bound. Function addNode() adds new peers from the underlying overlay to the set of replicated entities. To bootstrap new peers to the overlay, the peer, currently promoting other peers, copies a set of random

```
repeat
    N_b ← getSize()
    f_r ← (N_b − N_e)/T_i
    N_e ← N_b − T_m · f_r
    if N_e ≥ S then
        removeCloud()
        if N_e > R then
            p ← (N_e − R)/N_e
            leaveOverlay(p)
    else if N_e < S then
        if N_e ≤ C then
            addCloud()
        recovery()
    T_i ← idleTime(N_e, f_r, T_i)
    wait T_i
```

Algorithm 3.1: Resource management algorithm

```
procedure idleTime(N_e, f_r, T_i)
    if N_e ≤ C then
        T_i ← T_min
    else if f_r > 0 then
        T_i ← (S·k)/f_r − T_m
    else if T_i < T_max then
        T_i ← T_i + Δ
    return T_i

procedure recovery()
    N_a ← ⌈S(1+k)/N_e − 1⌉
    for j ← 1 to N_a do
        V ← copyRandom(view)
        if N_e ≤ C then
            V ← V ∪ {cloud}
        addNode(V)
```

Algorithm 3.2: idleTime and recovery procedures

descriptors from its own local view copyRandom($view$) and sends them as a V to the new peers. A local view (indicated as $view$) is a small, partial view of the entire network [99].

When the overlay size is below critical ($N_e \leq C$), a peer first adds to the local view a reference to the cloud (function addCloud()) and then starts the recovering process. Moreover, the reference to the cloud is also added to a set of random descriptors V that is sent to the new peers. Hence, in case of a reliability risk, the system autonomously switches the protocol back to the CLOUDCAST (i.e. it includes a reliable cloud node into the overlay).

Concluded this phase, the peer has to compute how much it has to wait before the next monitoring phase starts; this idle time T_i is computed by function idleTime().

If the overlay size is lower than C, the idle time is set to a minimum value T_{min}. As a consequence, the monitoring phase starts in a shorter period of time. Reducing of the idle period allows the system to respond quickly to the overlay changes and keep the overlay size in a safe range. In case the amount of replicas has decreased ($f_r > 0$), the idle time is computed as the ratio of the deviation range $S \cdot k$ and the expected failure rate f_r. The ratio is decreased in T_m time entities needed for the next overlay monitoring phase. If the result T_i is less

Table 3.1: Parameters used in the evaluation

Parameter	Value	Meaning
n	$0 - 1024$	Total number of peers in the underlying network
R	60	*Redundant* threshold
S	40	*Sufficient* threshold
C	10	*Critical* threshold
k	0.2	deviation from *sufficient* threshold
σ_{cyclon}	10s	Cycle length of CYCLON
$\sigma_{entropy}$	10s	Cycle length of anti-entropy
$\sigma_{failure}$	1s	Cycle length of peers failure
c	20	View size
g	5	Cyclon message size
Δ	1s	Incremental addition step for the idle period T_i

than T_{min}, the final T_i is set to T_{min}. When either the number of replicas is stable or it has increased ($f_r \leq 0$), the idle period is increased by a fixed amount Δ. The duration of the idle phase is limited by the parameter T_{max}, which is decided by the system administrator.

It is important to note that, in order to increase the amount of overlay peers in the network, a peer has to add them faster then the failure rate. In fact, the promoting rate of new peers depends on several factors, such as network topology and physical characteristics of the network.

3.3 Evaluation

The repairing protocol has been evaluated through an extensive number of simulations based on event-based model of PEERSIM [65].

Unless explicitly stated, the parameters that are used in the current evaluation are shown in Table 3.1. Such parameters are partially inherited from the CYCLON [99], CLOUDCAST [63] and Aggregation protocols [40]. The choice of the thresholds parameters in Table 3.1 is motivated by graphical representation; in reality, the protocol is able to support both smaller and bigger sizes of the overlay.

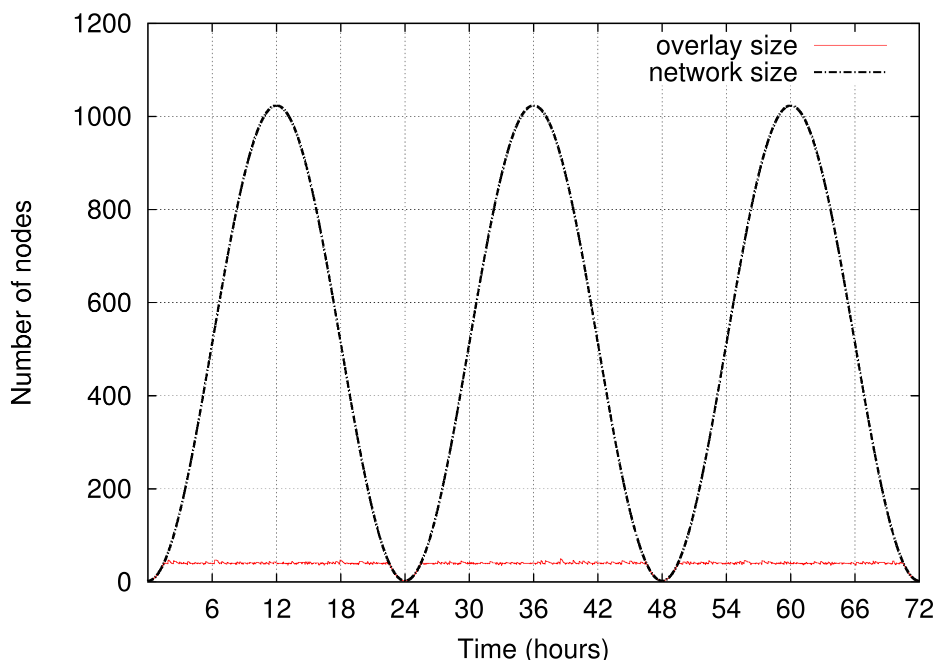


Figure 3.2: Overlay size in case of network oscillates daily between 0 and 1024 peers. The single experiment lasted 3 days.

The first aspect that has to be evaluated is protocol scalability: is our protocol able to support overlay availability in spite of network size oscillation? Figure 3.2 and Figure 3.3 shows the behaviour of the overlay when network size oscillates between 0 and 1024 peers in a one-day period. The average peer life time is around 3 hours.

When the network size grows, the overlay peers promote new peers to the overlay and its size growths until reaching the sufficient threshold. Then the protocol supports the sufficient overlay size with k deviation. When all peers leave the network, the overlay size falls to zero and contains only the cloud peer. This behaviour repeats periodically.

Figure 3.4 shows the amount of existing links to the cloud in the overlay. The top figure shows the network size oscillation and the bottom one represents the cloud in-degree for CLOUDCAST (thin line) and our protocol (thick line). As you can see, compared with CLOUDCAST, our protocol significantly reduces the utilization of cloud resources without implications on the overlay reliability. When the network size is enough to provide the *sufficient* overlay size, references to the cloud are removed; otherwise the overlay peers include cloud node

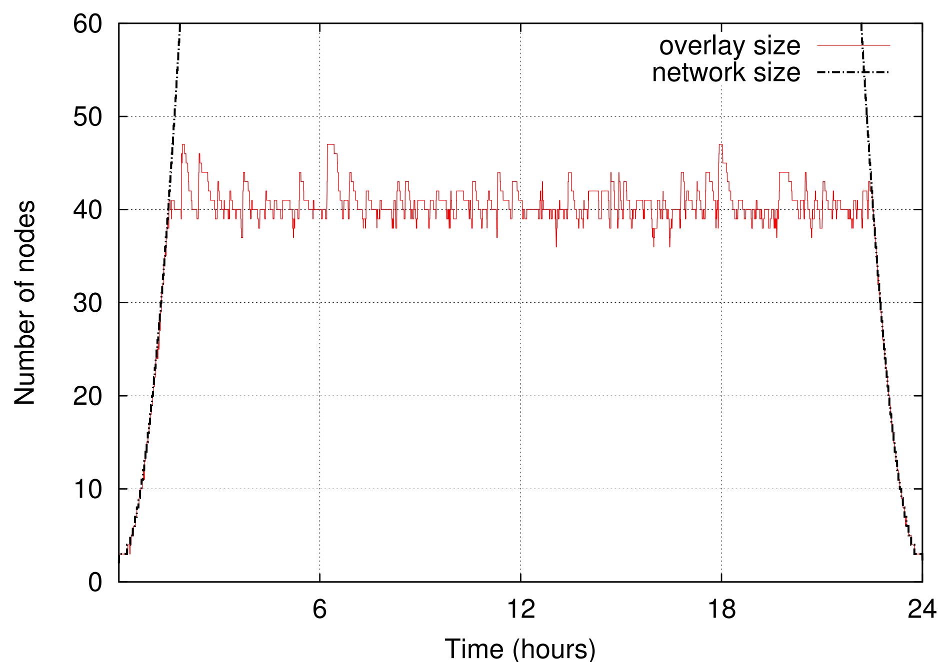


Figure 3.3: Overlay size in case of network oscillates daily between 0 and 1024 peers. Zooming over a single day.

into the overlay.

For system monitoring we adopt the work of Jelasity et al. [40]. Figure 3.5 shows the ability of the protocol to support the target overlay size under the various network churn rates and epoch intervals. The X-axis represents the duration of the epoch intervals. The Y-axis represents the deviation of the overlay size from the target *sufficient* threshold. A churn rate $p\%$ for the overlay means that at each second, any peer may abruptly leave the overlay with probability $p\%$. In this evaluation we do not consider that peers can rejoin the overlay.

As we can see from Figure 3.5, the lines correspond to 0%, 0.1% and 0.01% shows approximately the same behavior. These results proof that the system successfully supports the target deviation ($k = 0.2$). However, with the churn rate 1% the system is not able to support the overlay with an epoch interval larger then 30 seconds. Nevertheless, it must be noted that the churn rate of 1.0% corresponds to an average lifetime of less than 2 minutes.

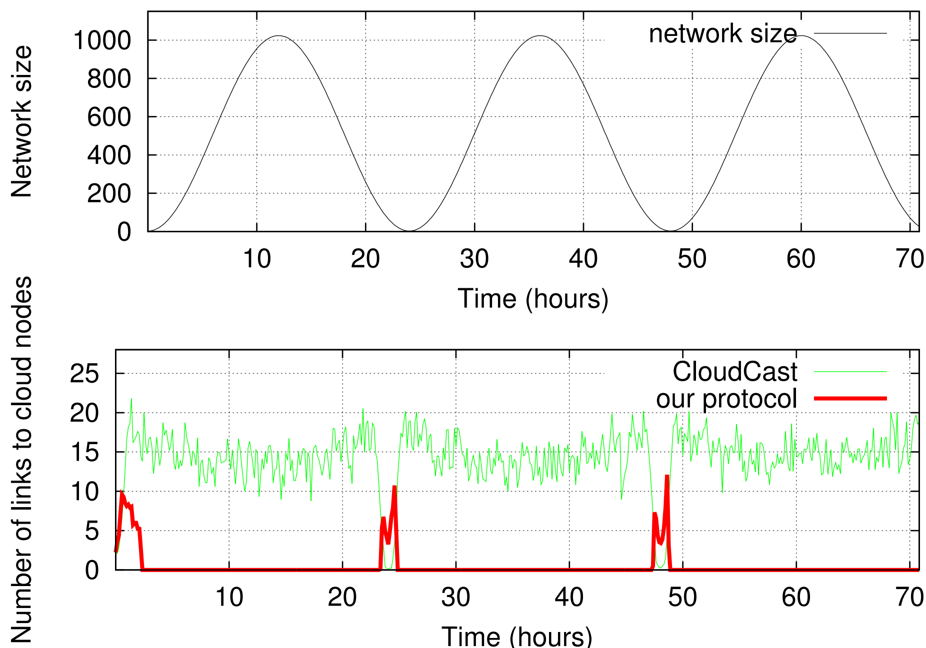


Figure 3.4: Cloud in-degree for CLOUDCAST and our protocols in case of network oscillates daily between 0 and 1024 peers.

3.4 Related Work

One of the most important limitations in P2P-based storage systems is connected with the difficulty to guarantee data reliability. P2P nodes leave or fail without notification and stored data can be temporarily or permanently unavailable.

A popular way to increase data durability and availability is to apply redundancy schemas, such as replication [50, 19], erasure coding [103] or a combination of them [71]. However, too many replicas may harm performance and induce excessive overhead. Therefore it becomes a relevant issue to keep the number of replica around a proper threshold, in order to have predictable information on load and performance of the system.

Two main approaches exist for replica control: (1) proactive and (2) reactive. Proactive approach creates replicas at some fixed rate [86, 71], whereas in reactive approach a new replica is created each time an existing replica fails [50, 19].

The work of Kim [50] proposes a reactive replication approach that uses lifetime of nodes to select where to replicate data. When the data durability is

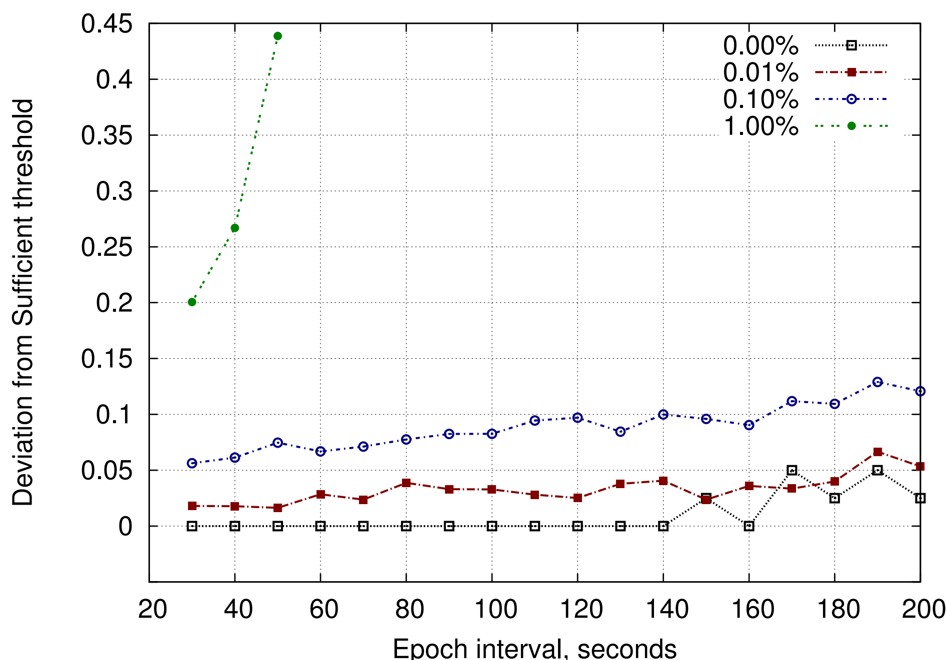


Figure 3.5: Deviation of the overlay size from the *sufficient* threshold under different levels of churn with variable epoch intervals.

below a threshold the node does replicas in nodes whose lifetime is long enough to assure durability. Conversely, Chun et al.[19] proposes a reactive algorithm, where data durability is provided by selecting a suitable amount of replicas. The algorithm responds to any detected failure and creates a new replica if the amount of replicas is less then the predefined minimum.

Nevertheless, these approaches are not taking in account the redundancy of replicas in the network. In fact, increasing the number of replicas affects bandwidth and storage consumption [8].

The problem of replica control is a relevant problem and one of the most complete work in this area is presented by Duminuco et al. [23]. To guarantee that data is never lost the authors propose an approach that combines optimized proactive with pure reactive replication. The approach is able to dynamically measure churn fluctuations and adapt the repair rate accordingly.

Nevertheless, Duminuco et al. consider the estimation of recovering rate, whereas in our work the attention is devoted for the periods of recovery restarting and network overloads caused by system monitoring. Moreover, Duminuco et al in their work do not consider the distributed mechanism of regulation the

overlay population.

Finally, recent works in the field do not consider the oscillation of the network size and the situation when P2P resources are not enough or not available. To solve these problems we use the combination of both communication models: P2P and cloud Computing. Moreover, we propose a mechanism that automatically includes or excludes the cloud into the overlay, by adapting the system to the network state.

Chapter 4

Video Streaming

Peer-to-peer (P2P) overlays have lowered the barrier to stream live events over the Internet, and have thus revolutionized the media streaming technology. However, satisfying soft real-time constraints on the delay between the generation of the stream and its actual delivery to users is still a challenging problem. Bottlenecks in the available upload bandwidth, both at the media source and inside the overlay network, may limit the quality of service (QoS) experienced by users. A potential solution for this problem is to *assist* the P2P streaming network by a cloud computing infrastructure to guarantee a minimum level of QoS. In such approach, rented cloud resources (*helpers*) are added on demand to the overlay, to increase the amount of total available bandwidth and the probability of receiving the video on time. Hence, the problem to be solved becomes the minimization of the economical cost, provided that a set of constraints on QoS are satisfied. The main contribution of this chapter is CLIVE, a cloud-assisted P2P live streaming system that demonstrates the feasibility of these ideas. CLIVE estimates the available capacity in the system through a gossip-based aggregation protocol and provisions the required resources from the cloud to guarantee a given level of QoS at low cost. We perform extensive simulations and evaluate CLIVE using large-scale experiments under dynamic realistic settings.

4.1 Problem statement

We consider a network consisting of a dynamic collection of *nodes* that communicate through message exchanges. Nodes could be *peers*, i.e., edge computers belonging to users watching the video stream, *helpers*, i.e., computational and

storage resources rented from an IaaS cloud, and the *media source* (*source* for short), which generates the video stream and starts its dissemination towards peers.

Each peer is uniquely identified by an ID, e.g., composed by IP address and port, required to communicate with it. We use the term *swarm* to refer to the collection of all peers. The swarm forms an *overlay network*, meaning that each peer connects to a subset of peers in the swarm (called *neighbours*). The swarm is highly dynamic: new peers may join at any time, and existing peers may voluntarily leave or crash. Byzantine behaviour is not considered in this work.

There are two types of helpers: (i) an *active helper* (AH) is an autonomous virtual machine composed of one or more computing cores, volatile memory and permanent storage, e.g., Amazon EC2, and (ii) a *passive helper* (PH) is a simple storage service that can be used to store (PUT) and retrieve (GET) arbitrary pieces of data, e.g., Amazon S3. We assume that customers of the cloud service are required to pay for computing time and bandwidth in the case of AHs, and for storage space, bandwidth and the number of PUT/GET requests in the case of PHs. This model follows the Amazon's pricing model.

We assume the source generates a constant-rate bitstream and divides it into a number of *chunks*. A chunk c is uniquely identified by the real time $t(c)$ at which is generated. The generation time is used to play chunks in the correct order, as they can be retrieved in any order, independently from previous chunks that may or may not have been downloaded yet.

Peers, helpers and the source are characterized by different bounds on the amount of available download and upload bandwidth. A peer can create a bounded number of download connections and accept a bounded number of upload connections over which chunks are downloaded and uploaded. We define the number of *download*, and *upload slots* of a peer p as its number of download and upload connections, respectively. Thanks to the replication strategies between different data centers currently employed in clouds [31], we assume that each PH has an unbounded number of upload slots and can serve as many requests as it receives. Preliminary experiments using PlanetLab and Amazon Cloudfront show that this assumption holds in practice.

We assume that peers are approximately synchronized; this is a reasonable assumption, given that some cloud services, like Amazon AWS, are already synchronized and sometimes require the client machines to be synchronized as

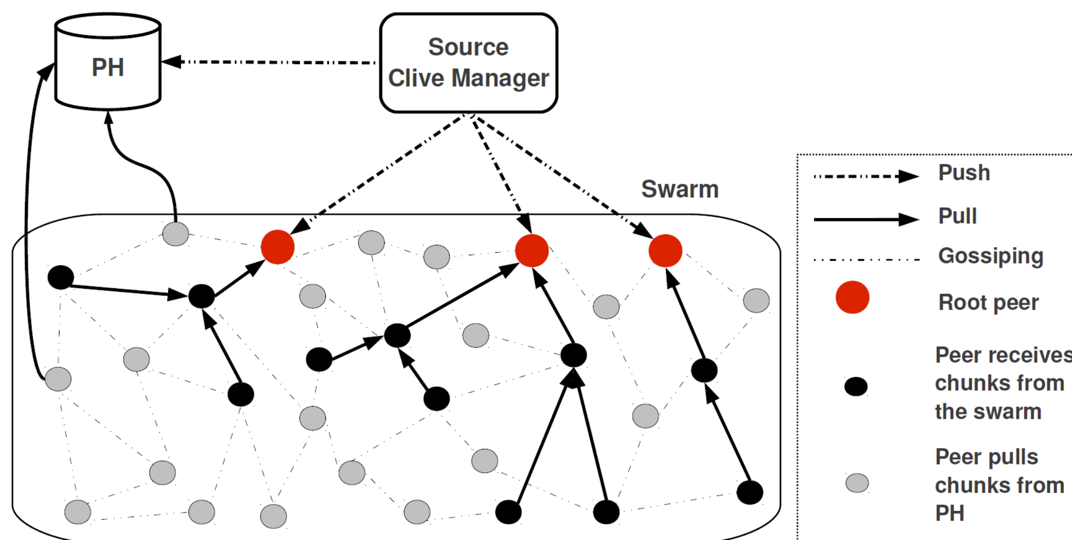


Figure 4.1: The baseline model.

well.

The goal of CLIVE peers is to play the video with predefined *playback delay* (the time between the generation of the video and its visualization at the peer) and *playback continuity* (the percentage of chunks that are correctly streamed to users). To reach this goal, CLIVE is allowed to rent PH and/or AH resources from the cloud.

Deciding about which and how much resources to rent from the cloud can be modelled as an optimization problem, where the objective function is to minimize the economic cost and the constraints are the following:

1. the maximum playback delay should be less than or equal to T_{delay}, meaning that if a chunk c is generated at time $t(c)$ at the source, no peers will show it after time $t(c) + T_{delay}$;

2. the maximum percentage of missing chunks should be less than or equal to P_{loss}.

Note that different formulations of this problem are possible, such as fixing a limit on the amount of money to be spent and trying to maximize the playback continuity. We believe, however, that a company, willing to stream its videos, should not compromise on the users' experience, but rather exploit peers whenever possible and fall back to the cloud when peers are not sufficient.

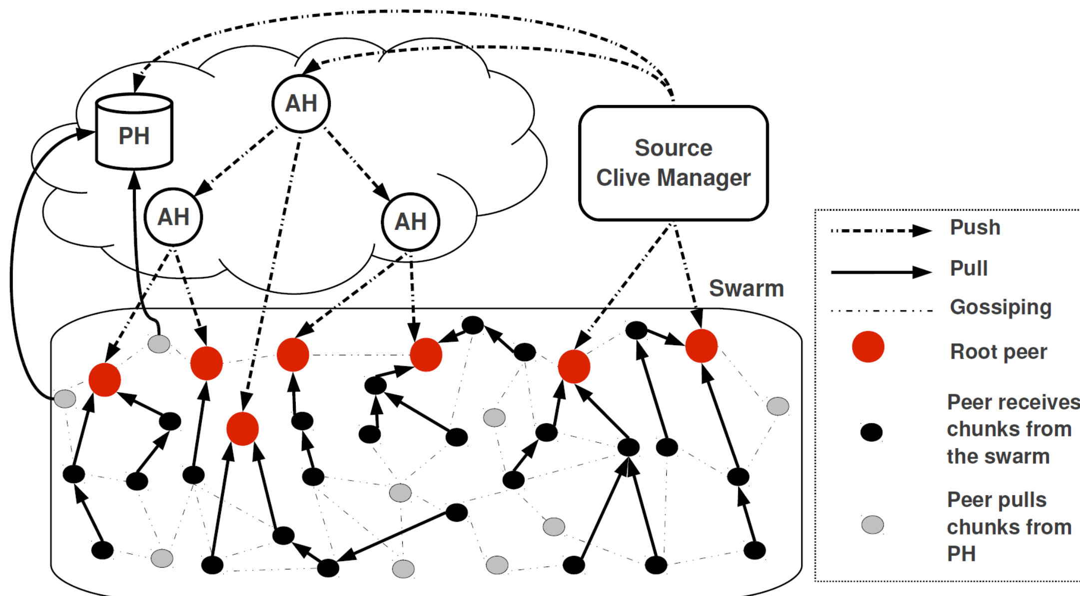


Figure 4.2: The enhanced model.

4.2 System architecture

The basic elements forming CLIVE have been already introduced: the media source, a swarm of peers, a single passive helper, and a number of active helpers. Aim of this section is to discuss how a such diverse collection can be organized and managed. We present two architectural models, illustrated in Figures 4.1 and 4.2. The *baseline* model (Figure 4.1) can be described as a P2P streaming protocol, where peers revert to PH whenever a chunk cannot be retrieved from other peers. The *enhanced* model (Figure 4.2) builds upon the baseline, by considering AHs and by providing a distributed mechanism to provision their number and appropriately organizing them.

In the rest of the section, we first discuss the baseline model, introducing the underlying P2P video streaming protocol and showing how it can be modified to exploit a PH. Then, we add the AHs into the picture and illustrate the diverse architectural options available when including them.

4.2.1 The baseline model

The baseline model can be seen as a P2P streaming service associated with a server – as simply as that. We introduce this model as a baseline for comparison and validation of our enhanced architectural model.

Note that the idea of augmenting a P2P video streaming application by renting cloud resources is general enough to be applied to several existing video streaming applications. We adopt a *mesh-pull* approach for data dissemination [108], meaning that peers are organized in an unstructured overlay and explicitly ask the missing chunks from their neighbours. Peers discover each other using a gossip-based peer-sampling service [21, 39, 73, 99]; then, the random *partial views* created by this service can be used by any of the existing algorithms to build the streaming overlay [27, 55, 70, 72].

In the mesh-pull model, neighbouring peers exchange their data availability with each other, and the peers use this information to schedule and pull the required chunks. There are a number of studies [16, 111] on chunk selection policies, but here we use the *in-order* policy, as in COOLSTREAMING [110], where peers pull the missing chunks with the closest playback time first.

The baseline model builds upon this P2P video streaming protocol by adding a PH (Figure 4.1). The source, apart from pushing newly created video chunks to the swarm, temporary stores them on the PH. If a peer p cannot obtain a chunk c from the swarm at least T_{lcw} time units before its playback time, it retrieves c directly from the PH. In other words, in order to guarantee a given level of QoS, each peer is required to have a predefined amount of chunks buffered ahead of its playback time, which is called the *last chance window* (LCW), corresponding to a time interval of length T_{lcw}.

