Adaptive Cloud Publish-Subscribe Services for Latency-Constrained Applications

Julien Gascon-Samson

Doctor of Philosophy

School of Computer Science
McGill University
Montreal, Quebec, Canada

October 2016

A thesis submitted to McGill University in partial
fulfillment of the requirements of the degree of
Doctor of Philosophy

Abstract

With the advent of very high-speed connections, modern web applications, smartphones and mobile applications, large-scale internet services have become ubiquitous and are part of our daily lives. Nowadays, in many of these services, such as social media, not only do users consume the contents, but they also contribute to the production of the contents. In addition, users want to be dynamically informed of changes to the contents in which they are interested in, notably by means of *push notifications*.

The publish/subscribe model is an efficient paradigm that can be leveraged in these contexts, as it provides a nice abstraction that allows for logically and efficiently decoupling content producers (publishers) from content consumers (subscribers). Publish/subscribe is typically provided as a service, in which subscribers register interest in (subscribe to) contents that they want to receive. Then, as publishers generate and submit contents in the form of publications to the service, the latter determines to which subscribers each publication should be sent to, and forwards each publication accordingly to the relevant subscribers. While multiple variants of the publish/subscribe paradigm have been described in the literature, this thesis is centered around topic-based publish/subscribe, which enjoys widespread usage in large-scale commercial systems.

Supporting large-scale topic-based publish/subscribe applications brings interesting research challenges, notably regarding the scalability and load balancing aspects, as some applications built on these systems can generate high message volumes. In addition, some specific applications impose additional constraints, such as multiplayer online games (MOG), in which publication delivery latencies must be kept below a given threshold, which can be particularly challenging when clients are distributed around the world. The cloud can be leveraged in these contexts, as a publish/subscribe service deployed in the

cloud can benefit from the large pool of resources that the cloud can provide in several geographical regions.

This thesis proposes a set of contributions in the general area of scaling cloud-based topic-based publish/subscribe systems. Our first contribution, Dynamoth, provides a scalable topic-based publish/-subscribe service that is tailored for latency-constrained applications. It provides a hierarchical scalability and load balancing model that exploits the intrinsic characteristics of the topic-based publish/-subscribe paradigm. In addition, Dynamoth also provides availability and fault tolerance in the event of server failures and provides several levels of reliability and ordering guarantees. Our second contribution, MultiPub, provides a global-scale topic-based pub/sub service tailored for the needs of applications with many clients around the world, and having strict latency constraints. As such, it allows one to impose latency constraints. MultiPub then continuously makes sure that these constraints are satisfied (if possible), by generating optimal configurations of cloud deployments spanning across several of the available regions. As cloud usage incurs bandwidth-related costs, and that different cloud regions exhibit different costs, MultiPub also attempts to reduce such costs by selecting the most cost-efficient configuration that respects latency constraints. On the other end, our third contribution, DynFilter, proposes a game-oriented topic-based publish/subscribe service that aims at limiting bandwidth usage in multiplayer and massively multiplayer online games. As DynFilter is game-specific, it exploits the conceptual spatial model of such games in order to inhibit the dissemination of publications that are of a lesser importance in a game setting, in a dynamic way, in order to achieve target bandwidth savings.

All of our experiments are run in the context of multiplayer online games, as the topic-based publish/subscribe paradigm fits well into the architectural model of such games. In addition, they are a good example of highly distributed, latency-constrained systems. As running experiments in the cloud is a challenging task, this thesis provides, as an additional contribution, a set of tools that were developed to assist in running large-scale, highly-distributed cloud-based experiments. Among these contributions is a full, reusable implementation of our Dynamoth platform, built according to software engineering principles.

In summary, we believe that this thesis provides new and innovative contributions in the cloud-based scalability of topic-based publish/subscribe.

Abrégé

Avec l'avènement des connexions haute-vitesse, des applications web modernes, des téléphones intelligents et des applications mobiles, les services internet à large échelle sont maintenant omniprésents et font désormais partie de notre quotidien. De nos jours, au sein de plusieurs de ces services tels que les médias sociaux, les utilisateurs ne sont plus de simples consommateurs de contenu, mais participent également à la production du contenu. De plus, les utilisateurs désirent être informés dynamiquement des changements au contenu qui les intéresse, notamment par le biais de notifications de type *push*.

Le modèle de publication/souscription représente un paradigme efficace qui peut être exploité dans ces contextes, puisqu'il fournit une abstraction qui permet de découpler de façon logique et efficace les producteurs de contenu (émetteurs) des consommateurs de contenu (souscripteurs). Le modèle de publication/souscription est typiquement fourni en tant que service, au sein duquel les souscripteurs expriment leur intérêt envers le (souscrivent au) contenu qu'ils souhaitent recevoir. Puis, au fur et à mesure que les émetteurs soumettent leur contenu sous la forme de publications au service de publication/souscription, ce dernier détermine à quels souscripteurs les différentes publications doivent être transmises, et procède au transfert des publications vers les bons souscripteurs. Plusieurs variantes du paradigme de publication/souscription ont été décrites dans la littérature. Cette thèse s'articule autour de l'une de ces variantes, le modèle de publication/souscription orienté-sujet, qui jouit d'une utilisation répandue dans les systèmes commerciaux à large échelle.

La prise en charge des applications à large échelle basées sur le modèle de publication/souscription orienté-sujet apporte des défis de recherche intéressants, notamment en ce qui a trait aux aspects d'évolutivité et de balancement de charge, puisque certaines applications construites sur ce modèle

peuvent générer un volume important de messages. De plus, certaines applications spécifiques imposent des contraintes additionnelles, telles que les jeux massivement multijoueurs en ligne (MOG), dans lesquels la dissémination des publications doit s'effectuer dans un délai strict, ce qui peut constituer un défi important surtout lorsque les clients sont géographiquement dispersés à travers la planète. L'utilisation du nuage informatique peut s'avérer bénéfique dans ces contextes, puisque le déploiement du service de publication/souscription dans cet environnement peut permettre au service d'avoir accès au large éventail de ressources que le nuage peut fournir dans plusieurs régions géographiques.

Cette thèse propose un ensemble de contributions dans le domaine général de l'évolutivité des systèmes de publication/souscription orientés-sujet dans le nuage. Notre première contribution, Dynamoth, propose un service évolutif de publication-souscription orienté–sujet qui est optimisé pour les besoins des applications contraintes en latence. Dynamoth propose un modèle hiérarchique d'évolutivité et balancement de charge qui exploite les caractéristiques intrinsèques du paradigme de publication/-souscription orienté-sujet. De plus, Dynamoth propose des propriétés de disponibilité et de tolérance aux pannes en cas d'échec de serveurs et propose différents niveaux de fiabilité et de garanties d'ordonnancement. Notre seconde contribution, MultiPub, propose un service de publication/souscription orienté-sujet à échelle globale conçu pour les besoins des applications avec des utilisateurs répartis à travers le monde, et ayant des contraintes strictes en latence. Dans cette optique, MultiPub permet aux applications d'imposer des différentes contraintes de latence. MultiPub s'assure alors sur une base continue que les contraintes définies sont respectées (si possible), en générant des configurations optimales de déploiements tirant parti d'un ensemble de régions infonuagiques disponibles. Puisque l'utilisation du nuage génère des coûts reliés à l'utilisation de la bande passante, et que ces coûts diffèrent au sein des différentes régions infonuagiques, MultiPub vise également à réduire les coûts en sélectionnant la configuration la plus économique qui respecte les contraintes en latence imposées. Dans un autre ordre d'idées, notre troisième contribution, DynFilter, propose un service de publication/souscription orienté-sujet conçu spécifiquement pour les besoins des jeux multijoueurs en ligne. DynFilter vise à limiter l'utilisation de la bande passante au sein de tels jeux. De par sa nature, DynFilter exploite le modèle conceptuel spatial propre aux jeux multijoueurs afin de limiter la dissémination des publications de moindre importance dans un tel contexte, de façon dynamique, afin d'atteindre des économies de bande passante ciblées.

L'ensemble de nos expérimentations ont été menées dans le contexte de jeux multijoueurs, puisque le paradigme de publication/souscription orienté-sujet s'harmonise bien avec le modèle architectural

de ces jeux. De plus, les jeux en ligne constituent un bon exemple de systèmes hautement distribués et contraints en latence. L'exécution d'expérimentations dans le nuage amène certes son lot de défis. Pour cette raison, cette thèse propose, en tant que contribution additionnelle, un ensemble d'outils qui ont été développés pour aider à l'exécution d'expérimentations à large échelle et hautement distribuées dans le nuage. L'une de ces contributions est une implémentation complète et réutilisable de notre plate-forme Dynamoth, qui a été construite selon des principles d'ingénierie logicielle.

En résumé, nous croyons que cette thèse amène des contributions nouvelles et innovatrices dans le domaine de l'évolutivité des systèmes de publication/souscription orientés-sujet et déployés dans le nuage.

Contributions

This section summarizes the main contributions that are part of this thesis. Note that our contributions are described in more details at section 1.2. The complete list of our publications can be found at the end of this thesis, including some contributions that are not part of this thesis.

Dynamoth [53, 55] Our Dynamoth contribution proposes a scalable and fault tolerant topic-based publish/subscribe service for cloud-based environments. Initial aspects of our work were first published in NetGames 2013 as a short paper / poster [55], and the core of our work was then published as a paper in ICDCS 2015 [53] (acceptance ratio of 12.8%). For these two contributions, I worked on the initial ideas and model, built the Dynamoth platform implementation, conducted the various experiments and wrote the paper, with the guidance of my advisors Bettina Kemme and Jörg Kienzle, with whom I had regular meetings and who collaborated towards reviewing the paper. A third year undergraduate student, Franz-Philippe Garcia, collaborated on implementing specific features of our model within our Dynamoth implementation, ran some experiments for the features he implemented and produced a technical report [52]. Note that our current publications do not include the availability and fault tolerance aspects. We are in the final stages of preparing a journal extension to Dynamoth that we plan on submitting to IEEE TPDS at the end of the Fall 2016 term. The contents of chapter 3 of this thesis is derived from our Dynamoth conference publication [53] and from our upcoming Dynamoth journal publication.

MultiPub Our MultiPub contribution proposes a latency and cost-aware topic-based publish/subscribe service for global-scale applications. In the context of MultiPub, I worked on the initial ideas and the model, and then I build two implementations of our system in order to run simulated and

cloud experiments, in order to write a paper (which is not published yet). My advisors, Bettina Kemme and Jörg Kienzle, once again provided me with their guidance, through regular meetings, as well as their feedback and collaboration on reviewing the paper. We plan on submitting to ICDCS 2017 in December 2016. The contents of chapter 4 is derived from our MultiPub paper to be submitted.

DynFilter Our MultiPub contribution proposes a game-oriented topic-based publish/subscribe service. In the context of DynFilter, I worked on the initial ideas, implemented the model, ran the experiments in the cloud and wrote the paper. My advisors also guided me throughout this project, once again through regular meetings, as well as through the feedback that they provided me with regarding the paper. Our results were published in NetGames 2013 [54]. The contents of chapter 5 is derived from our NetGames publication.

Acknowledgments

Foremost, I would like to thank my two advisors, Bettina Kemme and Jörg Kienzle, for the generous amount of time that they put into my progression as a PhD student and my training into the world of academia. Their support and contributions have been invaluable. I am very grateful of their encouragements, enthusiasm, and the fact that they were pretty much always available and that they always responded quickly whenever I had something to ask. I sometimes asked them for some things on very short notice (e.g., reference letters for funding applications), and they were always available and willing to help and accommodate. I am also grateful to them for letting me express new ideas and new research orientations, and for showing interest in pursuing these ideas. In addition to my supervisors, I am grateful to Professor Muthucumaru Maheswaran for his help, as part of my progress committee.

I would also like to thank my close friends who have always been there for me, as well as my lab colleagues, both from the Software Engineering lab and from the Distributed Information Systems lab, for their feedback, support and interesting discussions and exchange of ideas: Amir Yahyavi, Omar Alam, Chris Dragert, Matthias Schöttle, Nishanth Thimmegowda, Céline Bensoussan, Berk Duran, Yousuf Ahmad, Omar Asad, Cesar Canas, Joseph Dasilva, Masoume Jabbarifar and many more. I also wish to thank the students that I co-supervised and who collaborated to some of my projects: Joscha Lausch, Franz-Philippe Garcia, Fan Jin, Michael Coppinger, Aaron Uthayagumaran and Tristano Teneglia.

I want to express my gratitude to the organizations which provided me with funding to pursue my doctoral studies: the Fonds québécois recherche nature et technologies (FQRNT), the Fondation Desjardins, the Lime Connect / Google Foundation, the McGill School of Computer Science, the Caisse

Desjardins d'Ahuntsic and the Caisse Desjardins du Sault-au-Récollet, as well as the funding that my advisors provided me with. Without these sources of funding, realizing this thesis would not have been possible.

I should not forget to thank my family who was always around me, and who provided me with constant moral support, help, availability, affection and patience for the past 30 years: my mom Jocelyne, my dad Claude, my sister Catherine, my grandmother Janine, as well as Josée, France, Claudette, Jean-François, Louise, Alain, Denise and Micheline. This thesis would not have been possible without them.

Last but not least, I want to express my deep appreciation and gratitude to my partner Aimée for her constant encouragement, support, patience and understanding of sometimes (often) long working hours towards the completion of this thesis.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

Large-scale Internet services have become ubiquitous and are part of our daily lives. Nowadays, many people make use of online services on a daily basis, and even several times per day. Popular online services include emailing, social media, online news, gaming, online banking, streaming video contents, among many others. In addition, with the advent of smartphones, more and more people are connected at all times. Smartphones, combined with the ever-increasing popularity of social media, brought an important revolution: not only do users access *the contents*, but *the contents* now come to the users, through the form of *push notifications* and other similar mechanisms. Moreover, as we observe in many different applications, users are not limited to being content consumers anymore, but play an ever-increasing role in content production. This trend can be observed in many domains, such as, social media, blogs, video hosting services, in which users *define* the contents and consume the contents created by other users.

1.1 Motivation

Undoubtedly, supporting such large-scale, always-available and highly dynamic and distributed systems brings tremendous challenges, notably in regards to the software and hardware infrastructure that is needed. In the field of software and systems engineering, *paradigms* play an important role. Paradigms refer to abstracting certain facets of software systems into a set of well-known abstractions that can then be reused across different applications or domains. One such paradigm is the publish/-subscribe model, which allows for logical and efficient decoupling of content producers from content

subscribers. The publish/subscribe paradigm is very relevant in the context of large and Internet-scale applications in which users produce and consume contents, as it captures well the relationship between the different entities involved.

In an abstract form, publish/subscribe allows content consumers to express interest in and subscribe to contents produced by content producers. For example, in a social media setting such as Facebook, a user can post content to his/her wall. All interested users (*friends* in the Facebook terminology) can then express interest in this content. In a mobile application setting, a user might *subscribe* to receive weather alerts for his/her geographical region, *published* by weather service providers. One can quickly find many other applications that can benefit in a similar fashion from a publish/subscribe approach.

In principle, publish/subscribe can be offered as a service, where the service provider accepts and maintains subscription requests, and accepts publications and forwards them to relevant subscribers. Thus, a publish/subscribe service maintains a registry of all subscriptions and, upon receiving publications to be transmitted, matches each of these publications against the set of active subscriptions. Then, the service disseminates the publication to the set of interested subscribers. This thesis is centered around topic-based publish/subscribe, which is one particular flavor of publish/subscribe that enjoys widespread use in commercial systems. In topic-based publish/subscribe, subscribers register interest in *topics*. At the same time, publishers tag their publications with a topic. The publish/subscribe service then sends a publication for a particular topic to the subscribers that expressed interest in this topic. Note that beside topic-based publish/subscribe, some other flavors of publish/subscribe do exist and are described in section 2.1.

Scaling Publish/Subscribe Systems Finding matching subscribers and disseminating publications to subscribers incurs processing costs. Additionally, the dissemination process requires bandwidth. In the context of topic-based publish/subscribe, outgoing bandwidth is often a bottleneck, as the matching process is fairly simple while one single publication might need to be sent to many subscribers. Twitter is a notable example of a large-scale system that follows a topic-based publish/subscribe paradigm. In Twitter, over 7000 *tweets* (publications) are sent every second[1] by different users (publishers). Each tweet from a given user must be delivered by the service to all the followers (subscribers) of that user. In addition, the top users have close to 100 million followers [2], which means that every publication sent

[1] http://www.internetlivestats.com/twitter-statistics/

[2] http://twittercounter.com/pages/100, retrieved on October 2nd, 2016

by these users must be delivered to close to 100 million subscribers. Evidently, one single machine cannot process such a high volume of publications from a bandwidth perspective. Therefore, there is a need to scale such systems beyond one single machine in order to offer large-scale publish/subscribe services. One way to scale topic-based pub/sub systems, e.g., is to spread the set of active topics over the available machines, so that a single machine is only in charge of a limited amount of subscriptions and publications, and therefore has to disseminate only a subset of the overall publications. Different schemes exist, and are described in the next chapter.

The cloud can be used to support the scaling of publish/subscribe systems, as it can provide scalability by means of its virtually unlimited resources. However, important challenges lie in architecting such systems for cloud environments, especially regarding the distribution of the load/bandwidth (subscription, topics) among multiple machines, which is a major focus of this thesis.

Multiplayer Games: A Large-Scale Publish/Subscribe Application Multiplayer games represent a multi-billion dollar industry nowadays, with many gamers interacting through different devices, e.g., personal computers, game consoles and mobile devices. Large-scale and massive multiplayer online games exhibit a wide range of challenges of their own, including scalability challenges, as they are highly distributed systems featuring several hundreds and thousands of players within a shared virtual space. As several bodies of work demonstrated [70, 45, 25], the publish/subscribe paradigm can adequately model interest relationships and the exchange of information between the various in-game entities in the virtual world. Games also impose latency constraints on publish/subscribe systems, as information must be delivered quickly to players, in order to maintain the immersive experience that keep players engaged. For these considerations, this thesis considers games as a publish/subscribe application for our different experiments, as they provide several challenges and provide an intuitive model for our large-scale tests. In addition, part of this thesis aim at proposing publish/subscribe systems that are optimized for the needs of (large-scale) multiplayer games.

1.2 Thesis Contributions

This thesis makes three major research contributions in the general area of topic-based large-scale publish/subscribe systems. More specifically, our contributions address several scalability challenges, such as load balancing, latency optimization, cost minimization and bandwidth limitation. In addition,

this thesis also brings an additional engineering-related contribution, in the form of a set of tools that we developed to implement the models that we proposed and to assist in running large-scale experiments.

Dynamoth The *Dynamoth* system proposes a scalable and fault-tolerant topic-based publish/subscribe platform for cloud-based environments. To the best of our knowledge, very few cloud-based topic-based scalable publish/subscribe systems have been proposed in academia, and we think that Dynamoth provides a novel, relevant contribution in this area. A notable feature of Dynamoth is that it was designed with the needs of latency-constrained applications in mind, such as games, in which publications must be delivered as fast as possible. As such, it provides a flat architecture where publishers and subscribers communicate directly with publish/subscribe servers in the cloud, and where publications are delivered as fast as possible without having to go through a series of servers. Dynamoth proposes a hierarchical load balancer that adjusts the system configuration dynamically, including automatically adding/removing servers as needed. Dynamoth also provides a failure recovery mechanism that guarantees continuous message delivery despite failures of one or more servers, while providing several levels of guarantees regarding message delivery and ordering.

MultiPub The *MultiPub* system is an extension of Dynamoth and aims at minimizing costs and latency in a *global-scale* cloud setting for applications with strict latency needs. While Dynamoth already takes user-specified latency constraints into account, MultiPub can take advantage of cloud resources located in several regions of the world. By exploiting the locality of the users of the publish/subscribe system, MultiPub is able to split the load across the different available regions in an optimized manner, in order to come up with configurations that are as cost-efficient as possible, while meeting target latency constraints. In order to reach these goals, MultiPub offers a fine-grained load balancing model that takes the intrinsic characteristics of the topic-based publish/subscribe paradigm into consideration. To the best of our knowledge, MultiPub represents a novel contribution, as it is the first topic-based publish/subscribe system that considers both monetary costs reduction and latency optimization, in a global-scale cloud setting.

DynFilter *DynFilter* is a game-specific topic-based publish/subscribe system for the cloud. Considering the fact that games can consume large amounts of bandwidth, and that bandwidth incurs costs in the cloud, DynFilter aims at limiting bandwidth usage within games in order to meet predefined quotas. DynFilter's contributions lie in proposing a topic-based publish/subscribe model that takes advantage

of game semantics, and where the optimization criteria is bandwidth usage. As it was the case with Dynamoth and MultiPub, DynFilter is offered as a service and provides a real-time load balancer that adapts to the current load conditions monitored by the infrastructure.

Dynamoth and Software Tools In order to truly assert the relevance of our different contributions, and to compare them against other approaches, we built a full implementation of our approaches. Thus, we built the Dynamoth software platform, which allowed us to run our various large-scale experiments on topic-based pub/sub systems, both in cloud environments and in cloud-like environments. Experiments were run successfully with over 1200 pub/sub clients. The Dynamoth software platform was carefully engineered to be modular, extensible and reusable among other characteristics. We plan to release this platform as open source in a near future, so that other researchers can build upon it to develop their own pub/sub scalability models, and to run large-scale experiments. Dynamoth was successfully reused to implement the MultiPub and DynFilter models. In addition to Dynamoth, we also built a simulator implementing the MultiPub model (simulation-based results were compared against results obtained from our real implementation). We also developed a set of tools to assist in running very large-scale experiments in the cloud (Distmoth).

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 presents some relevant background notions, as well as relevant related work.

Chapter 3 presents our Dynamoth system and its main contributions in the area of load balancing, scalability, availability and fault tolerance of topic-based publish/subscribe systems.

Chapter 4 presents our MultiPub system and its general contributions towards cost and latency optimization in the context of such systems.

Chapter 5 presents DynFilter, our game-oriented topic-based pub/sub system that aims at restricting bandwidth usage in game-specific pub/sub applications.

All chapters motivate and describe the problem in hand, describe main contributions, present our concrete solution, describe its implementation and present a detailed performance evaluation that shows the feasibility and relevance of our approaches.

1.3 Thesis Organization

Chapter 6 presents the different software platforms and tools that we developed and that we provide as an additional contribution to this thesis: the Dynamoth platform, the set of tools that we developed to run our large-scale experiments (notably, Distmoth) and the implementation of our MultiPub simulator.

While our three chapters describing our research contributions (chapters 3, 4 and 5) contain their own conclusions, chapter 7 provides our overall conclusions to this thesis and opens areas for future work.

2

Background and Related Work

This section presents the common concepts that are relevant to the understanding of this thesis, as well the related work.

We introduce the publish/subscribe paradigm and the various languages of pub/sub in section 2.1. In section 2.2, we describe the general architecture of publish/subscribe systems. We then discuss in section 2.3 the different scalability approaches of publish/subscribe systems as well as the related work. In section 2.4, the reliability and optimization aspects of publish/subscribe systems are covered. In section 2.5, we describe some popular commercial and open-source large-scale publish/subscribe systems. Finally, in section 2.6, we discuss multiplayer games as one important application of publish/subscribe systems that brings several interesting challenges.

2.1 Publish/Subscribe Languages

The publish-subscribe (pub/sub) concept [31, 47, 48] is an extremely popular communication paradigm that is used across a wide range of application domains because it provides efficient and elegant ways to decouple content producers (publishers) from content consumers (subscribers). In a typical publish/-subscribe system, subscribers *subscribe* to contents that they are interested in receiving, and publishers *publish* messages (called publications) that are transmitted to relevant subscribers.

One of the fundamental challenges of publish/subscribe lies in *matching* publications to subscribers; that is, resolving to which subscribers any given publication should be sent to. The typical

modus operandi is for publishers to tag each publication with specific metadata, and to have subscribers subscribe to specific subscription predicates. The matching process then involves determining, for each publication issued by a given publisher *and* for each subscription registered by a given subscriber, whether the metadata tagged to the specific publication matches the specific subscription predicate; in which case the publication is delivered to the subscriber. Depending on the matching operation that must be performed, the matching process can range from being trivial, to being very complex and CPU-intensive.

The description given above is intentionally generic, as different *subscription languages* exist. In fact, the literature distinguishes between different types of publish/subscribe paradigms, the most popular ones being topic-based and content-based publish/subscribe. Some other more specialized flavors have also been proposed. These different flavors are described in the next subsections.

2.1.1 Topic-Based Publish/Subscribe

Topic-based publish/subscribe, sometimes referred to as *channel*-based communication, is perhaps the most popular publish/subscribe paradigm, due to its conceptual simplicity and its potential scalability. It is widely used in the industry, and many open-source [1] or commercial products exist. In topic-based pub/sub, the subscription predicate is a key, usually a string, referred to as the *topic*. Upon publishing, publishers tag their publications with a topic as metadata, so that subscribers subscribing to a given topic receive all publications tagged with this topic [47]. As a shortcut, we generally write that publishers *publish to* a given topic, which means that they issue publications that are tagged with the topic as metadata. Conceptually, the matching process in topic-based pub/sub is simple. Subscriptions can be stored in a hash table, with the topics as keys and the set of subscribers as values (for every topic). When a publication needs to be matched, one simply needs to find its topic in the hash table ($O(1)$ operation) and obtain the corresponding set of subscribers.