4.2.2 The enhanced model

If the P2P substrate does not suffice, the baseline model represents the easiest solution, but as our experiments will show, this solution could be too expensive, as an excessive number of chunks could end up being retrieved directly from the PH. However, even if the aggregate bandwidth of the swarm may be theoretically sufficient to serve all chunks to all peers, the soft real-time constraints on the playback delay may prevent to exploit entirely such bandwidth. No peer must lag behind beyond a specified threshold, meaning that after a given time, chunks will not be disseminated any more. We need to increase the amount of peers that receive chunks in time, and this could be done by increasing the amount of peers that are served as early as possible. The enhanced model pursues this goal by adding a number of AHs to the swarm (Figure 4.2).

AHs receive chunks from the source or from other AHs, and push them to other AHs and/or to peers in the swarm. To discover such peers, AHs join the peer sampling protocol [99] and obtain a partial view of the whole system. We use a modified version of CYCLON [99], such that peers exchange their number of upload slots along with their ID. AH chooses a subset of *root peers* (Figure 4.2) from their partial view and establish a connection to them, pushing chunks as soon as they become available. Root peers of an AH are not changed over time, unless they fail or leave the system, or AH finds a peer with more upload slots than the existing root peers. Clearly, a peer could accept to be a root peer only for one AH, to avoid to receive multiple copies of the same chunk. The net effect of adding AHs is an increase in the number of peers that receive the video stream early in time. The root peers also participate in the P2P streaming protocol, serving a number of peers directly or indirectly. The PH still exists in the enhanced model to provide chunks upon demand, but it will be used less frequently compared to the baseline model.

Architecturally speaking, an important issue is how to organize multiple AHs and how to feed chunks to them. There are two possible models:

- *Flat*: the AHs receive all their chunks directly from the source and then push them to peers in the swarm, acting just as bandwidth multipliers for the source.

- *Hierarchical*: the AHs are organized in a tree with one AH at the root; the source pushes chunks to the root, which pushes them through the tree.

The advantage of the flat model is that few intermediary nodes cause a limited delay between the source and the peers. However, the source bandwidth could end up being entirely consumed to feed the AHs; and more importantly, any communication to the cloud is billed, including the multiple ones from the source to the AHs. We, thus, decided to adopt the hierarchical model, also considering that communication inside the cloud is (i) extremely fast, given the use of gigabit connections, and (ii) free of charge [4].

One important question in the enhanced model is: *how many AHs to add*? Finding the right balance is difficult; too many AHs may reduce the PH load, but cost too much, given that they are billed for the used bandwidth, but also for each hour of activity. Too few AHs also increases the PH load, and as we show

in the experiments, increases the cost. The correct balance dynamically depends on the current number of peers in the swarm, and their upload bandwidth.

The decision on the number of AHs to include in the system is taken by the CLIVE *manager* (CM), a unit that is responsible for monitoring the state of the system and organizing the AHs. By participating in a decentralized aggregation protocol [40], the CM obtains information about the number of peers in the system and the distribution of upload slots among them. Based on this information, it adds new AHs or remove existing ones, trying to minimize the economic cost. The CM role can be played either directly by the source, or by one AH. A detailed description of the CM is provided in the next section.

4.3 System management

Based on the swarm size and the available upload bandwidth in the swarm, the CM computes the number of AHs that have to be active to minimize the economic cost. Then, depending on the current number of AHs, new AHs may be booted or existing AHs may be shutdown.

The theoretical number of AHs that minimize the cost is not so straightforward to compute, because no node has a global view of the system and its dynamics, e.g., which peers are connected and how many upload slots each peer has. Instead, we describe a heuristic solution, where each peer runs a small collection of gossip-based protocols, with the goal of obtaining approximate aggregate information about the system. The CM joins these gossip protocols as well, and collects the aggregated results. It exploits the collected information to estimate a lower bound on the number of peers that can receive a chunk either directly or indirectly from AHs and the source, but not from PH. We call this set of peers as *infected peers*. The CM, then, uses this information to detect whether the current number of AHs is adequate to the current size of the swarm, or if correcting actions are needed by adding/removing AHs.

In the rest of this section, we first explain how the CM estimates the swarm size and the upload slot distribution, and then we show how it calculates the number of infected peers using the collected information. We also present how the CM manages the number of AHs, based on the swarm size and the number of infected peers, and finally we explain the effect of T_{lcw}, as an important system parameter, on the cost and the QoS.

4.3.1 The swarm size and upload slot distribution estimation

All peers in the system, including the CM, participate in the aggregate computation (Algorithms 4.1 and 4.2). The procedure round() is called periodically by all peers, as well as by the CM to estimate (i) the current size of the swarm, (ii) the probability density function of the upload slots available at peers, and (iii) the T_{lcw} average (Section 4.3.4).

The size N_{swarm} of the current swarm is computed, with high precision, through the aggregation protocol [40]. On the other hand, knowing the number of upload slots of all peers is infeasible, due to the large scale of the system and its dynamism. However, we can obtain a reasonable approximation of the probability density function of the number of upload slots available at all peers.

Let ω be the actual upload slot distribution among all peers. We adopt ADAM2 [82] to compute $P_\omega : \mathbb{N} \to \mathbb{R}$, an estimate probability density function of ω. $P_\omega(i)$, then, represents the proportion of peers that have i upload slots w.r.t. the total number of peers, so that $\sum_i P_\omega(i) = 1$. ADAM2 is a gossip-based algorithm that provides an estimation of the cumulative distribution function of a given attribute across all peers.

For our algorithm to work, we assume that each peer is able to estimate its own number of upload slots, and the extreme values of such distribution are known to all and static. Otherwise, a simple mechanism proposed by Haridasan and van Renesse [34] can adjust the set of entries for the case where the extreme values of a variable are unknown. The maximum value is shown by $maxSlot$.

Our solution, summarized in Algorithms 4.1 and 4.2, is based on the gossip paradigm: execution is organized in periodic rounds, performed at roughly the same rate by all peers, during which a push-pull gossip exchange is executed [42]. During a round, each peer p sends a REQ message to a peer q, and waits for the corresponding RES message from q. Information contained in the exchanged messages are used to update the local knowledge about the entire system, which is composed by the following information:

- a *partial view*, or *view* for short, of the network, stored in variable $subview$, that represents a small subset of the entire population of peers,
- a *slot vector (SV)*, which is used to obtain an approximate and up-to-date information about the attribute distribution,

- a *local value (LV)*, which is used by peers to estimate the network size.

```
procedure init()
    int T_lcw ← T_rtt
    int slot ← the number of peer's slots
    int maxSlot ← maximum number of slots in the system
    for i ← 0 to maxSlot do
        if i = this.slot then
            this.SV[i] ← 1
        else
            this.SV[i] ← 0
    if this = CM then
        this.LV ← 1
    else
        this.LV ← 0

procedure round()
    this.view.updateAge()
    q ← selectOldest(this.view)
    this.view.remove(q)
    pSubview ← this.view.randomSubset(this.view)
    pSubview.add(this)
    send ⟨REQ, pSubview, this.SV, this.LV, this.T_lcw⟩ to q
```

Algorithm 4.1: Estimating the swarm size, upload slot distribution, and T_{lcw} average.

Partial views are needed to maintain a connected, random overlay topology over the population of all peers to allow the exchange of information. We manage the views through the CYCLON peer sampling service [99]. Each view contains a fixed number of *descriptors*, composed by a peer ID and a timestamps. During each round, a peer p identifies the node q with the oldest descriptor in its view, based on the timestamps through selectOldest() in Algorithms 4.1 and 4.2. The corresponding descriptor is removed, and a subset of p's view is extracted through procedure randomSubset(). This subset is sent to q through a REQ message. Peer q that receives the REQ message, replies with a RES message that similarly contains a number of descriptors randomly selected from its local view.

Whenever p receives a view from q, it merges its own view with the q'view through procedure mergeView. Peer p iterates through the received view, and

on event receive $\langle \text{REQ}, pSubview, pSV, pLV, pT_{lcw} \rangle$ **from** p **do**
- $qSubview \leftarrow this.view.$randomSubset$(this.view)$
- **send** $\langle \text{RES}, qSubview, this.SV, this.LV, this.T_{lcw} \rangle$ **to** p
- **for** $i \leftarrow 0$ **to** $maxSlot$ **do**
  - $this.SV[i] \leftarrow \frac{this.SV[i]+pSV[i]}{2}$
- $this.LV \leftarrow \frac{this.LV+pLV}{2}$
- $this.T_{lcw} \leftarrow \frac{this.T_{lcw}+pT_{lcw}}{2}$
- mergeView$(this.view, qSubview, pSubview)$

on event receive $\langle \text{RES}, qSubview, qSV, qLV, qT_{lcw} \rangle$ **from** q **do**
- **for** $i \leftarrow 0$ **to** $maxSlot$ **do**
  - $this.SV[i] \leftarrow \frac{this.SV[i]+qSV[i]}{2}$
- $this.LV \leftarrow \frac{this.LV+qLV}{2}$
- $this.T_{lcw} \leftarrow \frac{this.T_{lcw}+qT_{lcw}}{2}$
- mergeView$(this.view, pSubview, qSubview)$

procedure mergeView$(view, sentView, receivedView)$
- **forall** n **in** $receivedView$ **do**
  - **if** $this.view$ contains n **then**
    - $view.$updateAge(n)
  - **else if** $view$ has free space **then**
    - $view.$add(n)
  - **else**
    - $m \leftarrow sentView.$poll$()$
    - $view.$remove(m)
    - $view.$add(n)

Algorithm 4.2: Estimating the swarm size, upload slot distribution, and T_{lcw} average.

adds the descriptors to its own view. If p's view is not full, it adds the peer to its view, and if a peer descriptor to be merged already exists in p's view, p updates its age, if it is newer. If p' view is full, p replaces one of the peers it had sent to q with a peer in the received list. The poll method returns and removes the last peer from the list. The net effect of this process is the continuous shuffling of views, removing old descriptors belonging to crashed peers and epidemically disseminating new descriptors generated by active ones. The resulting overlay network, where the neighbours of a peer are the peers included in the partial view, closely resembles a random graph, characterized by extreme robustness

and small diameter [99].

LV is a local float value maintained at peers and at the CM. Initially, it equals zero in all peers, and equals one in the CM. In addition to LV, each peer also maintains SV, which is a vector with $maxSlot + 1$ entries, such that the index of each entry shows the number of slots, i.e., from 0 to $maxSlot$. Initially, all entries of SV at peer p are set to 0, except p's one, which is set to 1.

In each shuffle request, peer p sends its LV and SV values, along with its view. When q receives a REQ message from p, it replies with a message containing a subset of its views, SV, and LV. Peer q, then, goes through the received SV and updates its own SV entries to the average of the values for each entry in both SVs, i.e., $q.SV[i] \leftarrow (q.SV[i] + p.SV[i])/2$. Peer q also updates its LV to $(q.LV + p.LV)/2$. Likewise, peer p updates its SV and LV, when it receives RES from q. After a few exchanges, all peers and the CM find the distribution of slots in their own SV, such that entry i in SV shows the probability of peers with i slots. They can also compute the swarm size locally as:

$$N_{swarm} = 1/LV \tag{4.1}$$

4.3.2 Estimating the number of infected peers

The number of peers that can receive a chunk from either the swarm, the source or one of the AHs is bounded by the time available to the dissemination process. This time depends on a collection of system and application parameters:

- T_{delay}: No more than T_{delay} time units must pass between the generation of a chunk at the source and its playback at any of the peers.
- $T_{latency}$: The maximum time needed for a newly generated chunk to reach the *root peers*, i.e., the peers that directly receive the chunks from either the source or the AHs, is equal to $T_{latency}$. While this value may depend on whether a particular root peer is connected to the source or to an AH, we consider it as an upper bound and we assume that the latency added by AHs is negligible.
- T_{lcw}: If a chunk is not available at a peer T_{lcw} time units before its playback time, it will be retrieved from the PH.

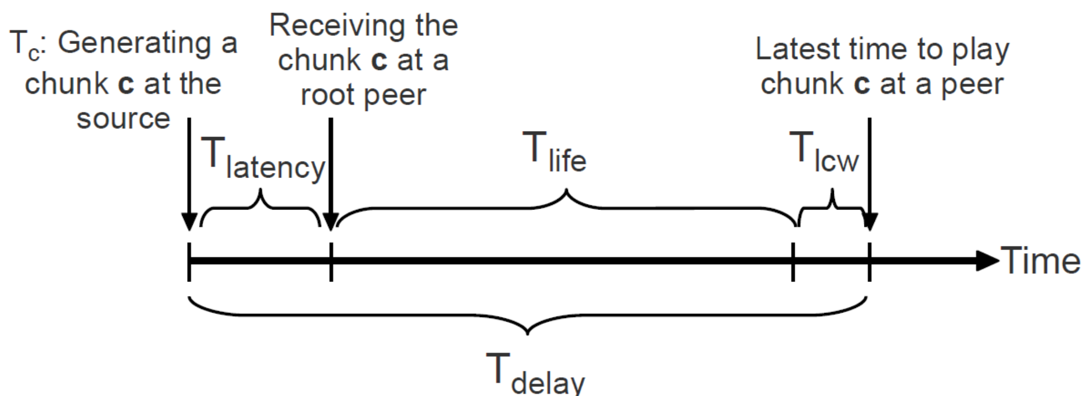


Figure 4.3: Live streaming time model.

Therefore, a chunk c generated at time $t(c)$ at the source must be played at peers no later than $t(c) + T_{delay}$, otherwise the QoS contract will be violated. Moreover, the chunk c becomes available at a root peer at time $t(c) + T_{latency}$, and it should be available in the local buffer of any peer in the swarm by time $t(c) + T_{delay} - T_{lcw}$, otherwise the chunk will be downloaded from the PH (Figure 4.3). This means that the lifetime T_{life} of a chunk at root peers is equal to:

$$T_{life} = (T_{delay} - T_{latency}) - T_{lcw} \tag{4.2}$$

Whenever a root peer r receives a chunk c for the first time, it starts disseminating it in the swarm. Biskupski et al. in [7] show that a chunk disseminated by a pull mechanism through a mesh overlay follows a tree-based diffusion pattern. We define the *diffusion tree* $DT(r, c)$ rooted at a root peer r of a chunk c as the set of peers, such that a peer q belongs to $DT(r, c)$, if it has received c from a peer $p \in DT(r, c)$.

Learning the exact diffusion tree for all chunks is difficult, because this would imply a global knowledge of the overlay network and its dynamics, and each chunk may follow a different tree. Fortunately, such precise knowledge is not needed. What we would like to know is an estimate of the number of peers that can be theoretically reached through the source or the current population of AHs.

The chunk generation execution is divided into rounds of length T_{round}. Chunk uploaded at round i becomes available for upload to other peers at round $i + 1$. The maximum depth, $depth$, of any diffusion tree of a chunk over its T_{life} is computed as: $depth = \lfloor T_{life}/T_{round} \rfloor$. We assume that T_{round} is bigger than the

```
procedure size(DENSITY P_ω, int depth)
    int min ← +∞
    repeat k times
        min ← min(min, recSize(P_ω, depth))
    return min;

procedure recSize(DENSITY P_ω, int depth)
    int n ← 1
    int slots ← random(P_ω)
    repeat slots times
        n ← n + recSize(P_ω, depth − 1)
    return n
```

Algorithm 4.3: Lower bound for the diffusion tree size.

average latency among the peers in the swarm. Given $depth$ and the probability density function P_ω, we define the procedure $\mathsf{size}(P_\omega, depth)$ that executes locally at the CM and provides an estimate of the number of peers of a single diffusion tree (Algorithm 4.3). This algorithm emulates a large number of diffusion trees, based on the probability density function P_ω, and returns the smallest value obtained in this way.

Emulation of a diffusion tree in a swarm is obtained using the recursive procedure $\mathsf{recSize}(P_\omega, depth)$. In this procedure, variable n is initialized to 1, meaning that this peer belongs to the tree. If the depth of the tree is larger than 0, another round of dissemination can be completed. The number of upload slots is drawn randomly by function $\mathsf{random}()$ from the probability density function P_ω. Variable n is then increased by adding the number of peers that can be reached by recursive call to $\mathsf{recSize}()$, where the depth is decremented by 1 at each step before the next recursion.

At this point, the expected number of infected peers, i.e., the total peers that can receive a chunk directly or indirectly from AHs and the source, but not from the PH, N_{exp}, is given by the total number of root peers times the estimated diffusion tree size, $N_{tree} = \mathsf{size}(P_\omega, depth)$. The number of root peers is computed as the sum of the upload slots at the source, $Up(s)$, and AHs, $Up(h)$, minus the number of slots used to push chunks to the AHs themselves, as well as to the PH, which is equal to the number of AHs plus one. Considering $\mathcal{AH}$ as the set of all AHs, we have:

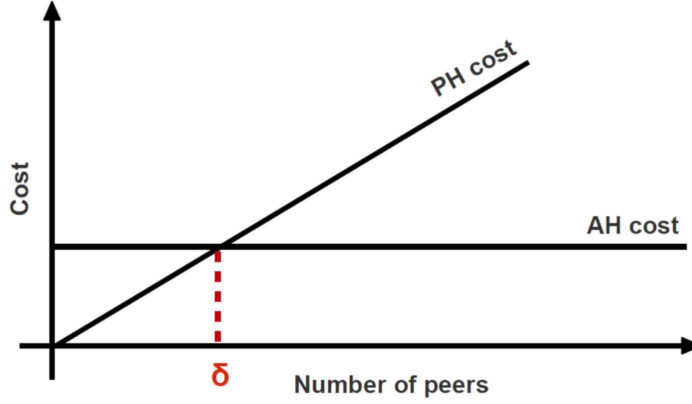


Figure 4.4: Calculating the number of peers that is economically reasonable to serve with PH utilization instead of running an additional AH.

$$N_{exp} = \left(Up(s) + \sum_{h \in \mathcal{AH}} Up(h) - (|\mathcal{AH}| + 1) \right) \cdot N_{tree} \tag{4.3}$$

4.3.3 AHs management model

We define the cost C_{ah} of an AH in one round (T_{round}) as the following:

$$C_{ah} = C_{vm} + m \cdot C_{chunk} \tag{4.4}$$

where C_{vm} is the cost of running one AH (virtual machine) in a round, C_{chunk} is the cost of transferring one chunk from an AH to a peer, and m in the number of chunks that one AH uploads per round. Since we utilize all the available upload slots of an AH, we can assume that $m = Up(h)$. Similarly, the cost C_{ph} of pulling chunks from PH per round is:

$$C_{ph} = C_{storage} + r \cdot (C_{chunk} + C_{req}) \tag{4.5}$$

where $C_{storage}$ is the storage cost, C_{req} is the cost of retrieving (GET) one chunk from PH and r is the number of chunks retrieved from PH per round. C_{chunk} of PH is the same as in AH. Moreover, since we store only a few minutes of the live stream in the storage, $C_{storage}$ is negligible.

Figure 4.4 shows how C_{ah} and C_{ph} (depicted in Formulas 4.4 and 4.5) changes in one round (T_{round}), when the number of peers increases. We observe that C_{ph}

increases linearly with the number of peers (number of requests), while C_{ah} is constant and independent of the number of peers in the swarm. Therefore, if we find the intersection of the cost functions, i.e., the point δ in Figure 4.4, we will know when is economically reasonable to add a new AH, instead of putting more load on PH.

$$\delta \approx \frac{C_{vm} + m \cdot C_{chunk}}{C_{chunk} + C_{req}} \tag{4.6}$$

The CM considers the following thresholds and regulation behaviour:

- $N_{swarm} > N_{exp} + \delta$: This means that the number of peers in the swarm is larger than the maximum size that can be served with a given configuration, thus, more AHs should be added to the system.
- $N_{swarm} < N_{exp} + \delta - Up(h) \cdot N_{tree}$: Current configuration is able to serve more peers than the current network size, thus, extra AHs can be removed. $Up(h) \cdot N_{tree}$ shows the number of peers served by one AH.
- $N_{exp} + \delta - Up(h) \cdot N_{tree} \leq N_{swarm} \leq N_{exp} + \delta$: In this interval the system has adequate resource and no change in the configuration is required.

The CM periodically checks the above conditions, and takes the necessary actions, if any. In order to prevent temporary fluctuation, it adds/removes only single AH in each step.

4.3.4 Discussion on T_{lcw}

T_{lcw} is a system parameter that has an important impact on the quality of the received media at end users, as well as on the total cost. Finding an appropriate value for T_{lcw} is challenging. With a too small T_{lcw} peers may fail to fetch chunks from PH in time for playback, while a too large T_{lcw} increases the number requests to PH, thus, increases the cost. Therefore, the question is how to choose a value for T_{lcw} to achieve (i) the best QoS with a (ii) minimum cost.

Impact of T_{lcw} on the QoS

As we mentioned in Section 4.2, each peer buffers a number of chunks ahead of its playback time, to guarantee a given level of QoS. The number of buffered

chunks corresponds to a time interval of length T_{lcw}. The length of T_{lcw} should be chosen big enough, such that if a chunk is not received through other peers, there is enough time to send a request to PH and retrieve the missing chunk from it in time for playback.

The required time for fetching a chunk from the PH depends on the round trip time (T_{rtt}) between the peer and the PH, and thus it is not the same for all the peers. Therefore, each peer measures T_{rtt} locally, which consists of the latency to send the request to the PH, plus the latency to receive the chunks at the peer's buffer. A peer should send a request for a missing chunk to PH no later than T_{rtt} time units before the playback time, otherwise, the retrieved chunk is useless. Therefore, each peer sets the minimum value of T_{lcw} to T_{rtt}.

While T_{lcw} is a local value at each peer, T_{life}, which is used by the CM to calculate the number of infected peers, depends on the T_{lcw} value (Equation 4.2). Therefore, the CM should be aware of the average T_{lcw} among peers. To provide this information to the CM, all peers, including the CM, participate in an aggregation protocol to get the average of T_{lcw} among all peers. Algorithms 4.1 and 4.2 show that in each shuffle, a peer sends its local T_{lcw} to other peers, and upon receiving a reply it updates its T_{lcw} to the average of its own T_{lcw} and the received one.

Impact of T_{lcw} on the cost

Equation 4.5 shows that the cost of PH increases linearly with the number of requests in each round. On the other hand, increasing T_{lcw} increases the PH cost in a round, as peers send more requests to PH. To have a more precise definition of PH cost in Equation 4.5, we replace r, which is the number of received requests at PH, with $\delta \times l$, where l is the normalized value of T_{lcw} at the CM by the average T_{lcw} (achieved in the aggregation protocol), i.e., $l = \frac{T_{lcw}}{\text{avg}T_{lcw}}$, and δ is the number of peers sending requests to PH in a round. Therefore, Equation 4.6 can be rewritten as follows:

$$\delta \approx \frac{C_{vm} + m \cdot C_{chunk}}{l \times (C_{chunk} + C_{req})} \tag{4.7}$$

The CM uses the average T_{lcw} to tune T_{lcw} of the system. If the CM finds out that changing T_{lcw} can decrease the cost, without violating the QoS, it floods the new value of T_{lcw} to the peers. To do that, it sends the new T_{lcw} to the directly

connected peers, and each peer forwards it to all its neighbours, expect the one that it receives the message from. In the flooding path, the peers with smaller T_{lcw} than the received one update their local T_{lcw}. However, if T_{lcw} at a peer is bigger than the received value, it does not change it, as its current T_{lcw} is the minimum required time to get a chunk from PH. Note that in Equation 4.6, we assumed the local CM T_{lcw} equals the aggregated average of T_{lcw}, thus, $l = 1$.

Figure 4.4 shows the relation between T_{lcw} and PH cost. The higher T_{lcw} is, the steeper the PH cost line is. This means that the cost of PH increases faster with the bigger T_{lcw}, since PH receives more requests in a shorter time. Moreover, smaller values for T_{lcw} push the PH cost line toward the x-axis. For example, if T_{lcw} is zero, i.e., peers never use PH, the PH cost is zero and the line overlaps the x-axis. However, we cannot set T_{lcw} to zero, since we use PH as a backup of the chunks to guarantee the promised QoS.

Increasing the T_{lcw} not only increases the PH cost, but also increases the total system cost. We see in Section 4.3.3, that the CM uses two parameters to manage the AHs, (i) the value of δ, and (ii) the number of infected peers. As Equation 4.7 shows, the higher T_{lcw} is, the smaller δ is. On the other hand, according to Equation 4.2, increasing T_{lcw} decreases T_{life}, and consequently, decreases the number of infected peers. Hence, increasing T_{lcw}, decreases both δ, and number of infected peers, and as a result the CM adds more AHs to the system, according to the management model in Section 4.3.3, which increases the total cost.

To summarize, we can say that the best value for T_{lcw} is the aggregated average T_{lcw}, where $l = 1$. Decreasing T_{lcw} below the average T_{lcw}, i.e., $l < 1$, decreases the QoS at peers, as they may fail to fetch chunks from PH before their playback time. On the other hand, although increasing T_{lcw} may increase QoS, it also increases the cost ($l > 1$). Hence, the CM never broadcasts new value of T_{lcw} to the system.

4.4 Experiments

In this section, we evaluate the performance of CLIVE using KOMPICS [3], a framework for building P2P protocols that provides a discrete event simulator for testing the protocols using different bandwidth, latency and churn scenarios.

Table 4.1: Slot distribution in freerider overlay.

Number of slots	Percentage of peers
0	49.3%
1	18.7%
2	8.4%
$3-19$	5.2%
20	6.8%
Unknown	11.6%

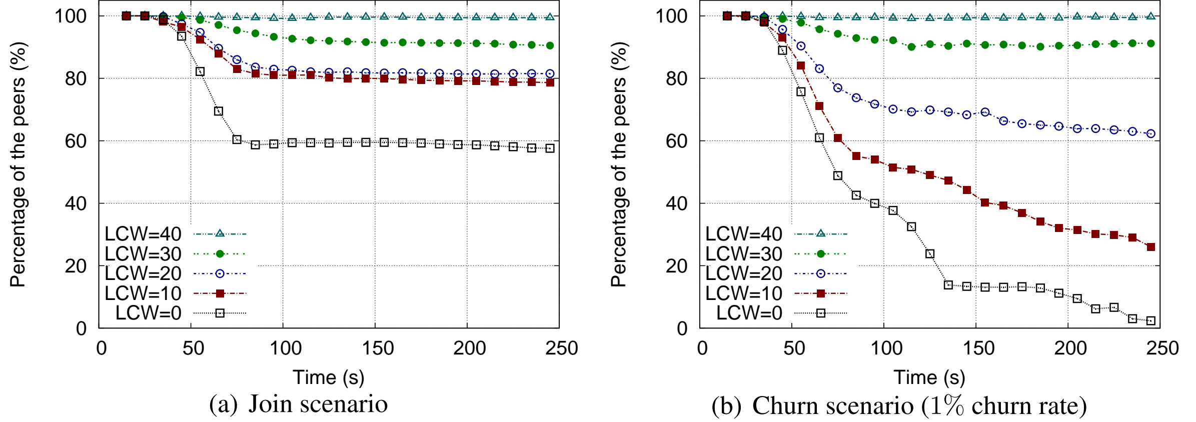
4.4.1 Experimental setting

In our experimental setup, we set the streaming rate to 500kbps, which is divided into chunks of 20kb; each chunk, thus, corresponds to 0.04s of video stream. Peers start playing the media after buffering it for 15 seconds, and T_{delay} equals 25 seconds. We set the bandwidth of an upload slot and download slot to 100kbps. Without loss of generality, we assume all peers have enough download bandwidth to receive the stream with the correct rate. In these experiments, all peers have 8 download slots, and we consider three classes of upload slot distributions: (i) *homogeneous*, where all peers have 8 upload slots, (ii) *heterogeneous*, where the number of upload slots in peers is picked uniformly at random from 4 to 13, and (iii) *real trace* (Table 4.1) based on a study of large scale streaming systems [88]. As it is shown in Table 4.1, around 50% of the peers in this model do not contribute in the data distribution. The media source is a single node that pushes chunks to 10 other peers. We assume PH has infinite upload bandwidth, and each AH can push chunks to 20 other peers. Latencies between peers are modelled using a latency map based on the King data-set [33].

In our experiments, we used two failure scenarios: *join-only* and *churn*. In the join-only scenario, 1000 peers join the system following a Poisson distribution with an average inter-arrival time of 10 milliseconds, and after joining the system they will remain till the end of the simulation. In the churn scenario, approximately 0.01%, 0.1% and 1% of the peers leave the system per second and rejoin immediately as newly initialized peers [94]. However, unless stated otherwise, we did the experiments with 1% churn rate to show how the system

performs in presence of high dynamism.

4.4.2 The effect of T_{lcw} on system performance

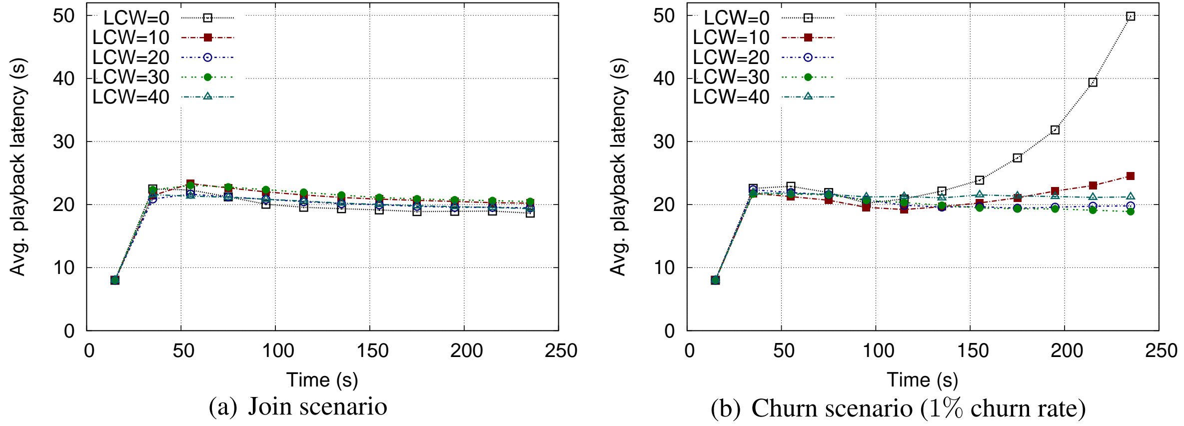


(a) Join scenario (b) Churn scenario (1% churn rate)

Figure 4.5: The percentage of the peers receiving 99% playback continuity with different values of T_{lcw} (measured in number of chunks).

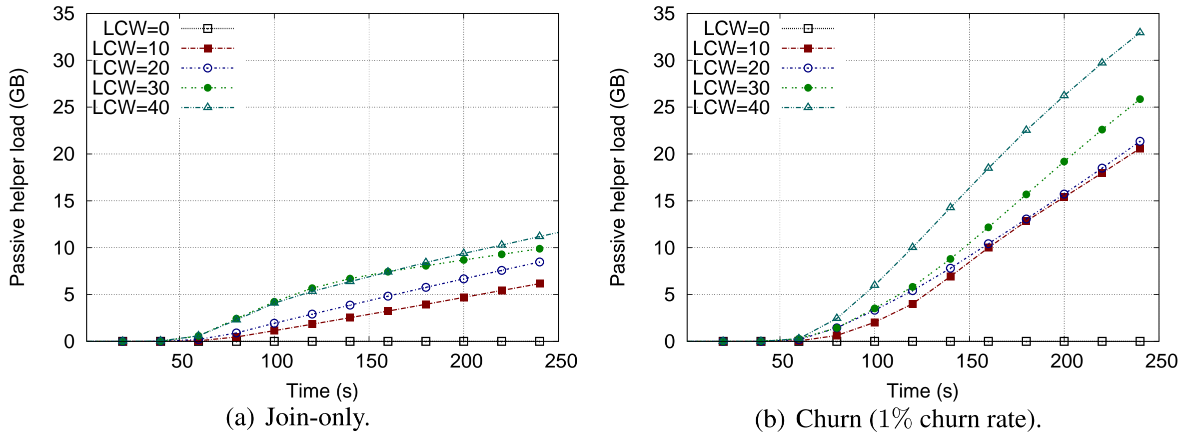
In the first experiment, we evaluate the system behaviour with different values for T_{lcw}, measured in number of chunks. In this experiment, we measure *playback continuity* and *playback delay*, which combined together reflect the QoS experienced by the overlay peers. Playback continuity shows the percentage of chunks received on time by peers, and playback delay represents the difference, in seconds, between the playback point of a peer and the source.

For a cleaner observation of the effect of T_{lcw}, we use the homogeneous slot distribution in this experiment. Figure 4.5 shows the fraction of peers that received 99% of the chunks before their timeout with different T_{lcw} in the join-only and churn scenarios (1% churn rate). We changed LCW between 0 to 40 chunks, where zero means peers never use PH, and 40 means that a peer retrieves up to chunk $c + 40$ from PH, if the peer is currently playing chunk c. As we see, the bigger T_{lcw} is, the more peers receive chunks in time. Although for any value of $T_{lcw} > 0$ peers try to retrieve the missing chunks from PH, the network latency may not allow to obtain the missing chunk in time. As Figure 4.5 shows, all the peers retrieve 99% of the chunks on time when $LCW = 40$. Given that each chunk corresponds to 0.04 seconds, $T_{lcw} = 40$ implies 1.6 seconds.



(a) Join scenario (b) Churn scenario (1% churn rate)

Figure 4.6: Average playback delay across peers with different values of T_{lcw} (measured in number of chunks).



(a) Join-only. (b) Churn (1% churn rate).

Figure 4.7: The cumulative PH load with different values of T_{lcw}.

The average playback delay of peers is shown in Figure 4.6. In the join-only scenario, playback delay does not depend on T_{lcw}, while in the churn scenario we can see a sharp increase when T_{lcw} is small.

4.4.3 PH load in different settings

Here, we measured PH load or the amount of fetched chunks from PH with different T_{lcw} values and churn rates. Figures 4.7(a) and 4.7(b) show the cumulative load of PH in the join-only and churn scenarios (1% churn rate), respectively. As we see in these figures, by increasing T_{lcw}, more requests are sent to PH, thus, increasing its load. Figure 4.8 depicts the cumulative PH load over

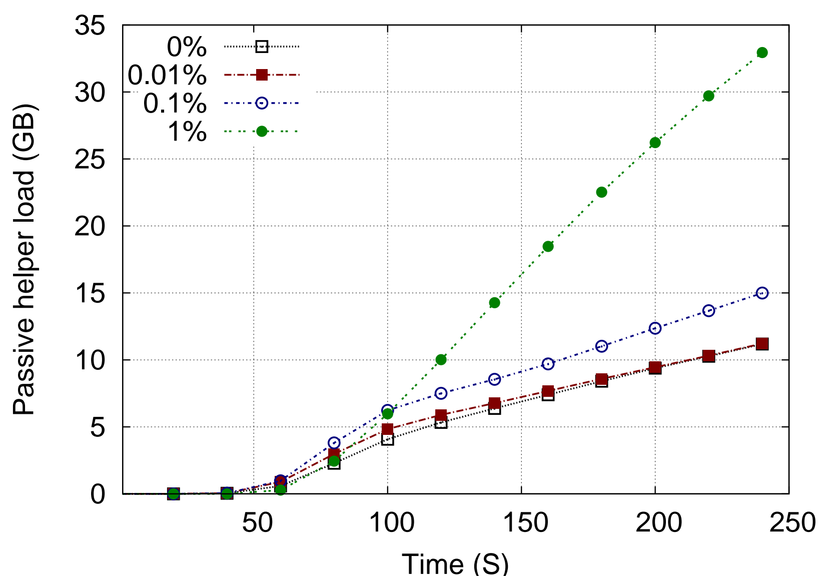
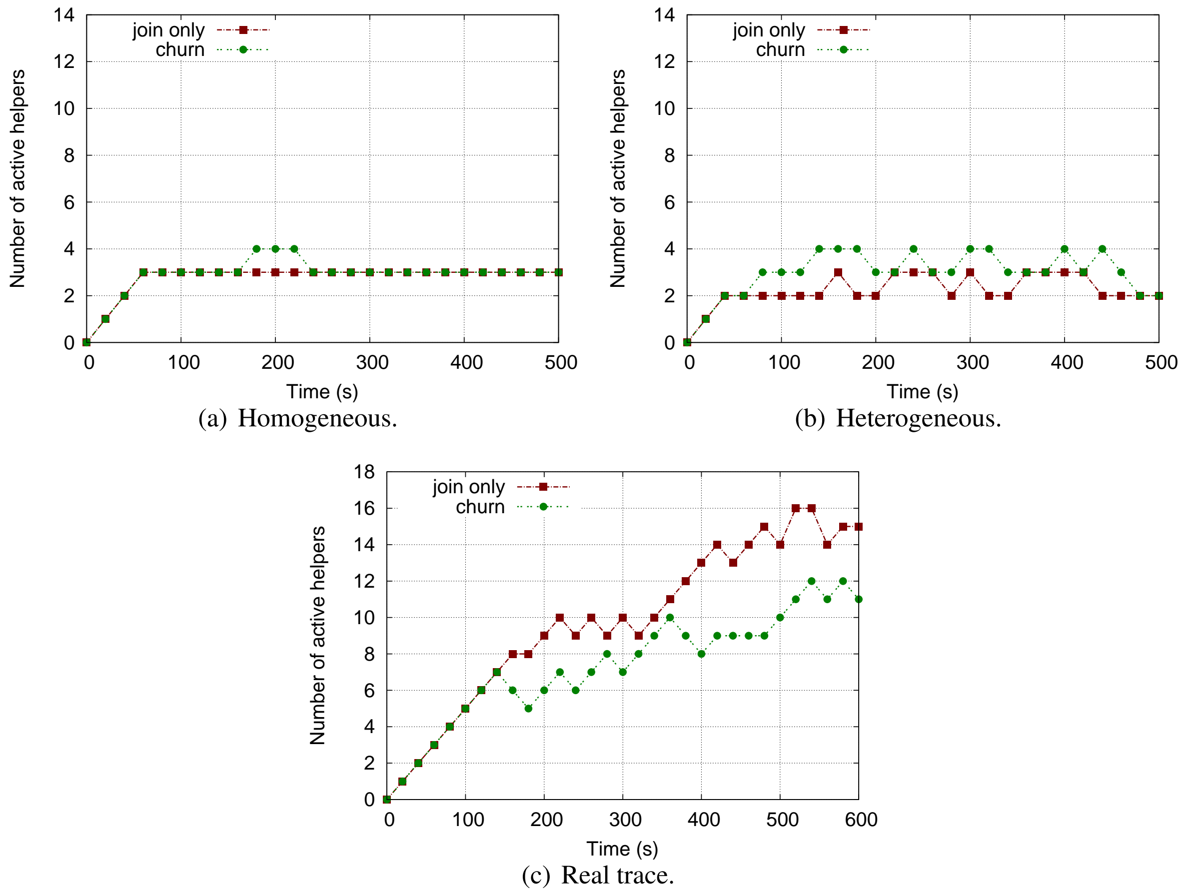


Figure 4.8: The cumulative PH load with different values of churn rates($LCW = 40$ chunks).

time for four different churn rates and LCW equals 40 chunks. As the figure shows, there is no big change in PH load under low churn scenarios (0.01$\%$ and 0.1$\%$), which are deemed realistic in deployed P2P systems [94]. However, it sharply increases in the presence of higher churn rates (1$\%$), because peers lose their neighbours more often, thus, they cannot pull chunks from the swarm in time, and consequently they have to fetch them from PH.

4.4.4 Economic cost

In this experiment, we measure the effect of adding and removing AHs on the total cost. Note, in these experiments we set LCW to 40 chunks, therefore, regardless of the number of AHs, all the peers receive 99% of the chunks before their playback time. In fact, AHs only affect the total cost of the service. In Section 4.3, we showed how CM estimates the required number of AHs. Figure 4.9 depicts how the number of AHs changes over time. In the join-only scenario and the homogeneous slot distribution (Figure 4.9(a)), the CM estimates the exact value of the peers that receive the chunks on time using the existing resources in the system, and consequently the exact number of required AHs. Hence, as it is shown, the number of AHs will be fixed during the simulation time. However, in the heterogeneous and real trace slot distributions (Figures 4.9(b) and 4.9(c)),

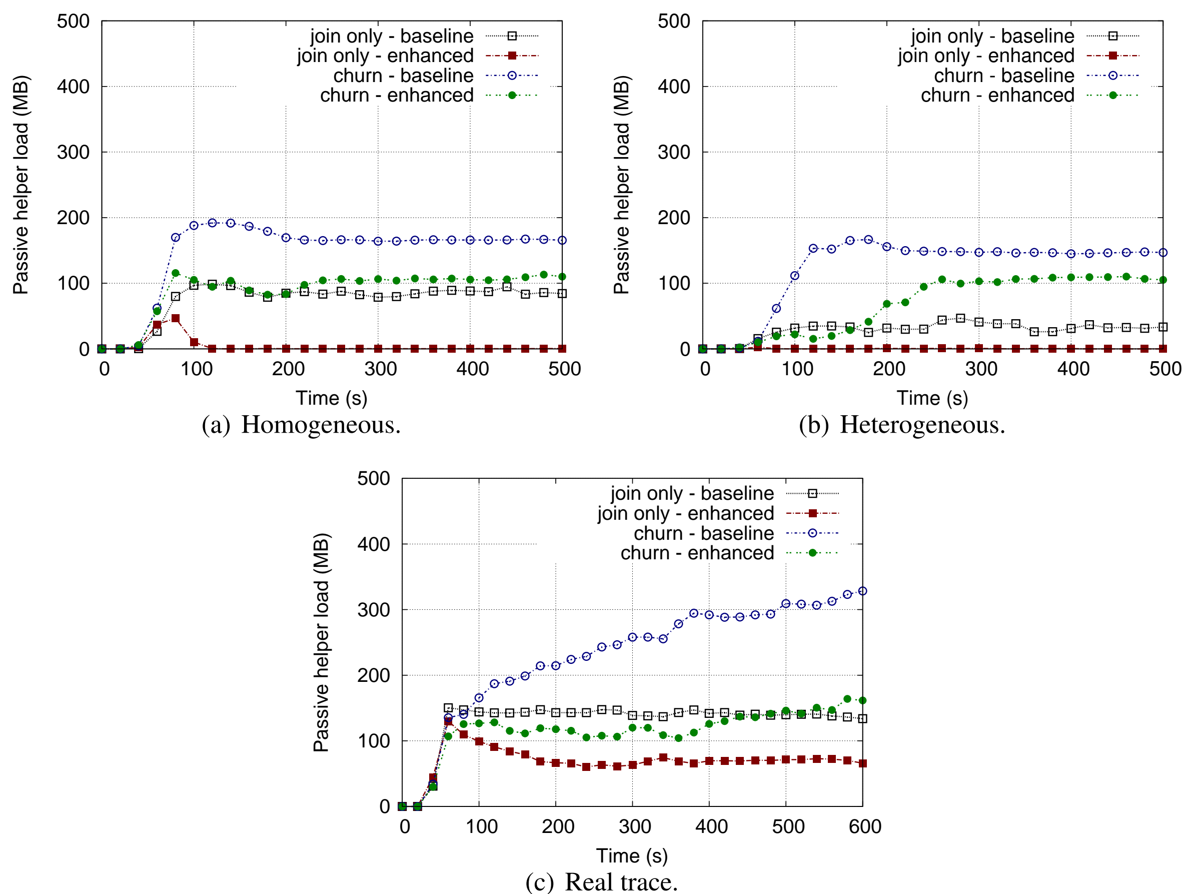


(a) Homogeneous.
(b) Heterogeneous.
(c) Real trace.

Figure 4.9: Number of AHs in different settings and scenarios.

CM estimation changes over time, and based on this, it adds and removes AHs. In the churn scenario (1% churn rate), CM estimation also changes over the time, thus, the number of AHs fluctuates.

Related to this, we see how PH load changes in different scenarios in the baseline and enhanced models (Figure 4.10). Figure 4.9(a) shows that three AHs are added to the system in the join-only scenario and the homogeneous slot distribution. On the other hand, we see in Figure 4.10(a), in the join-only scenario, with the help of these three AHs (enhanced model), the load of PH goes down nearly to zero. It implies that three AHs in the system are enough to minimize PH load, while preserving the promised level of QoS. Hence, adding more than three AHs in this setting does not have any benefit and only increases the total cost. Moreover, we can see in the join-only scenario, if there is no AH in the system (baseline model), PH load is much higher than the enhanced



(a) Homogeneous. (b) Heterogeneous.

(c) Real trace.

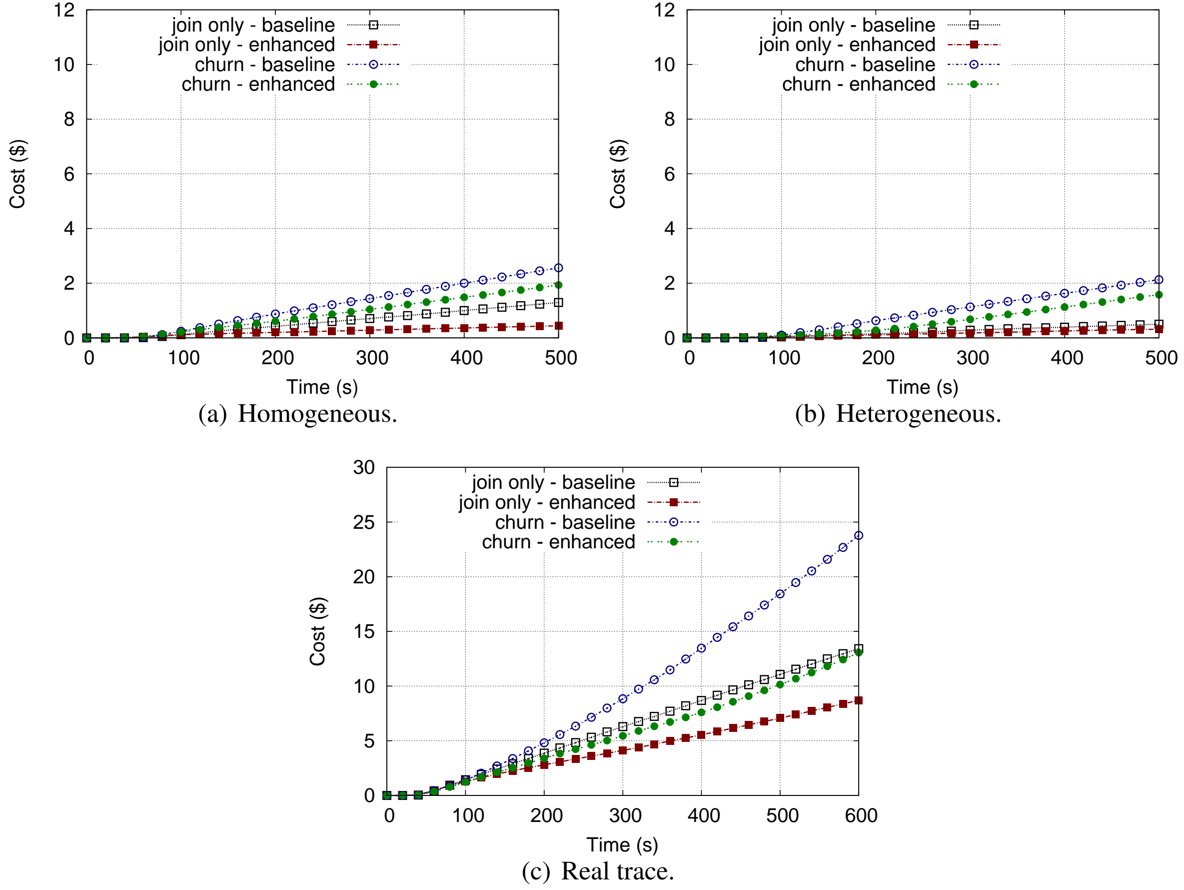
Figure 4.10: PH load in different scenarios with dynamic changes of the number of AHs.

model, e.g., around $90mb$, $40mb$, and $130mb$ per second in the homogeneous, heterogeneous, and real trace, respectively. The same difference appears in the churn scenario.

Figure 4.11 shows the cumulative total cost over the time in different scenarios and slot distributions. In this measurement, we use Amazon S3 as PH and Amazon EC2 as AHs. According to the price list of Amazon [1], the data transfer price of S3 is 0.12$ per GB, for up to 10 TB in a month. The cost of GET requests are 0.01$ per 10000 requests. Similarly, the cost of data transfer in EC2 is 0.12$ per GB, for up to 10 TB in a month, but since the AHs actively push chunks, there is no GET requests cost. The cost of a large instance of EC2 is 0.34$ per hour.

Considering the chunk size of $20kb$ $(0.02mb)$ in our settings, we can measure

[1] http://aws.amazon.com/, [Online; accessed Nov-2012]



(a) Homogeneous. (b) Heterogeneous. (c) Real trace.

Figure 4.11: The cumulative total cost for different setting and scenarios.

the cost of PH in Amazon S3 per round (second) according to the Formula 4.5:

$$\begin{aligned} C_{ph} &\approx r \cdot (C_{chunk} + C_{req}) \\ &\approx \frac{r \times 0.02 \times 0.12}{1000} + \frac{r \times 0.01}{10000} \end{aligned} \tag{4.8}$$

where r is the the number of received requests by PH in one round (second). The cost of storage is negligible. Given that each AH pushes chunks to 20 peers with the rate of $500kbps$ ($0.5mbps$), then the cost of running one AHs in Amazon EC2 per second according to Formula 4.4 is:

$$\begin{aligned} C_{ah} &= C_{vm} + m \cdot C_{chunk} \\ &= \frac{0.34}{3600} + \frac{20 \times 0.5 \times 0.12}{1000} \end{aligned} \tag{4.9}$$

Figure 4.11 shows the cumulative total cost for different slot distribution settings. It is clear from these figures that adding AHs to the system reduces the total cost, while keeping the QoS as promised. For example, in the high churn scenario (1% churn rate) and the real trace slot distribution the total cost of system after 600 seconds is 24$ in the absence of AHs (baseline model), while it is close to 13$ if AHs are added (enhanced model), which saves around 45% of the cost.

4.4.5 Accuracy evaluation

In this section we evaluate the accuracy of our estimations in form of evaluating the accuracy of upload slot distribution, and the accuracy of estimating the number of infected peers.

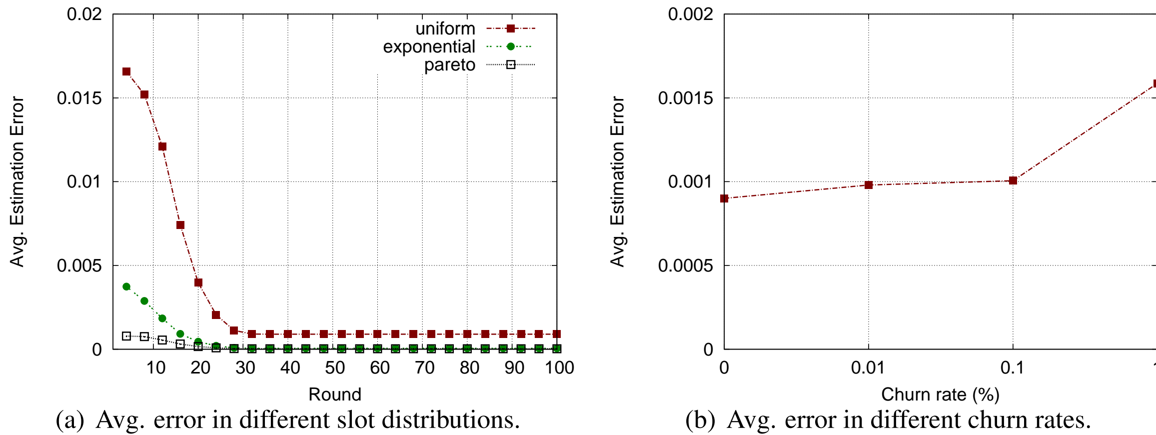
Upload slot distribution estimation

Here, we evaluate the estimation of upload slots distribution in the system. We adopt the Kolmogorov-Smirnov (KS) distance [84], to define the upper bound on the approximation error of any peer in the system. The KS distance is given by the maximum difference between the actual slot distribution, ω, and the estimated slot distribution, $E(\omega)$. We compute $E(\omega)$ based on P_ω for different number of slots. Since the maximum error is determined by a single point (slot) difference between ω and $E(\omega)$, it is sensitive to noise. Hence, we measure the average error at each peer as the average error contributed by all points (slots) in ω and $E(\omega)$. The total average error is then computed as the average of these local average errors.

We consider three slot distributions in this experiment: (i) the uniform distribution, (ii) the exponential distribution ($\lambda = 1.5$), and (iii) the Pareto distribution ($k = 5, x_m = 1$). Figure 4.12(a) shows the average error in three slot distributions, and Figure 4.12(b) shows how the accuracy of the estimation changes in different churn rates.

Number of infected peers estimation

Finally, we evaluate the estimation accuracy of the number of infected peers. Figure 4.13 shows the real number of infected peers and estimated ones in three



(a) Avg. error in different slot distributions.

(b) Avg. error in different churn rates.

Figure 4.12: Avg. estimation error.