Conceptually, topic-based publish/subscribe can be viewed as a specialization of attribute-based publish/subscribe, which is described in the next section. It can also be seen as an evolution of group multicast systems [44], in which clients can join logical *groups* and receive all messages sent towards that particular group.

Applications of Topic-Based Publish/Subscribe Mostly due to its simple data model, topic-based pub-sub is widely used across various application domains, such as traffic alert systems, mobile device notification frameworks (such as Google Cloud Messaging (GCM) used for sending push notifications to Android devices), chat/instant messaging systems, extreme weather alert systems, social networks, and many more.

In the Twitter example described previously, users can express interest in the identifier of other Twitter users (*followers*). Upon publishing, Twitter users then tag their publications with their own identifier, so that followers receive these publications. Facebook can also be viewed as relying on the publish/subscribe paradigm. For instance, in Facebook's instant messaging service (*Messenger*), one can create group conversations. Relevant Facebook participants of a given conversation then subscribe to receive messages from this conversation, and tag their outgoing messages (publications) accordingly (for instance, with an unique identifier representing the conversation) so that messages are delivered properly to relevant subscribers. At a high level, the Facebook friend system itself can also be viewed under a publish/subscribe paradigm: relevant users subscribe to other users that they are interested in (*friends*). Then, upon a given Facebook user publishing, the publication is delivered to all of that person's friends.

Multiplayer online games also constitute a popular application of topic-based publish/subscribe systems. In such applications, topics are used to convey significant amount of information at a high frequency rate. Multiplayer games will serve as our example application throughout this thesis, both to demonstrate the use and publish/subscribe as well as to evaluate the load of such systems, as they provide a convenient, popular and yet easy to understand model. More details about the networking aspect of multiplayer games, as well as how they relate to topic-based publish/subscribe, are discussed in section 2.6.

2.1.2 Content-Based Publish/Subscribe

Content-based publish/subscribe is a generalization of topic-based publish/subscribe in which the matching process is done over the contents of the publications themselves, which yields much more freedom and flexibility [84, 47, 88, 73, 10, 15, 50, 108]. In the most common scenario, publications contain a set of attribute/value pairs, and subscription predicates are expressed as a range over values for a given set of attributes. A publication then matches a subscription if its attribute values satisfy the

subscription predicate.

2.1.2.1 Content-Based vs Attribute-Based

A dichotomy exists in the literature, however, as some authors prefer to use the term *attribute-based* publish/subscribe for subscription languages involving attribute/value pairs, considering the fact that content-based publish/subscribe models can go *beyond* attribute/value pairs [73]. For instance, in [91], publications are in the form of XML files, and subscriptions are expressed as queries over the whole XML contents, thereby allowing for more powerful content-based subscriptions. In any case, and independently of the subscription language, it is always possible that publications consist of metadata and an opaque payload, and subscription queries are expressed over the metadata, which can be arbitrarily complex (topics, attribute-value pairs, semi-structured data, etc.). Alternatively, it can be the case that publications have no metadata, but only payload, and that subscriptions are expressed over the contents of the payload. In regards to the matching itself and the overall architecture of a publish/-subscribe service, however, there is little difference between the two. As a matter of fact, most work in the literature refer to attribute-based systems as content-based, where the attributes are either in the payload or as metadata. Throughout this thesis, we will follow the same line of reasoning and consider attribute-based systems as part of the large family of content-based systems.

2.1.2.2 Typical Content-Based Example: Stock Auctions

The domain of stock auctions maps well to the content-based publish/subscribe paradigm, as this paradigm allows for very fine-grained subscriptions. Under a topic-based paradigm, one would be limited to registering interest in a *string* value only; for instance, a given stock symbol (ex: `MSFT`, for Microsoft), or an index (ex: `NASDAQ`). Of course, multiple subscriptions could be registered for multiple symbols or indices. Subscribers might however be interested in receiving publications matching a more elaborate predicate, such as all stock quotes with a closing value greater than a given amount ($> 100\$$) *and* belonging to the `NASDAQ` index, which is not possible with topic-based pub/sub. Content-based pub/sub fills the shortcomings of topic-based pub/sub by allowing for more fine-grained subscription predicates. As an example, the predicate $\{\text{openingValue} < 50;\ \text{closingValue} > 100;\ \text{index} = \text{NASDAQ}\}$ could be used to subscribe to receive all publications related to stocks of the `NASDAQ` index, with an opening value below 50 and with a closing value above 100. Publication $\{\text{openingValue} = 30; \text{closingValue} = 200; \text{index} = \text{NASDAQ}; \text{city} = \text{Montreal}; \text{symbol} = \text{MSFT}\}$

would then match the predicate and thus, would be delivered to the subscriber(s) who registered this specific subscription.

2.1.2.3 Limitations of Content-Based Publish/Subscribe

While more flexible, an important drawback of content-based publish/subscribe is that it involves a more elaborated matching process. As opposed to topic-based publish/subscribe where one simply needs to lookup a given key in a hash table, one would need to match every publication against all subscriptions. Some optimizations are possible (a whole branch of the literature describe optimizations that can be applied to the matching process of attribute-based and content-based systems), but the matching process is never as simple and as fast as for topic-based pub/sub. For some applications which require very fast matching and dissemination of a large and recurrent amount of publications, such as games, we believe that topic-based pub/sub might yield to better performance; hence this thesis focuses on such systems in the context of games.

2.1.3 Specialized Publish/Subscribe Paradigms

In addition to content-based (attribute-based) and topic-based publish/subscribe, there exists some other, more specialized publish/subscribe paradigms. Spatial pub/sub systems are a special class of pub/sub systems where subscribers subscribe to regions in a virtual space [62]. Such systems are targeted towards geographical or spatial-based systems. GraPS [24] proposes a graph-based publish/-subscribe paradigm, where the application domain is represented as a graph. Subscriptions and publications contain queries that determine a subgraph and a publication matches a subscription if their subgraphs overlap.

2.2 Basic Architecture of Publish/Subscribe Systems

Publish/subscribe primitives (typically *publish*, *subscribe* and *unsubscribe*) are typically exposed by a library that offers a *publish/subscribe interface*. Such a library typically connects to a middleware platform that offers a publish/subscribe service at a low-level. For instance, in our implementation of our Dynamoth platform used throughout this thesis, we used the Jedis Java library which provides topic-based publish/subscribe (among other services). The library connects to an instance of a Redis server which provides the low-level pub/sub service that we use.

Thus, because multiple clients can interact with a pub/sub service, the typical pattern is to provide publish/subscribe as a service that accept client connections (subscribers and publishers) through a client-specific library, as it is the case with Redis. A publish/subscribe service typically performs the following tasks:

- accepting connections from publishers and subscribers;
- receiving and storing subscriptions;
- receiving and matching publications;
- forwarding publications to interested subscribers.

In the case of topic-based publish/subscribe, the matching process is trivial; however, the delivering of publications is the major overhead. For content-based publish/subscribe, however, the matching overhead can be considerable, as the matching process can be more elaborated.

2.3 Scalability of Publish/Subscribe Systems

For simple applications, a given publish/subscribe service might be able to function properly with only one server. However, as the number of subscribers, subscriptions and publications grow, the network often becomes a bottleneck, and scalability beyond one server might be needed. Scaling publish/subscribe systems brings a whole spectrum of challenges. This section presents the main publish/subscribe scalability approaches found in the literature. First, broker-based scalability approaches are presented. Then, scalability approaches that are built on a peer-to-peer paradigm are presented, followed by scalability approaches that are designed for the cloud. A classification of the main publish/subscribe scalability approaches is presented in table 2.1.

2.3.1 Broker-Based Scalability

A common way to scale publish/subscribe systems is to organize the available servers into a mesh-like topology. In the context of pub/sub scalability, this is referred to as broker-based publish/subscribe, and the individual servers are called *brokers*. Typically, brokers play two important roles: (1) they handle subscriptions and publications, like a typical publish/subscribe server and (2) they route messages to/from other brokers and to/from clients (subscribers/publishers). The broker overlay defines how

	Topic-based	Content-based
Broker-based	Dynatops (2013)	Cheung et al. (2010), MEDYM (2005), GEM (2016), MOVE (2012), Kyra (2004)
P2P-based	Scribe (2002), Tera (2007), PolderCast (2012), Tamara (2007), SpiderCast (2007), Chockler (2007)	Most approaches based on Pastry/Chord: PAPaS (2012), Li et al. (2010), PastryStrings (2006), DPS (2006), Sub-2-Sub (2006), Zhang et al. (2015) Tam et al. (2004), Meghdoot (2004), Terpstra et al. (2003), Bianchi et al. (2007), Chand et al. (2005)
Cloud-based	*To the best of our knowledge: no cloud-specific research approaches found in the literature*	BlueDove (2011), StreamHub (2013), E-StreamHub (2014), SEMAS (2014), SREM (2014), Zhang et al. (2010)

Table 2.1: Publish/Subscribe Scalability Approaches

the brokers are interconnected, and to which broker any given client connects to. In some systems, the broker overlay is built on top of a peer-to-peer substrate such as Pastry [89], or Chord [97, 98]. One characteristic of these peer-to-peer systems is that they dynamically build overlays connecting all nodes despite failures or reconfigurations. Each node is guaranteed to be connected to other nodes, and if a message has to be sent from one node to another, routing paths are established so that a message typically do not need to traverse more than a logarithmic number of nodes in the overlay. We will describe the details of the Pastry overlay in section 2.3.2.

2.3.1.1 Topic-Based Broker-Based Scalability

Dynatops Dynatops [110] is a broker-based publish/subscribe system that is built on the peer-to-peer lookup protocol Chord [97, 98] in order to structure it's broker overlay. Dynatops builds a multicast dissemination tree over a broker overlay where publications go from one or several brokers to reach all subscribers. However, the broker nodes are not peer-to-peer nodes despite the use of the Chord peer-to-peer overlay: they are assumed to be dedicated nodes. Dynatops proposes algorithms to group subscribers with similar interests on the same set of brokers. Dynatops is designed to handle scenarios where subscriptions are short-lived. As such, Dynatops features a "reconfiguration manager", which is a centralized component that is in charge of analyzing the pub/sub environment (rate of outgoing publications, change of subscriptions) and adapting the broker overlay to react promptly to changes.

As part of the reconfiguration process, Dynatops aims at minimizing the number of brokers that publications have to go through in order to reduce delivery times. However, for latency-constrained applications, we think that a flat approach such as our Dynamoth platform that we propose at chapter 3 might be more suitable since all communications occur only in two hops (from the client to the server and directly back to clients). Also, Dynatops is not specifically designed for cloud environments and does not support dynamically adding/removing broker nodes; therefore, it's scalability model is limited to reconfiguring the system with the set of pre-allocated broker nodes.

2.3.1.2 Content-Based Broker-Based Scalability

Several non cloud-based approaches for scaling content-based publish/subscribe systems have been proposed in the literature [29, 37, 36, 28, 86]. Although such systems are not cloud-specific, some of them could nevertheless be deployed in a cloud setting.

Load Balancing for Content-Based Systems In [37], Cheung and Jacobsen, based on their previous work [36], propose a load-balancing protocol for broker-based content-based pub/sub systems that is built over the PADRES content-based publish/subscribe system [50] (described in section 2.5). In their system, brokers are organized into clusters featuring edge brokers which subscribers are connected to, as well as cluster-head brokers where publishers send their publications to. Each broker continuously monitors it's own load. If the load becomes too high, then a load balancing session is established with another broker that can accept supplementary load, through a *mediator* component. As part as this load balancing session, the two brokers involved exchange detailed load information for every subscription and determine which subscriptions should be exchanged. The load balancing protocol notably proposes several load offloading algorithms that can address several performance metrics.

MEDYM MEDYM [28] proposes a dynamic broker-based routing overlay for matching and disseminating publications in content-based publish/subscribe systems. Each broker is responsible for a given set of subscriptions, and publications are forwarded between brokers by building dynamic multicast trees until all suitable brokers for a given publication have been reached.

Several additional contributions discuss further the problem of scaling the content matching and filtering processes of broker-based content-based publish/subscribe systems, such as GEM [49], MOVE [86] and Kyra [29].

2.3.2 Peer-to-Peer Scalability

Peer-to-peer approaches are interesting as they do not require the use of a centralized infrastructure, which can lead to cost reductions. In fact, peer-to-peer publish/subscribe approaches distribute the load of the pub/sub service over the users of the service themselves, and not over a centralized infrastructure, as it was the case for the models described at the previous section. Peer-to-peer systems are not a new concept, as many applications relying over this paradigm, such as file sharing, have been proposed over the years.

An advantage of peer-to-peer approaches is that they can somehow be more resilient in case of failure compared to a central infrastructure. The disadvantage of these approaches is that because the overlay is built over a large number of nodes (all users), the network is much larger than if only dedicated brokers are used. As a result, a message will likely traverse many more nodes as it would be the case in an overlay with dedicated brokers. Furthermore, as clients come and leave, the overlay is much more volatile. Thus, this severe drawback can make peer-to-peer-based systems impractical for some applications, such as latency-dependent multiplayer online games.

Several peer-to-peer approaches have been proposed in the literature for scaling publish/subscribe systems . Many of these approaches are built on well-known peer-to-peer substrates such as Pastry [89] or Chord [97, 98]. They give a better idea of the main principles, we discuss the base architecture of Pastry before we introduce the peer-to-peer pub/sub approaches. Chord follows a similar approach to Pastry.

2.3.2.1 Pastry

Pastry [89] is a general-purpose pure[1] peer-to-peer substrate that is designed to support many different applications, most commonly for file systems. Pastry's architecture is based on a ring, where each node is placed on the ring according to a uniquely assigned identifier. Each Pastry node maintains a routing table that contains the IP addresses and unique identifiers of a set of other Pastry nodes. The routing table is organized in such a way that it allows any given node to route a given message towards any other node in the Pastry ring. More specifically, upon a Pastry node receiving a message addressed towards a given identifier, it finds the node in its routing table that is numerically closest to the target

[1] As opposed to hybrid systems, no central entity is required.

identifier, and forwards the message to that node. The process is repeated until the message reaches its intended destination, which is the node that is globally numerically closest to the target identifier. The size of the routing table is defined by a configuration parameter, which in turn defines the bound on the number of hops needed to transmit a given message to any another arbitrary node (larger routing tables allows for delivering in less hops). Typically, the number of hops needed is logarithmic to the number of nodes.

The most common application for such a peer-to-peer overlay is to store files over a distributed set of nodes. Each file is stored in the node whose identifier is closest to the identifier of the file. The request for the file is then redirected through the overlay as described above. Peer-to-peer overlays are designed to be dynamic, so they can be used for a variety of applications.

Upon adding a node, a portion of the load (e.g. set of files) of the two neighbor nodes is transferred to the new Pastry node, as this new node then becomes in charge of the identifiers in the keyspace that are closest to its own identifier. Similarly, upon removing a node, the two neighbor nodes absorb the load of the removed node, since they become responsible for the new portions of the keyspace that are closest to them. The Pastry library notably provides the basic infrastructure for nodes joining/leaving the system, reorganizing the routing tables, and also supports up to a given amount of failing nodes (for instance, by replicating files on neighbor nodes), which can help ensuring consistency.

2.3.2.2 Topic-Based Peer-to-Peer Scalability

Several peer-to-peer topic-based pub/sub scalability approaches have been proposed in the literature [32, 39, 38, 94, 14, 78, 84]. These approaches are summarized in the following paragraphs.

Scribe Scribe [32] implements topic-based publish/subscribe over Pastry and was one of the first attempts at proposing a decentralized multicast overlay architecture. In Scribe, a node acts as a rendez-vous point for a given group (topic) and stores the list of members of the group/topic. A subscriber node creates a "path" towards the rendez-vous node. All nodes perform forwarding of publications to deliver publications in a reverse tree-like manner until all subscribers connected to each node along the path have been reached, following the Pastry topology.

Tera Tera [14] proposes a peer-to-peer two-layered overlay approach for large-scale topic-based publish/subscribe systems. At the lower overlay layer, Tera builds and manages a graph structure that in-

terconnects all nodes. For each topic in the system, an overlay is created at the higher layer, where one node (subscriber) is elected as the leader and is in charge of dispatching messages to all other subscribers. Publications from publishers are routed from the lower layer to the appropriate leader at the higher layer.

PolderCast PolderCast [94] is another scalable peer-to-peer topic-based pub/sub system where all subscribers for a given topic are interconnected using a ring overlay, but with additional random links. Any publication reaching a subscriber can then reach other subscribers in a linear fashion (worst case) or faster using the additional links. PolderCast notably aims at being more reliable than other peer-to-peer approaches such as Scribe that notably depends on single rendez-vous points for topics, which can be points of failures.

Other Approaches While Tamara [78] is not a new topic-based publish/subscribe system per se, it aims at improving the efficiency of such systems by proposing a topic clustering / grouping mechanism. Their model attempts to group topics with a similar set of subscribers together into *topic-clusters* in order to reduce maintenance and dissemination costs. Extensive cost computations are made on a regular basis to determine whether a given topic should be merged with a given topic-cluster and whether two given clusters should be merged together. This cost computation is a bit in the same spirit as our work on Dynamoth although the cost values are more blurry.

SpiderCast [39] and Chockler [38] are further P2P topic-based pub/sub systems that use distributed protocols to optimize the routing overlay.

2.3.2.3 Content-Based Peer-to-Peer Scalability

Several peer-to-peer attempts at providing content-based publish/subscribe can be found in the literature [10, 74, 9, 13, 103, 109, 100, 59, 102, 22, 33]. As this is the case with topic-based systems, many of these systems are built on top of overlay substrates such as Pastry [89] ([100, 9, 74]), Chord [97, 98] ([102, 59]) and CAN [87] ([59, 109]).

2.3.3 Cloud-Based Scalability

Cloud computing is an umbrella term and paradigm which refers to having computing resources (processing power, storage, etc.) and even software stacks *rented* from a provider. The typical pattern is

that third party hosting providers have a large pool of servers in datacenters connected via high speed data links. Customers may wish to rent some *amount* of computing resources from the pool to perform some tasks or to offer a given service to end-users. Instead of having dedicated computing resources, renting computing resources from the pool can often be a cost saver. Scalability is a key concept of cloud computing because customers can dynamically rent resources depending on their needs. Nowadays, many big providers offer a wide range of cloud computing services, such as Google, Amazon, Microsoft, Apple.

As this is the case in many domains, the cloud can be used to efficiently scale publish/subscribe systems. To the best of our knowledge, all research-oriented cloud-specific publish/subscribe scalability approaches that we were able to find were for content-based systems. The following subsections first describe the various cloud service models, and then describe the main cloud-based scalability approaches found in the literature.

2.3.3.1 Cloud Service Models

The literature distinguishes between three cloud computing levels of service: Infrastructure as a Service, Platform as a Service and Software as a Service.

Infrastructure as a Service (IaaS) Under the Infrastructure as a Service (IaaS) model, virtual machines (VMs) are provided to the end user. VMs are deployed on *hypervisors* which are server nodes hosted at the provider's datacenters. VMs-to-hypervisors assignations are managed by the cloud provider using load-balancing algorithms. Mistral [66] is one example of an IaaS platform. Users must manage the software stack and typically have full root access. This is a replacement for dedicated servers.

Platform as a Service (PaaS) Under the Platform as a Service (PaaS) model, the cloud provider provides a computing platform to its customers, which are typically developers, with an already established software stack. The software stack can include databases (relational or non-relational), web servers and engines (Apache, PHP, .net, etc). VMs are abstracted. A key advantage of the PaaS model is that the allocated infrastructure automatically scales to match the user's needs in a typically transparent manner: if more VMs are needed to support a given PaaS service, then more VMs are transparently allocated, since it is generally the responsibility of the PaaS service to auto-scale and reorganize itself whenever required. Non-relational ("NoSQL") databases [99] play a key role under PaaS architectures.

A key example of a PaaS platform is Amazon's Dynamo platform, which is a non-relational replicated key-value store [42, 43]. The various cloud-based publish/subscribe systems that we describe in the next section also fall under the PaaS model. In addition, our contributions to this thesis in terms of providing scalable cloud-based services also fall in this category: Dynamoth (chapter 3), MultiPub (chapter 4) and DynFilter (chapter 5) are all PaaS services that provides topic-based publish/subscribe for given applications.

Software as a Service (SaaS) Under the Software as a Service (SaaS) model, the cloud provider offers high level, ready to use software to the end-users. Such software is accessed through a thin-client or more recently, via a web browser. Popular examples are well-known online email clients such as Gmail or Outlook and online office/collaborative suites such Google Docs, Office 365, etc. Scaling online services offered under the SaaS model ultimately require the scaling of the underlying infrastructure and platform.

2.3.3.2 Consistent Hashing

Consistent hashing is a popular mechanism in cloud-based approaches where the load has to be distributed across several nodes in a dynamic fashion; that is, nodes need to be easily added and removed and partial load needs to be moved from one server to the other.

Under a typical consistent hashing model, multiple virtual identifiers are assigned to each node in the ring of nodes. For instance, in a ring with 100 nodes, with each node having 100 virtual identifiers, then there would be 10,000 identifiers in the ring, mapping to the 100 nodes. Even without considering the addition or removal of nodes, the mapping process in itself of a given key to a virtual identifier, then to a given physical identifier, will then tend to be more uniform, compared to a non-consistent hashing-based approach. For instance, if the nodes are used to store files, they will be more uniformly distributed across the nodes. Another major effect of consistent hashing is that upon adding/removing a node, consistent hashing will allow for the load to be spread more equally towards many nodes, and not only towards the two neighbor nodes.

If used for topic-based publish/subscribe, consistent hashing can be used to map topics to different servers in a naive way; for instance, by determining the virtual identifier that is closest to the name of the topic by some distance measurement, and then assigning the topic to the server that hosts this virtual identifier. Upon the current servers becoming overloaded, additional servers can be spawned,

and a portion of the load (topics) can be transferred from all current servers to the new server. Assuming N servers, upon introducing an additional server, all current servers would then give $1/N$th of their load to the new server. Similar holds when removing servers. Note that part of Dynamoth's contribution (chapter 3) lies in providing a scalability model that allows for more flexibility than consistent hashing, and we compare against consistent hashing in some of our experiments.

2.3.3.3 Content-Based Cloud Scalability

Basically, all cloud-specific research proposals in content-based publish/subscribe fall under the platform as a service model [73, 16, 15, 76, 75, 108].

BlueDove The BlueDove system [73] proposes a brokerless, two-layered scalable content-based (attribute-based) pub/sub system which features multi-dimensional subscriptions over a k-dimensional attribute space. The attribute space for each dimension is split over a set of matching servers. A set of dispatching servers are in charge of forwarding subscriptions and publications to the matching servers. Subscriptions are forwarded to all relevant matching servers responsible for any of the attribute ranges of the subscription (across all dimensions). Publications are forwarded to the most appropriate matching server among any of the possible match. This forwarding mechanism takes load balancing into consideration, so that the load can be split evenly across matchers.

While the mapping of attribute ranges to matchers across all dimensions is done in a *distributed hash table*-like manner, BlueDove goes beyond a pure distributed hash table approach by altering the mapping, in order to adapt to the skewness of the data, where some attribute ranges might more popular.

E-StreamHub Building on their previous work [16], the authors in [15] propose E-StreamHub, a scalable 3-layered hierarchical stream processing model that provides a content-based publish/subscribe platform in the cloud. According to the authors, the stream processing model and the decoupling and specialization of the tasks of the content-based publish/subscribe middleware lead to better performance. Each layer acts as a service that is responsible for a specific task, takes as input either a subscription/publication request (1st layer) or the output of the previous layer (2nd and 3rd layers). The output of the layer is then forwarded to the next layer (1st and 2nd layer) or to one or more subscribers (3rd layer).

Furthermore, each layer contains a set of "slices", which are independent instances of the appropri-

ate service for that slice. The number of slices in each layer can change in order to adapt to variations in measured load (load balancing), by exploiting the scalability of the cloud.

The first layer (partitioning) partitions the subscription space across the different slices of the second layer. Subscriptions are forwarded to the appropriate slice of the second layer, by means of hashing, where publications are forwarded to all slices, so that matching can be done. The second layer (filtering) performs matching of the publications in regards to the local subscriptions. Each slice forwards the resulting set of matching local subscribers to slices of the third layer (dispatching), who dispatch the publications to the subscribers.

System reorganization and migration operations are performed in a reliable way with the help of ZooKeeper [64], which is a middleware that provides reliable data storage and coordination services for highly distributed systems with many concurrent users.

SEMAS SEMAS [76] is a two-layered cloud-based scalable matching service for content-based (attribute-based) publish/subscribe applications. In SEMAS, subscription and publication requests are first sent to dispatching servers, who forwards them to matching servers. In the case of publications, matching servers forward the publication to relevant subscribers.