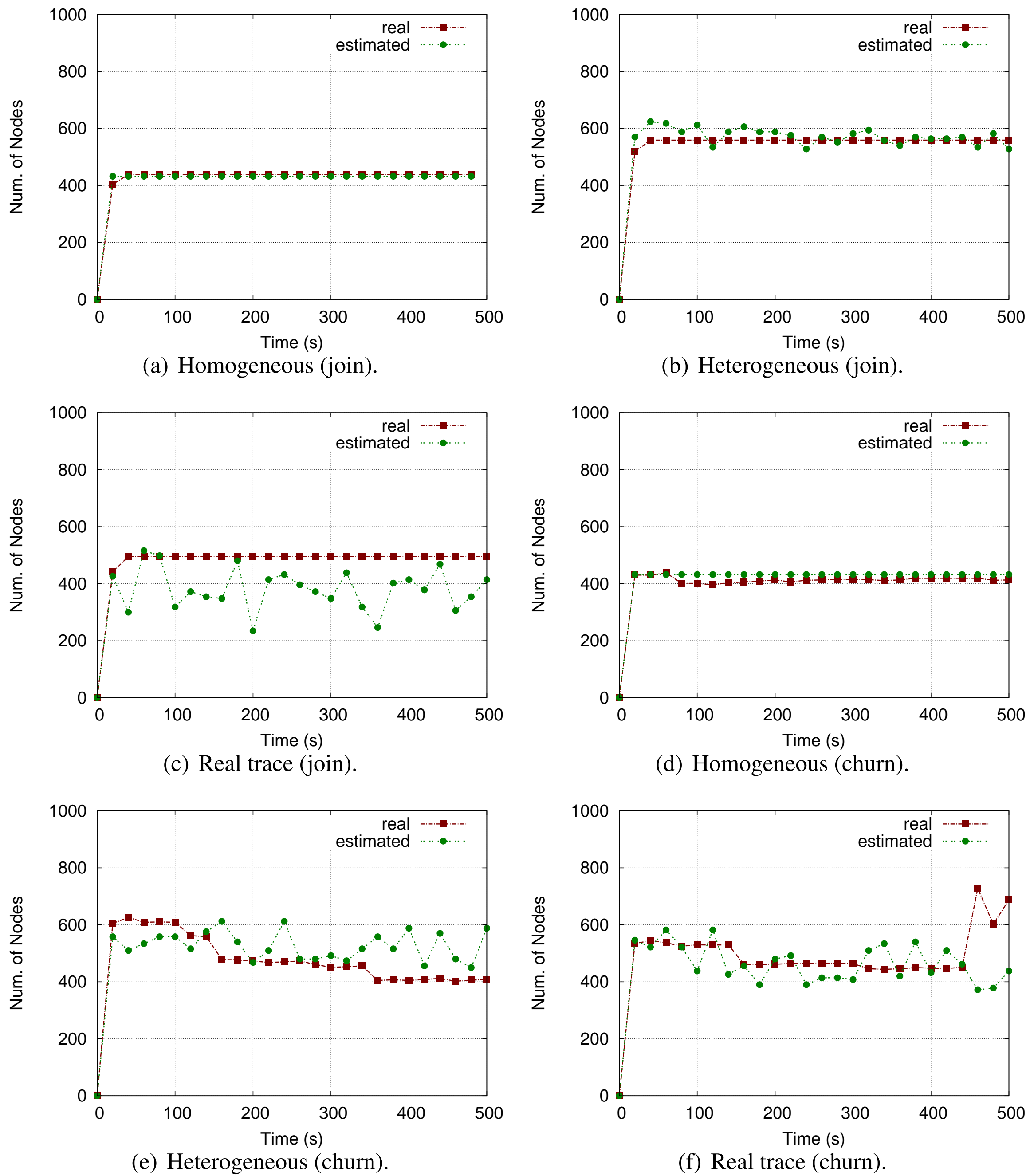
upload slot distributions and in join and churn scenarios. As shown in the homogeneous and heterogeneous slot distributions, our estimation of the number of infected peers closely fits the real number of such peer. However, in the real trace slot distribution, it may happen that a peer without upload slot connects directly to the source and prevents other peers to join the system, or on the other hand, a very high upload bandwidth peer joins close to the source and serves many other peers. That is why we see more difference between the real and estimated number of infected peers in the real trace slot distribution.

4.5 Related work

4.5.1 Content distribution

Although P2P algorithms are emerging as promising solutions for large scale content distribution, they are still subject to a number of challenges [108]. A typical problem is the bottleneck in the aggregated upload bandwidth in the overlay [53] that can lead to a low stream quality with disruptions. The bottleneck is caused by asymmetric bandwidth of the clients: the download bandwidth is usually much higher than upload.

In order to address these issues some recent research works propose to use the hybrid architecture combining Content Delivery Networks (CDN) and P2P overlays [57, 35]. Most of these works are focused on the reducing the use of CDN servers via utilization of P2P available resources whenever it is pos-



(a) Homogeneous (join). (b) Heterogeneous (join).

(c) Real trace (join). (d) Homogeneous (churn).

(e) Heterogeneous (churn). (f) Real trace (churn).

Figure 4.13: The comparison between the real number of infected nodes and the estimated ones.

sible. This kind of infrastructures is similar with the baseline model we consider. However, in addition we propose an enhanced architecture that allows to regulate the amount of upload bandwidth in the swarm via the utilization of additional cloud resources.

The most relevant work with respect to ours is LIVESKY, developed by Yin et al. [107]. The authors proposed a commercial deployment of the hybrid architecture, addressing several key challenges, including dynamic resource scaling while guaranteeing stream quality. However, while the authors consider redirection of the users according to available upload resources, we propose an approach for managing the amount of available upload bandwidth via swarm feedback processing.

One more possible approach to increase the upload capacity is to use peer helpers [101, 109]. The helper role can be played by *idle* [102] or *restricted* [52] users. Idle users are peers with spare upload capacity that are not interested in any particular data, while restricted users are users with limited rights to use the network service. Another approach suggests to exploit dedicated servers as helpers to accelerate content distribution [61, 95, 96], where the servers cache and forward content to other peers. Montresor and Abeni [63] introduced an alternative way to use dedicated servers. They proposed to merge P2P and cloud storage to support information diffusion.

In addition to these solutions, Wu et al. proposed a queuing model in [104] to predict the dynamic demands of the users of a P2P video on demand (VoD) system providing elastic amounts of computing and bandwidth resources on the fly while minimizing the cost. Similarly, Jin et al. presented a cloud assisted system architecture for P2P media streaming among mobile peers to minimize energy consumption [44]. Unlike all the described approaches, our work exploits cloud computing and storage resources as a collection of active and passive helpers. The combination of both types of helpers together with an effective resource management, distinguishes our approach from previous work.

4.5.2 Self-monitoring and self-configuration systems

Self-monitoring and *self-configuration* mechanisms are essential to manage large, complex and dynamic systems in an effective way. Self-monitoring detects the current states of system components, while self-configuration is aimed to adapt system configuration according to the received information.

Self-monitoring allows the system to have a view on its current utilization and state. One of the popular approaches for monitoring P2P overlays is decentralized aggregation [40]. For example, ADAM2 [82] presents a gossip-based

aggregation protocol to estimate the distribution of attributes across peers. Similarly, Van Renesse and Haridasan [34] propose a distribution estimation mechanism, which can be used to aggregate not only the values of different peers but also how the values are ranked in relation with each others. Another system that uses aggregation is CROUPIER [21], which is a NAT-aware peer sampling service and use aggregation protocol to estimate the ratio of open peers in the network.

Self-configuration is the process that autonomously configures components and protocols according to specified target goals, e.g., reliability and availability. To self-tune according to on-going state, the system can use an external component that controls the system either via control loops [48] or in a decentralized way [5, 41, 47]. A relevant example of a self-configuration mechanism is T-MAN [41], an overlay topology management that uses a ranking function exploited locally by each peer to choose its neighbors. Another system, proposed by Kavalionak and Montresor [47], considers a replicated service on top of a mixed P2P and cloud system. This protocol is able to self-regulate the amount of cloud storage resources utilization according to available P2P resources. However, the main goal of the proposed approach is to support a given level of reliability, whereas in our work we are interested in an effective data dissemination that allows to self-configure the amount of active and passive cloud resources utilization.

Chapter 5

Virtual game environment

Cloud computing has recently become an attractive solution for massively multiplayer online games, also known as MMOGs, as it lifts operators from the burden of buying and maintaining large amount of computational, storage and communication resources, while offering the illusion of infinite scalability. Yet, cloud resources do not come for free: a careful orchestration is needed to minimize the economical cost. In this chapter, we describe a novel architecture for MMOGs that combines an elastic cloud infrastructure with user-provided resources, to boost both the scalability and the economical sustainability provided by cloud computing. Our system dynamically reconfigures the platform while managing the trade-off between economical cost and quality of service, exploiting user-provided resources whenever possible. Simulation results show that a negligible reduction in the quality of service can reduce the cost of the platform up to 60% percent.

5.1 Architecture

This section introduces the overall structure of the proposed MMOG hybrid architecture. In order to motivate our design choices, we first briefly review the main characteristics of a classical client/server architecture for MMOGs.

Players connect to a centralized server by means of a *game client*, whose main task is to show on the screen the visual representation of the virtual environment and to map the actions of the player (i.e. movements and/or interactions with objects) into communications with the server. The virtual environment is populated with *entities*, which can be *avatars* representing players or

objects that can be manipulated. The player actions can be classified as *positional actions* and *state actions* [38]. The former correspond to the movement of entities across the virtual environment, while the latter correspond to changes to their state, e.g. the act of closing a door or collecting an object.

When an action occurs, the server must spread the information to the other players. However, not all players are interested in all actions. In fact, players only receive actions executed inside an area centered at the their virtual position, called *area of interest* (AOI). It is a task of the server to dynamically update and maintain the AOI of the players.

5.1.1 Distributed MMOG

Distributed MMOG architectures have to devise strategies to divide the virtual world into *regions*, and to find a proper assignment of these regions to multiple *nodes* (we use this generic term to indicate either peers run by the user, or servers run by the MMOG operator, as well as virtual machines run inside a cloud). Some architectures adopt a spatial division of the virtual environment into regions, and assign all the entities in a region to a node. Region-based partitioning is efficient for AOI resolution; since entities are clustered according to their spatial position, identifying entities included in an AOI is a relatively easy task. However, entities distribution is normally not spatially uniform, due to the presence of *hotspots*, i.e. regions with high concentration of entities. One of the drawback of hotspots is that they generate a large amount of load on the nodes managing them. Furthermore, due to the spatial division, positional actions may trigger a change into entity-region assignment, implying the transfer of entities among nodes. This may reduce the interactiveness of the game, given that entities are not accessible during transfers, in fact denying any possible state action on it. This is even more critical when considering the rate of transfer that in turn depends on the rate of positional actions (usually high) and the dimension of the regions.

By comparison, an hash-based entity assignment presents complementary characteristics. Since the association of an entity to a node does not depend on the position of the entity, positional actions do not trigger any migration of objects among nodes. Also, due to the random assignment, entities in a hotspot are managed by several nodes, whose load is uniformly distributed. However,

this solution makes AOI resolution impractical (the objects of an AOI may be spread among different nodes) and therefore it is rarely used in practice.

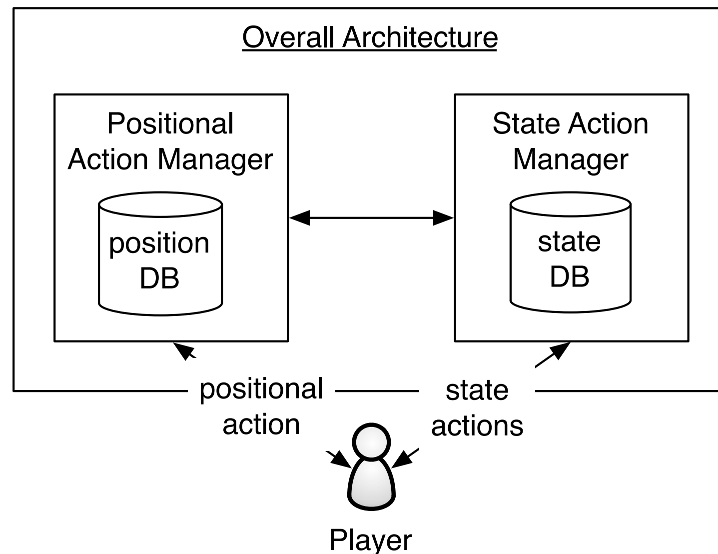


Figure 5.1: Overall architecture

5.1.2 The proposed architecture

In order to retain the advantages of both the entity assignment strategies discussed above, we propose a distributed MMOG architecture (shown in Figure 5.1) that exploits two components, each one managing a different kind of actions. The *positional action manager* (PAM), which we previously presented in [13]), manages the positions of the entities by organizing a epidemic-based distributed overlay among players. The *state action manager* (SAM), which is the focus of this chapter, stores the entity state and is organized according to a random entity-to-node assignment. This assignment strategy enables to handle the state of the entities without any transfer of them across nodes due to positional actions. Such transfers may anyway occur, but instead of being triggered by positional actions, they are usually performed to optimize the distribution of the entities (and as a consequence, of the load) among the nodes.

5.1.3 State action manager

In order to build and maintain an overlay for the management of the entity state in the MMOG, the state action manager (see Figure 5.2) is based on a distributed

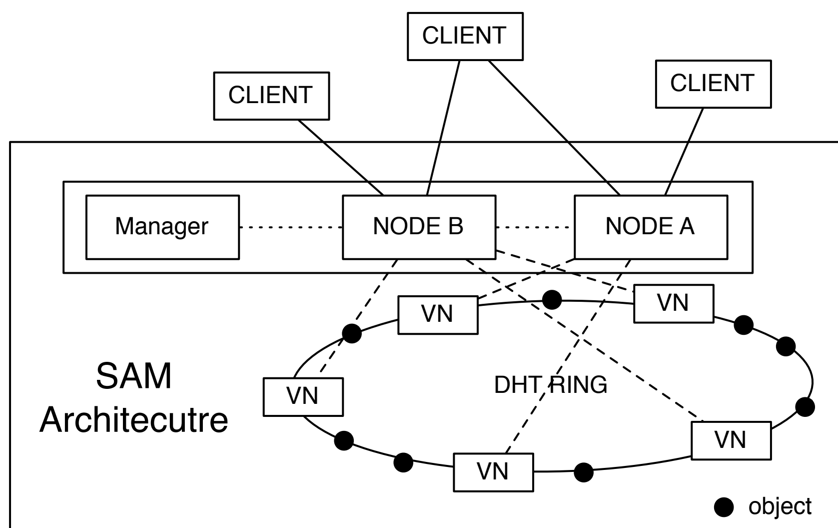


Figure 5.2: Black dots are the objects inside the virtual environment. VN boxes corresponds to virtual nodes. Node A manages 2 VNs, whereas node B manages 3 VNs. Client connects to the nodes to modify and read the objects. The manager has a global knowledge of the state of the node and the VNs.

hash table (DHT) [93, 91]. A typical DHT manages a logical address space, whose size is large enough to avoid clashes among items (i.e. a common size is 2^{160}). Each entity of the MMOG (avatar, objects) is assigned with an address in such space, which we refer to as its ID. The IDs are uniformly assigned to balance the distribution of the entities in the address space. The address space is partitioned among the nodes, together with the associated entities. Nodes are then connected to each other by an overlay, for routing and synchronization purposes. The overlay is built to guarantee $O(\log N)$ bounds, where N is the number of nodes in the DHT, both for the routing hops and for the size of the routing tables.

In addition to the typical DHT mechanisms, we adopt the *virtual node* (VN) paradigm over DHTs proposed by Godfrey et al. [30], to introduce a clear separation between the logical and the physical nodes. Each virtual node is in charge of an address range of the DHT. Several virtual nodes may be allocated on the same physical node. From a client perspective, a virtual node acts as a state server for a set of entities. Since the entities that a client is interested in may be managed in principle by different virtual nodes, each client may have multiple simultaneous connections to them. For instance, in Figure 5.2, a client is connected with node A and B at the same time. As a limit situation, each player can

connect to a different node per each object, so that the number of connections for each player is bounded by the amount of entities in its AOI. Nevertheless, it is worth noticing that not all the connections are active altogether, since each VN pushes the updates of the entities only when they happen.

In our architecture, we define the *load* of a VN as the upload bandwidth consumed to broadcast entities state to its associated clients. The load depends on the amount of entities that correspond to the VN and the amount of clients accessing them. The load changes over time, according to the interaction pattern of the avatars. Moreover, load may be unbalanced due to the presence of more popular entities. For instance, objects belonging to an hotspot receive an higher amount of updates.

The proposed architecture also includes an additional module, called *manager*, whose goal is to distribute the load among the nodes, so to exploit their heterogeneity. DHT nodes periodically notify the manager with their own load information. The manager periodically computes new assignments node-VNs based on the received information and, if necessary, the enrollment or the disposal of nodes from the DHT. In this scenario, the adoption of the VN paradigm yields concrete advantages: (i) more powerful nodes may receive an higher number of VNs than less powerful ones, (ii) heavy loaded nodes may trade VNs with unloaded ones, (iii) in case of a physical node failure, its VNs are possibly transferred/reassigned to different, unloaded, physical nodes, so reducing the risk of overloaded nodes. Moreover, migrating VN is easy and light. Their migration does not affect the organization of the address space at the DHT level. It only requires the exchange of data managed by the VN as well as the update of the mapping between the logical identifier of the VN and the physical address of the node hosting it.

5.1.4 Virtual nodes

One of the main advantages of the virtual node approach is the possibility to easily move entities across the nodes of the DHTs. This ability is a fundamental requisite for enabling proactive load distribution mechanisms. To better understand the advantages on exploiting virtual nodes, let us spend a few words on the load distribution in classical DHTs (i.e. that does not employ virtual nodes). There are essentially two ways to dynamically distribute the load in classical

DHTs:

1. *Move nodes*. An unloaded node (i.e. A) moves to a precise address of the DHT, so to unload a heavy loaded node (i.e. B). This operation requires A to leave the DHT and rejoin in a position so that part of the load form B is transferred to A. Even if this approach may work in a general situation, it is too time consuming and creates too overhead for a live application as a virtual environment. To fully understand the process, let us consider C as the successor of A (i.e. the node that is after A in the ring-shaped space of the DHT)[1]. When A leaves, C becomes responsible of the address space left free by A. This information must be spread in the DHT, so that the routing for the former A address space points correctly to C. In addition, before leaving, A must transmit all the data on its entity descriptors to C. When A joins the DHT and becomes the predecessor of B, this information must be spread to the DHT to adjust routing path. B also must send to A the entity descriptors that are in the new address space of A. In addition, A must build its routing table, in order to be part of the overlay. In summary, this process requires two entities transferring (from A to C and from B to A), to spread new information about 3 nodes and to build a new routing table. All these operation take time, and, most important, imply a large number of transferred data during which the entities are not reachable from clients.

2. *Move descriptors*. This technique requires moving the entity descriptors among nodes to distribute the load. Practically, an entity descriptor changes its ID in the ring-shaped address of the DHT. During the transfer of the descriptor, the entity is not accessible by clients. However, the most relevant drawback of such approach is that any time a client accesses to a new entity, it must query the DHT for its position. This requires to wait up to $\log N$ steps, which may be too long with an high number of nodes.

With virtual nodes, load distribution is lighter and more flexible with respect to a classical DHT. Node directly exchange virtual nodes (a process that we call *virtual node migration*), which offers the following advantages:

[1]We consider Chord in this example, but with small differences the following considerations are valid for other DHT implementation as well

- the ID of the entity does not change over time;
- it is possible to transfer load without nodes to leave the DHT;
- a virtual node that has moved does not have to rebuild its entire routing table. In fact, moving a virtual node requires only to stabilize a few routing paths, which is less than in a classical DHT system;
- it is possible to partially increase or decrease the load of a node.

During a VN migration, entities of the VN cannot be accessed. In other words, players cannot interact with the objects inside the VN that is migrating. Also, it can be the case of a player modifying the state of the object locally, just to see it reverted back when the migration of the VN is completed. To this end, it is important to keep the transition time as short as possible, in order to provide an acceptable level of interactivity for the VE clients.

5.1.5 Replication and fault tolerance

In a distributed system, the need of replication comes from the intrinsic unreliability of nodes. Since we target an heterogeneous system including both peer and cloud nodes, a fair orchestration of replication is a relevant issue. Our approach is based on the reasonable assumption that, in general, cloud nodes can be considered *reliable* whereas peer nodes are *unreliable*, due to the high degree of churn which characterizes P2P systems. This difference is mainly due to the lack of control over peers, which are prone to unexpected failures, and may leave the system abruptly. On the other hand, cloud nodes generally belong to a stable infrastructure based on virtualization, and this greatly increases their robustness and flexibility.

In order to cope with the unreliability of peers, we propose that every VN assigned to a peer is always replicated. The replica, called *backup virtual node* (bVN), is then assigned to a trusted resource, i.e. a cloud node. To keep the state of the bVN up-to-date with the original, peers send periodic updates to the cloud nodes. The replica schema adopted is *optimistic* [83], i.e. players can access to entities without previous synchronization between the regular VN and the relative bVN. This schema leads to *eventual consistency*, favouring availability over consistency of the entities. The periodic updates from the peer to

cloud for synchronizing bVN add further bandwidth requirements. However, the synchronization is performed at relatively large intervals (e.g. 30 seconds) with respect to the player updates, to reduce the required bandwidth.

The presence of bVNs guarantees a certain degree of availability in case of peer failures. Consider a peer P that manages a single VN and cloud node C that manages the respective bVN. When P departs from the system, either abruptly or gracefully, C becomes the new manager of the primary replica. As a consequence, clients connected to P must now connect to C. In the case of a gracefully departure of P, P itself may inform them about the new role of C; otherwise, the involuntary departure of P can be detected either by C, since it receives no more updates from P, or from the DHT neighbors of P, due to the repairing mechanism of DHTs. These nodes are able to notify the clients to send their notification to C.

5.2 Problem statement

Let time be subdivided in discrete time steps of length Δt and denoted by $t \in \mathbb{N}$. Let VN be the set of virtual nodes in the system. Then, $\forall v \in VN$ we define $v_{load}(t)$ as the bandwidth consumed by the virtual node v between the time step of length Δt between $t-1$ and t, with $v_{load}(0) = 0$. Similarly, $\forall v \in VN$ we define $v_{ent}(t)$ as the number of entities managed by v at time t, with $v_{ent}(0) = 0$. Each virtual node in VN is assigned to a node.

The set $N(t)$ contains the nodes that are in the system at the time t. An arbitrary node $n \in N(t)$ is characterized by the following invariant properties: (i) bandwidth capacity n_{cap}, (ii) bandwidth cost n_{bcost}, (iii) renting cost n_{rcost}. Over time, nodes are assigned with virtual nodes. We indicate with $n_{VN}(t)$ the set of virtual nodes assigned to a node n at time t. The outgoing bandwidth load imposed on a node at time t is indicated as $n_{load}(t) = \sum_{v \in n_{VN}(t)} v_{load}(t)$. From this, we define $n_{LF}(t) = n_{load}(t)/n_{cap}$ as the load factor of n at t. Finally, we indicate with $n_{pl}(t)$ the number of players served by n at time t.

We consider two different kinds of nodes, virtual machines rented from a cloud and peers provided by users. User-provided resources have no associated cost for bandwidth and renting, while cloud nodes are assigned with a pricing model taken from Amazon EC2 [2]), with the assumption that we can charge

[2]Amazon Elastic Compute Cloud: "http://aws.amazon.com/ec2/"

Table 5.1: Table of symbols

Symbol	Description
t	generic time step
Δt	length of a time step
VN	set of virtual nodes
$v_{load}(t)$	upload bandwidth load in byte of v at time t
$v_{ent}(t)$	entities managed by v at time t
$N(t)$	set of nodes at time t
n_{cap}	capacity of n
n_{bcost}	upload bandwidth cost per byte of n
n_{rcost}	renting cost of N at t
$n_{load}(t)$	upload bandwidth load in byte of n at time t
$n_{lf}(t)$	load factor of n at time t
$n_{vs}(t)$	the set of virtual nodes managed by n
$n_{pl}(t)$	player being served by n at time t
$\alpha(t)$	$[0, 1]$ QoS of the platform at time t
α_{thres}	QoS threshold
$\beta(t)$	cost in USD of the platform at time t
$DU(\Delta t)$	delayed updates during Δt
$U(\Delta t)$	total updates during Δt
z	number of updates per second
$E[latency_{i,j}]$	expected latency between nodes i and j
$E[fail_n]$	expected failure probability for node n

cloud nodes per unit of time Δt. Hence, it is possible to compute the cost per time unit as the sum of the bandwidth cost and the renting cost of the nodes, according on the bandwidth consumed at time t. The total system cost $\beta(t)$ is computed as follows:

$$\beta(t) = \sum_{n \in N(t)} ((n_{load}(t) * n_{bcost}) + n_{rcost}) \tag{5.1}$$

In this formulation the cost due the upload bandwidth changes over time. Rather, the renting cost depends on the number of cloud nodes exploited.

5.2.1 Quality of service

Each virtual node offers a service that is comparable to a publish/subscribe system [36]. Players subscribe to nodes, send inputs and receive back updates at a

fixed rate. The specific rate depends on the particular MMOG genre, typically in the order of few updates per second. Here we define this frequency as $z = 4$, corresponding an update every 250ms which fits medium-paced MMOGs [20]. When updates are delayed, players may perceive a clumsy interaction with the virtual environment. If the number of consequent delayed updates is large, players might not be able to interact with the environment at all. To favour a fully interactive virtual environment, the rate of updates should be as stable as possible. We consider the capacity of the nodes to provide a constant rate of updates as the metric for the *Quality of Service* (QoS).

In general, there are two main causes for delayed updates: (i) the network infrastructure between the node and the client, and (ii) the ability of the node to send the updates in time. In the first case, since we assume the Internet as the communication media, latency spikes and jitter are responsible of delayed updates. In this thesis we do not consider this issue, since it is general for any architecture. Moreover, several solutions (such as LocalLag [59]) have been proposed to mitigate the effects of network delays in MMOGs.

In this chapter we consider the second case (i.e. the delays generated by servers), as it is greatly affected by the exploitation of user-provided resources. In fact, peers are more prone to delay updates rather than a datacenter server, given their smaller reliability and limited bandwidth capability.

As QoS measure, let $\alpha(t) \in [0, 1]$ be the fraction of updates the nodes send within time t for all the entities whose state changed at time $t-1$. For example, if a given time t the nodes successfully sent only half of the updates for all the entities whose state is changed at time $t - 1$, then $\alpha(t) = 0.5$. Let us define $U(t)$ as the total number of updates from $t - 1$ to t and $DU(t)$ as the number of delayed updates from $t - 1$ to t. Then:

$$\alpha(t) = 1 - \frac{DU(t)}{U(t)} \tag{5.2}$$

There are three cases in which the nodes might incur in delayed updates:

- *Virtual node migration.* During a migration, the service is unavailable for the time the entities are transferring between nodes. In this case, the number of delayed updates depends on the *migration time* (MT, the times for a VN to migrate between nodes), which is discussed in Section 5.4.2.

- *Overloading*. When a node is overloaded, it simply does not have enough bandwidth to send updates. As a consequence, it either drops or delays some updates.
- *Failures*. When a node crashes and a back up node takes its place, the players need time to "know" the new updates provider. During this time the players are not receiving new system updates.

In a sense, migrations and overloading can be seen as a "necessary evil", as they trade some QoS in exchange of more flexibility. Hence we do not take them into account when we compute the target QoS for virtual node assignment. The migration is a graceful process that we can tune to minimize the affection on the QoS. In particular, the size of the virtual nodes can be chosen so that their MT remains under a definite time threshold. The details of this aspect and the tuning are discussed in Section 5.3.3.

Node overloading happens when the prediction function would compute an under-estimation of the load. In this case, less node than the necessary are recruited, causing a reduction in the QoS. To give the possibility to the operators to control the overloading, we define LF_{up} as an upper bound for the nodes load factor. By setting this parameter, operators can force the nodes to work under their capacity, as a node n will accommodate up to $LF_{up} * n_{cap}$ load. The LF_{up} parameter has to be carefully selected, as a high value can cause overloading, whereas a low value may cause resource over-provisioning. Ideally, LF_{up} should be able to cope with the error of the prediction function without affecting the economical cost of the infrastructure. An empirical evaluation of these factors, as well as the tuning of the LF_{up}, is presented is Section 5.4.

Due to the above considerations, we consider failures as the main cause of delayed updates. Delayed updates from failures depend on (i) the probability that a node fails during Δt and (ii) the time the infrastructure needs to recover from the failure, and (iii) the number of player accessing the node at the time of failure.

Assuming a model that describes failures of the nodes over time, we define $E[fail_n]$ as the expected probability for a node to fail during an arbitrary time step. When a node crashes, the backup node becomes the new node (see Section 5.1). The *failure time* (FT, in seconds) is the time that goes from the failure of a node to the moment the players have the information about the new node.

Let us define T_f as the time-out needed for a backup node to notice the failure of the node. If a node does not communicate for T_f seconds with the backup node, it is considered failed. Also, let us define as $E[latency]$ as the expected latency between the backup node and the players. Then, $FT = T_f + E[latency]$. According to the definitions above we have:

$$DU(t) = (FT \times z) \sum_{n \in N(t)} (E[fail_n] \times n_{pl}(t)) \tag{5.3}$$

To let operators control the QoS, we then define the system-wide parameter α_{thres}. It represents the percentage of successful updates that must be kept by the platform. For instance, with $\alpha_{thres} = 0.99$, only 1% of updates can be delayed due to failures. Note that α_{thres} indirectly controls the assignment of the virtual nodes between user-provided and cloud nodes.

5.2.2 Problem statement

Our aim is to provide an assignment of the virtual nodes to the nodes that respects the bound defined by the operator to control the quality of service, while keeping the economical cost as low as possible. Hence, we define the problem of assigning virtual nodes to nodes as follows:

Problem Statement. *Find an assignment of virtual nodes to nodes to minimize $\beta(t)$ such that: $\alpha(t) \geq \alpha_{thres}$, and $n_{LF}(t) < LF_{up}$ for every $n \in N(t)$.*

5.3 Virtual node allocation

The task of the *manager* is to compute a virtual node assignment respecting the constraints defined in the problem statement in the previous section.

The work of the manager is divided into time intervals, which we refer to as *epochs*. Figure 5.3 shows the management of two consecutive epochs. During an epoch, the manager executes the following: (i) instantiates or releases on-demand nodes from the cloud, and migrates the virtual nodes according to the assignment plan done in the prior epoch (Section 5.3.3), and (ii) computes the new assignment for the next epoch (Section 5.3.2). The new assignment is computed with an heuristics based on the load prediction for the next Δt time

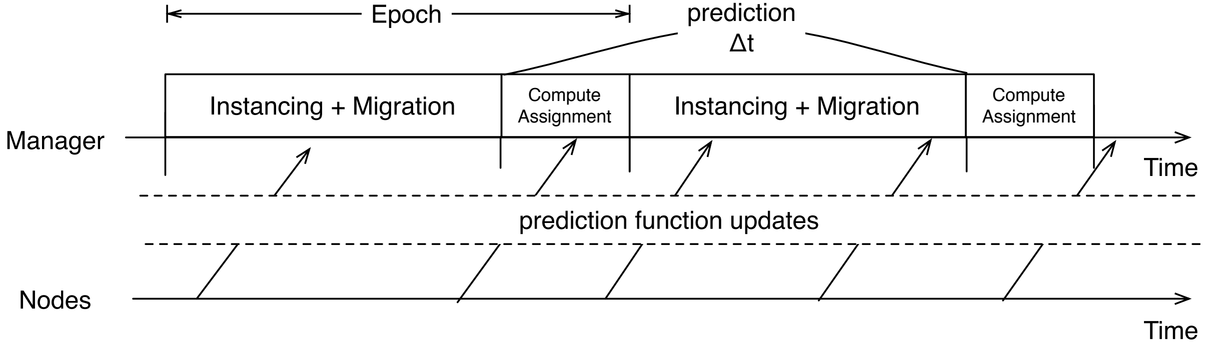


Figure 5.3: Time management

units. Over time, the manager receives updates from the nodes about their load. These updates are not synchronized with the epochs. If an update arrives when the new assignment computation is already started, it will be considered in the next epoch.

The duration of an epoch (which we refer to as τ_{epoch}) must be tuned to accommodate the instantiation time provided by the cloud platform chosen, which normally is in the order of few minutes [58]. Due to the fact that we use an heuristics to compute the assignments, τ_{epoch} is largely occupied by the instantiation time. As a consequence, in the following we assume that $\Delta t \approx \tau_{epoch}$.

5.3.1 Load prediction

The manager computes the load of the virtual nodes by using a prediction mechanism for each of them. The manager stores, for a virtual node v, the data necessary to forecast the load of v at arbitrary time. We refer to this data as L_v.

For example, by considering a prediction mechanism based on a simple exponential smoothing:

$$v_{load}(t+1) = \alpha(\sum_{i=0}^{T} v_{load}(i)(1-\alpha)^{i-1}) \tag{5.4}$$

then L_v would be the historical observations of the load of v up to time t. Over time, the *manager* receives renewed load estimation functions from the nodes. Indeed, L_v is computed locally by each node, and then sent to the manager (see Algorithm 5.1). Periodically nodes check the error between the observed load value and the predicted value computed using L_v on the manager. If the error

is larger than a predefined threshold ξ_{est}, then L_v is updated and sent to the manager.

```
Data: managerAddress, the IP of the manager
repeat
    foreach v ∈ VN do
        if |predictedLoad(L_v) − observedLoad| ≥ ξ_est then
            L_v ← update(observedLoad)
            msg ← add(v, L_v)
    if msg.size ≠ 0 then
        send(msg, managerAddress)
until true
```

Algorithm 5.1: Server's load estimation

In our implementation we exploited an *exponential smoothing* function [29] to predict load trends. This model assured a good prediction power in spite of its simplicity. Nevertheless, the described approach in principle allows us to apply a wide range of statistical models for the load estimation, as for example autoregressive models for data prediction such as ARMA or ARIMA [62]. The choice of the model depends on the expected data fluctuations and the desired accuracy of the prediction ξ_{est}. ξ_{est} represents a reasonable error in the load estimation due to the choice of the estimation model. High accuracy estimation models predict the load trend for large times interval Δt ahead. On the other hand, these models require intensive computation and are not suitable for fast-pace applications like virtual environments. An in-depth analysis of different prediction mechanisms is left as future work.

5.3.2 Virtual Nodes Assignment

The virtual node assignment is computed exploiting an heuristics and according to the predicted system state and the thresholds defined by the operator. For the sake of presentation, we divide the heuristics in two sub-tasks, *virtual node selection*, and *destination selection*.

Virtual Node Selection The aim of this task is to mark the virtual nodes to be migrated, adding them to vn_{pool}. Note that in this phase the manager works on an in-memory representation of the system, and that the actual migrations are executed once the virtual node assignment plan is defined.

```
input : LF_up, upper load factor threshold
input : P_size, the min amount of VNs to consider per epoch
input : α_tresh, QoS threshold
output: vn_pool, the list of virtual nodes to migrate
// Add VNs of overloaded nodes
foreach n ∈ N(t) do
    while n_LF(t + 1) > LF_up do
        vn_pool ← maxDerivative(n_vn(t))

// Add VNs to control QoS
while α(t + 1) ≥ α_tresh do
    VP ← v ∈ VN : v is assigned to a user resource
    vn_pool ← maxDerivative(VP)

// Add backed up VN
vn_pool ← vn_pool ∪ backUp()
// Removing VN from unused clouds
if size(vn_pool) < P_size then
    VC ← v ∈ VN : v is assigned to a cloud resource
    Sort VC in ascending order according to nodes predicted load factor
    vn_pool ← (P_size − size(vn_pool)) VNs from VC;

// Anyway perturb the system
if size(vn_pool) < P_size then
    vn_pool ← (P_size − size(vn_pool)) random VN
```

Algorithm 5.2: Virtual Nodes Selection

The pseudo-code of this task is presented in Algorithm 5.2. Initially, the manager marks virtual nodes from overloaded nodes. Note that the manager compares the predicted load factors of the node at time $t + 1$ against LF_{up}. The removal order of the virtual node considers the derivative of the virtual nodes' load trend. A virtual node with an high derivative would probably have a burst in the load soon, and migrating it may avoid overloading. Hence, the virtual nodes with the highest derivative are marked for migration as first.

Afterwards, the manager marks virtual nodes for migration until $\alpha(t + 1)$ is over the α_{thresh} defined by the operator. In this phase, only virtual nodes assigned to user-provided resources are considered. Moreover, the manager adds the virtual nodes currently managed by the back-up cloud nodes to vn_{pool} (see Section 5.1.5).

If, after these steps, the number of virtual nodes in vn_{pool} is less than P_{size}, additional virtual nodes are taken from cloud nodes with the lowest predicted load factor. This would lead to a removal of the unused cloud resources over time. If the size of the vn_{pool} is still lower than P_{size}, additional random virtual nodes are marked, to guarantee a constant level of perturbation to the system, useful to avoid being stuck in local optimal solutions.

```
input : vn_pool, the list of virtual server to migrate
input : LF_up, upper load factor threshold
output: Actions, the list of migrations to execute

foreach v ∈ vn_pool do
    Chosen = Null
    node_pool ← node_pool ∪ (n ∈ N(t) : (n ⊕ v)_LF(t) < LF_up)
    if node_pool is ∅ then
        Chosen ← recruitNewCloud()
    else
        node_pool ← (n ∈ node_pool : α(t + 1) given (n ⊕ v) > α_thres)
        if node_pool = ∅ then
            Chosen ← recruitNewCloud()
        else
            Sort node_pool ascending according the cost
            Chosen ← node_pool.getFirst()
    Actions ← migrate(v, Chosen)

executeActions()
releaseUnusedCloud()
```

Algorithm 5.3: Destination Selection

Destination Selection The aim of this task is to assign the virtual nodes from vn_{pool} to nodes. The idea is to find, for each virtual node in vn_{pool}, a set of candidate nodes ($node_{pool}$), and then assign the virtual node to the node candidate that minimizes the cost. The pseudo-code of this task is presented in Algorithm 5.3. Note that in the code we use the notation $n \oplus v$ to indicate a system where the node n would manage the virtual node v.

For each virtual node v in vn_{pool}, the manager first selects the node candidates such that, if assigned v, their predicted load factor would be less than LF_{up}. If no node satisfies this requirement, a new cloud node is recruited and v is assigned to it. Otherwise, the manager removes from $node_{pool}$ all the nodes

that would decrease the QoS below α_{thres}. If no candidates remain in $node_{pool}$ after this further selection, a new cloud node is recruited. Otherwise, among the candidates left in $node_{pool}$, the manager selects the one minimizing the cost. Whenever all the virtual nodes are assigned, the manager performs all the migrations (see next section) and releases unused cloud nodes.

5.3.3 Migration

At the start of the epoch, the manager executes the migrations that comes as output from the assignment computation of the previous epoch.

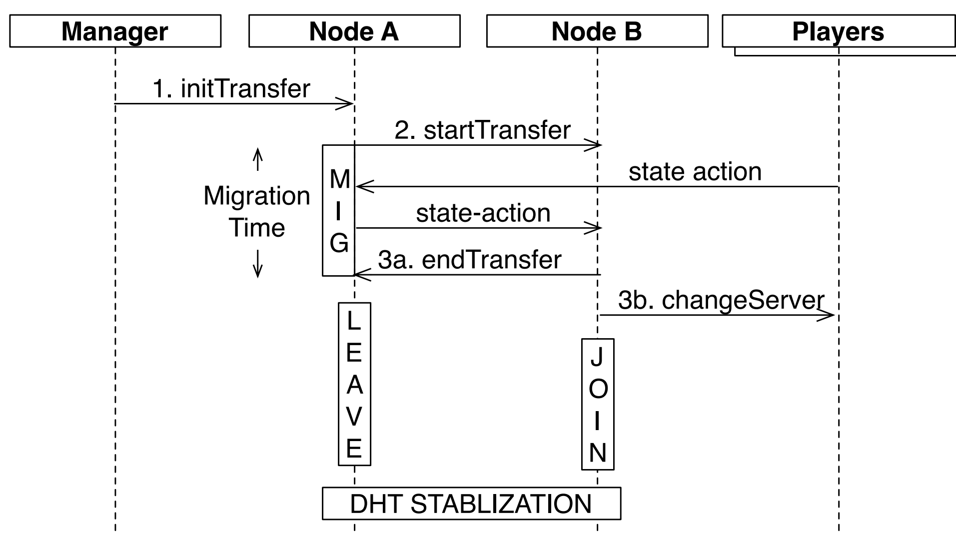


Figure 5.4: Migration of a VN from the node A to node B

The migration procedure has been originally presented in our prior work [14]. We briefly explain it here with an example. Suppose that a virtual node V migrates from a source node A to a destination node B. The actions involved (presented in the sequential diagram of Figure 5.4) are the following:

1. The manager send a reference to V and the address of recipient node B to node A.

2. A sends V to B, together with the list of users connected to V. In the transient time that is needed to complete the transfer, players still send entity update messages to A, which in turn forwards them to B. Note that in this transient period, entities may go out-of-sync and, as a consequence, players may perceive some visual inconsistencies.

3. Once received the message, node B notifies the clients that it has became the manager of V. From this point on, clients are able to modify the state of the entities included in V. However, the routing tables of the DHT have to be updated to assure correct routing resolutions.

4. To this end, V executes a *join* operation having B as target in order to update its references in the DHT. This operation updates the routing table of the node that are in the path from V to B, still leaving dangling references to A as the manager of V. To make consistent all references, the stabilization process of the DHT is executed.

5. Finally, a *leave* operation is executed by V on A in order to complete the process.

Since during the transfer the objects inside the VN are not accessible to the clients, this phase might affect users experience. To avoid this problem, it is important to limit the migration time of a virtual node, which largely depends on its size. A detailed tuning of the virtual nodes dimension is described in Section 5.4.2.

5.4 Experimental Results

The first part of this section presents the characteristics of the workloads used in our simulations. Then, we propose an empirical analysis to tune the dimension of the virtual servers and the maximum capacity threshold. Finally, we discuss the simulation results that consider cost and QoS varying different parameters, such as the number of players, the QoS threshold, and the churn level.

5.4.1 Workload Definition

A realistic simulation of the bandwidth load is central to properly evaluate a MMOG infrastructure[3]. The bandwidth load is sampled according to a discrete time step model. We define each step t to have a duration equal to Δt. For each step we compute the outgoing bandwidth requirement for the broadcasting

[3]We consider the load related to the management of the users. We do not take into account the bandwidth consumed for other tasks, like backup management, intra-server communications, and other services at application level (e.g. voice over IP).

of the entities of the players. The pricing model for bandwidth and renting is the one of Amazon EC2. In the following, we briefly describe the aspects considered when building the synthetic workload for our experimental setup; a more complete analysis is provided in [15])

Mobility models for players. Avatars move according to realistic mobility traces that have been computed exploiting the mobility model presented by Legtchenko et al. [54], which simulates avatars movement in a commercial MMOG, Second Life [1]. We have presented this implementation, as well as a comparison with other mobility models, in [11]. In the model, players gather around a set of *hotspots*, which usually corresponds to towns, or in general to points of interest of the virtual world. A circular area characterized by a center and a radius defines each hotspot. Movements are driven by a finite-state automaton, whose transition probabilities are taken from the original paper [54].

The objects distribution. To place objects over the virtual environment, we use the same space characterization of hotspot areas used by the mobility model. A fraction of the objects is placed inside hotspot areas, so that their concentration follows a Zipfian distribution [69], with a peak in the hotspot center. The rest of the objects is randomly placed outside of hotspots. Figure 5.5 shows a snapshot of the placement of avatars and objects in the virtual environment.

The variation of the players number over time. Evaluating how the infrastructure adapts itself to variations in the number of players is an important task. In particular, since the load is in direct correspondence with the number of players, we are interested on how an increasing (and decreasing) load is managed by the infrastructure. We used two variation patterns of the players number. The first simulates the arrival and the leaving of a player according to a seasonal pattern, like the one described in Figure 5.6. In this pattern, the minimum value is set equal to the 10% of the maximum. The second pattern considers a stable number of users, i.e. their number does not change during the simulation.

We generated two different workload from these patterns. In the rest of this section, we refer to the workload with the seasonal pattern as **W1**, and to the pattern with no variation as **W2**.

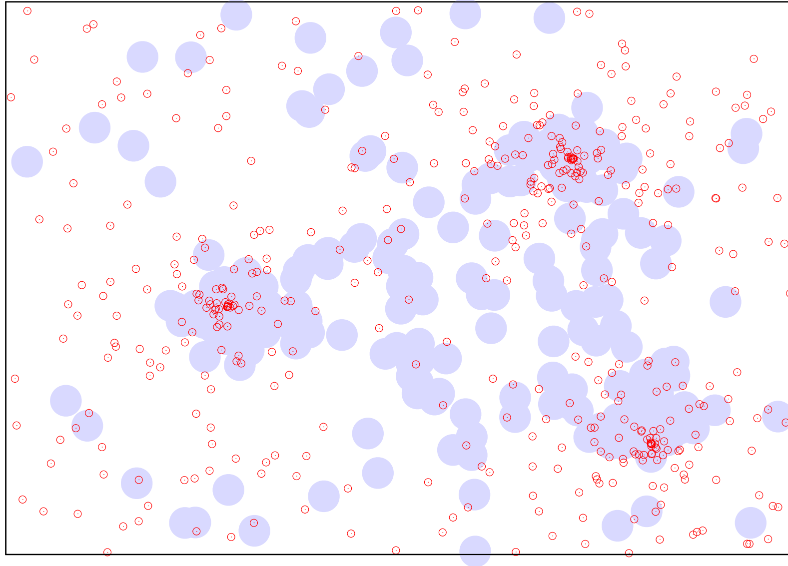


Figure 5.5: Objects and avatars placement in the virtual environment

5.4.2 Tuning the Virtual Nodes Dimension

As we stated in Section 5.2, the *migration time* (MT) may affect the interactivity of the virtual environment. The more time a migration takes, the more the users perceive the virtual environment as "frozen".

The MT of a virtual node mostly depends on its size; indeed, it is important to tune this size to minimize this problem. The size of the virtual node depends on several factors: (i) the size of the routing table, (ii) the number of the clients accessing the virtual node, and (iii) the number of entities handled. In the following we enter in details of these three aspects.

The size of the routing table. The routing table of the virtual node contains the references to other DHT nodes, and depends on the particular DHT implementation chosen. In general, the routing table size is logarithmic with respect to the number of nodes participating in the DHT [91]. In our implementation we use Chord DHT [93] where each entry of the table is composed by a DHT-ID (160 bits) and a IP (32 bits). By considering a large DHT with 10K virtual nodes, the routing table contains 14 entries. Hence, the size of the routing table is 336 bytes.

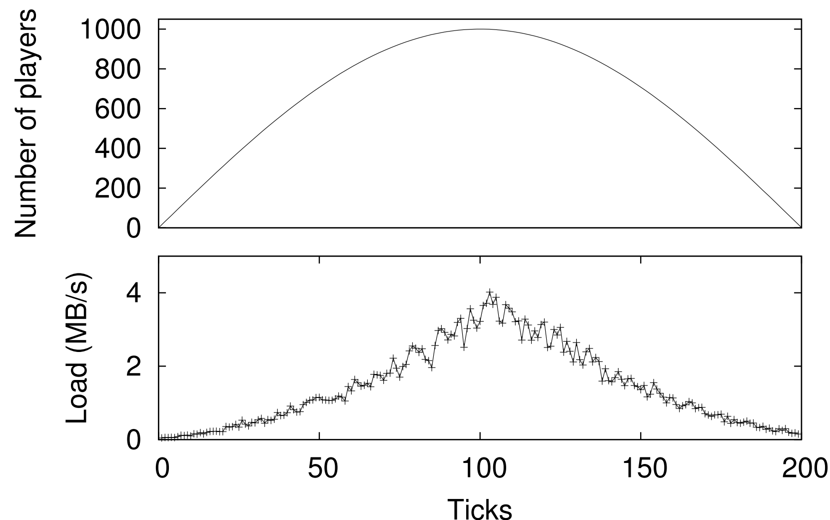


Figure 5.6: Number of players over time (up) and correspondent load variation (down)

The number of the clients accessing the virtual node. Any virtual node maintains a list of accessing clients. In order to estimate the number of connected clients per virtual node entity, we conducted an empirical analysis. We counted the clients per entity per minute (hence, we consider a quite large timespan) in a simulation with synthetic generated avatars movements. The movements and the placement of the objects in the virtual environment were generated as described in Section 5.4.1. Figure 5.7 shows the histogram of the clients (in percentage) plotted in a log-log scale. The trend of the plot resembles a power law, i.e. a function of the form $y(x) = Kx^{-\alpha}$. By fitting the data, we derived $K = 0.5$ and $\alpha = 1.4$ (the corresponding function is also plotted in the figure). We exploited a number generator based on this function to generate the number of accessing clients list for a virtual node, which is then multiplied for the size of the entry. Each entry of this list contains a UID (32 bits), a IP (32 bits) and a port (32 bits).

The number of entities handled. The content of an entity is composed by (i) a UID (32 bits), (ii) a DHT-ID (160 bits), (iii) a point representing the two-dimensional position in the virtual environment of the entity ($32 + 32$ bits), and (iv) a list of attributes, where to each entry name (32 bits) corresponds a respective values (64 bits). Let us assume that the dimension of the attribute list

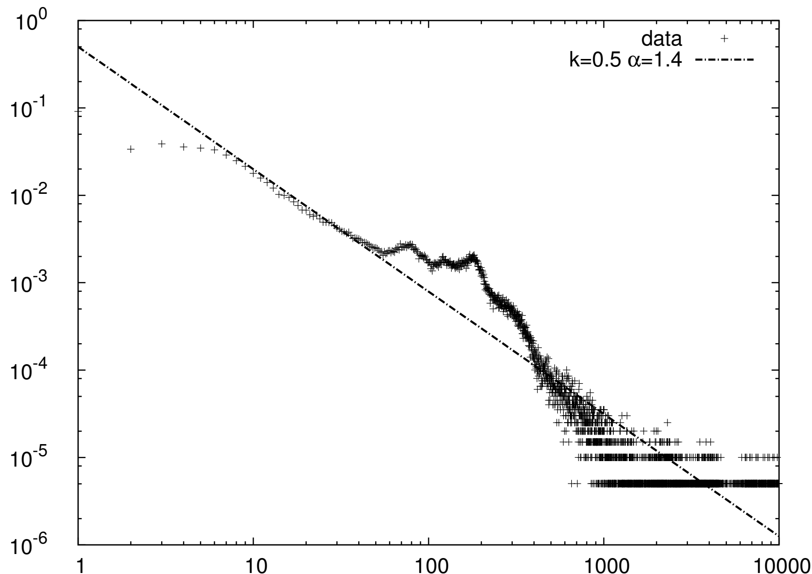


Figure 5.7: Average clients per entity per minute plotted in log-log

is arbitrarily fixed for all entities to 10 elements. We argue that this value is a good average estimate to contain enough information for a general MMOG. Summing up, each entity descriptor has a size of about 140 bytes.

We aim to determine the maximum size of a virtual node such that in the 95% of the cases the MT takes less than the interactivity delay users may tolerate in response of their actions. This delay spans from few hundreds of milliseconds in fast-paced MMOGs up to two second in slow-paced MMOGs [20]. Here we stay in the middle, and consider the interactivity delay under 1 second as tolerable, which fits medium-paced game genre.

In order to find the maximum size of a virtual node we needed a MT model. The size of the virtual nodes were generated by considering the aspects (size of routing table, accessing clients and entities handled) detailed above. To model MT we have exploited (with some minor modification) the model for TCP latency proposed by Cardwell et al. [10]. This TCP latency model requires Round Trip Times (RTTs) as input parameter. We model RTT delays according to the traces of the king dataset [33]. The probability density function (PDF) of the RTTs is shown in Figure 5.8.