Like BlueDove [73], SEMAS also considers a k-dimensional attribute space and maps the whole attribute space to different *clusters*. Each "spatial" cluster corresponds to a given matcher node (a given matcher can be in charge of multiple clusters) and is assigned using a consistent hashing technique in order to reduce redistribution overhead.

SEMAS addresses two important challenges: (1) it proposes a mechanism (HPartition) to handle subscription skewness (hot regions of the k-dimensional attribute space) while guaranteeing that each publication can properly be processed by only one matcher. This is accomplished by dividing the more active portions of the subscription space into multiple smaller clusters that can be remapped to different matchers. (2) it provides a mechanism (PDetection) to automatically adjust matching capacity elastically to adapt to variable publication arrival rate, by adding or removing brokers or by modifying cluster-to-broker mappings. Note that the subscription space is still always assigned to matchers using consistent hashing. SEMAS proposes a mechanism to maintain normal processing of publications during reconfiguration by using the "current" configuration until all nodes are ready to use the "new" configuration. The switch happens synchronously using a mutex-based approach. The synchronization

aspect is also an important issue [72] that we address in Dynamoth, in which we propose resilient reconfiguration protocols (section 3.4). In addition, Dynamoth also proposes a mechanism to address subscription and publication skewness in topic-based pub/sub systems (section 3.2.4).

SREM SREM [75] is a broker-based scalable matching service for content-based pub/sub networks. In SREM, all brokers are directly reachable through the Internet and can be located in different datacenters. Publishers and subscribers connect directly to the brokers, and forwarding mechanisms are setup to route publications and subscriptions among the different datacenters and brokers.

SREM builds upon SEMAS and proposes a k-dimensional subscription model also inspired by BlueDove [73]. Brokers are arranged into clusters in a n-dimensional layered tree space. Clusters at the lowest level (level 0) contain all available n_B brokers. At the next level (level 1), two clusters contain half of the brokers ($\frac{n_B}{2}$). At level 2, 4 clusters contain $\frac{n_B}{4}$ brokers. The process is repeated for all n levels. Each cluster maps to a portion of the subscription space. This scheme implies that as we go higher in the hierarchy, clusters are responsible for a reduced portion of the subscription space. At the highest level n, any broker inside a cluster can perform the matching for any publication within that subscription space. Upon receiving a given publication or subscription, SREM describes an approach that starts at the lowest level (level 0) and bubbles up towards clusters at the last level, in order to determine a set of brokers that can handle the matching.

Another contribution of SREM is an improved mechanism to divide hot spots in popular subspaces of the subscription space into smaller subspaces, which is inspired by the HPartition technique of SEMAS. This allows for a more uniform mapping on the higher-level clusters and brokers.

Deploying Non-Cloud Systems in the Cloud In [108], the authors conduct a performance evaluation of running two popular non-cloud-based pub/sub systems based on broker networks - PADRES [50] and OncePubSub [65] - in the cloud using different approaches (black box, grey box and white box). Under the black box approach, resource usage is monitored on the virtual machines (VMs). When certain performance thresholds are reached, workload migration across hosts occurs using live VM migration techniques. However, no new VMs are added nor removed. Under the grey box approach, which has been implemented for PADRES, the workloads are monitored. When it is determined that a given broker will be overloaded, a new replica broker is spawned, and a proxy broker transparently redirects subscriptions and publications across the two brokers, so that the system still thinks that

	Approaches
Reliability, Fault Tolerance and Availability	Yoon et al. (2011), Kazemzadeh et al. (2009), Pubily (2012), GEPS (2016), Chang et al. (2012)
Latency and QoS Optimization	Bellavista et al. (2014), JMS, DDS, PubSubCoord (2015), DCRD (2011), IndiQoS (2005), SES (2014), Setty et al. (2014)
Reducing Monetary Costs	Setty at al. (2014), Heinze et al. (2013)

Table 2.2: Publish/Subscribe Reliability and Optimization Approaches

these two brokers are in fact only one. Finally, the white box approach takes the inner working of the publish/subscribe system into consideration to perform load balancing, and allows for adding and removing VMs. This approach has been implemented with OncePubSub only.

2.4 Reliability and Optimization of Publish/Subscribe Systems

Besides providing scalability, a publish/subscribe service should provide some reliability, fault tolerance and availability properties in order to handle various types of failures that can occur. In addition, some publish/subscribe-based applications might exhibit specific requirements in terms of acceptable latencies or QoS properties, or in terms of reducing monetary costs incurred by the use of the pub/sub service. This section presents an overview of the main publish/subscribe reliability and optimization approaches described in the literature, which are outlined in table 2.2.

2.4.1 Reliability, Fault Tolerance and Availability

Providing fault reliability, fault tolerance and availability brings interesting research questions, especially when it comes to synchronizing many nodes and providing delivery guarantees and proper message ordering despite system failures. A seminal contribution in this area are the Lamport clocks [72], which was one of the first papers to address the problem of synchronizing time in distributed systems.

In the area of fault-tolerant publish/subscribe systems, there have been a few approaches described in the literature [67, 68, 34, 107, 90]. Some of these research questions are also addressed in our

Dynamoth system (chapter 3). This section presents some of the approaches discussed in the literature.

2.4.1.1 Modeling of Delivery Guarantees

[107] proposes a formal modeling of publication delivery and ordering guarantees and appropriate reconfiguration operations for broker-based, content-based pub/sub systems. The model considers two levels of delivery guarantees: (1) Strong Complete Delivery, where all publications from any given publisher must be delivered to all subscribers, and (2) Weak Complete Delivery, where all publications who do not go through a faulty broker must respect the strong complete delivery property. In addition, the model also considers two message ordering guarantees: (1) Strong Ordered Delivery, where all publications from any given publisher must be delivered *in order* to all interested subscribers and (2) Weak Ordered Delivery, where all publications who do not go through a faulty broker must respect the strong ordered delivery property.

These delivery and ordering guarantees proposed in this model bear some similarity with the different levels of guarantees that Dynamoth provides (see section 3). However, the details differ considerably, as the Dynamoth architecture is fundamentally different (topic-based instead of content-based, and flat instead of broker-based). Furthermore, the model presented here assumes that all publications not already forwarded to a faulty broker are delivered, where Dynamoth can guarantee that all publications are transmitted in case of a faulty server (section 3.5).

2.4.1.2 Broker Overlay Reconfiguration

As seen in section 2.3.1, some broker overlay configurations might not be optimal for a given workload, and it might be beneficial to optimize the overlay. In the case [107] presented above, the authors propose a set of reconfiguration operations that can be applied to optimize the overlay, which are chosen based on the desired level of delivery and ordering guarantees.

[67] proposes an approach to reconstruct the broker topology upon broker failure in broker-based content-based publish/subscribe systems. The broker overlay is organized into a mesh-like layout similar to [110] or Scribe-like [32] approaches, where publications are disseminated in a multicast tree-like manner. The system supports up to δ consecutive failing brokers, and upon broker B failing, traffic from neighbor brokers is re-routed in such a way to avoid B. The process is repeated if adjacent brokers are failing. The failure handling protocol however requires all publications to be confirmed by

brokers and relayed to upstream brokers. While duplicates can still happen, they are detected and eliminated. A recovery protocol is also provided in the event where a failed broker is restored, so that the subscriptions can be properly reassigned and the topology restored.

The Publiy system [68] is similar in spirit and also provides reliability and fault tolerance mechanisms by having upstream brokers hold publications until successfully delivered by downstream brokers. It also establishes optimized links to reduce the number of hops for publication delivery.

GEPS [90] proposes an approach to reconstruct the topology of content-based pub/sub systems built on a tree topology following the failure of one or more brokers. This approach also ensures that no messages are lost, and that messages are still delivered during failure recovery in order to minimize latencies.

2.4.1.3 Byzantine Faults

In [34], the authors propose a set of protocols to address various Byzantine faults that can occur in cloud-based pub/sub systems, notably for subscribers, publishers and brokers. For instance, a faulty broker might delay the delivery of some publications, reorder them, corrupt subscriptions or publications. A publisher could send irrelevant publications or flood with a large amount of publications, where a subscriber might perform a large amount of unnecessary subscriptions to generate more traffic.

2.4.2 Latency and QoS Optimization

Some applications relying on a publish/subscribe paradigm require meeting strict latency bounds. A popular example of such applications are multiplayer online games, in which the gameplay quality can be severely degraded if publications are not delivered within a reasonable delay (section 2.6). In some cases, it is necessary to bound the delivery times of publications, which means that steps have to be taken to guarantee that such publications are delivered within a maximal time frame. In order to support latency-constrained applications, the publish/subscribe middleware must be designed to take the latency needs of such applications into consideration.

As this is the case in many other domains, the latency requirements of pub/sub-based applications can also be expressed in terms of *quality of service* (QoS) requirements. As such, some QoS-aware publish/subscribe systems have been proposed in the literature, which provide various QoS guarantees.

2.4 Reliability and Optimization of Publish/Subscribe Systems

[19] surveys several QoS-based approaches in the context of wide-scale pub/sub systems. Besides bounded delivery times, several QoS properties can be specified, such as guaranteed delivery, reliability and uniqueness. In some QoS-based systems, an expiration time can be set for publications, and all expired publications are rejected by subscribers [8]. A major drawback of many QoS-based approaches is that they require a network which supports QoS properties. While LANs and WANs can be designed to support QoS guarantees, this is generally not the case for Internet communications, which operates on a best-effort basis.

Our Dynamoth (chapter 3) also aim at reducing message delivery times by employing several mechanisms. Our MultiPub contribution (section 4) goes further in that direction by allowing one to impose delivery time constraints on topics, in a cloud-based global-scale setting. It is important to mention, however, that our systems still operate on a best-effort basis, due to the aforementioned lack of QoS on Internet-scale communications.

The following subsections highlights some of the main contributions in the area of optimizing latency and providing QoS guarantees in the context of publish/subscribe systems.

Data Distribution Service (DDS) The Data Distribution Service (DDS) from the OMG group [6] proposes the specification of a rich software architecture for QoS-constrained data delivery schemes, with an emphasis on pub/sub applications. The architecture provides support for both topic-based and content-based publish/subscribe, and specifies several QoS properties that an implementation of the standard can follow. However, since DDS is only a specification, the expected performance is completely dependent on specific implementations of the standard, as well as the QoS properties that implementers choose to follow. In addition, while DDS implementations can be cloud-based, the standard itself is not specifically tailored for cloud environments.

PubSubCoord The PubSubCoord system [11, 12] provides a multi-layered, broker-based cloud coordination system for WAN-scale topic-based publish/subscribe systems. An instance of a publish/-subscribe service is deployed in each local network. A set of cloud-based routing brokers interconnect all local publish/subscribe services to provide a bridge allowing publishers and subscribers in different networks to be interconnected. Thus, for publishers and subscribers in the same network, the local publish/subscribe service is directly used (one hop), while for publishers/subscribers in different networks, one or more routing brokers have to be used (two or more hops).

Since the routing layer is cloud-based, it can horizontally scale in terms of the amount of brokers needed, although the scaling mechanism is not precisely described. Similarly, vertical scaling is also provided by properly mapping topics to brokers (load balancing) and building an efficient inter-broker overlay.

PubSubCoord also provides some degree of fault tolerance among routing brokers using ZooKeeper [64] to assist in performing coordination. However, fault tolerance is not provided for local publish/-subscribe systems. The authors claim that the performance in terms of delivery times in wide-area networks can be impacted by two main factors: (1) the network congestion, which they claim can be alleviated by the cloud scalability and an efficient mapping of topics to brokers and (2) by the inter-broker forwarding mechanism. For latency-critical applications, the different local publish/subscribe systems can be interconnected, which results in one less hop.

Local publish/subscribe systems use the Data Distribution Service (DDS) specification, and some of the QoS policies of the DDS standard are used to express the QoS, latency and fault tolerance needs of PubSubCoord-based applications.

DCRC DCRD [58] is a routing algorithm for broker-based topic-based pub/sub systems which selects the most optimal path towards recipients, in order to meet QoS requirements. Publications are dynamically forwarded from the publisher to the subscriber through a series of interconnected brokers. Subscribers can specify a delay requirement with their subscriptions. Each broker dynamically decides to which broker the publication should be forwarded next by taking into consideration the delay requirements, inter-broker latencies as well as the list of failed brokers. Since brokers must confirm the transmission of each publication to the upstream broker, faulty brokers can be detected. Therefore, DCRD can guarantee the transmission of each publication even in the presence of faulty brokers and thus also provides some fault tolerance. Changes to the broker topology are also propagated between brokers.

IndiQoS [30] is yet another middleware that proposes QoS guarantees, in broker-based publish-subscribe systems. However, it specifically requires a QoS-enabled network, which makes it impractical for Internet and/or cloud deployments. In [61], the authors propose SES, a flat scalable pub/sub-based event dissemination platform built on the XMPP protocol. SES notably monitors the QoS in terms of publication latency and attempts to minimize such latencies. In [92], the authors propose a model that aims at maximizing topic-based pub/sub subscribers' satisfaction in resource-constrained

environments.

2.4.3 Reducing Monetary Costs

Some publish/subscribe applications generate massive amounts of messages, which in turn generate significant network traffic. Section 2.3 already discussed the challenges of load balancing pub/sub systems over several machines. Another problem that stems from high data usage is that in a cloud setting, network usage translates to costs, as outgoing data is typically billed based on the amount used. For data-hungry applications deployed in a cloud setting, costs can become very expensive.

In addition to having strict latency needs, multiplayer games are also good examples of data-hungry applications, as they generate a large amount of messages at a very frequent interval. To the best of our knowledge, very few approaches [93, 60] covered the aspect of reducing cloud-related costs incurred by publish/subscribe systems, which is an important contribution that our MultiPub system (discussed at chapter 4) provides and, indirectly, our DynFilter system (discussed at chapter 5). The two relevant approaches that we found are discussed below.

Setty and al. [93] propose a model for efficient resource allocation in the context of topic-based pub/sub systems, in order to reduce cloud-related costs and satisfy subscribers' interest. Their approach notably takes into consideration the VM-related costs, the bandwidth costs and the outgoing bandwidth capacity of the cloud VMs in order to perform efficient load balancing. The optimization criteria is the total costs. Evaluations have been run with a simulation approach using traces from Twitter and Spotify. Similar to them, MultiPub also solves an optimization problem, but our constraints consider message delivery time, and we consider a global-scale deployment of our service in many cloud regions. Furthermore, MultiPub is implemented as a prototype and evaluated in real cloud setting.

[60] aims at measuring the cloud-based costs incurred of running stream processing applications in a cloud setting. Their approach notably takes into consideration the costs of multiple cloud providers.

2.5 Popular Commercial and Open-Source Publish/Subscribe

While we are aware of very few research papers that present cloud-based topic-based pub/sub scalability, several enterprises provide large-scale pub/sub systems in the cloud as a service. Google offers

a scalable pub/sub system that delivers messages at low latency with guaranteed delivery [7], for all kinds of applications. Google Cloud Messaging [4] is Google's topic-based pub/sub infrastructure that allows services to send push notifications to Android devices through the cloud. Amazon SNS [2] is another example of topic-based pub/sub service for push notifications. Microsoft Azure [27] is Microsoft's Cloud platform that offers a variety of services including a topic-based publish/subscribe interface.

Apache Hedwig [3] and Kafka [71] are two popular publish/subscribe open source systems which offer some form of scalability by allowing for manual addition and removal of nodes. However, they cannot be qualified as elastic because automatic addition/removal of nodes based on measured load is not done. PADRES [50] is the well-known University of Toronto's scalable content-based publish/subscribe platform developed for research purposes that is used across a wide range of projects.

2.6 Multiplayer Games as a Publish/Subscribe System

Multiplayer games constitute good applications of topic-based publish/subscribe, since the richness of this paradigm and the logical decoupling of content producers from content subscribers map well to the game semantics. In addition, games bring several scalability challenges, as they often generate large amounts of publications at a very frequent interval, and they typically exhibit strict latency needs. For these considerations, we decided to use game applications throughout this thesis for illustration and demonstration purposes.

In the context of a multiplayer or massive multiplayer online game, players are located in the same virtual world space. In order for the game to operate properly, players typically have to receive game state update messages at a frequent interval [40, 104, 21, 80, 82] from other players and other relevant in-game entities. Generally, upon receiving a state update, the game client usually performs some kind of game-dependent processing and updates the graphical display.

2.6.1 Latency Requirements of Multiplayer Games

The dissemination infrastructure of multiplayer games must ensure that all relevant state update messages are delivered in a timely fashion in order to meet the strict requirements of the various types of games, while coping with the highly variable bandwidth needs of such games.

Due to their fast-paced nature, first person shooter (FPS) games typically require up to 20 game states updates per second, which corresponds to update messages every 50 ms. Studies have shown that the gameplay quality degrades significantly [40, 104, 80, 18] when latencies go beyond 150ms. On the other end, massive multiplayer online games (MMOG) have more relaxed requirements, accepting latencies of 150-200ms and even more, depending on the nature of the game.

2.6.2 Bandwidth Needs of Multiplayer Games

The amount of players in a given game at any given time is not constant: players are more likely to be playing at certain periods of the day or of the week [101]. Also, because massive multiplayer online games (MMOGs) feature large-scale virtual worlds, players in such games often exhibit flocking behavior [85, 35], where a large amount of players gather towards the same popular locations on the map (towns, popular quests, etc.). Flocking can draw a lot of outgoing bandwidth from the servers since the number of messages that need to be transmitted within the flocking area increases in a near-quadratic way. If not handled properly, it can cause a game to collapse[2]. Over the years, there has been extensive attempts in the literature at proposing scalable multiplayer game architectures [56, 79, 70, 45]. [35] notably attempts to mitigate the flocking phenomenon in MMOGs. Thus, because of those phenomena, bandwidth use within a game is subject to variation over time.

2.6.2.1 Adjusting the Contents and the Frequency of Update Messages

In the simplest case, players receive state updates from all other players and virtual objects. Assuming that every player generates messages at a regular interval, such a scheme does not scale well, as the number of messages delivered grows $O(n^2)$ to the number of players in the game. As each message delivered generates bandwidth, servers can quickly become overloaded.

In the same research orientation as our DynFilter work, presented in chapter 5, some approaches have been proposed that aim in reducing bandwidth usage in games [104, 21, 106]. Two seminal approaches are WatchMen [104] and Donnybrook [21], which propose mechanisms to reduce bandwidth use in peer-to-peer-based games by reducing the rate at which updates are delivered for players located outside of other player's vision range. More specifically, full updates are sent only to the *k*-most in-

[2]http://www.gamespot.com/articles/blizzard-addresses-warlords-of-draenor-server-prob/1100-6423584/ [Aug 18, 2015]

terested players within a given player's vision range. Players located in the vicinity but not in the list of k-most interested players receive different, less frequent messages, that contain data used to aid in performing dead reckoning which are techniques that are used to approximate the position of players or other objects in presence of high latency or infrequent updates [111, 77, 83, 69]. Finally, players located very far receive sporadic state update messages. Thus, these approaches do not only vary the frequency of message delivery, but also the contents of state updates. In contrast, our proposed system, DynFilter, automatically adjusts the update frequency depending on the number of players in the vicinity. It is also worth mentioning that DynFilter is cloud-based, unlike these two approaches that are peer-to-peer-based.

2.6.2.2 Interest Management: Limiting Game-Related Messages

As explained at the previous section, it might not be necessary for all players to receive state updates regarding all other players and in-game entities. At a high level, interest management techniques are used to restrict the information that any given client/player p will receive, and make sure that p will not receive state updates that are not relevant based on the current state / context of p. Most interest management techniques exploit the topology of the map to determine in which clients and in-world objects any given client p might be interested in (area of interest - [20]). The various interest management approaches fall into two main categories: *object-based* interest management and *tile-based* interest management.

Object-Based Interest Management In object-based interest management, p registers interest in all entities located within a fixed radius of p. This approach typically involves a centralized interest manager component [70] continually tracking and recomputing the interest sets for all players, which might be a more CPU-intensive process. However, it offers high granularity.

Tile-based Interest Management In tile-based interest management, the virtual world is split into a set of interconnected tiles (usually triangular, but other shapes are also possibles such as squares or hexagons), and p registers interest in all tiles (partially) located within a certain radius of p. The interest management process is simplified since the interest manager only has to keep track of the tile in which every player (or other relevant in-game entity) is located.

The two approaches described at the previous section (Watchmen and Donnybrooke) were exam-

ples of object-based interest management. On the other hand, the Mammoth system [70, 45] proposes a hybrid approach where players first express interest in tiles, which are then used to communicate a list of entities that the player will individually subscribe to. Some approaches also take into account the topology of the map to make sure that a given client cannot be interested in an area which would be invisible to him (for instance, an area hidden by a wall or inside/outside a house) [70, 23].

2.6.2.3 Using Topic-Based Publish/Subscribe to Model Interest Management in Games

The publish/subscribe paradigm can be used to express interest relationships between the various in-game entities [70, 80]. More specifically, the topic-based publish/subscribe paradigm can be used to model interest subscriptions and message dissemination for the two approaches mentioned in the previous section.

Object-based Interest Management In the case of object-based interest management, a topic name (string) is generated for every player p and every relevant entity (e.g., `Player202` or `Container5032`). p receives a list of all players/entities in its area of interest and establishes individual subscriptions to all relevant object-topics. Upon being notified of objects to be added or removed from p's interest set, p invokes proper subscriptions and unsubscriptions. p publishes its state updates to its own player topic.

Tile-based Interest Management In tile-based interest management, a topic name (string) is generated for every virtual tile (e.g., `Tile23`). p automatically determines in which tile it is located, or gets this information from the infrastructure. p then automatically subscribes to the topic corresponding to the tile in which it is located, and publishes its own state updates on the same topic. The problem arises where p might be located near the edge of two or more tiles. p should then subscribe not only to one tile, but to a set of tiles located around p.

It is worth noting, however, that other pub/sub paradigms than topic-based publish/subscribe can be used to model interest management in games. Notably, in [26], the authors evaluate and compare how different pub/sub architectures (topic-based and content-based) can be used in the context of MMOGs to efficiently process message delivery as well as some other game-related tasks such as interest management. On the other end, [63] and [80] are two systems that make use of spatial publish-subscribe to perform interest management in games, as well as to scale such games. The various contributions to this thesis (Dynamoth, MultiPub and DynFilter) adopt the tile-based approach as it was conceptually

simpler from an implementation and experimental standpoint.

2.6.3 Cloud Gaming

Cloud gaming [96, 41, 95] has been a hot research topic in the last few years. As an alternative to providing dedicated cloud-based services for games, cloud gaming involves running whole games (clients and servers) in the cloud: game players play using thin clients that stream a live video of the game and transmit user input back to the server. Cloud gaming brings interesting properties such as having all game-related state updates sent over an internal network (clients and servers in the same cloud), thus potentially leading to less finer-grained interest management, an increased security and the ability to play on multiple devices with minimal porting efforts. However, there are also drawbacks such as high cloud bandwidth use (which might be very expensive) and high client bandwidth use, which might be limited and might be very costly on mobile data plans, despite using degradation techniques [41]. Latencies will also be higher [96], which might be problematic for latency-sensitive games such as FPS.

3

Dynamoth: Scalable and Available Publish/Subscribe

Dynamoth [53, 52] is a scalable, elastic cloud-based topic-based middleware that supports any number of publishers, subscribers and publications. Any application making use of general purpose topic-based publish/subscribe is supported, with a specific emphasis on latency-constrained applications such as multiplayer and massive multiplayer online games.

The Dynamoth approach differs in several aspects from the various scalable topic-based publish/-subscribe systems found in the literature (section 2.3). Notably, such approaches were either peer-to-peer-based or broker-based, and not really designed for the cloud (although some cloud-based approaches were presented for scaling content-based publish/subscribe systems, described in section 2.3.3). As mentioned before, to the best of our knowledge, no cloud-specific topic-based publish/subscribe scalability approaches were proposed in the literature. In this regard, we think that Dynamoth and its numerous characteristics, such as its flat publish/subscribe approach, as well as its scalability and fault tolerance models, bring interesting novel contributions in the area of scaling such systems.

3.1 Dynamoth's Main Contributions

Dynamoth's main contribution lies in providing a transparent horizontal and vertical cloud scalability mechanism in the context of topic-based publish/subscribe systems by proposing a dual-layer load balancer that operates at two hierarchical levels: at the system level (macro load balancing) and at the

topic-level (micro load balancing).

At the *system-level,* Dynamoth proposes a dynamic mechanism to distribute the responsibility for individual topics across multiple publish/subscribe servers. Whenever the popularity of some topics changes, new topics are introduced or topics are removed, Dynamoth dynamically adjusts the load on individual servers. Furthermore, when the total load increases or drops significantly, Dynamoth automatically adds or removes publish/subscribe servers by spawning/despawing nodes in the cloud to optimize cloud infrastructure-associated costs. At the *topic-level*, Dynamoth is capable of handling cases where specific topics have extremely high load, possibly orders of magnitude larger than other topics. Such situations can happen if a topic has a very large number of publishers, subscribers and/or publications.