Based on the MT model we conducted experiments to study the influence of a size of virtual node on MT. Figure 5.9 shows that virtual nodes managing less

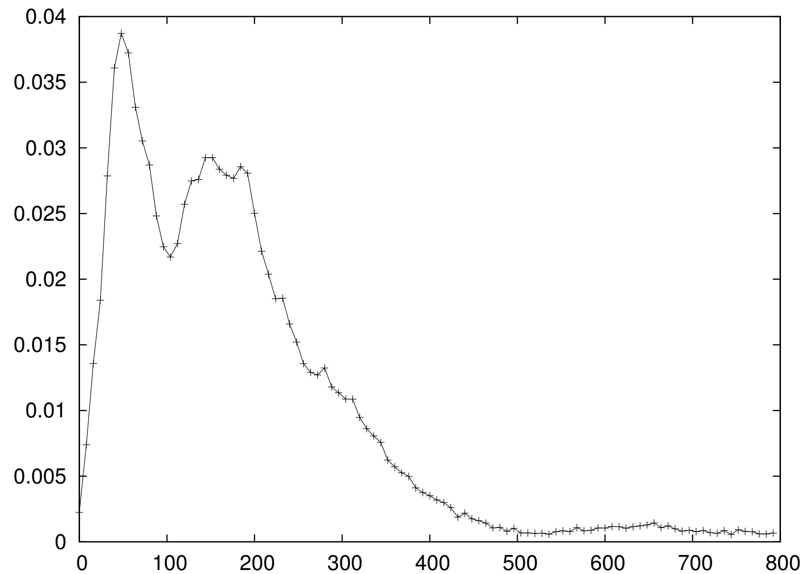


Figure 5.8: Probability density function of RTTs

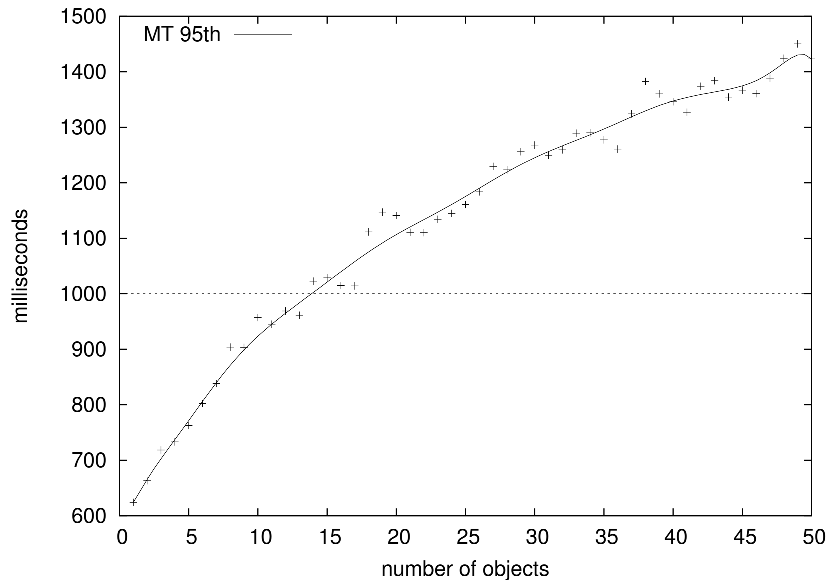


Figure 5.9: 95th percentile of MT with different amount of objects

than 15 entities have an MT less than one second with the 95th percentile. In the following, we use 15 as a maximum number of entities per virtual node.

More generally, this result may be used in two ways. Given a virtual environment with a predictable number of entities, it is possible to define the minimum number of virtual nodes to employ. On the other side, if it is a necessity to have

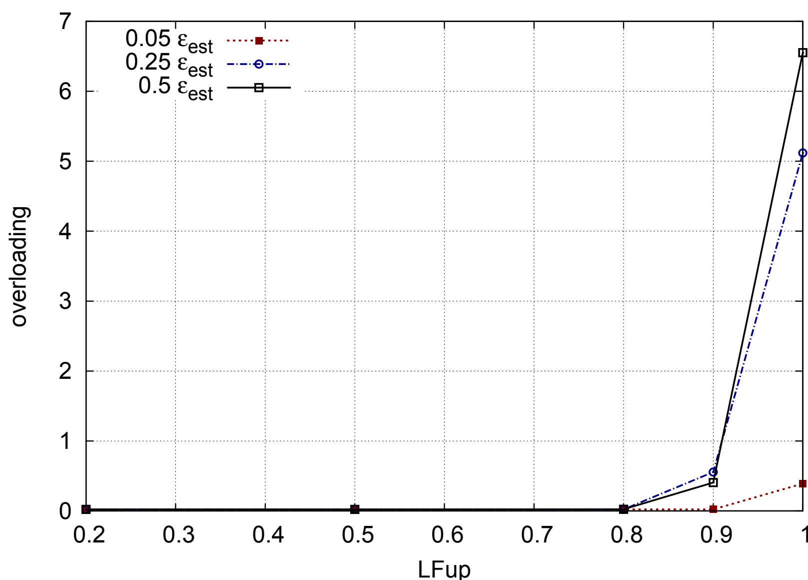


Figure 5.10: Percentage of overloading over LF_{up} for different eps

a specific number of virtual nodes, it is possible to know the maximum number of entities the system can support.

5.4.3 Tuning the Capacity Threshold

In this section we discuss the evaluation of two parameters: (i) the maximum node capacity LF_{up} (see Section 5.2) and (ii) the error of the load prediction ξ_{est} (see Section 5.3). A proper tuning of these parameters is essential to avoid nodes overloading as well as resource under- and over-provisioning.

Figure 5.10 shows the percentage of overloading load for different LF_{up} values. As we can see from this figure, for a low LF_{up} the selection of epsilon is irrelevant. In particular, until $LF_{up} = 0.8$ the percentage of overloading remains around 0%. Nevertheless, the $\xi_{est} = 0.05$ allows to tune $LF_{up} = 0.9$ without nodes overloading and shows better results in terms of resource utilization. By comparison, with $\xi_{est} = 0.25$ or higher, traces of overloading starts at $LF_{up} = 0.8$. However, if we use all the capacity of the nodes ($LF_{up} = 1$), the percentage of overloading grows up to 7%. This can be explained with the tendency of our prediction mechanism to under-predict the load of the virtual nodes, causing the utilization of less resources then needed.

As we can see in Figure 5.11, the system cost decreases as LF_{up} increases.

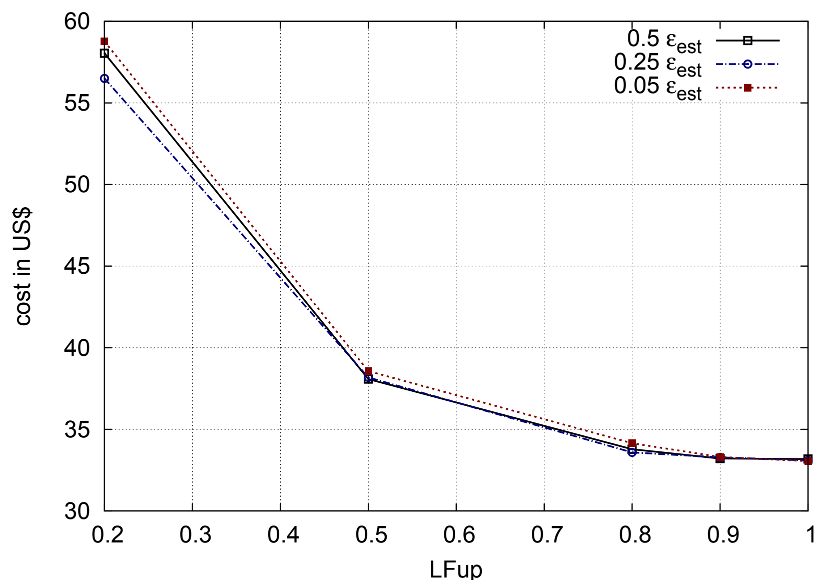


Figure 5.11: Total simulation cost over LF_{up} with different eps

Nevertheless, for a high LF_{up} ($LF_{up} = 0.9$) the selection of the ξ_{est} is not relevant. The fact that with $\xi_{est} = 0.05$ we yield better results in terms of resource utilization, candidates $LF_{up} = 0.9$ and $\xi_{est} = 0.05$ as good values for the parameters. Note that these results hold for the chosen prediction mechanism, and should be recomputed if other prediction techniques are used.

5.4.4 Cost over the number of players

In this section, we analyse the cost per minute over time and the total cost of a simulation with the seasonal (**W1**) and fixed (**W2**) access patterns, and with different number of players (from 1 up to 10 thousand). The QoS threshold, α_{thres}, is set to 0.95, $LF_{up} = 0.9$ and $\xi_{est} = 0.05$.

Figure 5.12 and 5.13 show the cost per minute over time with **W1** and **W2** respectively. From the figures is evident that the cost largely depends on the number of players. The cost with **W1** has evident peaks and falls, and the cost per minute grows accordingly with the number of players. In this sense, the ability to exploit user-provided resources in situation of low load yields a clear advantage. On the contrary, the cost with **W2** is more stable, as the load is imposed only by the movements of the players in the virtual world and not by their access patterns. However, it is interesting to notice that the cost growing

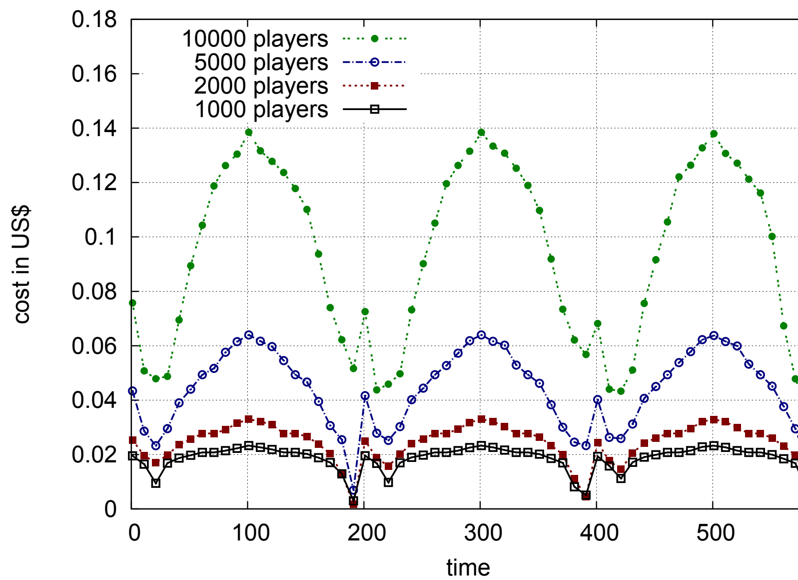


Figure 5.12: Cost per minute over time considering workload 1

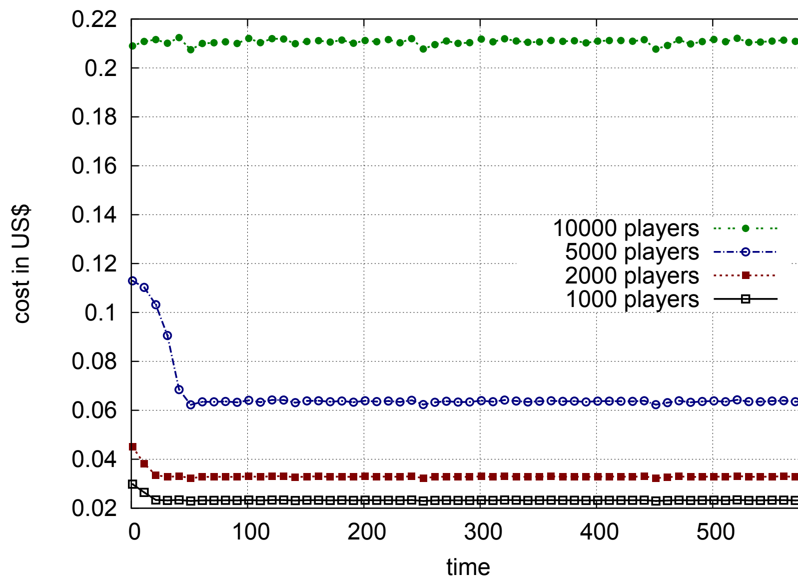


Figure 5.13: Cost per minute over time considering workload 2

is not linear in terms of number of players and the cost for 10k users is almost 4 times higher than for 5k. This occurs since with a dramatic growing of load, user-provided resources became less able to serve virtual nodes.

Figure 5.14 and 5.15 depict the cost of a ten hours simulation for different number of players for **W1** and **W2** respectively. As it was expected, the cost with

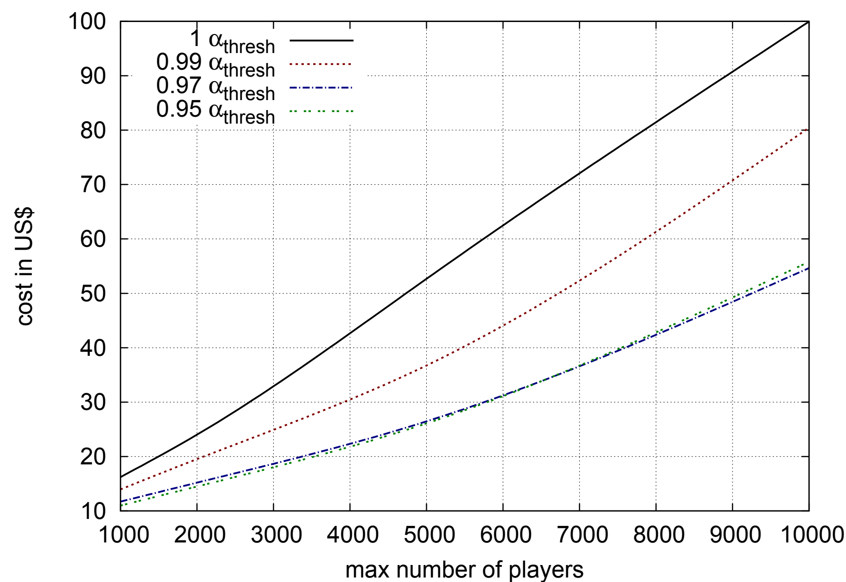


Figure 5.14: Total cost with workload 1

W2 is higher due to the larger number of concurrent players participating to the virtual world. Moreover, as we can see from the figures, a lower QoS threshold allows to significantly reduce the total system cost. However, if for **W1** the difference of total cost between $\alpha_{thres} = 1$ and $\alpha_{thres} = 0.99$ is 20%, it drops to 1% with **W2**. Furthermore, the difference from $av = 1$ and $\alpha_{thres} = 0.95$ is around 60% in **W1**, whereas it is still 1% in **W2**. This suggests that with a fixed, maximum number of players, some virtual nodes are too heavily loaded to be managed by user-provided resources, even if those resources are available. This results in a larger utilization of cloud resources, which in turn increases the cost. Nevertheless, the system with **W2** shows good results in case of medium size load (around 5k users) and allows to reduce up to 60% of the cost between $av = 1$ and $\alpha_{thres} = 0.95$.

Figures 5.14 and 5.15 also introduce another interesting observation. The cost of the two workloads when α_{thres} is 0.95 and 0.97 are basically the same. This suggests that a further reduction in the α_{thres} would have no impact on the cost of the platforms. This occurs because no virtual nodes can assigned to user-provided resources, due to their limited bandwidth capabilities. The correspondence of results with α_{thres} equal to 0.95 and 0.97 is also present in the next section.

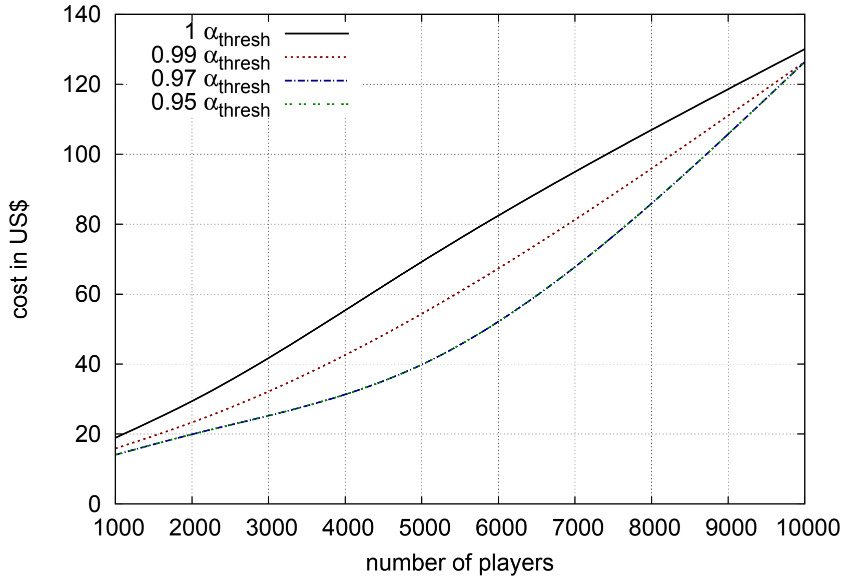


Figure 5.15: Total cost with workload 2

5.4.5 Qos and Cost Trade-off

One of the strength of our approach is to provide MMOG operators the ability to set the desired level of QoS of the platform. The operator chooses the *QoS threshold*, α_{thres}, which affects the assignment of virtual node to node. In general, the higher the threshold, the less virtual nodes are assigned to user-provided resources. As a consequence, this threshold indirectly controls the operational cost of the platform.

To evaluate the trade-off between cost and QoS we considered the workload **W1**, $\xi_{est} = 0.05$, $LF_{up} = 0.9$ and 5k players. Figure 5.16 shows how the architecture maintains the QoS in the system above the specified QoS level. At first, as we can see from the figure, the architecture successfully maintains a level of QoS above the threshold. However, the system QoS never reaches the threshold of 0.95. This is because all the virtual nodes that can be assigned to user-provided resources have been already assigned. One more point is when the level of the QoS raises up, regardless of the threshold. These increments occur when cloud resources are exploited to supply the low number of user-provided resources. For instance, around the 200th and 400th ticks according to the seasonal pattern there are no available user-provided resources to support the system (see Figure 5.6).

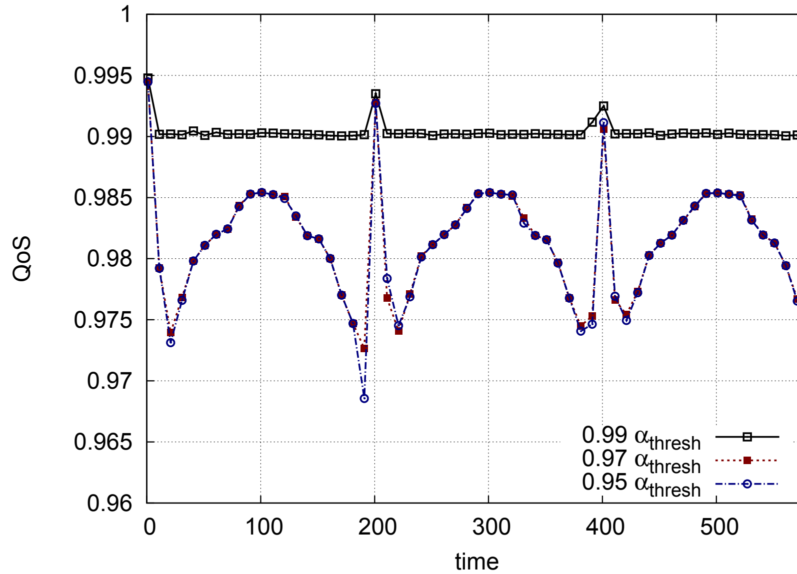


Figure 5.16: QoS over time with different QoS thresholds

Figure 5.17 shows the cost per minute for different QoS thresholds. In the situations of peak load and lack of user-provided resources the utilization of cloud increases, in fact increasing the cost as well. This suggests that on-demand resources are essential in order to support the MMOG.

5.4.6 Behaviour over different churn levels

In our work we consider two types of churn: from the seasonal trend of the accesses (Figure 5.6) and from players that interchange during a cycle. Although both kinds of churn affect performances, in our evaluation we varied only the percentage of players interchange during a cycle, the churn level in the following.

Figure 5.18 shows the QoS over time with different levels of churn. In the experiments we used $\alpha_{thresh} = 0.95$. The figure suggests that the level of churn directly affects the minimum level of QoS reached. In case of high churn level, the amount of user-provided resources is reduced to maintain an acceptable level of QoS. For instance, with a churn level of 0.05 the QoS stays around 0.99, whereas with a churn level of 0.5 the QoS drops to the threshold parameter 0.95. This is confirmed in Figure 5.19, which measures the cost at the same conditions. As we can see with a churn level of 0.5 the cost is significantly

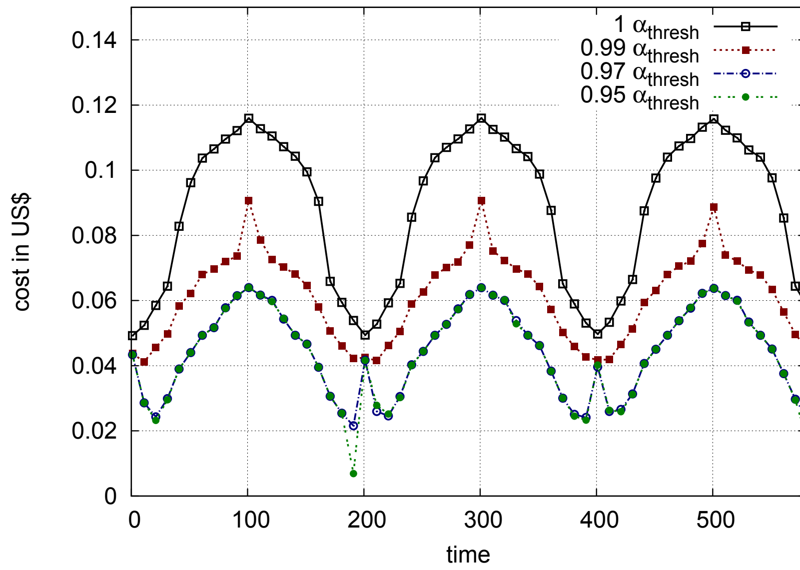


Figure 5.17: Cost per minute with different QoS thresholds

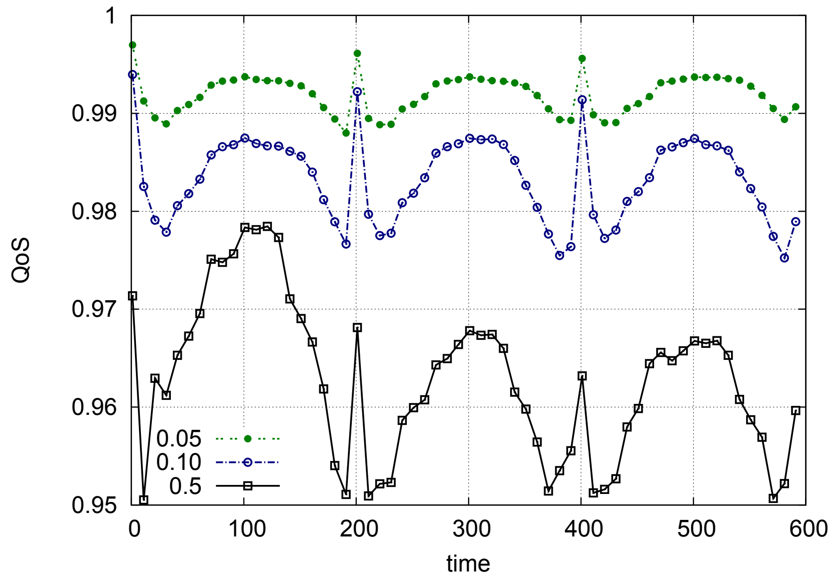


Figure 5.18: QoS over time with different churn levels

higher than in the other cases.

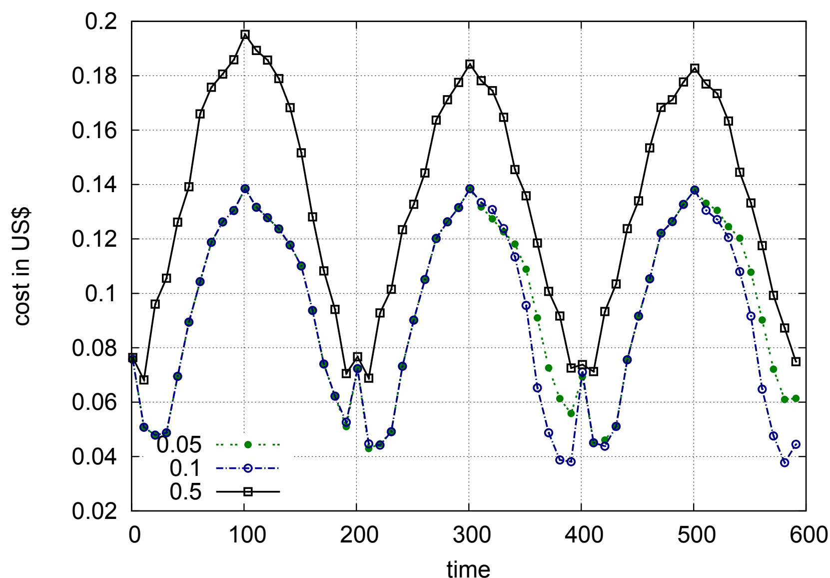


Figure 5.19: Cost over time with different churn levels

5.5 Related work

Our work can be compared with two orthogonal areas of research, namely hybrid architectures for MMOGs and integration of P2P and cloud computing. In the rest of this section, we collect and compare approaches on these fields that, to the best of our knowledge, are more relevant with respect to our proposal.

5.5.1 Hybrid MMOG architectures

Last decade has seen the rising of several P2P-based architectures targeting large-scale on-line games. Among them, the hybrid MMOG architectures focused on the combination of centralized resources with P2P paradigms to support MMOGs. Unlike our contribution, these works do not explicitly consider cloud computing as centralized resources and hence they lack an economic model. To the best of our knowledge, this work is the first proposing the integration of P2P and cloud computing as a solution for MMOGs.

Hybrid MMOG architectures aim to exploit and combine user-provided resources (peers) and centralized resources (servers) in a seamless infrastructure.

A central issue for hybrid MMOGs is to define a partitioning of the virtual environment, so that it is possible to assign specific tasks to peers. A widely-

used method is *spatial partitioning*, i.e. the virtual environment is divided into *regions* or *cells*, whose dimension can be either fixed or variable. These regions are in turn assigned to a peer or a server, that becomes the *manager* of the entities in such regions.

The work proposed by Kim et al. [51] belongs to the first category. The authors consider square regions, initially managed by a central server. The first peer with enough computational and bandwidth capabilities to enter a cell becomes the cell manager. Afterwards, a fixed number of peers entering the same cell act as backup managers in order to increase failure robustness. Similarly, Barri et al. [6] proposes an hybrid system including a central server and a pool of peers. The central server runs the MMOG and, as soon as it reaches the maximum of its capacity, it delegates part of the load to the peers.

Other approaches employ a *functional* partitioning of the MMOG, where only subset of functions are delegated to peers. For instance, Chen and Muntz [18] proposes a functional partition of the MMOGs tasks. Central servers are responsible for user authentication and game persistence while manage only regions characterized by high-density user interactions, whereas peers support low-density interaction regions.

Jardine and Zappala [38] provide a distinction between *positional* and *state-changing* actions. They propose an hybrid architecture where peers manage positional actions, that are more frequent and prone to be maintained locally. Central servers handle state-changing actions, that are not transitory and require a larger amount computational power.

By comparison, our general architecture exploits a similar characterization of actions, while the functional partitioning is done at the level of components rather than at resource type. In other words, we employ two different components that respectively manage positional and state-changing actions.