In addition to Dynamoth's scalability properties, Dynamoth also provides high availability and fault tolerance using an efficient failure detection and publication recovery mechanism, as a second major contribution.

In summary, the Dynamoth platform provides the following contributions:

- We provide a scalable topic-based publish/subscribe infrastructure where topics are distributed across many publish/subscribe servers. Clients are made aware of the topic assignments so that they can send their publications and subscriptions to the correct publish/subscribe servers, leading to low latency as no indirections occur.
- Our approach provides load-balancing and elasticity at the system-level. Topic assignments can change, and servers can be added or removed from the configuration on the fly as the workload patterns change. Reconfigurations do not interrupt message processing, and messages are guaranteed to be received by all subscribers despite the reconfiguration.
- Our approach provides load-balancing and elasticity at the topic-level. Highly-loaded topics can be replicated across several publish/subscribe servers in order to avoid overload or overly high response times.
- Dynamoth also provides performance-driven availability and fault tolerance by proposing a transparent, efficient failure detection and recovery mechanism. Different guarantee levels are supported for every topic. Notably, Dynamoth can guarantee that all publications over a given topic are delivered in FIFO order in the event of server failure.

- We have implemented Dynamoth on top of an existing open-source publish/subscribe system, namely Redis, without any changes to Redis itself. Thus, we can take advantage of an already existing, highly-optimized publish/subscribe system. We believe that the concepts presented in this paper could be implemented on top of other publish/subscribe systems, as long as they offer the standard publish/subscribe interface: `subscribe`, `unsubscribe` and `publish` operations.

We have evaluated the performance of Dynamoth by conducting extensive experiments over a massive multiplayer game application instead of simulations which can produce less reliable results in the context of multiplayer games [46]. Our results indicate that Dynamoth performs significantly better than a consistent hashing-based approach (section 2.3.3.2), and that it is able to adapt quickly to complex workloads that continuously change. Notably, our experiments on this large-scale game application revealed the following:

- Dynamoth is able to handle 60% more simultaneously active players with the same set of publish/subscribe servers than the consistent hashing approach.
- Dynamoth is properly able to handle large-scale workloads that are subject to high variation over time, while minimizing the number of required publish/subscribe servers, and keeping average latency low.
- Dynamoth is able to promptly detect and recover from server failures, and ensure that all potentially missed publications are retransmitted without hindering the performance of the publish/subscribe infrastructure. Furthermore, the performance of the different guarantee levels that Dynamoth provides is compared.

3.2 System Model

In the following subsections, we describe the main highlights of the Dynamoth architecture. The software implementation of Dynamoth is described thoroughly in section 6.1.

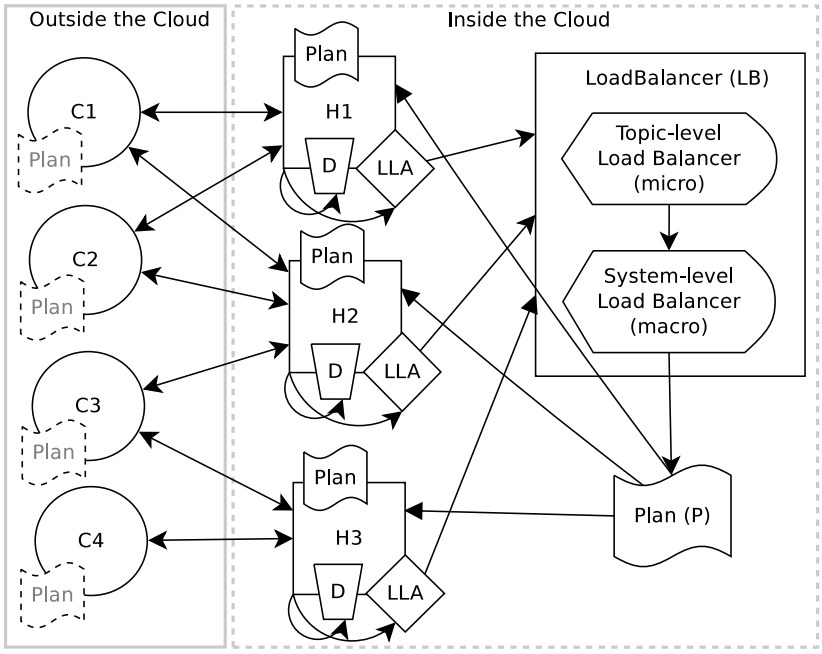


Figure 3.1: Dynamoth Architecture

3.2.1 Naming Conventions

Throughout this thesis, we usually refer to P as a publisher, S as a subscriber and T as a topic within a publish/subscribe system. A publication message is denoted as M. A client to the publish/subscribe system is denoted as C. A given client C can be a publisher to a set of topics and a subscriber to a set of topics. The two set of topics can overlap, which means that a given client can be at the same time a publisher and a subscriber to a common set of topics. In the case of multi-server publish/subscribe systems, H (host) refers to a publish/subscribe server.

3.2.2 Architecture

The Dynamoth architecture is depicted in figure 3.1. The core of the system is a set of standard, independent publish/subscribe servers (`H1` to `H3` in the figure) that handle message dissemination between all clients. In our implementation, each server represents an instance of the Redis [1] pub/sub server software, but it should be simple to replace Redis by any other publish/subscribe middleware with a standard API.

In our approach, we deploy on each node in the cloud a standard publish/subscribe server and two further components, a *local load analyzer* (LLA) coupled with a *dispatcher* (D). The local load analyzer performs real-time monitoring of the load on the pub/sub server. The dispatcher module is needed during system reconfiguration to guarantee that messages are forwarded to all subscribers.

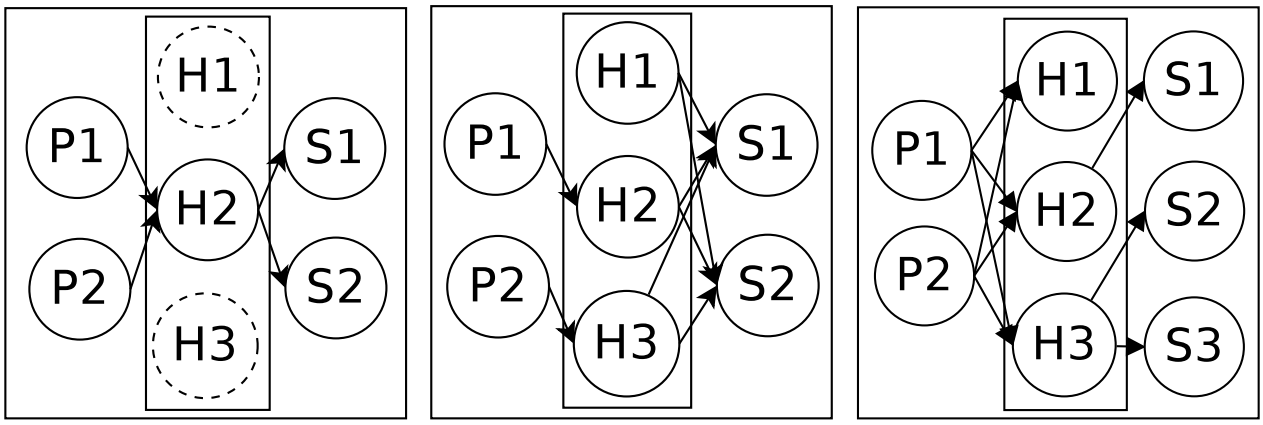
There exists one load balancer node in the cluster that gathers input from all the local load analyzers and aggregates all the metrics. It determines if a configuration change is needed (e.g. whether some publish/subscribe server is overloaded). If this is the case, it determines how to balance the load in the system. To this aim, Dynamoth proposes the concept of a *plan,* which is used to resolve to which publish/subscribe server a given publication or subscription should be sent to. The plan is a more elaborate version of a lookup table where the keys are the topics and the values are the list of servers that should be used for each topic. Whenever a new plan is generated, it is propagated to the dispatchers located on the publish/subscribe server nodes. The dispatchers need this plan to ensure that all messages are forwarded to all subscribers during reconfiguration.

Clients interact with the system through the Dynamoth client library. The client library exposes a standard publish/subscribe API. In our implementation, it corresponds to the original Redis API. The Dynamoth client library uses a client-specific plan to determine to which of the publish/subscribe servers to send publications and subscriptions to. The actual sending of messages is done using the standard Redis client library.

3.2.3 Delivery and Ordering Guarantees

In the absence of failure, Dynamoth always provides reliable delivery for all topics. That is, the messages sent by a publisher on a given topic are received by all subscribers of this topic unless a component (publisher/subscriber/server) fails. Furthermore, it provides FIFO ordering, that is, the messages of an individual publisher to a specific topic are delivered to all subscribers in the order they were sent.

In case of server failures, Dynamoth offers a variety of guarantees. For every topic, a specific level of reliability can be set. If subscribers for a given topic can tolerate lost messages in the unlikely event of a server failure, a more efficient failure handling mechanism can be set for that topic. In the case where subscribers must receive all publications on that topic despite server failures, a more rigid yet less efficient failure handling mechanism can be used. Similar holds for ordering: one can define for any given topic whether, in case of failures, FIFO order must be maintained or a more efficient



(a) No Replication (b) All-Subscribers Replication (c) All-Publishers Replication

Figure 3.2: Topic Replication Strategies

best-effort ordering can be used.

As long as there are no failures or reconfigurations, messages are delivered exactly once in FIFO order simply because we use TCP as underlying communication layer. However, care has to be taken when the system configuration changes or failures occur. To handle these situations, we tag all messages sent by publishers with the identifier of the publisher (sender) and a sequence number. Thus, the combination of identifier and sequence number provides a unique message identifier, and the sequence numbers will help us deliver messages in FIFO order in case of configuration changes. We will give more details as we discuss the different aspects of Dynamoth.

3.2.4 Mapping Topics to Publish/Subscribe Servers

Since Dynamoth is brokerless (direct link from publishers and subscribers to pub/sub servers) and makes use of multiple servers in the cloud, we need an efficient way of determining to which server any given topic should map to. As such, Dynamoth supports three approaches of how to assign topics to publish/subscribe servers (see figure 3.2). In the figure, we consider a topic T, a set of publishers (P-nodes) that will be publishing to T, a set of subscribers (S-nodes) that will be receiving publications flowing through T (subscribers of T) and a set of publish/subscribe servers (H-nodes) that will be used to route publications from the publishers to the subscribers.

In most cases, a topic T is assigned to one publish/subscribe server *H*, and clients send subscription requests and publications for topic T to H (figure 3.2a). This single-server mapping will work for most topics. However, in some scenarios, the number of subscribers, publishers and/or publications on a given topic T might be too large for only one publish/subscribe server. For instance, if a given topic T has a very large number of subscribers, then this might lead to too many simultaneous connections on the publish/subscribe server. This can also lead to important increases in the message processing delay since the publish/subscribe server has to send the same messages to all subscribers at the same time. If T has too many publications, then this could cause too many messages to flow on T, thus potentially overloading the publish/subscribe server (even if the server is in charge of that sole topic), or overflowing individual connections between the publish/subscribe server and a subscriber.

To address the requirements of such heavy-loaded topics, Dynamoth performs load-balancing at the topic-level by allowing for more than one publish/subscribe server to map to any given topic. We refer to this as topic *replication*. Dynamoth proposes two topic replication approaches depending on the overload situation. In both approaches, several publish/subscribe servers are responsible for the topic, but they differ in the way subscribers subscribe to the topic and how publishers publish their messages. In both cases, it is important that all subscribers receive all publications regardless of which publish/subscribe server has been used to process the publication.

3.2.4.1 All-Subscribers Replication

With all-subscribers replication, all subscribers send their *subscription requests* for T *to all* the publish/subscribe servers responsible for T while *publishers* send their message for T *to only one* of the servers responsible for T. In figure 3.2b, the three servers H_1, H_2 and H_3 can be used to process publications flowing through T; each publisher sends its publications through a random server (in this case, P_1 publishes to H_2 and P_2 publishes to H_3) and all subscribers have subscriptions to T on all servers H_1, H_2 and H_3 thus making sure that whichever server is used, all subscribers receive all publications. This replication scheme is relevant if there is a very high number of publishers and/or messages to be transmitted on T but the number of subscribers is within the limit of what a single publish/subscribe server can handle in terms of connections. An example from gaming could be a topic which is used by clients to send position updates within a tile of a virtual world. The publishers are the players that control avatars located in the tile, publishing position and state updates at a high frequency. The subscribers are the game servers responsible, for example, to perform interest management for the tile. With all-

subscribers replication, a player chooses a random publish/subscribe server for its publications, thus distributing the high message load over several servers.

3.2.4.2 All-Publishers Replication

With all-publishers replication, a subscriber *subscribes to only one* of the publish/subscribe servers in charge of T, while *publishers* send their publications *to all* servers. In figure 3.2c, once again all three servers H_1, H_2 and H_3 can be used to process publications flowing through T. Each publisher sends its publications to all servers, which requires sending n messages (n being the number of publish/subscribe servers that handle T), while each subscriber subscribes to T on one specific publish/subscribe server, and thus, receives the message once. This replication scheme is relevant if there is a very high number of subscribers for topic T, but a relatively low number of messages, because each publication message is sent n times. An example from gaming could be some form of broadcast topic through which a game server communicates some world-wide events that are of interest to all players. Without replication, a single publish/subscribe server might take a long time to disseminate such a publication to all subscribers, violating response time requirements. In contrast, when the message is sent to many publish/subscribe servers, the publish/subscribe servers can forward the message in parallel to the many subscribers.

3.2.4.3 Message Ordering with Replication

While Dynamoth is in a stable state (no reconfiguration), all messages of a publisher P to T are received in sending order without any extra precaution. When there is only one server H for a topic T, this is trivially true, as we use TCP connections between P and H, and H and all subscribers of T, and H processes the messages in the order it receives them from P. With all-subscribers replication, P chooses a random server H_i among all available servers for T, and then sends all its publications to H_i. Thus, all subscribers receive all publications from P in FIFO ordering; thus the FIFO ordering guarantee is preserved for any given publisher. For all-publishers replication, a subscriber, as it is only connected to one of the servers H_i, will receive all the messages from H_i, and H_i handles the publisher's messages in FIFO order.

3.2.5 Bootstrapping and Initial Conditions

Initially, a Dynamoth system contains a set of one or more publish/subscribe servers and an initial global plan ("plan 0") which does not provide any specific topic mapping. When a plan does not contain mapping information for a topic (at startup and whenever new topics are dynamically created), it uses a consistent hashing algorithm to map the topic to a publish/subscribe server. This allows for some form of initial form of load distribution among all servers. Over time, the plan is updated when topics are assigned to publish/subscribe servers because of load balancing and topic replication. All dispatchers on the publish/subscribe server nodes always have an up-to-date copy of the complete current global plan.

3.2.5.1 Initial Consistent Hashing

By default and when a given topic T doesn't need to be included as part of a load balancing operation, the publish/subscribe server responsible for topic T is determined through consistent hashing. Let $H_{1,}H_{2,}..., H_{N_H}$ be the set of all available publish/subscribe servers, and let $V_{1,}V_{2,}...V_{N_V}$ be a set of virtual identifiers. Each server is assigned a set of these virtual identifiers. Typically, N_V is considerably larger than N_H, i.e., each server is assigned a large number of virtual identifiers. Let $\mathrm{map}(V_i)$ be a function that, given a virtual identifier V_i as input, returns a server. The $\mathrm{map}(V_i)$ function uses a consistent hashing algorithm so that the removal of one server leads only to the remapping of the virtual identifiers that were assigned to that server, and when adding a server, an equal number of identifiers assigned to each of the old servers gets reassigned to the new server. That is, a remapping only remaps $\frac{V_H}{N_H}$ identifiers. It is also assumed that $\mathrm{map}(V_i)$ is deterministic.

Algorithm 3.1 explains the mechanism by which the default server H_d^T is selected for topic T considering N_V virtual identifiers. First, the identifier of the topic is hashed using a common hashing function so that no two topics share the same integer hash value (T_{hash}). Then, an appropriate virtual identifier T_V is selected (modulo operation). Finally, we use the map function to determine to which server T_V maps to.

function SelectDefaultServer (N_V, T)
begin
  $T_{\text{hash}} \leftarrow \text{hash}(c)$;
  $T_V \leftarrow T_{\text{hash}} \bmod N_V$;
  $T_d \leftarrow \text{map}(T_V)$;
  return T_d;
end

Algorithm 3.1: Selecting Default Server

3.2.5.2 Maintenance of a Client's Local Plan

Each client C maintains a client-specific partial plan $P(C)$, which we refer to as the *local plan.* The local plan is a partial view of the global plan which is known by the load balancer, local load analyzers and dispatchers. At connection time, the local plan is empty and C uses the consistent hashing mechanism described above (section 3.2.5.1) to determine to which server subscriptions and publications for a given topic T should be sent. If the local plan is out of date and as a result an incorrect server is chosen, the server ensures that the subscription/publication reaches the correct server. Furthermore, it informs the client about the correct publish/subscribe server(s) for topic T. Thus, over time, C updates its plan $P(C)$ with the correct topic/server assignments gradually as it becomes more aware of the assignments of various topics.

Similarly, whenever the global plan changes, C is informed in this lazy manner. A consequence of the local plan mechanism is that at any time $P(C)$ only contains information about topics that the client actually uses. Assuming that in large-scale settings, each client only interacts with a small subset of all topics, this approach keeps the plan information at the client side as small as possible. Minimizing the local plan size also enables the middleware to support multiple applications concurrently (in a gaming context, that could be many independent instances of a multiplayer game). The exact mechanisms in which client plans are updated over time and during reconfiguration are described in section 3.3.2.

3.3 Load Monitoring and Plan Generation

To perform load balancing and create new plans, Dynamoth must know the current load on all publish/-subscribe servers. Moreover, it must estimate as precisely as possible the load distribution that will be obtained after rebalancing occurs based on the current load. Our framework is able to accurately mon-

itor and measure the load for every topic, on every publish/subscribe server, with minimal overhead, without the need to alter the pub/sub server software (Redis in our case).

3.3.1 Load Monitoring: Local Load Analyzers

To enable load monitoring, each node that runs a publish/subscribe server also runs a *local load analyzer (LLA)*. The role of the LLA is to continuously gather extensive load metrics for every topic managed by the publish/subscribe server. The recorded metrics for every time unit t (t is one second in our experiments) include the number and list of publishers, the number of publications, the number and list of subscribers, the number and size of sent messages, and the incoming and outgoing number of bytes transmitted.

The LLA is notified when the publish/subscribe server receives new subscriptions and unsubscriptions. This allows the LLA to discover new topics and keep track of the subscribers. In order to collect all metrics, the LLA registers as an "observer" to every topic hosted onto the local publish/subscribe server, and therefore receives a copy of every publication. The fact that the LLA runs locally on the same machine as the publish/subscribe server greatly reduces communication overhead and does not use any local bandwidth. Our empirical observations showed that: (1) running the LLA module had very limited CPU overhead and (2) the outgoing bandwidth of the publish/subscribe servers got saturated much more quickly than the CPU. This second observation can be explained by the fact that most publications will be sent to many subscribers. Therefore, our rebalancing algorithm doesn't take CPU load and incoming bandwidth into account since through our experiments, they were not a limiting factor, except for some specific cases where there is a huge amount of subscribers for a given topic and a significant amount of publications. This can lead to high CPU usage and is handled by using topic replication.

All LLAs send an aggregate update message at a predefined interval to the Load Balancer node. This message contains all metrics for all topics for all time units t_i since the transmission of the last update message, as well as additional information such as the theoretical maximum outgoing bandwidth supported by that server node, as well as the measured outgoing bandwidth on the network interface. Since both the LLA and the load balancer are likely to be located in the same LAN/cloud, this period update message will likely have very limited impact on the bandwidth.

For additional details regarding our implementation of the LLA component within Dynamoth, the

reader is encouraged to refer to section 6.1.4.

3.3.2 Load Balancer: Generating a New Plan

Upon receiving the metrics from all Local Load Analyzers (LLAs) for every time unit *t*, the Dynamoth *load balancer* (LB) first computes the load ratio for all publish/subscribe servers. The load ratio LR_i for a given server i is defined as the measured outgoing bandwidth M_i divided by the maximum outgoing bandwidth supported by the server B_i (eq. 3.1).

$$LR_i = \frac{M_i}{B_i} \tag{3.1}$$

The LB then decides if a new plan should be generated or if the current plan should be kept. New plans are generated only after at least t_{wait} time units have elapsed since the last plan generation to make sure that most of the configuration overhead of the last plan change is completed before the next one is triggered. A new plan is generated using the Dynamoth rebalancer module in a two-step process: (1) topic-level rebalancing (subsection 3.3.2.1) and (2) system-level rebalancing (subsection 3.3.2.2).

3.3.2.1 Topic-level Rebalancing

In this step, the LB checks the number of publishers, subscribers and publications on each topic and determines whether some topics could benefit from replication (all subscribers or all publishers). Algorithm 3.2 outlines how the LB determines whether any given topic should use any of the replication schemes. The first step involves computing the *publication-to-subscribers ratio* (P_{ratio}) (line 1) and checking whether this ratio is above a given threshold (line 3). We also check whether we have a minimum amount of publications before triggering replication (line 3), since replication makes sense in cases where a given topic uses significant resources that cannot be managed by one publish/subscribe server. If both conditions are true, then the all-subscribers replication scheme is used. The number of servers $N_{servers}$ that should be used is then computed (line 4), and all subscribers will connect to all $N_{servers}$ servers. A similar approach is used to determine if the all-publishers scheme should be enabled instead (lines 2;7-9) and decide on the number of servers. At this step, we ensure that we have a minimum amount of subscribers to make sure that replication is relevant. If the conditions regarding all-subscribers and all-publishers are not met, then replication is not used for this topic (or is canceled

if it was active).

One corner case is the case where the amount of publications and subscribers are both very large (not shown in our algorithm due to space constraints); our system will then use the all-subscribers scheme since the all-publishers scheme causes publications to be sent to all publish/subscribe servers, which is more costly.

Upon enabling a given replication scheme for a given topic, or if replication is already enabled for this topic but the LB determines that additional servers should be used (P_{ratio} or S_{ratio} increases), then the load balancer selects the least-loaded servers first from the pool of available servers not already mapped to the topic. Similarly, if replication servers need to be freed (the LB determines that replication is not needed anymore or that the number of servers can be reduced), then the busiest servers are freed first.

begin
  P_{ratio} = $\#\text{publications}/\#\text{subscribers}$;
  S_{ratio} = $\#\text{subscribers}/\#\text{publications}$;
  if P_{ratio} > $\text{AllSubs}_{\text{threshold}}$ and $\#\text{publications} > \text{Publication}_{\text{threshold}}$ **then**
    N_{servers} = $P_{\text{ratio}}/\text{AllSubs}_{\text{threshold}}$;
    replicate(ALL_SUBSCRIBERS, N_{servers});
  end
  else if S_{ratio} > $\text{AllPubs}_{\text{threshold}}$ and $\#\text{subscribers} > \text{Subscriber}_{\text{threshold}}$ **then**
    N_{servers} = $S_{\text{ratio}}/\text{AllPubs}_{\text{threshold}}$;
    replicate(ALL_PUBLISHERS, N_{servers});
  end
  else
    replicate(NO_REPLICATION);
  end
end

Algorithm 3.2: Determining Whether Replication Should Be Used

3.3.2.2 System-level Rebalancing

In this step, the LB analyzes the load on each publish/subscribe server. In general, Dynamoth can perform two types of load rebalancings: (1) a high-load rebalancing, which is needed when one or more publish/subscribe servers are overloaded in order to bring the load down, and (2) a low-load

rebalancing, which takes place in the case where one or more publish/subscribe servers are underloaded in order to free servers that are not required anymore with the ultimate goal of shutting them down. Because publish/subscribe servers are most likely deployed in the cloud, the LB aims at being efficient regarding the number of servers that need to be used in order to save costs, while maintaining adequate performance. The two next subsections explain our current system-level load balancing algorithms for high-load and low-load rebalancing. In real commercial systems, more elaborate heuristics could be used.

3.3.2.3 High-Load Rebalancing

If there is a publish/subscribe server H_i with a load ratio LR_i that exceeds a given threshold LR^{high}, then a new *high-load* plan $P*$ *must* be generated so that $P*$ ensures that the load returns below a safe threshold for all servers. If this is not possible, then one or more additional servers have to be allocated from the cloud.

Algorithm 3.3 describes our heuristic for generating a plan to reduce the load on overloaded servers. The algorithm repeats as long as there is at least one publish/subscribe server with an estimated load ratio above LR^{high}. The publish/subscribe server with the highest load ratio (H_{max} with load ratio LR_{max}) is selected. Then, as long as the *estimated* load ratio $\overline{LR_{max}}$ remains above a certain threshold LR^{safe}, we do the following: (1) obtain the publish/subscribe server with the lowest load ratio (H_{min} with load ratio LR_{min}); (2) obtain the busiest topic T^{out}_{max} on H_{max}; (3) migrate this topic from H_{max} to H_{min} in the new plan $P*$, and (4) estimate the load ratio $\overline{LR_{max}}$ (on H_{max}) that we would get if $P*$ was applied. Of course, the estimated load on the server that receives the topic will be recalculated as well to make sure that we do not overload that server.

3.3.2.4 Low-Load Rebalancing

If the global load ratio (averaged LR_i for all publish/subscribe servers i) is below a given threshold, then one or more servers can be freed. This operation is less critical for performance reasons, but nevertheless essential for cost saving purposes. Topics from the lowest loaded server are slowly migrated to the other servers as long as the load on the other servers stays below a given limit. When a server has no more topics, it is deallocated. If at any point the global load ratio increases such that it becomes higher than the low-load threshold, then the low-load rebalancing will be interrupted (and, if needed, a

```
begin
    P* = P.copy();
    while true do
        (H_max,LR_max) = max(LR_i ∀ H_i);
        if LR_max < LR^high then
            return P*;
        end
        overline(LR_max) = LR_max;
        while overline(LR_max) ⩾ LR^safe do
            (H_min,LR_min) = min(LR_i ∀ H_i);
            T_max^out = getBusiestTopic(H_max);
            P*.migrate(T_max^out, H_max → H_min);
            overline(LR_max) = estimateLR(P*);
        end
    end
end
```

Algorithm 3.3: Generating a New High-Load Plan

high-load rebalancing can be triggered). The detailed low-load rebalancing algorithm, similar in spirit to the high-load rebalancing algorithm, is not described in details in this section. It is nevertheless fully implemented in our Dynamoth implementation.

For all algorithms, the values of the various threshold parameters were determined empirically based on the capabilities of the machines at our disposal. Of course, with different hardware, those values would most likely need to be adjusted. In future work, one could explore the idea of having a mechanism to automatically set and update thresholds based on real-time conditions.

3.4 Reconfiguration

This section describes the lazy propagation mechanism used by Dynamoth to propagate plan changes to relevant clients of the pub/sub system.

3.4.1 Overview

Upon determining that a new global plan $P*$ should be applied, all stakeholders (local load analyzer and dispatcher components, as well as subscribers and publishers) need to be informed. However, sending a new global plan to all clients at reconfiguration time would create a huge bottleneck and message overhead. Furthermore, global plans contain information about all topics, while individual clients are likely only interested in a few of these topics and therefore should only receive partial plan information on a need-to-know basis. Thus, we use a lazy scheme, as introduced in sections 3.2.5.1 and 3.2.5.2 where, at connection time, clients use consistent hashing to determine publish/subscribe servers and get to know the true server that should be used for a given topic only when they actually send their first message for this topic. Similarly at reconfiguration time, their partial plans are only updated on a need-to-know basis using a lazy propagation technique.

For this to work and to not loose any subscriptions and publications that are sent to the wrong publish/subscribe servers, the servers must be able to handle wrongly addressed messages. The following subsections give an overview of Dynamoth's lazy reconfiguration process. More details are given at the next section (3.4.2).

3.4.1.1 Initialization

Whenever a client does not have any server information about a topic, it sends the publication/subscription request to the server determined by consistent hashing. If this is not the correct server, the server sends a message back to the client informing it about the correct server. The client updates its local plan and then sends the message to the correct server.

3.4.1.2 Subscriber Change

Whenever a plan change moves a topic T from server H_0 to server H_1, all subscribers need to be informed and move their subscriptions accordingly from H_0 to H_1. We don't do this immediately for all topics, because this could lead to a spike of unsubscriptions and subscriptions at the time of reconfiguration, possibly causing performance bottlenecks. In order to stagger the reconfiguration of topics, we notify subscribers of the switch of topic T together with the first publication on T after the plan changes.

3.4.1.3 Publishing on an Old Server

As this is the case with subscribers, publishers are also not informed immediately of a plan change. In fact, in our system, there is actually no central authority that would know the content of the local plans of the clients, as they all are maintained individually by the clients themselves. Instead, when a topic T has moved from server H_0 to H_1, and H_0 receives a publication message on T, it informs the publisher about the change, so it can update its plan and send its next message to the correct server. At that time, both H_0 and H_1 might have subscribers for T, as some but not all of the subscribers might already have updated their subscriptions. Therefore, H_0 forwards the message to all subscribers of T still connected to it, and also sends the publication to H_1 so that it can deliver the message to all subscribers of T already connected to H_1, in a timely fashion (which is more efficient then denying the publication and asking the publisher to resubmit to H_1).

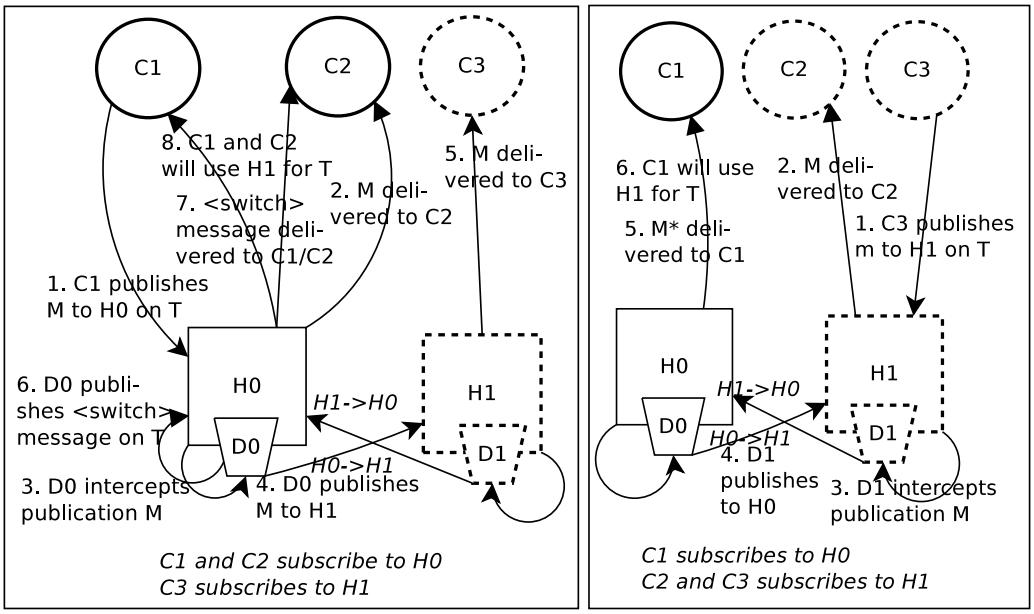
3.4.1.4 Publishing on the New Server

Finally, a publisher might already know the new location of T and send a publication for T to H_1 while there are still some subscribers connected to H_0. Therefore, H_1 forwards the publication not only to its local subscribers but also to H_0 so that it can disseminate it to the subscribers still connected to H_0. Such forwarding needs to occur until H_0 does not have any subscribers anymore, which is after the expiration of a timer (discussed in section 3.4.2.5).

For replicated topics, reconfiguration is more complicated as there are multiple publish/subscribe servers that are in charge of publications and subscriptions for a given topic. However, in principle, it follows the same line of reasoning as described above. For details, we refer the reader to Franz-Philippe Garcia's technical report [52].

3.4.2 Reconfiguration Details

A challenge in our system is that we rely on ready-to-use publish/subscribe servers that we do not want to alter. Thus, the forwarding functionality is implemented in the dispatchers that are collocated on the publish/subscribe server nodes as described in the following subsection and illustrated in figure 3.3. This section and the subsections give additional details and illustrate the reconfiguration process.



(a) Incorrect Server H_0 (b) Correct Server H_1

Figure 3.3: Handling Publications during Reconfiguration

3.4.2.1 Reconfiguration Setup

Each dispatcher has connections to all other publish/subscribe servers in order to be able to forward messages. Whenever a new global plan P_* is created, the load balancer sends it reliably to all dispatchers. Upon receiving a new plan P_* such that server H_0 was responsible for topic T in the old plan and server H_1 in the new plan P_*, the dispatchers of both H_0 and H_1 subscribe locally to topic T to receive all publications. Furthermore, the dispatchers intercept all subscription and unsubscription requests submitted to their local publish/subscribe servers.

3.4.2.2 Incorrect Publish/Subscribe Server

Figure 3.3a illustrates by example what happens if a publication message M on topic T goes to an incorrect publish/subscribe server H_0 (step 1). The publication is first sent to the subscribers still using H_0 for T (C_2 in our example). As the dispatcher D_0 is also subscribed to T, it also receives the publication. If this was the first publication on T after the new plan P_* was received, D_0 publishes a <*switch to* H_1> message to T on H_0 in order to ask all subscribers to switch to H_1. H_0 then forwards

this switch message to all subscribers, who then update their local plan and transfer their subscription to H_1 (steps 6, 7 and 8). D_0 also publishes the original publication to T on the new server (H_1) which delivers it to its own subscribers, if any (steps 3, 4 and 5). Note that some steps might execute concurrently (steps 2 and 3, or steps 4 and 6 for example).

3.4.2.3 Correct Publish/Subscribe Server

The case where a publication M on T is sent to the correct publish/subscribe server H_1 is much simpler (step 1 in figure 3.3b). H_1 delivers it to local subscriber C_2 (step 2). D_1 also receives M, and publishes M to T on the old server H_0 (steps 3 and 4). Finally, H_0 delivers M to C_1.

3.4.2.4 Client Subscribing and Moving a Subscription

If a client has outdated plan information for a topic T (using consistent hashing or having an outdated plan), then this client might send a subscription request for T to an incorrect server H_0. The dispatcher for H_0 then notifies the client that it subscribed to T on an incorrect server. Upon receiving such a message, the client immediately updates its local plan, subscribes to T on H_1 and unsubscribes from T on H_0. The same process is used when a client is asked to move an existing subscription: it subscribes to the topic on the new server, and then unsubscribes from the same topic on the old one (so that no publications are lost).

3.4.2.5 Duration of Forwarding and Dispatcher Subscriptions

An important question is how long dispatchers should subscribe to topics and forward messages. The dispatcher on H_1 must forward messages it receives for T to H_0 as long as there are still clients that are subscribed to H_0 instead of H_1. Thus, in order to avoid unnecessary forwarding, the dispatcher on H_0 notifies the dispatcher on H_1 as soon as there are no subscribers for T remaining on H_0. The dispatcher on H_1 then stops forwarding messages.

The dispatcher on H_0 forwards to H_1 messages for T it receives from publishers that don't know yet about the switch. In principle, it could simply send to the publisher the information about the new server H_1, and the publisher could then republish the message on H_1. However, for performance and cost reasons, H_0 forwards the message directly to H_1, because this communication happens inside the cloud. Over time, there will be less and less publishers that publish on the wrong publish/subscribe

server (H_0). To avoid requiring that the dispatcher on H_0 must be subscribed to T forever, we employ a timeout mechanism.

Each client C configures a timer for each topic T in its local plan. The timer is reset whenever C sends or receives a publication on topic T, or when the server for T changes in C's local plan. When the timer expires and the client is not subscribed to T, then the client removes T's entry from the plan. Should C later try to subscribe to T or send a publication to T, it connects to the server that is determined through consistent hashing (just like at the initialization phase, discussed in sections 3.2.5.1 and 3.4.1.1). The dispatcher on H_0 uses the same timer. It sets the timer for T when a new plan moves T from H_0 to another server H_1. It stops receiving publications on T and forwarding messages to H_1 when the timer expires, because at this time no client will have the outdated information for T anymore.

The dispatcher of the default server H_d^T, i.e., the server for a topic T determined by consistent hashing, is always subscribed locally to T. Thus, it can determine when publications for T are sent erroneously to H_d^T and let the senders know the real server that is responsible for T. With this, whenever a client sends a message to the wrong server (either a server based on an outdated plan or the server determined through consistent hashing), the dispatcher on that server receives the message and informs the client about the reconfiguration.

3.4.2.6 Ordering and Delivery Guarantees during Reconfiguration

During reconfiguration, using TCP is not enough by itself to guarantee FIFO ordering and exactly once delivery. Assume a topic T is moved from server H_0 to server H_1. Publisher P still sends its first message M_1 after reconfiguration to H_0. H_0 forwards it to H_1, but also tells P about the change. Therefore, P sends its next two messages M_2 and M_3 to the new server H_1, which delivers them to its local subscribers, but also forwards them to H_0 for subscribers that have not yet migrated. It can happen that a subscriber S subscribes to H_1 just after H_1 has sent out M_2, but before it has sent out M_3. Note that S first subscribes to H_1 and only when this is confirmed it will unsubscribe from H_0. Thus, S will receive M_2 from H_0 and M_3 from H_1, that is, it will receive all messages. But as these two messages arrive from two different servers, it can happen that M_3 arrives before M_2. It might also happen that both H_0 and H_1 send M_3 to S if S doesn't unsubscribe quickly enough. We have to guarantee that both messages are delivered only once to the application layer at S, and in the correct order.

Therefore, as mentioned before, the underlying client library automatically adds sequence numbers to the publisher's messages. If messages arrive at the subscriber in an incorrect order as described above, our library delays their delivery to the application until missing messages are received. If messages arrive twice, duplicates are discarded; thus, we guarantee that FIFO ordering is maintained for all messages sent by any given publisher. Note that such out of order or double reception can only occur during reconfiguration. Thus, adding sequence numbers is only needed during reconfiguration (e.g., from the time the publisher gets notified that it should send to a new server until the timer that we described in the previous section has expired).

In addition to being used as part of the reconfiguration process, the duplicate message elimination strategy is also used as part of our failure recovery process, discussed in the next section. While we have implemented elimination of duplicates, we have not fully implemented the reordering of messages during reconfiguration so far, due to the added complexity. As we are in the process of open-sourcing Dynamoth, we would certainly welcome patches from the community in this regard.

3.5 Availability

Dynamoth is designed to provide availability by remaining fully functional in the event of server failures. To reach that goal, the various Dynamoth components, both server- and client-based, are able to detect server failures and take appropriate reconfiguration measures. The failure-handling mechanism is completely transparent to the users of the Dynamoth library. Note that we only consider server failures, as client failures are out of the control of the publish/subscribe service. Subscribers that fail are simply excluded and do not receive any messages that are delivered during their downtime.

Guaranteeing reliability and/or FIFO delivery when failures occur make failover more expensive, and can slow down message delivery significantly. In some contexts, in particular for latency-constrained applications, such an additional delay might not be acceptable. For instance, in gaming applications, a published state update often replaces a previous state update. Thus, a best effort delivery of publications might be sufficient. Therefore, the Dynamoth client library allows one to specify, for any topic T, whether or not delivery and/or FIFO needs to be guaranteed during failure handling periods. In particular, we support three options: (i) reliable and FIFO delivery, (ii) reliable delivery but no ordering guarantee, (iii) best effort.

3.5.1 Failure Assumptions

Our failure handling components make several assumptions in regard to crash and asynchrony behavior.

3.5.1.1 Fault Model

Our system can handle server failures, but assumes no network failures (no message loss, no network partitions). We assume that servers fail by crashing, meaning that all processes on the server machine fail simultaneously, i.e, the publish/subscribe middleware (Redis) and all related components (local load analyzer, dispatcher). We can handle simultaneous server failures, and in principle, as long as one server is running, the service is available.

3.5.1.2 System Load

During recovery, the load of failing servers is transferred to non-failed servers. Therefore, Dynamoth assumes that the supplemental load can temporarily be handled properly by the remaining servers, which is a necessary condition to bring the system back to a fully functional state. After recovery, the Dynamoth load balancer is invoked and can trigger reconfiguration steps including the spawning of additional cloud servers, if needed.

3.5.1.3 Bounded Message Delivery Time

We follow the traditional failure detection approach and suspect a server to have failed if we do not receive an expected message within a certain time interval. That is, we assume an upper bound on the delay of any message. In particular, we assume that any message sent by any publisher P to a publication server H, or from a publication server H to any subscriber S, is delivered and processed within an interval of T_{send} (we assume the processing time to be a small, constant value). We further assume a bounded communication time T_{server} between servers in the cloud. Naturally, $T_{server} < T_{send}$, since all servers are co-located in the same cloud. As the Internet is inherently asynchronous, our failure detection modules on the publisher and subscriber clients might wrongly suspect a server to have failed while it is operational. However, our system can easily handle such wrong suspicions by clients.

3.5.2 Overview

In the following subsections, we first describe the case of a single server failing and no topic replication. Reconfiguration when multiple servers fail and reconfiguration for replicated topics are described in sections 3.5.7 and 3.5.8. Let H_f be the failing server. The failure handling mechanism can be decoupled in three main steps: the *failure detection phase*, the *resubscription phase* and the *replay phase*, followed by an optional *load balancing* phase.

3.5.2.1 Failure Detection Phase

In the failure detection phase, all involved nodes detect the failure. We assume that servers detect the failure earlier than clients, and all agree whether it is really a failure, or H_f is simply being slow. The exact failure detection process is explained in section 3.5.3.

3.5.2.2 Resubscription Phase

During the resubscription phase, all subscriptions are reestablished towards alternate servers. The default alternate server for a topic is simply the server that is determined through consistent hashing, i.e., the same server that is responsible for a topic by default at system start (section 3.2.5.1), simplifying the reconfiguration process. The resubscription process is described in section 3.5.4.

3.5.2.3 Replay Phase

Finally, in the replay phase, missed publications, which are messages that were in transit when H_f failed, are resent, with or without ordering, in order to fulfill the delivery and ordering guarantees. In our model, the individual publishers are the ones who resend messages. We have decided to not use the traditional mechanisms of persistence or 3-way replication at the server side to avoid message loss as both have a very high overhead in terms of execution costs and execution time; which we consider prohibitive for latency-constrained applications. Instead, for topics where reliability is needed in the case of failures, the Dynamoth library at the publisher buffers recently sent messages and resends them should a server crash. This replay occurs in the order messages were originally sent to guarantee FIFO, or, if the publisher does not care about order, messages are resent in a more efficient way. The message replaying process is discussed in sections 3.5.5 and 3.5.6.

3.5.2.4 Load Balancing

Once the failover is completed, i.e., all topics that were originally assigned to the failed server have been reassigned and messages have been replayed, the new configuration is likely to not be completely balanced and/or might be close to an overload situation. After reconfiguration, the load balancer, which is suspended during the failure recovery process, is restarted. Then, if needed, the load balancer will generate a more optimal configuration taking into consideration the new set of available servers and the current assignation of topics, following its usual process, in order to fine-tune the allocation of topics to servers, and if needed, introduce new servers.

3.5.3 Detecting Server Failure within Bounded Time

Although TCP connections are capable of detecting failures, the standard TCP timeout interval is far too large to be useful in the context of Dynamoth. This section describes how Dynamoth detects all server failures within a time interval T_{detect}.

3.5.3.1 Server Failure Detection for Subscribers

To enable subscribers to detect server failures, Dynamoth servers ensure that they communicate with each subscriber at least once within each time interval T_{detect}. Let S be a subscriber subscribing to one or more topics on server H, and let $T_H(m_L)$ be the time that S received the last publication from server H (across any topic). Let $T^S_{current}$ be the current time at S. Then $T^H_{elapsed} = T^S_{current} - T_H(m_L)$ is the amount of time that has elapsed since the last message was received from H. If $T^H_{elapsed}$ exceeds T_{detect}, then subscribers suspects H to have failed.

To ensure that $T^H_{elapsed}$ does not exceed T_{detect} for servers that are functioning correctly, the LLA at H, that already monitors communication with each client, injects artificial "keep-alive" messages when needed. Let $T^H_{current}$ be the current time at server H and let $T_S(m_L)$ be the time the last publication was sent to subscriber S by server H (across any topic). As defined in section 3.5.1, T_{send} is the estimated maximum bound on the time needed to deliver and process a message. As $T^H_{current-}T_S(m_L)$ approaches $T_{detect} - T_{send}$, the LLA issues a keep-alive message to S, so that it is received before the T_{detect} timeout is reached on S. Thus, keep-alive messages are only needed in the case where no other publication was sent to S within the T_{detect} interval.

3.5.3.2 Server Failure Detection for Publishers

Let P be a publisher publishing to at least one topic on H. If P is also a subscriber to any topic managed by H, then P benefits from the failure-handling mechanism for subscribers described above. Since it is essential for publication playback that all publishers detect server failures in time, our client software simply subscribes to a dummy "keep-alive" topic on H in the case where P is not subscribed to any topic on H. If P eventually subscribes to a topic on H, then our client software simply drops the subscription to the "keep-alive" topic.

3.5.3.3 Server Failure Detection by other Servers

Within the cloud, we install a standard failure detection mechanism with agreement. Thus, if any server suspects another server H_f to have failed, it runs an agreement protocol with the other servers so that all servers take the same decision to either decide that H_f has failed or is still functional. We assume that in-cloud failure detection is much faster than any publisher or subscriber suspecting a failure.

3.5.4 Reestablishing Subscriptions to the Default Servers

Upon failure detection of server H_f by any given subscriber S, S reestablishes all subscriptions that it had to all topics managed by H_f towards alternate servers. Just as when a client connects to the system the first time, i.e., when it still has no information about who is serving a particular topic T, S simply sends its resubscription request for T to the default server H_d determined by consistent hashing as described in section 3.2.5. If a client wrongly suspects a server to have crashed, e.g., because of network partitioning, then the default server H_d informs S by rejecting the resubscription request.

We define T_{resub} as an upper bound on the amount of time that all subscribers need to reestablish all of their subscriptions. Thus, assuming the actual time of the failure of server H_f is t_f, then at the latest at time t_s (equation 3.2) it is guaranteed that all subscribers will have reestablished all subscriptions to all topics H_f was responsible for, to the default servers of these topics.

$$t_s = t_f + T_{detect} + T_{resub} \tag{3.2}$$

3.5.5 Best Effort Delivery

In the best effort approach, messages sent after the failure but before its detection by the publisher, are lost. A publisher, as it is also a subscriber (to at least the keep-alive topic), detects the failure within the detection window T_{detect} after the actual time of the failure t_f. As the actual detection time varies from client to client, a publisher/subscriber on topic T might detect a failure earlier than another subscriber. Nevertheless, if delivery guarantees are not important for a given topic T, the publisher can immediately start sending new publications towards the alternate server H_d. However, if it is required that at least the new messages sent on topic T should be transmitted to all subscribers, then the publisher has to wait until all subscribers have resubscribed to H_d, at time t_s (equation 3.2). In the most extreme case, the publisher has detected the crash basically at the time it occurred (begin of the detection window) and a subscriber at the end. Thus, a publisher should wait the time of $T_{detect} + T_{resub}$ after it has detected the failure by itself before starting to send new publications to H_d. For this purpose, the library at the publisher maintains for each topic a failover queue Q_f to which is appends all messages submitted by the publisher. The messages are then sent in FIFO order $T_{detect} + T_{resub}$ time units after the detection of the failure.

Note that the best effort approach guarantees FIFO delivery. All messages received before the failure are received in sending order as discussed in previous sections. Then, the set of messages in transition during failure detection and failover are lost. When a subscriber has successfully resubscribed and receives messages again, these messages are again received in the order they were sent. The next section (3.5.6) discusses the case where reliable delivery is enabled for topic T; that is, all messages including those in transition during failure detection and failover should be correctly delivered.

3.5.6 Retransmitting Missed Publications

In order to retransmit missed messages for topics for which reliable delivery is requested, the client library at the publisher appends during normal processing each message sent by the publisher to a local buffer queue Q_r for a certain amount of time, which we call the *playback window*. When the publisher detects a failure, the messages in Q_r are resent to the alternate server H_d (after the resubscription phase).

3.5.6.1 Playback Window

An essential question is how long publishers should retain publications for playback in order to guarantee that no publications are lost. Assuming t_f to be the time of failure, messages that were sent before time $t_f - 2T_{send}$ are guaranteed to have arrived at the subscribers (T_{send} time units to arrive at the server and another T_{send} time units to be sent by the server to the subscribers; we assume that a message is sent in parallel to all subscribers). A publisher detects a failure the latest at $t_f + T_{detect}$. Thus, Q_r must hold all messages sent within the last $T_{detect} + 2T_{send}$ time units. Messages sent before this playback window can be removed from the queue.

3.5.6.2 Replaying Past Publications

A publisher P starts resending messages in Q_r when it is certain that all subscribers have reestablished their subscriptions. As discussed above, this is guaranteed to have happened $T_{detect} + T_{resub}$ time units after P detected the failure itself. As it is possible that some subscribers have already received some of these messages before the failure, the library at the subscribers detect such duplicate messages in the same way duplicate messages were detected during reconfiguration (see section 3.4.2.6).

If FIFO order is required even during failure recovery; that is, if it is required that all messages are sent in FIFO order, including messages that have to be replayed, then P resends messages in Q_r in the order they were appended (ordered playback). In contrast, if no ordering is required, then we resend the messages in reverse order, that is, the message that was last appended to Q_r before the failure was detected is the first one to be sent. This is done to minimize delivery times for recent publications, since fresher publications are likely to be more relevant. In addition, it is more likely that older publications (at the beginning of the queue) have already been delivered at the subscribers.