With respect to the state of the art, we stay somewhere in the middle. On one hand, we consider some functionalities, like authentication, to be handled by centralized and full controllable servers. On the other hand, other functionalities may be mapped to both central servers and peers. This requires a more flexible, dynamic strategy for region distribution, to allow for a fine-grained management of the resources by the MMOG operator. Resources control is very important for our approach, because the seamless combination of cloud computing and P2P requires to keep under control the cost and effectively deal with the implicit

uncertainty related to peers.

5.5.2 Hybrid P2P and cloud architectures

Combined cloud and P2P infrastructures recently gained attention from the research community. These infrastructures aim to resolve the issues typical of pure P2P solutions.

A common problem is the asymmetry in the bandwidth capability of user-provided resources, where the download bandwidth is usually much higher than upload. In content distribution and information dissemination, a widely used approach to enable the upload of the content from user-provided resources is to use the so called *helpers*, i.e. additional cloud nodes supporting the delivery of content [61, 95, 96, 63].

A MMOG infrastructure can be seen as a content delivery system, where the content is represented the state of the virtual environment. In this sense, the addition of cloud to support user-provided servers in the management of the MMOG, is comparable to adding helpers. However, unlike classical content delivering, the state of the MMOG is modified by the players themselves, so servers shall manage not only the diffusion but also the consistency and correctness of the information.

An essential property in hybrid P2P-cloud architectures is *self-configuration*: components and protocols are autonomically configured according to specific target goals (such as reliability and availability). In dynamic contexts, self-configuration is often supported by forecasting the resource utilization [104] and orchestrating the leasing and releasing of pay-per-use resources. In one of our work [47] we exploited a resource prediction mechanism to self-tune the amount of replicas on top of an hybrid P2P and cloud system. We leveraged this experience to proactively assign the objects of the MMOG to the nodes of the infrastructure.

Chapter 6

NAT-traversal systems

The recent rise of P2P systems as building blocks for many distributed applications posed issues on their real applicability. Last decade most of the work in the field concentrated on improving P2P overlays characteristics, i.e., economical effectiveness, reliability and performance. Most of the proposed P2P solutions are based on the assumption that all the on-line peers can communicate with each other via Internet. Nevertheless, over last decades the Internet IP architecture has undergone steady changes. One of the most important changes is the spreading of NAT approaches, which have progressively led to the loss of end-to-end addressability.

The peer-sampling service is a fundamental mechanism for gossip-based communication protocols (Section 2.1). In this chapter we developed a specific protocol to run the peer sampling service in NAT systems. In details, we propose a robust NAT traversal solution that is assisted by cloud entities. As a matter of fact, our solution demonstrates how the combination of cloud and P2P technologies can solve the problem of P2P services application in the real Internet.

6.1 Problem statement

We consider a network that contains two type of nodes: private and public. Private nodes, i.e., *child* nodes, are the nodes that are behind NAT and cannot be accessed directly. Public nodes, i.e., *parent* nodes, are opened for an end-to-end communication. To allow child nodes to participate in P2P communication, each child node keeps a list of its parent nodes.

In our work we aim to implement the so called NAT-traversal hole punching technique [26]. Hence to contact a child node, the communication is done through the available *rendezvous server*, which role is played by one of the parent nodes belonging to the child. We do not consider here the adaptation of the hole pouncing technique to different types of NAT systems and communication protocols [79]. Rather, we concentrate on the problem of rendezvous servers overloading and reliability.

The number of the parent nodes assigned to each child clearly affects the availability of child nodes. For this reason, we introduce a measure of quality of service, which we define as the percentage of nodes with an *Availability Factor* (AF) larger than a predefined value. AF is the probability for a node in the system to be accessible for communication. More precisely, an on-line parent node is by definition always accessible (i.e. it has a public IP), therefore its AF is equal or really close to 1. At the same time, the AF of a child node depends on the failure probability of its parents, and ranges in the interval $[0..1]$. For example, if a child node has no alive parents, its AF is equal to 0.

Therefore, maintaining the AF as higher as possible for child nodes requires a large number of parent nodes. Unfortunately, this is not generally possible as it may happen that the total number of parent nodes is not enough to satisfy all the child nodes. In order to overcome this problem and increase the AF for child nodes we propose to use *cloud parents nodes* as always available and reliable parent nodes.

In case a child node has not enough reliable parents, it could use a cloud parent to be accessible from behind the NAT. However, renting a cloud node to perform as a parent comes with a cost. Intensive cloud utilization can hinder in the economical sustainability of the approach. Therefore it is important to strike a balance between the amount of cloud resource used and the overall QoS of the system.

More generally, in this chapter we address the issue of autonomously regulate the utilization of resources between the cloud and a set of available P2P public nodes, while maximizing the QoS. To this end, we propose a self-regulation mechanism that focuses on parent nodes management in cloud-assisted NAT-traversal. Our target is to meet a desired level of QoS while keeping the minimum possible economical cost.

We consider the AF as the critical parameter for parent selection mechanism.

As a consequence, the problem of a self-regulation mechanism can be divided into two parts: (i) modelling the AF, and (ii) self-regulation and cost minimization. In details:

- Each child node has to determine the number of parents $N_{parents}$ to keep. $N_{parents}$ is a function of the *Load Factor* (LF) (% of upload bandwidth channel of a parent node required to support child-parent connection) and its current and predicted AF.
- Determine the constraints for P2P parent selection considering parent stability. In particular, in case the parents are from the same subnetwork, their failure probability is not independent.
- The system has to self-regulate the amount of used cloud resources according to the current and predicted AF and cost.

6.2 Nat-traversal system model

Each node in the system is represented by a descriptor $\{N_{id}, Nat_{type}, \{P_i\}\}$, where N_{id} is the unique node identifier, Nat_{type} indicates a public/private type of the node and $\{P_i\}$ lists the parent nodes assigned.

The role of the parent can be played by P2P or cloud entities. We consider a model that allows the system to switch autonomously between P2P and cloud parent nodes. As an example, consider a PSS protocol where each child node has only a cloud node as parent. Child node's descriptor would look like $\{N_{id}, Nat_{type}, \{C\}\}$, where C is the cloud parent node descriptor. As the number of available P2P public nodes increases, it would be possible, based on the current AF level, to include additional P2P parent nodes into a child descriptor (i.e. $\{N_{id}, Nat_{type}, \{P_1, P_2, \ldots, P_i, C\}\}$) and even totally rely on P2P parent nodes (i.e. $\{N_{id}, Nat_{type}, \{P_1, P_2, \ldots, P_i\}\}$). Instead, when the number of public nodes is below some *Critical* threshold, to guarantee a reliable PSS and to avoid the nodes overloading, cloud parent nodes may be inserted back into the child node's descriptor (i.e. $\{N_{id}, Nat_{type}, \{P_1, P_2, \ldots, P_i, C\}\}$).

The critical threshold for the number of parent nodes depends on the probability of the parents to fail (we describe it in more details in Section 6.2.1). On one side, the higher the number of parent nodes, the lower the risk to be

offline at the same time for all of them. On the other side, a child periodically sends a keep alive message to its parents to keep the connection open. Hence, a high number of parents increases the network overhead, due to the traffic of keep-alive messages.

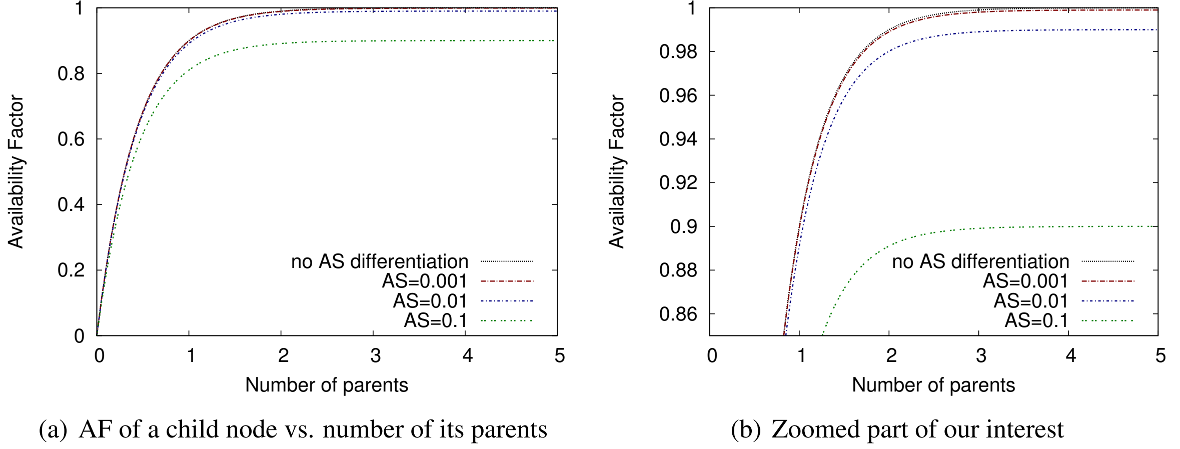
To avoid network overhead we introduce the *Redundant* threshold for the number of parent nodes. The redundant threshold limits the maximum number of parents that is reasonable to keep for a child node. The detailed analysis of the redundant threshold is presented in Section 6.2.2.

6.2.1 Modeling AF

In this section we present the model for the AF, which allows each child node to compute, according to the desired Qos, the number of parents to keep. To compute the AF for a child node we have to compute the probability that all its parents are off-line at the same period of time τ.

In our work we consider two issues that can lead to a parent failure: (i) Sub-network failure and (ii) network churn rate. We assume that each of the nodes belongs to some sub-network, for example to some Autonomous System (AS). We consider AS to be a connected group of some IP prefixes that is run by one or more network operators with single routing policy. As it is shown in the work of Sriram et al. [90], ASs can be subject to malicious attacks that can lead to AS isolations. In this case, isolated AS cannot support child nodes. Moreover, we also consider the actual churn rate of the network, that shows the percentage of nodes that leaves the network over a specific time interval.

More formally, each parent belongs to some AS_i that has a failure rate α_i. Moreover, each of the parents P_j has a failure rate π_j due to the network churn rate. As an example, we consider a descriptor of the child node with three parents P_1, P_2, P_3, in which $P_1, P_2 \in AS_1$ and $P_3 \in AS_2$. To compute the AF of such node, we have to define the probability that at least one of these parents is available during the next τ time interval. The probability that P_1 and P_2 are not available at the same time is defined with the multiplication of their nodes failure rates $\pi_1 \times \pi_2$. Hence the probability that at least one of them is available is $1 - \pi_1\pi_2$. At the same time, if AS_i is not available, all the nodes belonging to it are not available as well. Hence, at least one of the parents P_1 or P_2 is available only if the AS_1 is available at the same time: $(1 - \alpha_1)(1 - \pi_1\pi_2)$.



(a) AF of a child node vs. number of its parents

(b) Zoomed part of our interest

Figure 6.1: AF of a child node vs. number of its parents for different system configurations and AS failure rates.

In all other cases, either both parents are not available or the whole AS_1 is not available: $1-(1-\alpha_1)(1-\pi_1\pi_2)=\pi_1\pi_2+\alpha_1(1-\pi_1\pi_2)$. Following the same logic for AS_2, the probability that parents belonging to AS_2 are not available is $\pi_3+\alpha_2(1-\pi_3)$. Hence the probability that during the τ time no parents are available is $(\pi_1\pi_2+\alpha_1(1-\pi_1\pi_2))(\pi_3+\alpha_2(1-\pi_3))$. Thus the AF for this child node is $AF=1-(\pi_1\pi_2+\alpha_1(1-\pi_1\pi_2))(\pi_3+\alpha_2(1-\pi_3))$. To simplify the computation, we assume a network with churn rate of r. Hence, in our model $\pi_1=\pi_2=\cdots=r$. In the same way, the general formula for AF of a child node is:

$$AF = 1 - \prod_{i \in AS} [\alpha_i + r^{k_i}(1-\alpha_i)] \tag{6.1}$$

where k_i is the number of parents belongs to AS i.

Figures 6.1(a) and 6.1(b) show the impact on AF from ASs with different failure rates. To simulate a stress situation, we consider the network with high churn rate $r = 0.1$. In our experiment we consider the worst case of parents layout according to the AS systems. In other words, we consider all the parents belong to the same AS. We consider the following ASs failure rates: 0.001, 0.01, 0.1.

Moreover, Figure 6.2 shows the AF derivative changing with a number of parents. Here we consider two possible cases: (1) all the parents belong to the same AS (Equation (6.2)), and (2) no ASs are considered (Equation (6.3)).

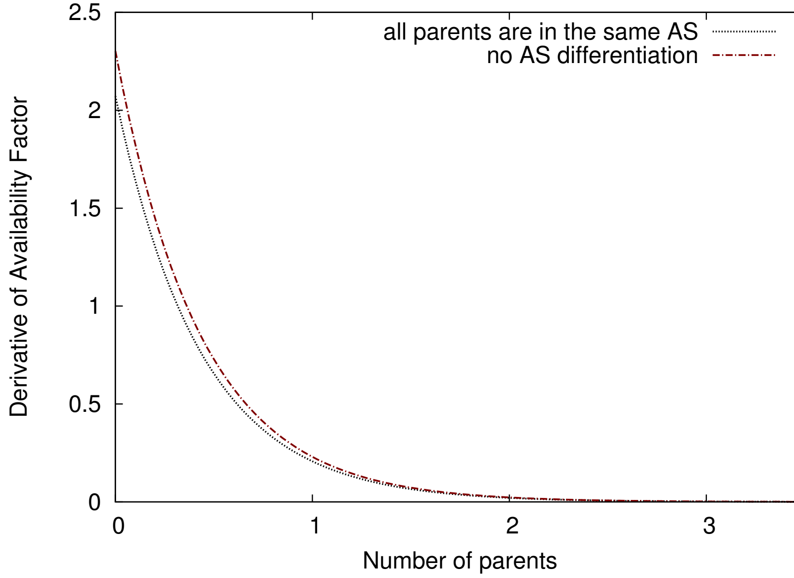


Figure 6.2: AF derivative for different number of parents in layout without AS and when all the parents belong to the same AS. $\alpha = 0.1$ and $r = 0.1$.

$$AF_1' = [1 - \alpha + r^k(1 - \alpha)]' = r^k ln(r)(\alpha - 1) \tag{6.2}$$

$$AF_2' = [1 - r^k]' = -r^k ln(r) \tag{6.3}$$

As we can see, in all the cases AF converges to its maximum when the number of parents is around 3. However, the maximum possible AF for each AS is different and limited by AS failure rate. Therefore the parents layout in terms of AS does not significantly influence on the optimum number of parents for a child node, but impacts on the maximum AF that can be reached. Arguing as above, we see that a number of parents greater than 3 yields high AF even in case of high network churn rate. Therefore, we set the critical threshold to 3.

6.2.2 The impact of LF

The support of our NAT-traversal protocol requires the communication of keep-alive messages between child and parent nodes. The higher the number of parents in a descriptor of a child, the more bandwidth is needed to support the keep-alive messages exchange.

The upload bandwidth usually is a bottleneck in P2P systems. Hence it is important to consider the consumption of parent node upload bandwidth caused by the maintenance of the NAT-traversal protocol LF_{NAT}:

$$LF_{NAT} = \frac{N_{private} m_{NAT} \overline{N_{parents}}}{N_{public} U} = \frac{m_{NAT} \overline{N_{parents}}}{\gamma U} \tag{6.4}$$

$$\gamma = \frac{N_{public}}{N_{private}} \tag{6.5}$$

$$\overline{N_{parents}} = \frac{1}{N_{private}} \sum_{p \in N_{private}} N_{parents}(p) \tag{6.6}$$

where m_{NAT} is the bandwidth rate of keep-alive messages, $\overline{N_{parents}}$ is the average number of parent nodes per child, $N_{private}$ is the number of private nodes in the network, N_{public} is the number of public nodes in the network and U is an average available upload bandwidth per parent node.

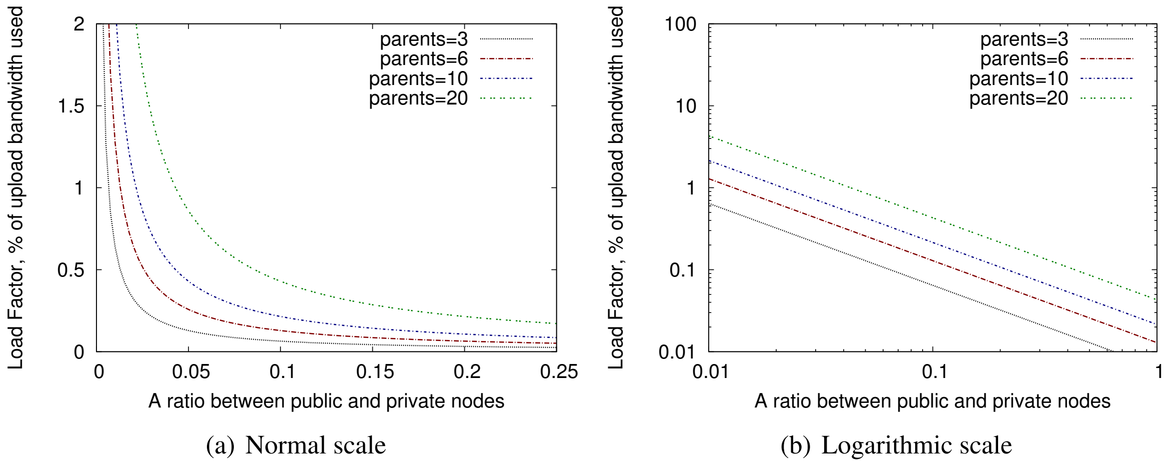
To compute the $Redundant$ threshold we consider the maximum load factor LF_{max} in the Equation 6.4:

$$Redundant = \lfloor \frac{LF_{max} \gamma U}{m_{NAT}} \rfloor \tag{6.7}$$

The Equation 6.7 shows that the $Redundant$ threshold is a dynamic parameter and has to be computed based on the current γ. Another important parameter to compute is $\gamma_{critical}$, that indicates the minimum required ratio between public and private nodes that satisfies $LF_{max} \leq 1\%$, $N_{parents} > Critical$:

$$\begin{cases} \gamma_{critical} = \dfrac{m_{NAT} \overline{N_{parents}}}{LF_{max} U} \\ N_{parents} > Critical \end{cases} \tag{6.8}$$

Case studies. In order to evaluate the model presented previously we consider a "stress test" for the system where keep-alive messages are sent every 30 seconds. In general, the interval between two consecutive keep-alive messages depends on the NAT type and can various from seconds to minutes. Therefore, to evaluate the upload bandwidth consumption caused by NAT-traversal support we chose a 30 seconds keep-alive interval as a reasonable stress condition. A



(a) Normal scale (b) Logarithmic scale

Figure 6.3: The LF impact for different ratio between public and private nodes depends on the average number of parents per child.

generic keep-alive message contains no payload and it is composed only by the header, which results in a TCP/IP message of 41 bytes.

Figure 6.3 shows a graphical representation of Equation (6.4), where $U = 512$ Kbps and $m_{NAT} = 11$ bps. The curves are shown for different $\overline{N_{parents}}$. The graph shows that the number of parent nodes used in a child descriptor does not significantly affect the LF. However, when γ is low (less than 0.5) and public nodes are few the risks of overloading increases.

Figure 6.4 shows the required minimum of γ to support an average number of parents per child $\overline{N_{parents}}$ with a maximum allowed LF_{max}.

It follows that a child with more parents in the descriptor should reduce its number of parents to avoid overtaking the maximum LF. At the same time, the number of parents cannot be less than the critical threshold (which is around 3 according to the AF model). The grey area on the Figure 6.5 corresponds to a "safe area" in which the maximum LF and the required AF are respected for $\gamma = 0.05$.

6.3 Nat-aware peer-sampling

In summary, each child node in the network executes the following tasks: (i) peer-sampling, (ii) parent-child connection support and (iii) NAT-traversal parents management.

For the peer sampling, we consider the CLOUDCAST peer-sampling ser-

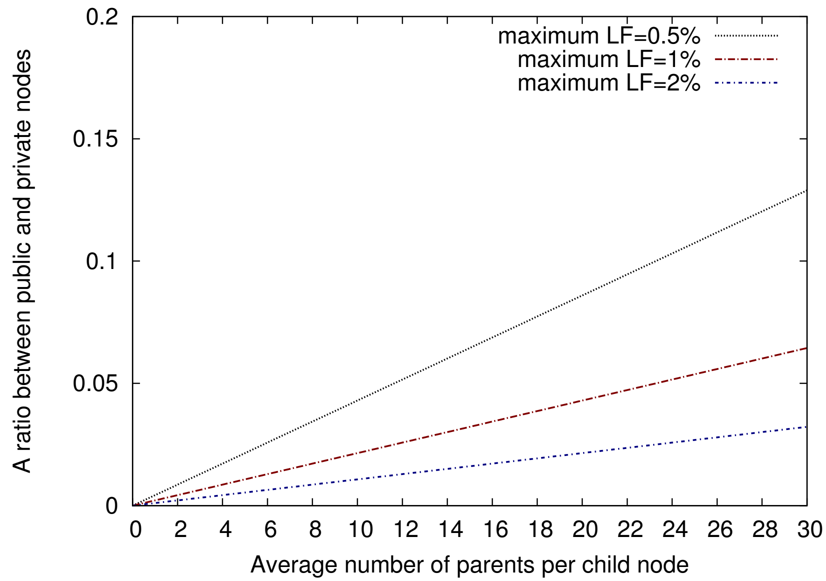


Figure 6.4: Maximum number of parent nodes in a child descriptor for maximum allowed LF

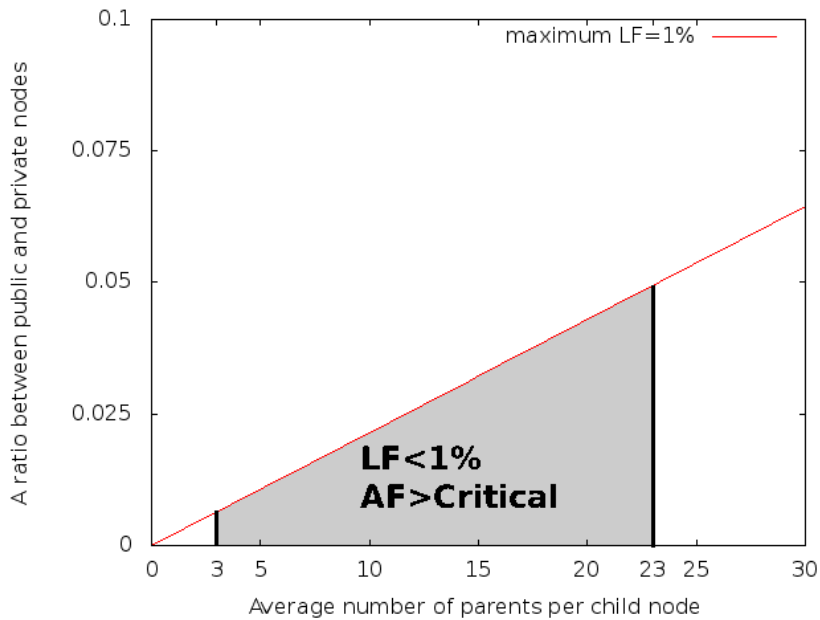


Figure 6.5: Redundant and Critical thresholds for $\gamma = 0.05$ and $LF_{max} = 1\%$

vice [63]. As we already mentioned, the communication with child nodes is done via one of its parents. Currently, we consider the model where the parent for a child contact is chosen randomly among a child parents list. The parent-child connection support is realized by simply allowing the child to periodically exchange keep-alive messages with the parents. Whenever a child node detects the failure of one of its parents it starts the NAT-traversal parents management protocol immediately.

The NAT-traversal parents management is the actual core of our work and is described in the next section.

6.3.1 Parents management

Pseudocode of the parents management is described in Algorithm 6.1. The algorithm execution is divided into time intervals ΔT, which a parameter of the system. Nevertheless, as we can see after, the minimum reasonable ΔT is equal to the time interval of the PSS shuffling.

```
repeat
    γ_i ← getCurrentRatio()
    γ_{i+1} ← getEstimation(γ_{i-1}, γ_i)
    Redundant_{i+1} ← ⌊ LF_max γ_{i+1} U / m_NAT ⌋
    if Redundant_{i+1} ≤ Critical then
        if Descriptor.Parents ! contain(cloud) then
            addParent(cloud)
    if N_parents > Redundant_{i+1} then
        removeParent()
    [candidates] ← findBetterParent(view)
    parentOptimization([candidates])
    AF_i ← getCurrentAF()
    AF_{i+1} ← getEstimation(AF_{i-1}, AF_i)
    if AF_{i+1} ≤ AF_Critical then
        addParent(cloud)
    execute(Actions)
    wait ΔT
```

Algorithm 6.1: Algorithm executed by child nodes

At the start of each cycle a child requests information about current ratio between public and private nodes in the network γ_i getCurrentRatio (Equation (6.5)). According to the data from previous $i-1$ and current i iterations the system estimates getEstimation of γ_{i+1} and computes the redundancy threshold $Redundant_{i+1}$ for the next $i+1$ protocol iteration (Equation 6.7).

In case (1) the estimated redundancy threshold is less or equal to the critical threshold and (2) the child descriptor does not contain any cloud as a parent node, then the child includes a cloud as one of its parents. Otherwise, if the current number of parent nodes is higher than the expected redundancy threshold, then a child virtually removes redundant parent nodes from its descriptor. The decision about the parent to remove is based on how the parent nodes impact on the AF. In particular, the most unreliable parents are the candidates to be removed.

Each child node monitors the partial view $view$ given by the PSS to find parent nodes with better characteristics findBetterParent, such as the connection latency or more stable behaviour. Such parents are called *candidates*. Afterwards, a child adds the candidate parents to its descriptor up to redundancy threshold or replaces the parents are in the descriptor with better ones from candidates parentOptimization (Section 6.3.2).

Finally a child computes the current AF_i and the expected AF_{i+1} for the next iteration getEstimation. In case of AF_{i+1} is less than the defined critical availability threshold $AF_{Critical}$, a child includes a cloud entity as a parent node to guarantee service reliability.

6.3.2 Parent Changing Policy

A child node continuously monitors the network to find "better" parents. A "better" parent is a parent that provides less connection latency or more stable behaviour in the network increasing AF of the child.

Whenever a child switches a parent, it takes τ period of time to update its descriptors in the network. Since we are using CLOUDCAST for PSS, nodes for shuffling are chosen according to their age and the nodes with the higher age in the partial view are contacted first. Hence, the τ period is limited to a number of cycles equal to the cache size [99]:

$$\tau \sim t \times c \tag{6.9}$$

where c is the cache size and t is the average time of a shuffling between two neighbours.