Note that we use a form of throttling to control replay. In particular, after each replay of a message, we introduce a delay of T_{delay} time units before resending the next message. This delay serves two purposes: 1) ensuring that alternate servers do not collapse due to the additional load of the replaying process and 2) ensuring that publications on other topics (handled by the same servers) do not suffer performance degradations.

3.5.6.3 Handling New Publications

Another important question is how to handle the messages that are submitted for publication on the publisher after failure detection. We already described that these messages are appended to Q_f (not to be confused with Q_r) and should only be sent after it is ensured that all subscribers have properly reestablished their subscriptions. Furthermore, in the case where reliable and FIFO order are required, we first have to resend all messages from Q_r (i.e., the potentially missed publications), and only then we can process the messages queued in Q_f. Queue processing (Q_r then Q_f) takes place until all messages have been sent. In other terms, new messages arriving while processing either Q_r or Q_f are appended to Q_f, until Q_f is empty, if FIFO is required. As a result, new publications are queued and eventually transmitted in the correct order *after all past publications have been replayed* to the alternate server.

In contrast, if ordering is not required, we can perform the sending of messages in Q_f concurrently with resending messages in Q_r (concurrent playback). Also messages that are submitted concurrently to this replay are sent out immediately.

In both cases, when the queues are empty, that is, when we reach a point when all past and queued new publications have been sent, normal processing of publications is restored so that future publications are sent normally.

Preserving FIFO ordering has the drawback that all new publications are delayed until all past publications have been (re)transmitted. In the case of concurrent playback, because new publications are sent concurrently to the playback of old publications, they do not suffer from additional delays.

3.5.7 Failing Servers while Reconfiguring

For simplicity, in the previous sections, we described the case with one failing server. At a high level, handling multiple failures, where a given failure happens prior to the completion of a failure recovery process, consists of applying our failure recovery mechanism recursively. This section briefly outlines how Dynamoth is able to handle such cascading failure scenarios.

Assume one of the alternate servers H_d crashes while the failover of H_f has not yet completed. Subscribers that were still in process of resubscribing to H_d will experience a timeout or error message when trying to connect to H_d. Subscribers that had already completed their subscription process will detect the new failure at most T_{detect} time units after the failure of H_d occurs. They simply have to find a

further alternate server $H_{d'}$, and reconnect to that server instead. This *second* alternate server is selected by an extension of our consistent hashing mechanism, which allows for generating a *deterministic sequence of alternate servers*. Assuming that the non-failed servers can support the additional load of the failed servers, considering N servers, we support $N_f = N - 1$ failing servers; thus, for any server H_f, we can generate a sequence of N_f alternate servers.

As long as there is no message replay, the reconnection of publishers follows the same logic. If reliable delivery is required for topic T, then we have to make sure that all messages are delivered. What has to be done depends how far a publisher P got in replaying messages before the failure of H_d. To not loose any messages, the replay mechanism simply has to use the same mechanism that is deployed during normal processing. We create a temporary queue $Q_{f'}$. All messages (re-)sent from Q_r and Q_f to the alternate server H_d are appended to $Q_{f'}$ and remain there for a time period of $T_{detect} + 2T_{send}$. After that time period we know that H_d has sent these messages to the subscribers. Should the publisher complete the replay (Q_r and Q_f are empty) before the failure of H_d, $Q_{f'}$ simply becomes the new Q_f, and from the view point of P, the crashes ofH_f and H_d are consecutive and not concurrent. Should Q_r or Q_f not yet be empty when P detects the failure of H_d, then the remaining messages are removed and appended to $Q_{f'}$, then $Q_{f'}$ becomes the new Q_f and the replay to $H_{d'}$ is started upon waiting $T_{detect} + T_{resub}$ time units.

That is, replayed publications and new publications during the replay are treated just as publications during normal processing. They are kept at the publisher as long as it is not guaranteed that the alternate server has sent them to the subscribers. Should the alternate server fail they will be replayed to the alternate server of the alternate server. Clearly, this process can be iteratively repeated for cascading failures.

While cascading failures lead to increased, yet temporary congestions in topics affected by the failing servers, the playback mechanism guarantees that all potentially missed publications are nevertheless correctly received by all subscribers, assuming that the subscribers and publishers remain connected to the Dynamoth system.

We would like to note that if a publisher fails during the replay mechanism, it might happen that some subscribers have received a message (before the server failure) while others have not (as the publisher failed before resending the message to the alternate server). This inconsistency can only occur if a publication server and a publisher fail shortly one after the other.

3.5.8 Reconfiguration with Replication

We previously described the reconfiguration process in the context of non-replicated topics. For replicated topics (All-Publishers and All-Subscribers), the process is altered as follows. Assuming topic T is replicated to servers $\mathcal{H} = \{H_1, H_2, \ldots, H_f, \ldots, H_n\}$; if H_f fails, then clients detect it according to the process described in section 3.5.3. Then, instead of finding an alternate server for H_f, H_f is simply removed from the list of replicated servers. Specific steps for the two replication schemes are as follows:

3.5.8.1 All-Subscribers Replication

Subscribers do not need to do anything beside remembering that H_f has become unavailable (and that the subscription to T on H_f is no longer valid). All publishers who had chosen H_f to be the server to which they sent their messages randomly select one of the remaining servers for T and republish potentially missed publications according to the procedure described previously. Note that they do not need to wait for subscribers to resubscribe as all subscribers are already subscribed to all remaining servers in $\mathcal{H}$. In that context, all-subscribers replication has the added benefit of providing extra redundancy, which can shorten recovery time.

3.5.8.2 All-Publishers Replication

All subscribers that were using H_f randomly select one of the other remaining servers for T to resubscribe. Publishers register the failure of H_f. If replay is required, they replay to all remaining servers in the same way as done for the non-replicated case.

3.6 Implementation and Experimental Setup

3.6.1 Implementation

Our Dynamoth implementation is highly modular and consists of over 150 Java classes and over 15,000 lines of code. It is described in more details in section 6.1. For the purpose of this chapter, all Dynamoth components and algorithms were implemented as described. All inter-component communications are done using the publish/subscribe primitives offered by the Dynamoth API. The dispatcher and local

load analyzer modules reuse the publish/subscribe interface, and standard Redis instances are used as the publish/subscribe servers. They are independent and do not communicate with each other. Because of this, any individual Redis instance can be replaced with any other publish/subscribe middleware as long as it support the basic publish/subscribe primitives.

As application we use a multiplayer online game (MOG) application. Mammoth [70] is a game engine for MOGs that was developed at McGill University as a testing and research framework. Researchers are able to use the system to conduct all kinds of experiments related to MOG games in a realistic environment. Mammoth has a very modular architecture that allows easy replacement of components. One such component is the network infrastructure, which handles the exchange of all messages between all nodes. Mammoth requires the network engine to provide a topic-based publish/-subscribe API as messages to updated players, messages to send replicas to clients and general broadcast messages by the server are sent using the publish/subscribe paradigm. Thus, we used Dynamoth as the network engine for Mammoth. For our experiments, we used a specific sub-game developed within Mammoth, namely RGame, in which players are controlled by a simple AI that repeatedly chooses a random point on the map, moves the player towards that point and then takes a short break. The game world is split into a set of square tiles. Players subscribe to the tile in which they are located in, and publish their own state updates on the tile. Thus, all players receive update messages from all other players in the same tile. As players continuously move around, this application generates many subscriptions and unsubscriptions to tiles, and update position publications on these same tiles.

3.6.2 Experimental Setup

Large-scale scalability and elasticity experiments were conducted on 80 machines of the labs of the School of Computer Science of the McGill University over which we distributed client and server nodes. Part of our decision was motivated by the bandwidth costs that would have been incurred if many large-scale experiments were run in a public cloud.

While each publish/subscribe server node received exclusive access to one of the lab machines, clients had to share machines in order to provide scalability results. All lab machines have dual or quad-core processors and at least 4 GB of RAM. Both client and server nodes were run in the same LAN. In order to emulate a typical cloud setup, where the publish/subscribe server nodes would run in the cloud connected by a LAN and the clients access the servers over a WAN, we adjusted our latency

measurements using the King dataset [57] (a study that gathered millions of latency measurements between arbitrary DNS servers). We filtered the dataset to keep only measurements from North America. For each inbound publication message, we do the following: (1) if the publication comes from an infrastructure node[1] and goes to a client node, then we sample one value from the dataset; (2) if the publication comes from a client node and is received by an infrastructure node, then we sample one value and (3) if the publication comes from a client node and is received by another client node, then we sample two values (round-trip). Each message received by the Redis middleware is put in a queue and is delivered to the application layer only after a timer corresponding to the sampled latency value(s) expires. Our experiments revealed that this delaying mechanism produced latency measurements comparable to what we could expect from running our infrastructure servers in the cloud.

On the other hand, availability experiments were run in a later stage in a real cloud setting. For these experiments, we used the Amazon EC2 cloud, with up to 20 virtual machines located in the `us-east-1` (Virginia) region. The King dataset was also used in a similar fashion to emulate proper client-to-cloud latencies, since both clients and publish/subscribe servers were run in the same cloud for convenience and cost-saving purposes.

3.7 Experiments

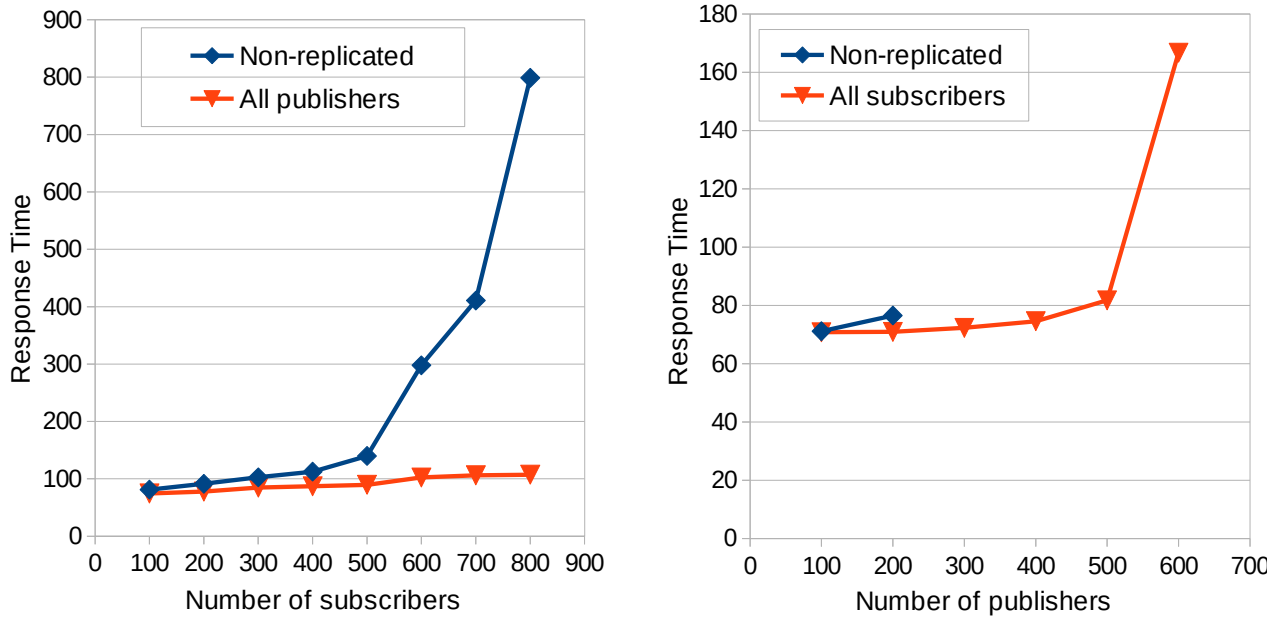
3.7.1 Topic-level Scalability

We first assessed the scalability of the topic-level (micro) load-balancing capabilities of Dynamoth by running experiments with both replication schemes proposed by Dynamoth: “all publishers” and “all subscribers”. For that, we ran a set of micro-benchmarks that focus on specific overloaded topics.

3.7.1.1 All Publishers

In this experiment, we connected up to 800 subscribers to a given topic T. In our setup, we ran 10 subscribers per machine. One publisher client sends 10 publication per second on topic T. This experiment was first attempted with replication disabled (only one publish/subscribe server was handling topic T) and then with replication enabled over 3 servers (3 publish/subscribe servers were in charge of server T). Under the “all publishers” model and under the replicated configuration, the publisher

[1] A node that would be usually located in the cloud: Local Load Analyzer, Dispatcher or Load Balancer



(a) All Publishers

(b) All Subscribers

Figure 3.4: Replication Experiments

was sending its publications to all 3 servers and every subscriber was randomly subscribing to T on one of the 3 servers. Figure 3.4a details response time results.

We observe that with 100 subscribers, both the non-replicated and the replicated configurations yield similar response times. Then, as the number of subscribers grows, the response time for the non-replicated configuration continuously increases. This is explained by the fact that sending the message to a large volume of subscribers takes more time if it is done only by one server. Finally, above 500 subscribers, the CPU is not able to process the flow of publications anymore and the performance decreases exponentially. Using 3-server replication, the response times remain very low. This is due to the fact that each server only needs to process and forward publications to a $\frac{1}{3}$ of the subscribers. Thus, our all-publishers replication mechanism allows our system to scale properly in situations where there would be many subscribers on a given topic.

3.7.1.2 All Subscribers

We attempted to connect up to 800 publishers sending 10 publications per second each on a given topic T, and only one subscriber. This experiment was ran with replication disabled and with replication enabled with 3 servers under the "all subscribers" model. Under this configuration, the subscriber was subscribing to T on all 3 servers and all publishers were publishing randomly to one of the 3 servers. Figure 3.4b details the response time results.

We observed that under the non-replicated configuration, we are able to support up to 200 publishers. After that, delivery of messages fails because the output buffer for the subscriber gets too full due to the high volume of publications. Under the replicated configuration, we are able to safely support nearly up to 600 publishers because each server processes only $\frac{1}{3}$ of the publications. This demonstrates that our all-subscribers replication mechanism allows our middleware to scale to support scenarios where there are large amounts of publications.

3.7.2 Scalability

This experiment aims at evaluating the scalability of our Dynamoth architecture and the effectiveness of our load balancer in the context of a large-scale latency-constrained game application. At the start of the experiment, some 120 players are active in the game and over time more and more players join the game; therefore increasing the load in the system. Overall, we attempted to connect up to

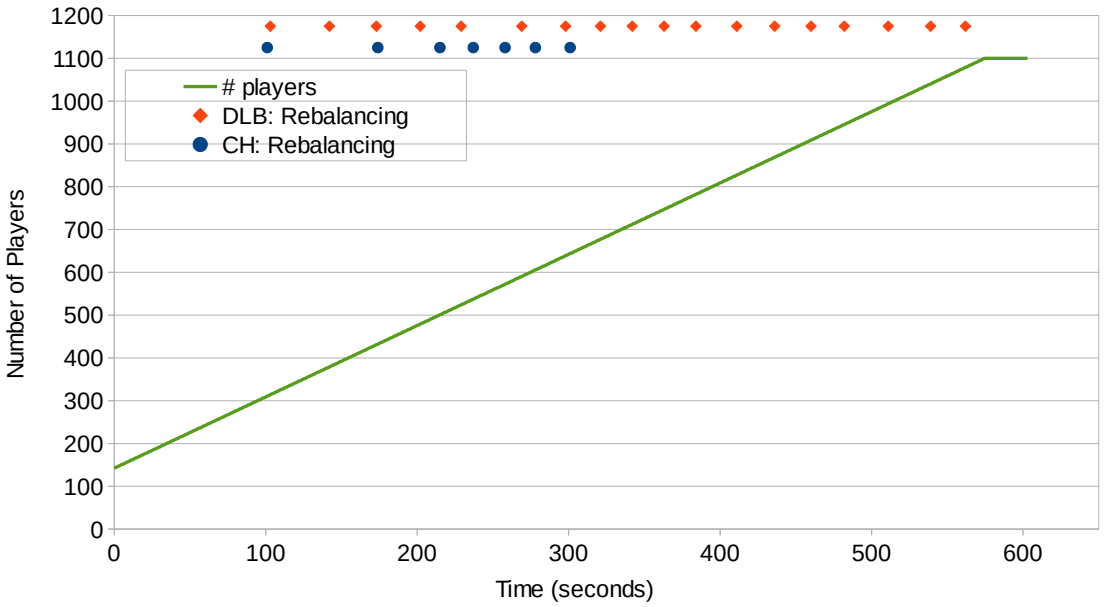


Figure 3.5: Number of Players

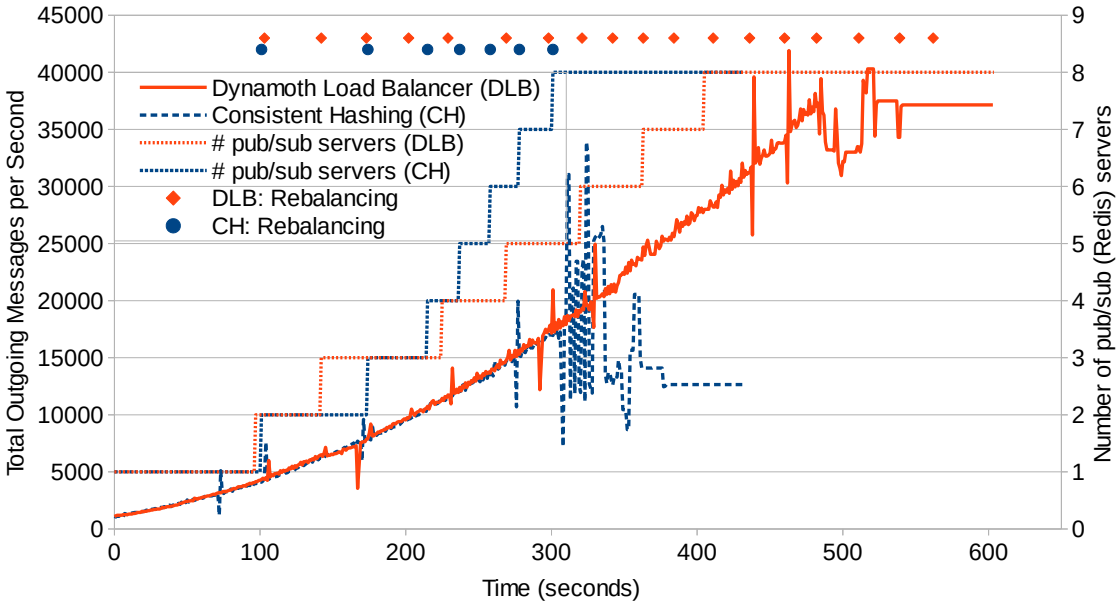


Figure 3.6: Total Outgoing Messages and Number of Publish/Subscribe Servers

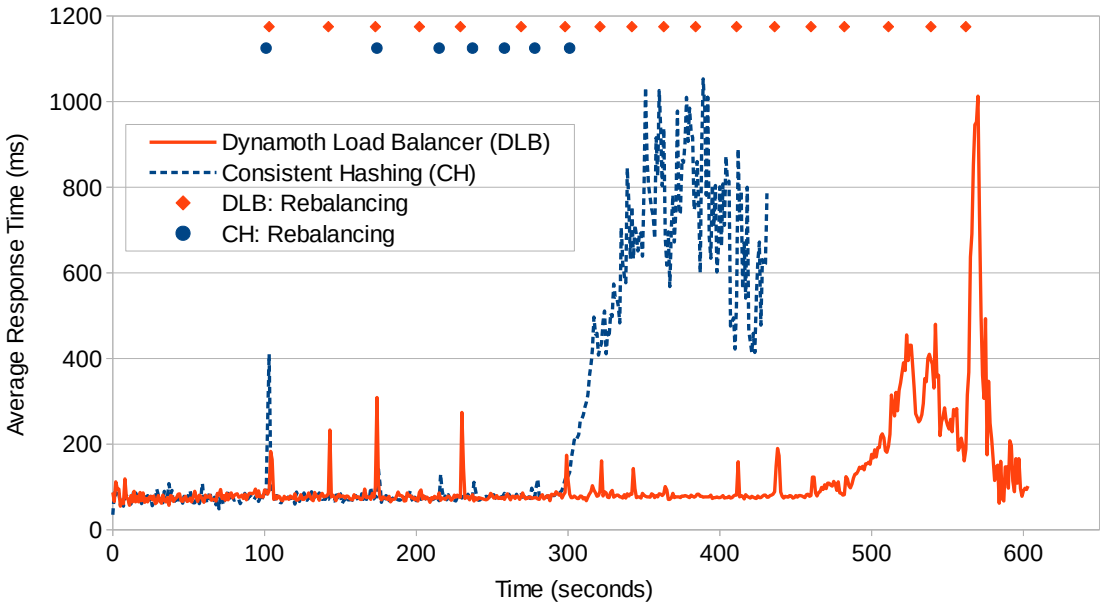


Figure 3.7: Average Response Time

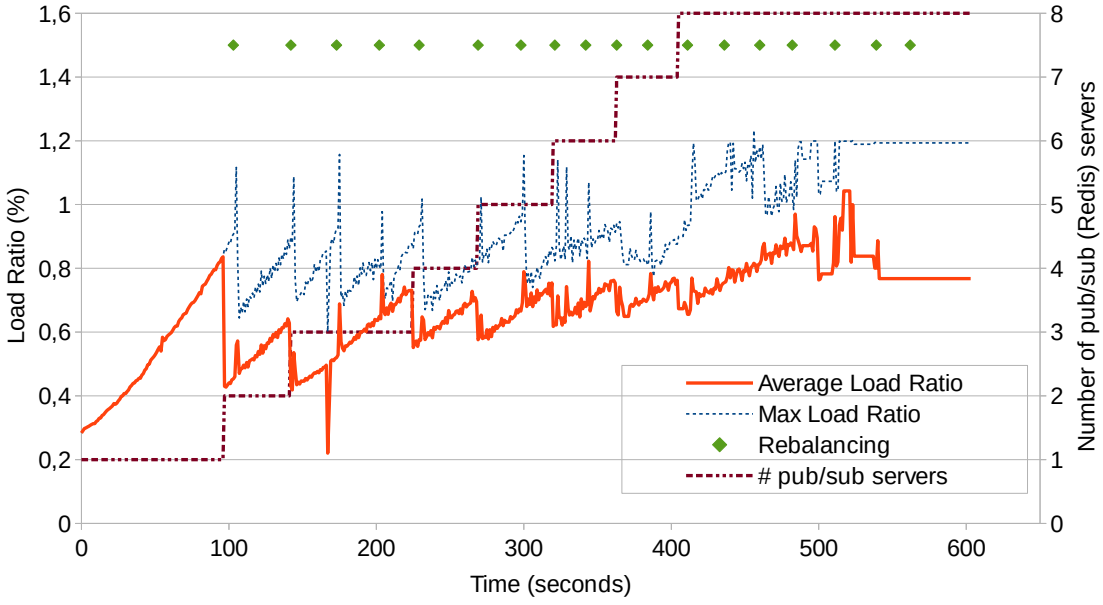


Figure 3.8: Dynamoth Load Balancer - Publish/Subscribe Server Load

1200 clients; once joined, each player sends 3 state updates (publications) per second. Up to 8 Redis publish/subscribe servers were available. We ran this experiment with our Dynamoth load balancer and we ran the same experiment again using only consistent hashing, the standard load balancing technique described previously in section 2.3.3.2.

Figures 3.5, 3.6 and 3.7 detail our results for experiment 2. In all figures, the time is shown on the X-axis (in seconds). Figure 3.5 plots the number of players active in the system over time showing how the players slowly join the game. Figure 3.6 plots the total number of messages transmitted per second throughout the whole system over time, as well as the number of Redis publish/subscribe servers (between 1 and 8) that were currently active, for both the Dynamoth and consistent hashing experiments. Finally, figure 3.7 plots the average response time experienced by clients over time (the time that elapses between the client publishing a state update and receiving the corresponding notification back from the publish/subscribe server). The diamonds and circles indicate the times where the load balancer triggered a reconfiguration respectively for the Dynamoth and consistent hashing approaches. In the context of a game, the playing quality will be optimal if the average response time remains below 150ms.

By using multiple Redis publish/subscribe servers, Dynamoth scales up to almost 1000 participants. We observe small spikes in the average response time around the time when new server is added and rebalancing takes place, but those bursts are only of short duration and the average response time is otherwise always maintained at an acceptable threshold (around 75ms). The bursts happen because the application of the new plan occurs at a time the servers are already overloaded, further increasing the load for a short amount of time, leading to an additional delivery delay for some messages. However, our lazy plan propagation approach keeps this impact of plan changes very low, as explained in section 3.4. The results further show that our load balancer is conservative and first reuses the pool of active servers before deciding to spawn new servers. This keeps cloud utilization costs low. Furthermore, even after the 8th and final server is deployed, the load balancer is still able to maintain an acceptable performance for a while just by applying incremental plan changes (without deploying additional servers).

With consistent hashing, only up to 625 players can be supported before performance deteriorates. This is due to the fact that consistent hashing doesn't take individual server loads into account when a rebalancing occurs. Servers shed $\frac{1}{N}$ of their load to a newly deployed server, irrespective of their current

load. As a result, highly loaded servers do not loose significant load and tend to become overloaded again soon. Furthermore, this technique has to spawn a new server every time a rebalancing occurs, which is not cost efficient in a cloud setting.