Two main cases drive the decision process about the updating of a child descriptor. The first case is when a child decides to change one of its parents for a better one. In this case we can use the mechanism that prevents the AF decreasing during the exchange period. Whenever a child decides to change an alive parent, the former parent remains the parent for the next τ time. In other words, for the next τ time the child has $N_{parents}+1$ available parents and the AF factor is temporarily increased. Hence, this process does not decrease AF, even temporary. The second case is when a child changes one of its failed parents. During the period of time τ following the parent failure, this peer has $N_{parents}-1$ available parents. Hence the AF of this peer is decreasing. Nevertheless, the probability of a parent failure is already counted in the AF model.

6.4 Related work

Gossip-based PSSs are a widely used building block to support P2P overlay networks. They provide peers with a random sample of the nodes in the network. Most of the PSSs rely on the assumption that any time each peer is able to communicate with any peer in its sample. Nevertheless, in the modern Internet architecture a large part of the peers sit behind the NATs [17]. Such nodes become under represented in the partial views and traditional PSS cannot guarantee the uniform randomness of the samples.

There are two main approaches to deal with NAT: relaying [73] and hole punching [26]. Relaying technique is based on sending messages via a third party relay. The disadvantages of this method are that it consumes resources of relay entity and increases the communication latency. Nevertheless, the relaying technique is the only one that deals with all the types of NAT. Hole punching enables two nodes to establish direct connection with the help of a rendezvous server, even if both of the nodes are behind NATs. This NAT traversal technique is preferable when large amount of traffic is expected to be sent between nodes, for example video streaming or on-line games systems.

One of the first work that deals with NATs for PSS is the work of Drost et al. [22]. In their ARRG protocol each node maintains a list of the nodes with whom it had a successful contact in the past. In case the shuffling request fails, the next node to shuffle is chosen from this list. Nevertheless, such approach fails to create random network samples, because the nodes that are in the list are contacted more frequently than others.

Kermarrec et al. [49] propose a NAT-aware PSS based on the hole punching. They propose to use both public and private nodes as rendezvous servers. Whenever two nodes shuffle the views they become the rendezvous server for each other. Hence, when a node initiates a gossip exchange with a private node it uses a chain of rendezvous servers to execute a hole punching. Nevertheless, the length of such chain is not limited making the approach sensitive for the network dynamics and increasing the connection latencies.

In our work we are concentrated on hole punching mechanism for NAT-traversal protocols. In particular, we focus on the application of autonomous resource management between two types of rendezvous servers (what we called parent nodes): P2P public and cloud computing nodes. Our approach, even if designed for the PSS, is general enough to be applied to a wide range of P2P-based network services.

Chapter 7

Conclusion

The goal of the thesis is the realization of an efficient and scalable way to manage resources between technologies as P2P and Cloud Computing. The effective autonomous management could open new horizons in applying P2P communication paradigm by reducing the economical effort on the service providers, while offering a level of service compatible with centralized architectures. We addressed the problem of autonomous management by developing a collection of protocols that tackle the issues of resource bottlenecks, reliability and scalability. In order to evaluate the effectiveness of our strategies, we have applied them to different distributed services and applications.

Our extensive simulations of autonomous P2P/Cloud resources management showed good results for all applications in terms of system scalability and reliability. However, reducing the economical costs in comparison with pure cloud utilization required the ability to trade some of the application QoS. For example, in the cases of the video streaming and multiplayer online games, interactivity and consistency are traded to achieve economical efficiency.

The first application where we applied the combination of P2P and Cloud is a replica management service. We have designed a protocol for replica management that is able to self-regulate the amount of cloud (pay-per-use) resources when peer resources (free) are not enough. The case study we considered shows the benefits in terms of reliability that can be obtained from such approach. The protocol has been tested in different scenarios, showing the effectiveness of the approach even in highly dynamic networks.

Further, we applied the self-regulation mechanism to address load balancing in video streaming. We described and developed CLIVE, a P2P live streaming

system that integrates cloud resources (helpers) whenever the peer resources are not enough to guarantee a predefined QoS. Two types of helpers are used in CLIVE, active ones (virtual machines participating in the streaming protocol, AH) and passive ones (represented by a storage service that provides content on demand, PH). CLIVE estimates the available capacity in the system through a gossip-based aggregation protocol and provisions the required resources (AHs/PHs) from the cloud to guarantee a given level of QoS at low cost. We implemented a prototype of CLIVE system based on Amazon's services like EC2 and S3. To demonstrate the feasibility of CLIVE, we performed extensive simulations and evaluated our system using large scale experiments under dynamic realistic settings. We showed that we can save up to 45% of the cost by properly choosing the right number of AHs compared to only using a PH to guarantee the predefined QoS.

In order to estimate the distribution of a global attribute value (i.e. the available capacity) in CLIVE, we developed a lightweight gossip-based protocol, where nodes periodically exchange their local information and update it to converge towards a global aggregate value. We compared our protocols with state-of-the-art solutions, like EQUIDEPTH [34] and ADAM2 [82]. We showed that the maximum error of the enhanced model is comparable to ADAM2, and both are smaller than EQUIDEPTH and the baseline. Moreover, we show that the average error of the enhanced model is also less than EQUIDEPTH, and finally we show that the total network overhead of the baseline and enhanced model are 1% and 10% those of the EQUIDEPTH and ADAM2, respectively.

We employed the concept of combining P2P and cloud computing for large scale applications, particularly designing an architecture support of large scale on-line games. Our architecture merges the different characteristics of P2P and cloud nodes to provide an efficient and effective provisioning of resources, load balancing, while scaling in economical cost and quality of service. Our approach tackles these issues by strategically migrating the load and recruiting new resources. We also provide the ability to control the behaviour of the platform. By selecting a desired QoS level, it is possible to control the amount of cloud and user-provided resources to exploit. This allows performing aggressive strategies in terms of cost reduction. We demonstrated that trading a small amount of QoS cuts the cost of the service up to 60%, with respect to a pure cloud solution.

As the last point, we approached the problem of NAT-traversal in P2P services by proposing an approach that exploits cloud resources. We designed a cloud-assisted solution for hole punching that increases the effectiveness of P2P-based services in real Internet network. Despite of the considered PSS example the approach can be applied to variety of P2P-based Internet services and applications.

However, while our research opens new possibilities to exploit heterogeneous resources in distributed systems, there is still work to be done. First of all, we would like to extend the evaluation of the NAT-traversal approach we described in this thesis. Then, we did the general assumption that the all participants of P2P network behave like honest contributors, while often the big part of the network clients, the so-called free-riders, in fact consuming much more resources than their actual contribution [2]. Another limitation of resource management is the additional bandwidth consumptions that are caused by the necessity of the system synchronization in order to manage the network. For example, in the multiplayer on-line game, additional bandwidth overhead is required to maintain the consistency of backup replicas. These limitations hinder the flexibility and generality of our work, but at the same time they open additional research paths that can be explored, such as:

- It would be interesting to consider scenarios in which the users cannot consume network resources without contributing themselves. The form of contributions can be different and can vary from computational resources to money.

- In our application scenarios we mostly consider one influencing parameter - upload bandwidth of nodes. Even if bandwidth is often the bottleneck of P2P networked applications, at the same time it would be interesting to consider also other important nodes characteristics, such as computational resources or node uptime and locality.

- Another interesting improvement would be to exploit the possibility to use different cloud providers at the same time. It would decrease the dependency of the application on the economical policy of only one cloud provide. Moreover, the economical effectiveness can be increased due to the

autonomous switching between cloud resources, according to the suitability of the price model and service conditions.

- A possible future direction could be to integrate together the cloud-assisted NAT-traversal mechanism with the described applications (e.g. CLIVE or on-line games) and deploy them in the real Internet network.

As ultimate analysis, our thesis proposed and validated the combination of cloud and P2P for a collection of heterogeneous applications with promising results. Therefore, we strongly believe that the combination of Cloud Computing and Peer-to-Peer is the next milestone for distributed P2P-based architectures.

Bibliography

[1] Second life website. http://secondlife.com/, [Online; accessed Jan-2013].

[2] E. Adar and B.A. Huberman. Free riding on gnutella. *First Monday*, 5(10), 2000.

[3] C. Arad, J. Dowling, and S. Haridi. Developing, simulating, and deploying peer-to-peer systems using the Kompics component model. In *Proc. of the Fourth Int. ICST Conf. on COMmunication System softWAre and middlewaRE (COMSWARE'09)*, NY, USA, 2009. ACM.

[4] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[5] O. Babaoglu and M. Jelasity. Self-* properties through gossiping. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3747–3757, 2008.

[6] I. Barri, F. Giné, and C. Roig. A Scalable Hybrid P2P System for MMOFPS. *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 341–347, February 2010.

[7] B. Biskupski, M. Schiely, P. Felber, and R. Meier. Tree-based analysis of mesh overlays for peer-to-peer streaming. In *Proc. of DAIS'08*, pages 126–139. Springer, 2008.

[8] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: pick two. In *Proc. of the 9th conf. on Hot Topics in Operating Systems - Volume 9*, Berkeley, CA, USA, 2003. USENIX Association.

[9] J. Buford, H. Yu, and E.K. Lua. *P2P Networking and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[10] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1742–1751. IEEE, 2000.

[11] E. Carlini, M. Coppola, and L. Ricci. Evaluating compass routing based AOI-cast by mogs mobility models. In *Proceedings of the 4th International Conference on Simulation Tools and Techniques*, pages 328–335. ICST, 2011.

[12] E. Carlini, M. Coppola, and L. Ricci. Probabilistic dropping in push and pull dissemination over distributed hash tables. In *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pages 47–52, 2011.

[13] E. Carlini, M. Coppola, and L. Ricci. Reducing Server Load in MMOG via P2P Gossip. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, 2012.

[14] E. Carlini, L. Ricci, and M. Coppola. Flexible load distribution for hybrid distributed virtual environments. *Future Generation Computer Systems*, 2012.

[15] Emanuele Carlini. *Combining Peer-to-Peer and Cloud Computing for Large Scale On-line Games*. PhD thesis, IMT Institute for Advanced Studies Lucca, 2012.

[16] N. Carlsson and D. Eager. Peer-assisted on-demand streaming of stored media using BitTorrent-like protocols. In *Proc. of NETWORKING'07*, pages 570–581. Springer, 2007.

[17] M. Casado and M.J. Freedman. Peering through the shroud: the effect of edge opacity on ip-based client identification. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, Berkeley, CA, USA, 2007. USENIX Association.

[18] A. Chen and R.R. Muntz. Peer clustering: a hybrid approach to distributed virtual environments. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 11. ACM, 2006.

[19] B.G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of the 3rd Conf. on Networked Systems Design & Implementation (NSDI'06)*, San Jose, CA, 2006. USENIX.

[20] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, 2006.

[21] J. Dowling and A.H. Payberah. Shuffling with a croupier: Nat-aware peer-sampling. In *Proc. of ICDCS'12*, pages 102–111. IEEE, 2012.

[22] N. Drost, E. Ogston, R. van Nieuwpoort, and H.E. Bal. ARRG: real-world gossiping. In *Proceedings of the 16th International Symposium on High-Performance Distributed Computing (HPDC-16 2007), 25-29 June 2007, Monterey, California, USA*, pages 147–158. ACM, 2007.

[23] A. Duminuco, E. Biersack, and T. En-najjary. Proactive replication in distributed storage systems using machine availability estimation. In *Conference on Emerging Network Experiment and Technology*, 2007.

[24] P.Th. Eugster, R. Guerraoui, A.M. Kermarrec, and L. Massouli. Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5):60–67, 2004.

[25] I. Eyal, I. Keidar, and R. Rom. LiMoSense – live monitoring in dynamic sensor networks. In *7th Int. Symp. on Algor. for Sensor Syst., Wireless Ad Hoc Networks and Autonomous Mobile Entities*, 2011.

[26] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, Berkeley, CA, USA, 2005. USENIX Association.

[27] R. Fortuna, E. Leonardi, M. Mellia, M. Meo, and S. Traverso. QoE in pull based P2P-TV systems: overlay topology design tradeoffs. In *Proc. of P2P'10*, pages 1–10. IEEE, 2010.

[28] D. Frey, R. Guerraoui, A.M. Kermarrec, and M. Monod. Boosting gossip for live streaming. In *Proc. of P2P'10*, pages 1–10. IEEE, 2010.

[29] E.S. Gardner. Exponential smoothing: The state of the art—part II. *International Journal of Forecasting*, 22(4):637–666, 2006.

[30] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured P2P systems. In *INFOCOM 2004. Twenty-third Conference of the IEEE Computer and Communications Societies*, pages 2253–2262. IEEE, 2004.

[31] S. Goel and R. Buyya. Data replication strategies in wide area distributed systems. Technical report, Idea Group Inc., Hershey, PA, USA, 2006.

[32] K. Graffi, A. Kovacevic, S. Xiao, and R. Steinmetz. SkyEye.KOM: An information management over-overlay for getting the oracle view on structured P2P systems. In *Int. Conf. on Parallel and Distributed Systems*, 2008.

[33] K.P. Gummadi, S. Saroiu, and S.D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proc. of the SIGCOMM Internet Measurement Workshop*, 2002.

[34] M. Haridasan and R. van Renesse. Gossip-based distribution estimation in peer-to-peer networks. In *Proc. of IPTPS'08*. USENIX, 2008.

[35] C. Hu, M. Chen, C. Xing, and B. Xu. EUE principle of resource scheduling for live streaming systems underlying CDN-P2P hybrid architecture. *Peer-to-Peer Networking and Applications*, 5(4):1–11, 2012.

[36] S.Y. Hu. Spatial Publish Subscribe. *Proc. of IEEE Virtual Reality (IEEE VR) workshop, Massively Multiuser Virtual Environment (MMVE)*, 2009.

[37] S. Idreos, M. Koubarakis, and Ch. Tryfonopoulos. P2P-DIET: an extensible P2P service that unifies ad-hoc and continuous querying in super-peer

networks. In *Proc. of the ACM Int. Conf. on Management of data (SIGMOD'04)*. ACM, 2004.

[38] J. Jardine and D. Zappala. A hybrid architecture for massively multi-player online games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, page 60. ACM, 2008.

[39] M. Jelasity and A. Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Proc. of ICDCS'04*, pages 102–109. IEEE, 2004.

[40] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3):219–252, 2005.

[41] M. Jelasity, A. Montresor, and O. Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, 2009.

[42] M. Jelasity, S. Voulgaris, R. Guerraoui, A.M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25, 2007.

[43] P. Jesus, C. Baquero, and P.S. Almeida. Fault-tolerant aggregation for dynamic networks. In *SRDS*, pages 37–43, 2010.

[44] X. Jin and Y. Kwok. Cloud assisted P2P media streaming for bandwidth constrained mobile subscribers. In *Proc. of IPDPS'10*, pages 800–805. IEEE, 2010.

[45] D. Jurca and R. Stadler. H-GAP: estimating histograms of local variables with accuracy objectives for distributed real-time monitoring. *IEEE Transactions on Network and Service Management*, 7(2):83–95, 2010.

[46] H. Kavalionak, E. Carlini, L. Ricci, A. Montresor, and M. Coppola. Integrating peer-to-peer and cloud computing for massively multiuser online games. *Peer-to-Peer Networking and Application*, pages 1–19, 2013.

[47] H. Kavalionak and A Montresor. P2P and cloud: A marriage of convenience for replica management. In *Proc. of IWSOS'12*, pages 60–71. Springer, 2012.

[48] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[49] A.M. Kermarrec, A. Pace, V. Quéma, and V. Schiavoni. NAT-resilient gossip peer sampling. In *ICDCS*, pages 360–367. IEEE Computer Society, 2009.

[50] K. Kim. Lifetime-aware replication for data durability in P2P storage network. *IEICE Trans. on Communications*, E91-B:4020–4023, 2008.

[51] K.C. Kim, I. Yeom, and J. Lee. HYMS: A hybrid mmog server architecture. *IEICE Transactions on Information and Systems*, E87:2706–2713, 2004.

[52] R. Kumar and K.W. Ross. Optimal peer-assisted file distribution: Single and multi-class problems. In *Proc. of HOTWEB'06*. IEEE, 2006.

[53] R. Kumar and K.W. Ross. Peer-assisted file distribution: The minimum distribution time. In *Proc. of HOTWEB'06*, pages 1–11. IEEE, 2006.

[54] S. Legtchenko. Blue Banana: resilience to avatar mobility in distributed MMOGs. *Networks*, pages 171–180, 2010.

[55] B. Li, S. Xie, Y. Qu, G.Y. Keung, C. Lin, J. Liu, and X. Zhang. Inside the new Coolstreaming: Principles, measurements and performance implications. In *Proc. of INFOCOM'08*, pages 1031–1039. IEEE, 2008.

[56] Y. Lu, B. Fallica, F.A. Kuipers, R.E. Kooij, and P.V. Mieghem. Assessing the quality of experience of Sopcast. *International Journal of Internet Protocol Technology*, 4(1):11–23, 2009.

[57] Z.H. Lu, X.H. Gao, S.J. Huang, and Y. Huang. Scalable and reliable live streaming service through coordinating CDN and P2P. In *Proc. of ICPADS'11*, pages 581–588. IEEE, 2011.

[58] N. Markatchev, R. Curry, C. Kiddle, A. Mirtchovski, R. Simmonds, and T. Tan. A cloud-based interactive application service. In *e-Science, 2009. e-Science'09. Fifth IEEE International Conference on*, pages 102–109. IEEE, 2009.

[59] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: Providing consistency for replicated continuous applications. *IEEE Transactions on Multimedia*, 6(1):47–57, 2004.

[60] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First Int. Workshop on Peer-to-Peer Systems*, IPTPS'01, pages 53–65, London, UK, 2002. Springer-Verlag.

[61] P. Michiardi, D. Carra, F. Albanese, and A. Bestavros. Peer-assisted content distribution on a budget. *Computer Networks*, 56(7):2038–2048, 2012.

[62] D.C. Montgomery, C.L. Jennings, and M. Kulahci. *Introduction to time series analysis and forecasting*, volume 526. Wiley, 2011.

[63] A. Montresor and L. Abeni. Cloudy weather for P2P, with a chance of gossip. In *Proc. of the 11th IEEE P2P Conference on Peer-to-Peer Computing (P2P'11)*, pages 250–259. IEEE, August 2011.

[64] A. Montresor and A. Ghodsi. Towards robust peer counting. In *Proc. of P2P'09*, pages 143–146. IEEE, 2009.

[65] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, September 2009.

[66] A. Montresor, M. Jelasity, and Ö. Babaoglu. Decentralized ranking in large-scale overlay networks. In *SASO Workshops*, pages 208–213, 2008.

[67] V. Nae, A. Iosup, S. Podlipnig, R. Prodan, D. Epema, and T. Fahringer. Efficient management of data center resources for Massively Multiplayer Online Games. *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2008.

[68] V. Nae, R. Prodan, A. Iosup, and T. Fahringer. A new business model for massively multiplayer online games. In *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering*, pages 271–282. ACM, 2011.

[69] M.E.J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[70] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Proc. of IPTPS'05*, pages 127–140. Springer, 2005.

[71] L. Pamies-Juarez and P. Garcia-Lopez. Maintaining data reliability without availability in P2P storage systems. In *Proc. of the 2010 ACM Symp. on Applied Computing*, SAC'10, pages 684–688, Sierre, Switzerland, 2010. ACM.

[72] A.H. Payberah, J. Dowling, and S. Haridi. Glive: The gradient overlay as a market maker for mesh-based P2P live streaming. In *Proc. of ISPDC'11*, pages 153–162. IEEE, 2011.

[73] A.H. Payberah, J. Dowling, and S. Haridi. Gozar: NAT-friendly peer sampling with one-hop distributed NAT traversal. In *Proc. of DAIS'11*, pages 1–14. Springer, 2011.

[74] A.H. Payberah, J. Dowling, F. Rahimian, and S. Haridi. Gradientv: Market-based P2P live media streaming on the gradient overlay. In *Proc. of DAIS'10*, pages 212–225. Springer, 2010.

[75] A.H. Payberah, H. Kavalionak, V. Kumaresan, A. Montresor, and S. Haridi. CLive: Cloud-Assisted P2P Live Streaming. In *Proc. of the 12th Conf. on Peer-to-Peer Computing (P2P'12)*, 2012.

[76] A.H. Payberah, H. Kavalionak, A. Montresor, J. Dowling, and S. Haridi. Lightweight gossip-based distribution estimation. In *Proc. of the 15th IEEE International Conference on Communications (ICC'13)*. IEEE, June 2013.

[77] A.H. Payberah, H. Kavalionak, A. Montresor, and S. Haridi. CLive: Hybrid P2P-Cloud Live Streaming System. *IEEE Transactions on Parallel and Distributed Systems*, 2013. Submitted 01-10-2013.

[78] A.H. Payberah, F. Rahimian, S. Haridi, and J. Dowling. Sepidar: Incentivized market-based P2P live-streaming on the gradient overlay network. In *Proc. of ISM'10*, pages 1–8. IEEE, 2010.

[79] R. Roverso, S. El-Ansary, and S. Haridi. NATCracker: NAT Combinations Matter. *Computer Communications and Networks, International Conference on*, pages 1–7, 2009.

[80] R. Roverso, S. El-Ansary, and S. Haridi. SmoothCache: HTTP-live streaming goes peer-to-peer. In *Proc. of NETWORKING'12*, pages 29–43. Springer, 2012.

[81] A.I.T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. of the IFIP/ACM Int. Conf. on Distributed Systems Platforms Heidelberg*, Middleware'01, pages 329–350, London, UK, 2001. Springer-Verlag.

[82] J. Sacha, J. Napper, C. Stratan, and G. Pierre. Adam2: Reliable distribution estimation in decentralised environments. In *Proc. of the 30th Int. Conf. on Distr. Comput. Systems (ICDCS'10)*, Italy, 2010. IEEE Computer Society.

[83] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[84] G. Schay. *Introduction to probability with statistical applications*. Birkhäuser, 2007.

[85] L. Schubert and European Commision. The Future of Cloud Computing: Opportunites for European Cloud Computing Beyond 2010, 2010.

[86] E. Sit, A. Haeberlen, F. Dabek, B.G. Chun, H. Weatherspoon, R. Morris, M.F. Kaashoek, and J. Kubiatowicz. Proactive replication for data durability. In *5th Int. Workshop on Peer-to-Peer Systems*, IPTPS'06, 2006.

[87] S. Spoto, R. Gaeta, M. Grangetto, and M. Sereno. Analysis of PPLive through active and passive measurements. In *Proc. of IPDPS'09*, pages 1–7. IEEE, 2009.

[88] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang. The feasibility of supporting large-scale live streaming applications with dynamic application end-points. *ACM SIGCOMM Computer Communication Review*, 34(4):107–120, 2004.

[89] K. Sripanidkulchai, B. Maggs, and H. Zhang. An analysis of live streaming workloads on the internet. In *Proc. of IMC'04*, pages 41–54. ACM, 2004.

[90] K. Sriram, D. Montgomery, O. Borchert, Okhee Kim, and D.R. Kuhn. Study of BGP peering session attacks and their impacts on routing performance. *IEEE J.Sel. A. Commun.*, 24(10):1901–1915, October 2006.

[91] R. Steinmetz and K. Wehrle. *Peer-to-Peer Systems and Applications*, volume 3485. Lecture Notes in Computer Science, 2005.

[92] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Computer Communication Review*, 31(4):149–160, 2001.

[93] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[94] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *Proc. of IMC'06*, volume 25, pages 189–202, 2006.

[95] R. Sweha, V. Ishakian, and A. Bestavros. Angels in the cloud: A peer-assisted bulk-synchronous content distribution service. In *Proc. of CLOUD'11*, pages 97–104. IEEE, 2011.

[96] R. Sweha, V. Ishakian, and A. Bestavros. AngelCast: cloud-based peer-assisted live streaming using optimized multi-tree construction. In *Proc. of MMsys'12*, pages 191–202. ACM, 2012.

[97] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21:164–206, 2003.

[98] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proc. of the IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, London, UK, 1998. Springer-Verlag.

[99] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[100] S. Voulgaris and M. van Steen. Epidemic-style management of semantic overlays for content-based searching. In *Proc. of the 11th Int. Euro-Par Conference*, volume 3648 of *LNCS*, pages 1143–1152, Lisbon, Portugal, August 2005. Springer.

[101] J. Wang and K. Ramchandran. Enhancing peer-to-peer live multicast quality using helpers. In *Proc. of ICIP'08*, pages 2300–2303. IEEE, 2008.

[102] J. Wang, C. Yeo, V. Prabhakaran, and K. Ramchandran. On the role of helpers in peer-to-peer file download systems: Design, analysis and simulation. In *Proc. of IPTPS'07*, 2007.

[103] Ch. Williams, Ph. Huibonhoa, J. Holliday, A. Hospodor, and Th. Schwarz. Redundancy management for P2P storage. In *Proc. of the Seventh IEEE Int. Symp. on Cluster Computing and the Grid*, CCGRID'07, pages 15–22, Washington, DC, USA, 2007. IEEE Computer Society.

[104] Y. Wu, C. Wu, B. Li, X. Qiu, and F.C.M. Lau. Cloudmedia: When cloud on demand meets video on demand. In *Proc. of ICDCS'11*, pages 268–277. IEEE, 2011.

[105] Praveen Yalagandula and Mike Dahlin. A scalable distributed information management system. In *Proc. of the Conf. on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'04)*, pages 379–390, New York, NY, USA, 2004. ACM.

[106] Zh. Yang, B.Y. Zhao, Y. Xing, S. Ding, F. Xiao, and Y. Dai. AmazingStore: available, low-cost online storage service using cloudlets. In *Proc. of the 9th Int. Workshop on Peer-to-peer Systems*, IPTPS'10, San Jose, CA, 2010. USENIX.

[107] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, Ch. Lin, H. Zhang, and B. Li. Design and deployment of a hybrid CDN-P2P system for live video streaming: experiences with LiveSky. In *Proc. of Multimedia'09*, pages 25–34. ACM, 2009.

[108] W.P.K. Yiu, X. Jin, and S.H.G. Chan. Challenges and approaches in large-scale P2P media streaming. *MultiMedia*, 14(2):50–59, 2007.

[109] H. Zhang, J. Wang, M. Chen, and K. Ramchandran. Scaling peer-to-peer video-on-demand systems using helpers. In *Proc. of ICIP'09*, pages 3053–3056. IEEE, 2009.

[110] X. Zhang, J. Liu, B. Li, and Y.S.P. Yum. CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming. In *Proc. of INFOCOM'05*, pages 2102–2111. IEEE, 2005.

[111] B.Q. Zhao, J. Lui, and D.M. Chiu. Exploring the optimal chunk selection policy for data-driven P2P streaming systems. In *Proc. of P2P'09*, pages 271–280. IEEE, 2009.