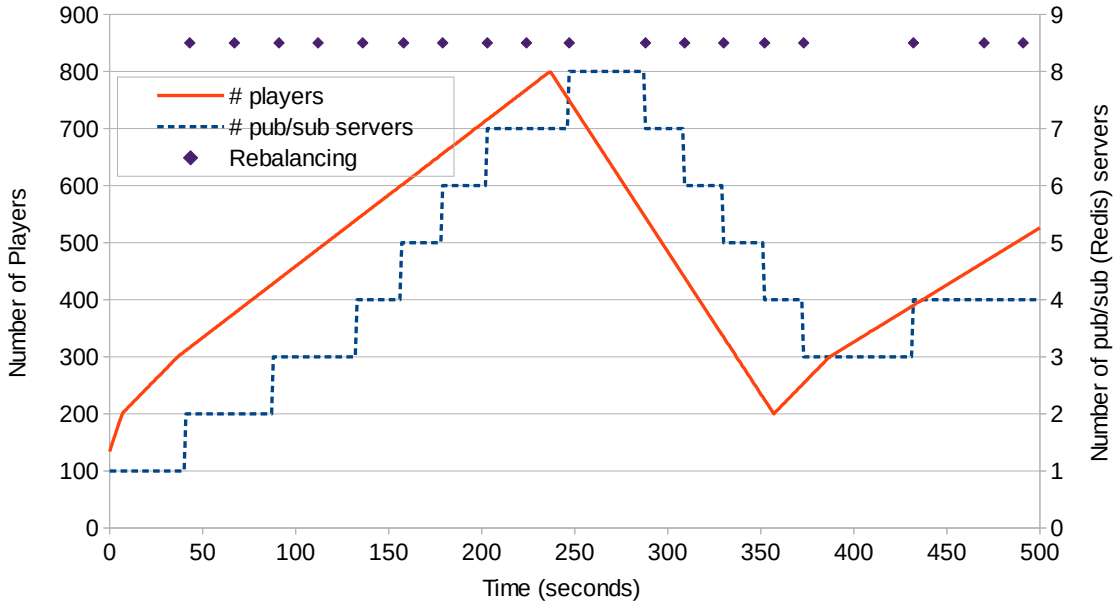
For that same experiment with Dynamoth, figure 3.8 plots the average load ratio (equation 3.1) of all active publish/subscribe servers, the load ratio of the busiest server as well as the number of Redis servers and the time points when a rebalancing occurred. A load ratio of 1 or below is safe. According to our observations, a Redis publish/subscribe server will fail when the load ratio exceeds 1.15. We can see that our load balancer is able to maintain the average load below 1 until the system as a whole becomes overloaded. It is also able to maintain the load ratio of the busiest server below 1 for most of the experiment.

3.7.3 Elasticity

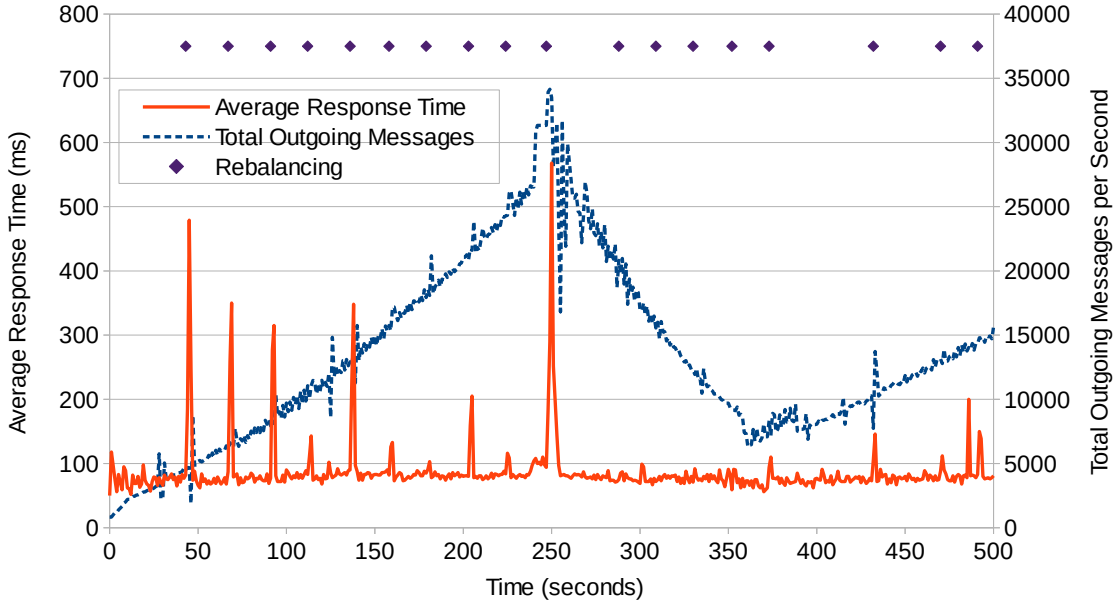
In this experiment, we show how the Dynamoth load balancer handles fluctuating real-time conditions. We first inject step by step 800 clients into the virtual environment; then we remove 600 (to reach 200); then we connect a little less than 400 additional clients (to reach almost 600). Figure 3.9 shows the measurements gathered during this experiment. Figure 3.9a shows the number of players as well as the number of Redis publish/subscribe servers that were being used at a given time. Figure 3.9b shows the average response time and number of outgoing messages over time. In both figures, the points in time where the Dynamoth load balancer triggered a rebalancing are denoted by a diamond.

We observe that as the number of clients increases, rebalancings occur, which sometimes require the addition of new servers. When the number of clients decreases, rebalancings also occur and are able to release servers again to save cloud infrastructure costs. Since those rebalancings have a lower priority, there is an observable delay between the time when the load decreases and the servers are removed. As in the previous experiment, when high-load rebalancings occur, we observe small spikes in average latencies as rebalancing adds additional load to already loaded servers. When scaling down, rebalancings do not cause spikes in average latencies because such rebalancings only occur when publish/subscribe servers are underloaded.

Overall, this experiment reveals that Dynamoth is correctly able to handle fluctuating workload patterns by providing adequate resources as needed.

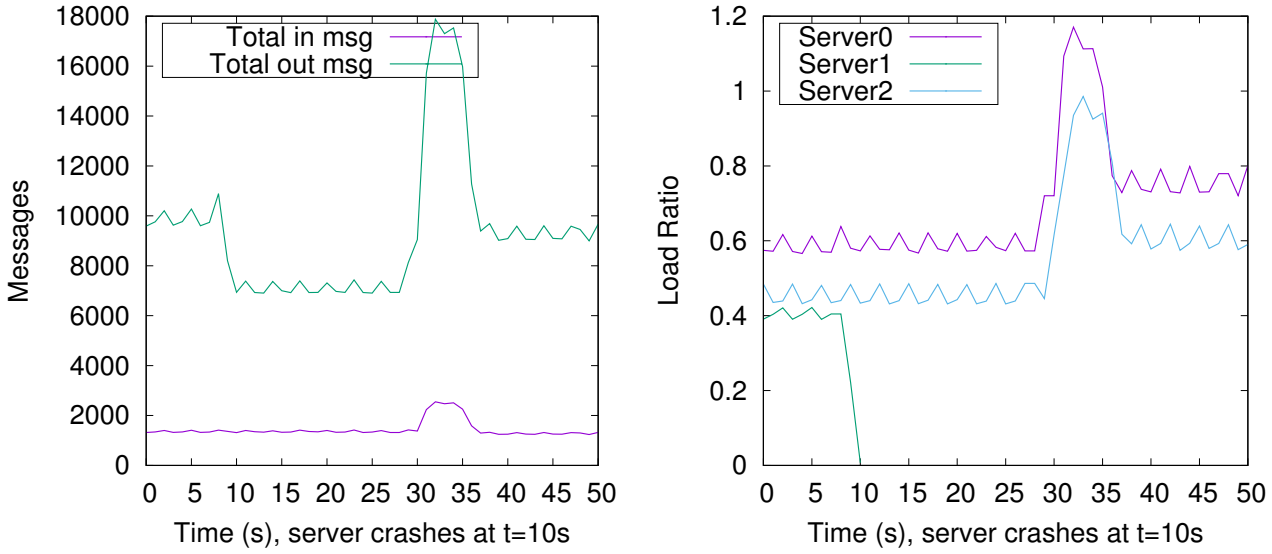


(a) Number of Players & Number of publish/subscribe Servers



(b) Average Response Time & Outgoing Messages

Figure 3.9: Handling a Varying Number of Players



(a) Total Incoming/Outgoing Messages

(b) Load Ratio on all Publish/Subscribe Servers

Figure 3.10: Availability: Messages and Load Ratio

3.7.4 Availability

This experiment aims at assessing the availability properties of Dynamoth. For each run, we started the system with up to 400 players and 3 pub/sub servers (0, 1 and 2) with a similar workload on 20 Amazon EC2 instances. About 1700 incoming messages per second were fed to the 3 pub/sub servers by all publishers, which were in turn delivered to multiple subscribers (about 10,000 outgoing messages per second - figure 3.10a) across 64 topics. The initial load ratio on each server was around 0.4-0.6, which meant that the outgoing bandwidth capacity on each server was used to 40%-60% (see the first seconds of figure 3.10b).

3.7.4.1 Failure Detection and Recovery

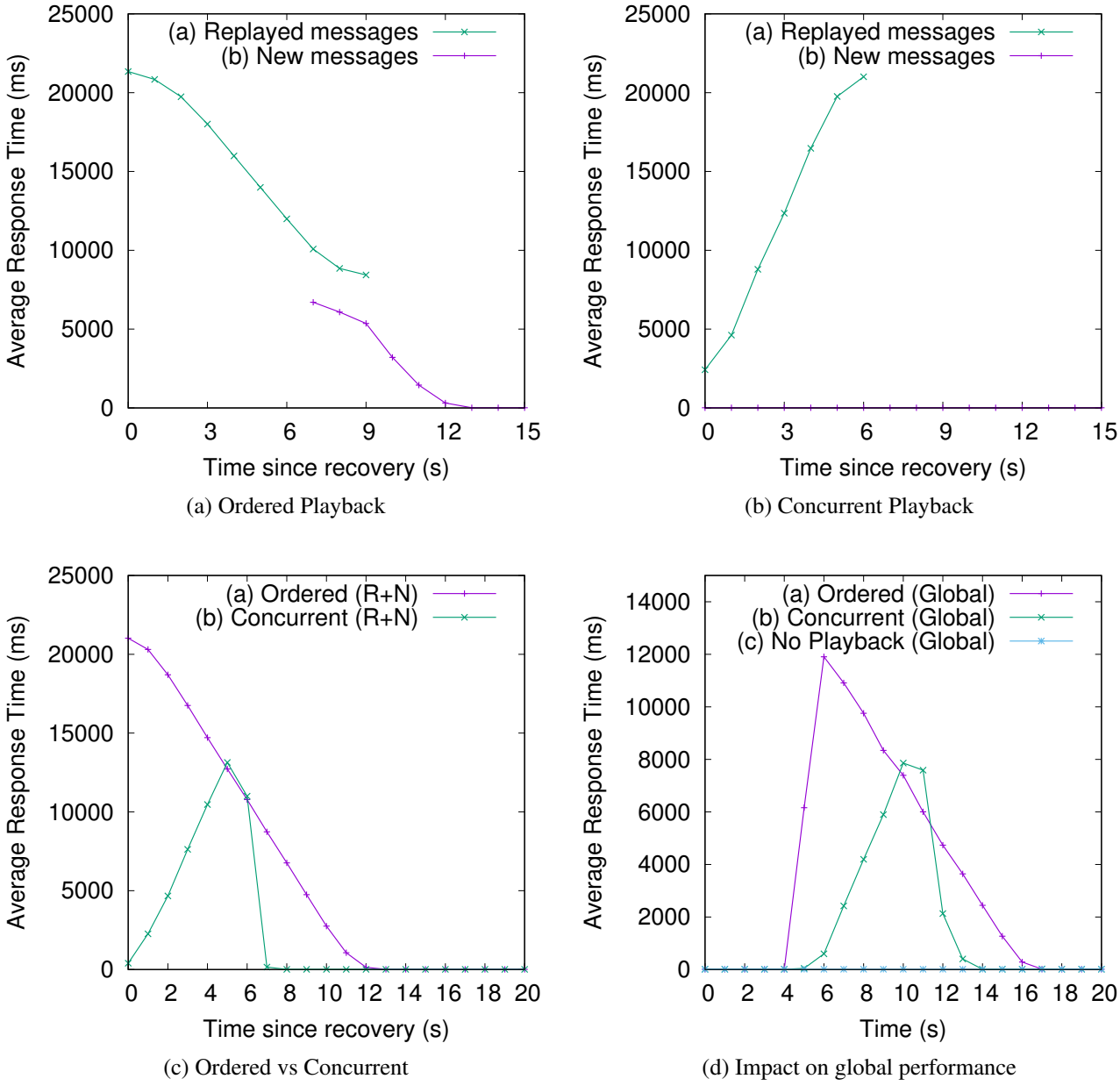
After approximately 10 seconds, server 1 crashes. On figure 3.10a, we can observe that the total number of outgoing messages drops by approximately 33%, but the number of incoming messages remains constant since publishers are still *attempting* to send the same amount of messages (they did not detect the failure yet). In figure 3.10b, we observe that server 1 becomes inactive (due to the failure); however,

servers 0 and 2 are still sending messages. Since this experiment was run in a controlled environment, after several trial runs, we determined that a failure detection time of 10 seconds ($T_{detect} = 10$) and a resubscribe time of 2 seconds ($T_{resub} = 2$) were adequate. To simplify the display of results, in this example, we made all players detect the failure near the end of the detection window (around 9 seconds after the crash, a bit before the expiration of $T_{detect} = 10$). Then, as described in section 3.5.6, they wait for $T_{detect} + T_{resub} = 12$ seconds before replaying their publications. Thus, playback occurs about 21 seconds after the crash, for all publishers (time $t = 31$ on the graph).

Consequently, on figure 3.10a, at $t = 31$ seconds, the amount of outgoing messages rises significantly (up to ~1800 messages per second) for a few seconds while old messages are replayed (assuming that message replaying is enabled for all topics), and then stabilizes again at 10,000 messages per second when the replaying of previous messages is completed. We observe that the incoming messages curve follows the same pattern. On figure 3.10b, we can see that the load ratio on servers 0 and 2 goes up since they "absorb" the load of server 1. During the recovery phase, the load goes up to ~1.20, which according to our experiments, was a safe maximum that servers could support without collapsing. In our experiment, we enabled the throttling feature of Dynamoth so that (1) message replaying did not overload clients and servers, and (2) processing of messages from other *non-failed* topics was not impacted. After message replaying completes, the load ratio of servers 0 and 2 stabilizes again, but at a higher level, (~70-80%), since their current load now includes the absorbed load of server 1. At this point, after recovery, the load balancer is restarted and can decide to run rebalancing operations, if needed.

3.7.4.2 Comparison of the different Guarantee Levels

In figure 3.11, we compare the different guarantee levels supported by Dynamoth: reliable/FIFO (ordered playback), reliable/NoOrder (concurrent playback) and best effort, as defined in sections 3.5.5 and 3.5.6. For all approaches, we monitored the average response times (latencies) at regular intervals (once per second). For subsections 3.11a, 3.11b and 3.11c, the x-axis corresponds to the time since resubscription, and we collected latency data only for messages sent across topics that have been remapped due to the failure. For figure 3.11d, the time starts a few second before recovery takes place and latency data was collected for all topics.



(a) Ordered Playback

(b) Concurrent Playback

(c) Ordered vs Concurrent

(d) Impact on global performance

Figure 3.11: Availability: Average Response Time Measurements

Reliable / FIFO (Ordered Playback) In figure 3.11a (Ordered Playback), we show average response times for (a) replayed messages only (sent while failing or during reconfiguration, but before resubscription, and had to be replayed) and (b) new messages only (sent *after* resubscription). The (a) curve shows that latencies are initially very high - over 21 seconds - which makes sense since the first messages that could not be delivered and had to be replayed were originally sent over 21 seconds ago, at the moment of the crash. Then, as more and more messages get sent at a faster rate (in order to quickly clear up the queue Q_r), response times go down. Between time $t = 7$ and $t = 9$, new messages start to be sent from some clients (curve (a)) concurrently to the sending of replayed messages by other clients, until a point where all clients only send new messages (curve (a) only). We observe that the first response time measurements for new messages are initially high, which is a consequence of the ordered playback approach where all new messages must be sent after all replayed messages have been sent, in order to preserve FIFO ordering guarantees. Eventually, at time $t = 13$, all queued messages (for both queues Q_r and Q_f) have been sent for all clients and response times return to their baseline level (recovery is completed and normal message processing is reenabled).

Reliable / NoOrder (Concurrent Playback) In figure 3.11b (Concurrent Playback), for (b) new messages only, latencies are always at their baseline threshold (ie. the curve is directly on the x-axis), since playback of new messages takes place concurrently to the playback of old publications. That also explains why new messages are sent as soon as subscriptions are reestablished (time $t = 0$). This figure demonstrates that enabling concurrent playback allows new publications to be sent without any additional delay (assuming that servers are able to handle the increased post-recovery load). For (a) replayed messages, we recall that Dynamoth replays the queue in reverse order in order to minimize response times for undelivered messages, since some messages at the head of the playback queue will probably already have been delivered before the failure. Reverse playback is reflected on the figure: messages initially have low response times, which slowly grows until a peak is reached, and then suddenly dropping to baseline values again (only new messages, curve aligned with x-axis).

Comparison of Ordered and Concurrent Playback Approaches Figure 3.11c compares the (a) ordered playback and (b) concurrent playback approaches for both replayed and new messages combined. We can see once again that the concurrent scheme leads to lower average response times and favors recent messages due to the concurrent playback of new messages and the replaying of missed messages in reverse order, which ultimately leads to a quicker completion of the failure recovery pro-

cess (at the expense of sacrificing FIFO ordering).

Impact on Global Performance of all Approaches Figure 3.11d shows a holistic view of the impact of all Dynamoth failure recovery mechanisms on the average response times observed for all messages sent through all topics, across all publish/subscribe servers. The (a) ordered and (b) concurrent curves exhibit similar trends as in figure 3.11c, with lower average response times, since the baseline response times observed for topics not impacted by the failure are also taken into consideration (this was not the case for figures 3.11a, 3.11b and 3.11c). Note that our throttling mechanism during playback is designed so that the replaying process does not impact the sending of messages across such topics. This figure also gives results for an experiment that we ran with the best effort approach (no delivery guarantee, since there is no playback of potentially missed messages, new messages are sent as soon as reconfiguration has completed). Figure 3.11d shows that the use of this recovery mechanism does not have any significant impact on the response times, since missed publications were simply not delivered and that the non-failed servers in our experiment were able to handle the increased load. As a consequence, the results for this experiment, which are illustrated by curve (c), are aligned with the x-axis since there was no significant performance impact.

3.8 Dynamoth Conclusions

We presented Dynamoth, our topic-based publish/subscribe middleware platform optimized for latency-constrained environments. Dynamoth uses independent publish/subscribe servers deployed in the cloud to handle the delivery of all publications in a broker-less way in order to minimize latencies. A major contribution is our hierarchical load-balancer which can perform rebalancings at the system-level (macro) and at the topic-level (micro). System-level load balancing enables our system to scale to arbitrary numbers of publishers, subscribers and publications in real time in order to adapt to the current load conditions. Additional publish/subscribe servers are dynamically allocated from the cloud when needed and removed when they are not required anymore. Topic-level load balancing (replication) allows our platform to handle special topics which exhibit high load patterns such as topics with extremely large numbers of publishers, subscribers and/or publications: such topics can be mapped to more than one publish/subscribe server. Dynamoth also proposes an elaborate propagation mechanism to notify all relevant clients of changes to topic assignments with very minimal impact on performance, while ensuring uninterrupted delivery of all messages. In addition, Dynamoth provides performance-

driven availability and failure recovery by proposing an approach to automatically recover failed subscriptions and publications in the event of a server failure. Finally, our failure recovery approach also provides different guarantee levels on a per-topic basis, such as the preservation of FIFO ordering if needed by the application.

We built an implementation of Dynamoth and ran extensive, large-scale experiments using a multiplayer game prototype as an application testbed. Our experiments reveal that Dynamoth is able to scale in an elastic manner as the number of subscribers, publishers and publications grow while maintaining low response time despite the very high variability in the workloads. When the load decreases, unnecessary resources are automatically released. Our results also revealed that Dynamoth was properly able to handle server failures and get back to a working state, while promptly recovering missed publications.

4

MultiPub: Latency and Cost-Aware Publish/Subscribe

The Dynamoth [53] system, described at the previous chapter (3), proposes a fully dynamic, scalable and available topic-based publish/subscribe service for cloud-based environments. While the service is tailored for general-purpose publish/subscribe applications, it nevertheless takes into consideration the needs of latency-constrained applications, notably due to its non broker-based flat architecture and its replication mechanism. However, Dynamoth's load balancing model was limited to scaling in terms of bandwidth. While preventing the over-saturation of servers certainly has benefits in terms of reducing response times, a limitation of Dynamoth in the context of latency-constrained applications is that it made no guarantees regarding publication delivery times (latency), in particular when clients are spread across the world.

MultiPub aims at solving this limitation of Dynamoth by proposing a fully dynamic global-scale topic-based pub/sub middleware, specifically tailored towards applications which require meeting strict delivery time bounds, such as multiplayer games, while attempting to optimize cloud-induced costs. MultiPub notably takes into consideration the fact that cloud providers have resources in several geographical regions.

4.1 MultiPub's Main Contributions

MultiPub provides two main contributions. The first major contribution is that it allows the user to set timing constraints on a per-topic basis, and ensures that such constraints are respected whenever possible. Depending on client locations, delivery bounds and cloud costs, a given topic can be managed by server(s) within a single or multiple cloud regions. As a second major contribution, MultiPub automatically finds the best configuration in terms of costs that does not violate delivery guarantees, and reconfigures whenever conditions change.

Our general contributions towards the optimization of global-scale publish/subscribe systems are summarized as follows:

- We provide a ready-to-use dynamic global-scale topic-based pub/sub middleware that can be deployed as a service in the cloud.
- Our approach takes advantage of cloud providers that offer resources in geographically dispersed regions and considers their specific characteristics (bandwidth-related costs and latencies), in order to generate an optimal configuration in terms of assigning regions to handle topics and clients. Two different message routing approaches are supported.
- Our system automatically collects and analyzes real-time measurements from all nodes and continuously reconfigures itself whenever a more appropriate configuration is found (more cost-efficient or more latency-minimizing). In the same spirit as Dynamoth, this analysis and reconfiguration process is entirely transparent to users of our platform.
- We have built a complete simulation package (MultiPubSimulator) to extensively evaluate our model under different scenarios.
- We have also built a prototype of MultiPub on top of Dynamoth [53], in order to evaluate MultiPub in a real cloud setting.
- We have gathered real latency measurements towards and between all regions of the Amazon EC2 cloud [5]. Combined with the King dataset, we derive realistic client latency values to feed both our simulator and MultiPub prototype.
- We have run several experiments to compare MultiPub against default, static pub/sub latency optimization approaches, demonstrating how it is able to reduce costs while meeting delivery

R	Region	Location	$EC2	$Inet
R_1	us-east-1	N. Virginia	0.02	0.09
R_2	us-west-1	N. California	0.02	0.09
R_3	us-west-2	Oregon	0.02	0.09
R_4	eu-west-1	Ireland	0.02	0.09
R_5	eu-central-1	Frankfurt	0.02	0.09
R_6	ap-northeast-1	Tokyo	0.09	0.14
R_7	ap-northeast-2	Seoul	0.08	0.126
R_8	ap-southeast-1	Singapore	0.09	0.12
R_9	ap-southeast-2	Sydney	0.14	0.14
R_{10}	sa-east-1	Sao Paulo	0.16	0.25

Table 4.1: EC2 Outgoing Bandwidth Costs

time bounds under various conditions. Real cloud-based experiments were run over multiple regions of the Amazon EC2 cloud, and results obtained from our full system implementation validated our simulation-based results.

To the best of our knowledge, MultiPub is the first attempt at proposing a pub/sub system that uses an optimization approach that considers both delivery times and cloud costs to determine cost-effective and time-constrained deployments in a global-scale setting.

4.2 MultiPub Model

MultiPub is a topic-based cloud publish/subscribe middleware that is tailored for latency-constrained, world-scale applications. MultiPub servers can be installed in as many cloud regions as necessary to serve clients. Given a set of regions, and a topic with its publishers and subscribers, MultiPub automatically determines which regions should be handling the topic considering that publishers and subscribers are located world-wide and have varying latencies towards each of the different regions. Inspired by Dynamoth's replication mechanism (section 3.2.4), MultiPub allows any given topic to be handled by more than one server; in this case, in different cloud regions.

The selection of which region(s) should be responsible for any given topic takes into account two factors. First, delivery times should be kept under a user-defined threshold, if possible. Second, since the use of the cloud incurs costs on a pay-per-use basis, MultiPub chooses, among the set of con-

figurations that can fulfill the delivery constraints, the configuration that is the cheapest in terms of cloud-related costs. Table 4.1 shows the 10 regions offered by Amazon EC2. Throughout this chapter, we use this region setup as an example for both our latency as well as our cost calculations. Note that MultiPub is not limited to cloud regions offered by a single provider; it could also work with regions offered by different providers [93].

4.2.1 Delivery Constraints vs. Cost Minimization

With MultiPub, a delivery time constraint can be specified for each topic T. The constraint specifies the maximum allowed delivery time (max_T) for a ratio of all publications received across T ($ratio_T$). For instance, $max_T = 200$ and $ratio_T = 95$ mean that 95% of all messages sent on T should be delivered within 200 ms or less. MultiPub makes sure that this constraint is respected, if possible. However, in some cases, it might be unrealistic. For instance, some clients might experience very high latencies due to the use of mobile or satellite connections, or the requested max_T threshold might simply be too low. If the constraint cannot be met, then MultiPub finds the most latency-minimizing configuration, irrespective of costs. A configuration for a topic T defines the regions which handle the topic, as well as for all publishers or subscribers, to which server(s) they should connect. The different configuration options are described at the next section (4.2.2).

If there is more than one configuration that fulfills the delivery constraint, then MultiPub chooses the one with minimal cost. The MultiPub cost model only considers bandwidth-related costs as this is by far the dominating factor. Given the simplicity of topic-based matching, the costs related to message dissemination are much higher than CPU costs. In contrast, this assumption could certainly be different for content-based systems which can be much more CPU-intensive.

In current cloud infrastructures, cloud inbound bandwidth is typically free, while there are different costs associated with outgoing bandwidth towards other cloud regions and outgoing bandwidth towards external clients. As the baseline for this paper, Table 4.1 details the costs in terms of outgoing bandwidth of the various EC2 regions. Column `$EC2` gives the costs of 1GB of outgoing data sent towards another EC2 region, while column `$Inet` lists the costs of 1GB of data sent towards any node on the Internet. We observe that the outgoing costs in some regions (Asia and South America) are very expensive compared to others. Thus, it might be worth avoiding to route topics through servers in regions with expensive bandwidth costs if the delivery constraints allow for such optimization.

4.2.2 Configuration Options

Supporting publishers and subscribers in different cloud regions, while ensuring that latency constraints are respected, can be challenging. Figure 4.1 presents three straightforward approaches which will serve as baseline and figure 4.2 shows our MultiPub dynamic approach. Assume the 10 Amazon EC2 regions and a topic T with 5 publishers, one each close to the regions R_1, R_3, R_5, R_8, and R_{10} respectively, and 5 groups of subscribers, each group being close to one of these five regions, and having 80, 5, 40, 25, and 2 subscribers respectively.

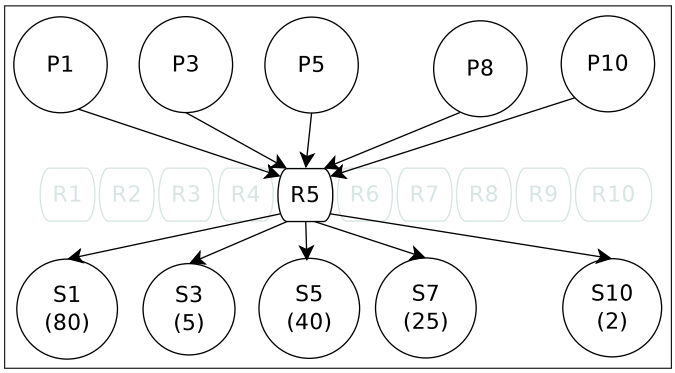
4.2.2.1 One Region

In the simplest case, the pub/sub middleware in only one region (figure 4.1a) is in charge of all publications and subscriptions on topic T. In this scenario, all publishers and subscribers for this topic connect to the service in this region. With this setup, some clients will experience very high latencies due to fact that some publications will travel for long and/or cross ocean distances twice. For instance, upon P_1 (close to region R_1, i.e., North Virginia) publishing to T, the publication needs to travel from P_1's location to the cloud region R_5 (long distance), where the pub/sub service then sends the publications to all subscribers. All subscribers except of those in group S_5 will receive the publication only after a long delay as they are far away from R_5. The advantage of this approach is, however, that it is less expensive, as R_5 is one of low-cost regions.

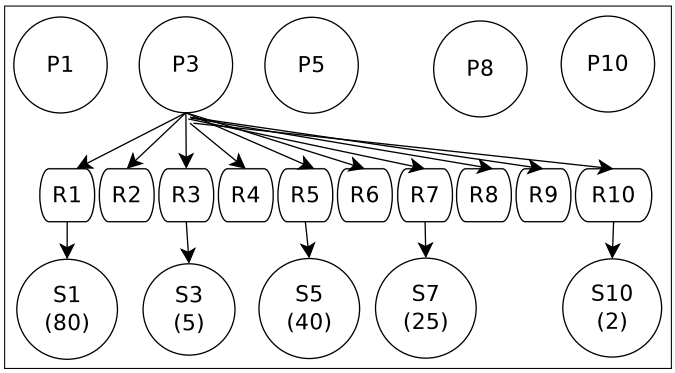
4.2.2.2 All Regions

This approach involves statically having pub/sub servers in all cloud regions handle topic T. This scheme allows for minimizing delivery times, since each subscriber automatically uses its closest, local cloud region (figures 4.1b and 4.1c). With the *direct delivery* approach (figure 4.1b), upon publishing, all publishers send all publications towards all regions (note that the figure only shows publications from one publisher to simplify). One problem of this approach is that all publishers must send their publications to all regions, which can be cumbersome, in particular as outgoing bandwidth might be a limiting factor for some publishers.

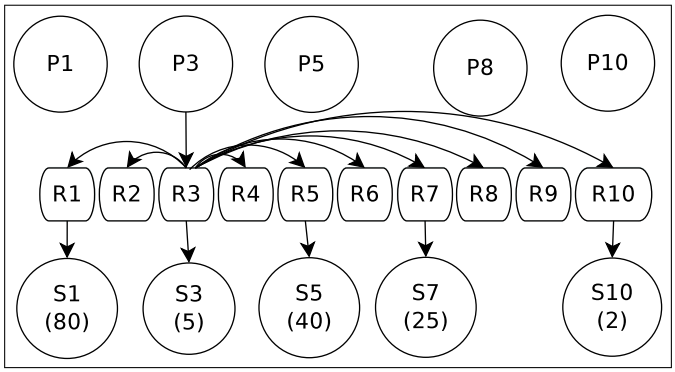
Routed delivery (figure 4.1c) is an alternative scheme where publishers only send towards their local, closest region (again, only one publisher is shown in the figure). The local region then forwards



(a) One Region



(b) All Regions / Direct Delivery



(c) All Regions / Routed Delivery

Figure 4.1: Typical Publication Delivery Approaches

the publication to the pub/sub service in all other regions. This scheme solves the issue of publishers publishing towards all regions, at the expense of increased delivery costs due to the additional outgoing, inter-region cloud bandwidth costs. Although the cost is usually smaller then sending to clients, it can still be significant.

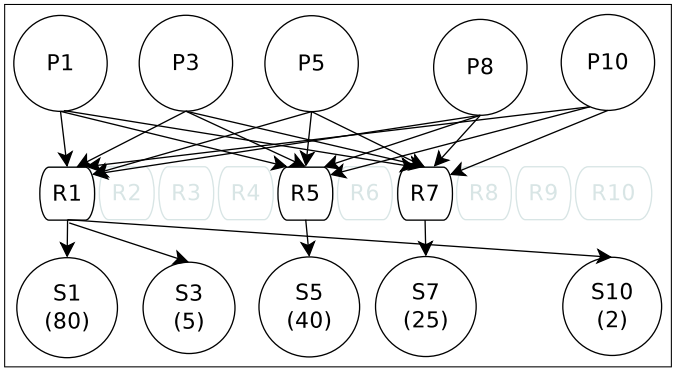
At first view it might appear that using the direct delivery approach will always yield lower latencies than routed delivery as messages always only travel two hops (from publisher to the service, and from there to the subscribers) while routed delivery has one additional hop (from cloud region to another cloud region). However, as inter-cloud links are often more optimized, the actual latencies can vary significantly, and thus direct delivery might have lower latency in some configuration while routed delivery can have lower latency in other configurations. Section 4.6.3.2 describes an experiment that compares direct against routed delivery.

An obvious drawback of the *all regions* approach is that this scheme forces all available regions to be used for topic T despite some regions having very few or no clients, thus consuming additional potentially unnecessary resources.

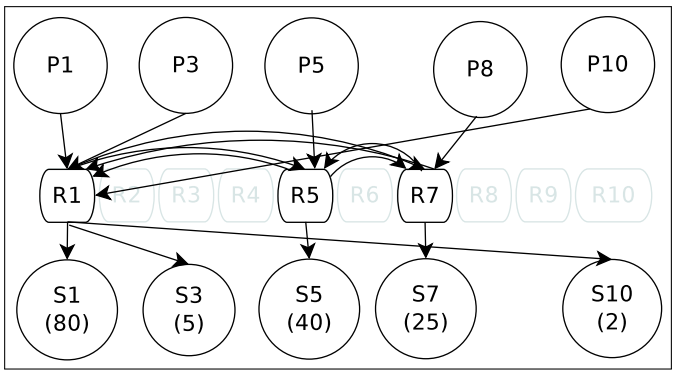
Furthermore, the approach is potentially considerably more costly than the one-region approach, as all regions are used, and some of them have considerably higher bandwidth costs than others. If delivery constraints are not violated, it would make sense to drop regions where bandwidth is very expensive (such as South America or Singapore).

4.2.2.3 MultiPub

MultiPub (figure 4.2) finds the best combination of cloud regions to use among all possible regions for topic T. MultiPub also determines whether direct or routed delivery should be used. Figure 4.2a shows an example of MultiPub with only 3 regions selected for topic T out of the 10 possible regions, using the direct delivery approach. Since there are very few subscribers in regions R_3 and R_{10}, MultiPub chooses to ignore these regions. Subscribers close to these two regions are assigned to region R_1 instead, since this region happens to be their second closest region. Figure 4.2b shows the same example, with routed delivery instead. Publisher P_3 publishes towards its closest available region (R_1 in Virginia, since R_3 in Oregon has not been selected by MultiPub for topic T). The MultiPub instance installed in R_1 forwards the publication to regions R_5 and R_8.



(a) MultiPub / Direct Delivery



(b) MultiPub / Routed Delivery

Figure 4.2: MultiPub Publication Delivery Approaches

4.3 System Architecture

The MultiPub cloud-based architecture, which is depicted in figure 4.3, spawns across multiple cloud regions, in order to be scalable and ensure that constraints are respected. In the following subsections, we describe the various architectural components of MultiPub.

4.3.1 Server Clusters

Each cloud region has an instance of the MultiPub service. In each cloud region, the pub/sub process matching itself can be performed by one server or by a set of servers, as this is the case with Dynamoth [53] (described at chapter 3). As such, MultiPub uses Dynamoth internally in each of the cloud regions, while MultiPub provides a global-scale overlay on top of the Dynamoth instances deployed in each region. In principle, we could use any scalable pub/sub platform that can be deployed in a single cloud.

We consider supporting multiple servers per region as an orthogonal problem since intra-region scalability can be managed locally, on a per-region basis, without altering the way in which MultiPub behaves. Dynamoth brings an adequate solution to this problem and offers an API that completely abstracts its inner working. Thus, we view each Dynamoth instance in each region as a black box, that can be abstracted from a conceptual point of view as a single server. Throughout this chapter, we assume, for simplicity, that there is a single server per region, and use the terms “service” and “server” interchangeable.

4.3.2 Assigning Regions to Topics

MultiPub allows topics to be handled by one or more regions. The mapping of topics to regions is expressed as a bit matrix, referred to as *assignment matrix*, where the columns are the regions and the rows are the topics. The row for topic T contains a 1 for each column representing a region that handles topic T, and a 0 for regions that do not handle topic T.

4.3.3 Region Managers

Each region has a *region manager* component that collects real-time data for every topic T maintained by the region including, the list of all publishers who publish to T, the list of subscribers who have

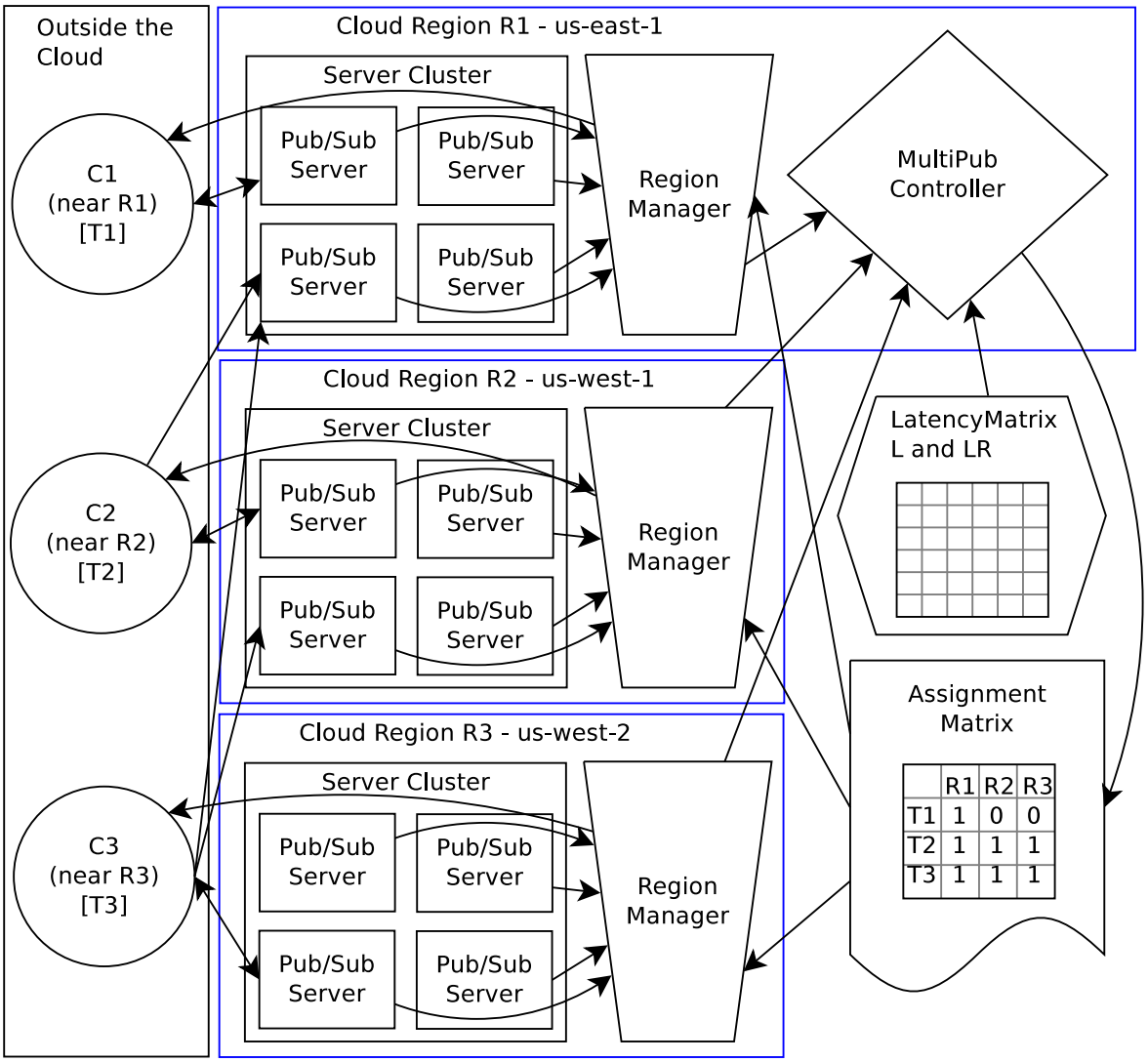


Figure 4.3: MultiPub Architecture

subscribed to T, as well as the number of messages and their sizes in bytes sent by each publisher over T. The design of the region managers and the statistics that they collect are inspired by the local load analyzers found in Dynamoth (section 3.3.1).

Data is collected throughout a collection interval, and then sent to the MultiPub Controller which can then determine overall delivery times and bandwidth costs over the last time interval. The collection interval is a configurable variable. It should be long enough to capture correctly the amount and sizes of messages sent by publishers, but short enough to not consider outdated information (for instance, publishers that stopped sending publications a while ago).

4.3.4 MultiPub Controller

The *MultiPub controller* is installed in one of the regions. It aggregates the data received from the region managers. Furthermore, it maintains for each topic T, the delivery constraint $< ratio_T, max_T >$ (where the $ratio_T$ percentile of messages must be delivered to subscribers within a time bound of max_T). Finally, it keeps track of the latencies between every client C in the pub/sub system (publisher or subscriber) and each of the cloud regions (L), as well as the latency between each pair of cloud regions (L^R) (both are described in section 4.4.2). These latency values are needed to compute an optimal solution. We will discuss later how we structure and maintain this information.

4.3.4.1 Solver

For each topic T, the *solver* subcomponent of the controller continuously recomputes an optimal configuration, considering as input the latest information about each topic collected by the region managers. If a better configuration as the current one is determined for a topic, the new configuration is sent in form of a bit vector to the region managers which then incorporate them into their assignment matrix.

4.3.4.2 Applying a New Configuration

The new configuration also has to be sent to clients of topic T (subscribers and publishers). If a region was added for this topic, and the new region is closer for some of the subscribers on T than any previous region of this topic, then those subscribers have to subscribe to T on the new region instead. Correspondingly, if a region was removed, then the subscribers that were currently connected to this

region must reconnect to the next closest region.

Similar holds for publishers with the routed delivery approach: they have to reestablish their connections if the region closest to them changed between the old and the new configuration. In contrast, with direct delivery, where it is necessary for publishers to publishers to all regions handling T, all publishers have to be informed as a configuration change means that the regions responsible for T have changed. Thus, the publishers have to send their publications to the new set of regions mapped to topic T.

For example, assuming a scenario with a topic T with 10 publishers and 10 subscribers in North America and 10 publishers and 10 subscribers in Europe, and only a server in Region R_1 assigned to T. The MultiPub controller determines that subscribers in Europe experience delivery times that are over the threshold for a significant portion of the publications that they receive (more precisely, for all publications sent from a publisher in Europe and received by a subscriber in Europe, as these messages cross the Atlantic twice). The controller then decides that T should now map to two regions: R_1 (us-east-1) and R_5 (eu-central-1) with a direct delivery approach in order to meet delivery constraints. All 20 publishers need to receive the new configuration so that they now send their publications to both regions, as well as the 10 subscribers in Europe as they have to resubscribe to the European region R_5. With this, any message crosses the Atlantic at most once.

4.3.4.3 Handling Configuration Changes

In our current implementation, for any given client C, the region manager in the region that is the closest to C in the old configuration has the responsibility of informing C of a configuration change. In the example above, this means, that the region manager of R_1 informs all publishers and all European subscribers of the change. One has to be careful that subscribers don't miss notifications when the configuration changes and before clients have successfully reestablished their connections to other regions. Dynamoth, that we use for scalability within each cluster, faces the same challenges when it comes to balancing the load among servers within each cloud region. Thus, the same reconfiguration mechanisms that are proposed in Dynamoth within the cloud (described in section 3.4), are used at global-scale for MultiPub.

Note that new clients can receive the current configuration for a topic through a simple lookup at any of the region managers. Also note that all *internal* system-related messages such as the sending

of the new configurations, the collection of data by region managers and the messages between region managers and the controller are sent using the standard MultiPub pub/sub interface over special system-related topics, and not through designated communication topics.

4.4 System Model

In this section, we describe our MultiPub pub/sub model more formally. In particular, assuming a particular configuration for a topic T, consisting of publishers, subscribers and a set of regions that handle T, we show how delivery times and the bandwidth costs over a given time period can be calculated. In the following sections, the descriptions apply to a single individual topic T, since topics are treated independently in MultiPub. The same computations can be done for any other topic. Also, we use the short term *region* to refer to the MultiPub service running in this cloud region.

4.4.1 Publishers, Subscribers, Regions and Publications

There are a total of N_R^{total} regions. Considering topic T, we denote with N_P the number of publishers for T, with N_S the number of subscribers to T and with N_R the number of regions that are assigned to T. $\mathbb{S} = \{S_1, \ldots, S_{N_S}\}$ is the set of subscribers for topic T, $\mathbb{P} = \{P_1, \ldots, P_{N_P}\}$ is the set of publishers for T, and $\mathbb{R} = \{R_1, \ldots, R_{N_R}\}$ is the set of regions serving T.

As discussed before, we consider both direct delivery where a publisher P sends a message to all regions in $\mathbb{R}$, and routed delivery where it sends it to only one region, denoted as R^P, who then forwards it to the other regions in $\mathbb{R}$. R^P is the region that is the closest (latency-wise) to P. All subscribers of topic T connect to only one region $R^S \in \mathbb{R}$, namely the closest. We denote with $\mathbb{S}^R \subset \mathbb{S}$ the subset of subscribers that use region R to receive publications on topic T, and with N_S^R the number of subscribers that subscribe to topic T on region R. Therefore, $N_S = \sum_{R \in \mathbb{R}} N_S^R$.

Given a publication message M sent by publisher P on a given topic T, M is sent to each of the regions $R \in \mathbb{R}$ (either by P directly or through R^P) and then each region R forwards it to all the N_S^R subscribers, leading to N_R messages towards all cloud regions handling T and a total of N_S messages towards all subscribers.

4.4.2 Latency Model

A subscriber S to topic T connects to $R^S \in \mathbb{R}$, which is the closest to subscriber S. The same case applies for publishers if routed delivery is enabled for topic T. To determine the qualifying region, we maintain a latency matrix L, where each row represents a client C (either a publisher or a subscriber) and each column a cloud region R. There is a total of N_R^{total} columns, which also include regions that do currently not serve topic T, since C might be using a one such region for any other topic than T, or the MultiPub controller might decide to add one such region to the list of regions assigned to T as part of a reconfiguration operation.

Entry L_{CR} indicates the expected one-way latency (message delivery time) between client C and region R (either direction). Thus, a subscriber S (publisher P) connects to $R^S \in \mathbb{R}$ ($R^P \in \mathbb{R}$) with the smallest $L_{SR^S}(L_{PR^P})$ value among all regions in $\mathbb{R}$. We assume L_{CR} to remain constant, but our model still holds if the value is updated over time at an infrequent rate. For instance, the infrastructure can monitor latency changes between every client C and every available region R periodically and update L_{CR} accordingly.

We also define a second latency matrix L^R, which represents one-way latencies between pairs of cloud regions. $L^R_{R_iR_j}$ denotes the latency between region R_i and region R_j. Obviously $L^R_{R_iR_i} = 0$. Results for the 10 regions of the Amazon EC2 cloud were determined and are presented in section 4.6.1.1. For simplicity (conceptually and from an implementation standpoint), we only considered the Amazon cloud, but our latency model could take multiple provider's latency metrics into consideration.

As mentioned previously, the MultiPub controller optimizes placement of topics on cloud regions by taking both L and L^R into consideration.

4.4.3 Publication Delivery Time

The total delivery time $D(M^{PS})$ of any given publication M sent from publisher P on topic T towards subscriber S depends on whether direct or routed delivery is used. For simplification purposes, we assume that the processing time of message M at any node is negligible which we believe is reasonable given that obtaining the list of subscribers for a given topic in topic-based publish/subscribe can be implemented as a simple lookup operation.

4.4.3.1 Expected Direct Delivery Time

Publisher P directly sends its publication M to all regions $R \in \mathbb{R}$ handling T, which then forward it to their local subscribers. Thus, for subscriber S connected to it's closest region R^S, delivery is done in two hops and is calculated using equation 4.1.

$$D_{Direct}\left(M^{PS}\right) = L_{PR^S} + L_{R^S S} \tag{4.1}$$

4.4.3.2 Expected Routed Delivery Time

Publisher P sends its publication M towards its local region R^P, i.e., the one which minimizes latency. R^P then forwards M to all other regions $R \in \mathbb{R}$ handling T. Each region then forwards M to their local subscribers. Thus, the delivery can be calculated using equation 4.2 and includes either two hops (for the subscribers with $R^S = R^P$, in which case $L^R_{R^P R^S} = 0$) or three hops.

$$D_{Routed}\left(M^{PS}\right) = L_{PR^S} + L^R_{R^P R^S} + L_{R^S S} \tag{4.2}$$

4.4.4 Publish/Subscribe Cost Model

As mentioned, the MultiPub cost model only considers bandwidth-related costs. In the following, we designate with $\alpha(R)$ the cost per outgoing byte from region R towards a different region (derived from column `$EC2` in table 4.1), and with $\beta(R)$ the cost per outgoing byte towards any client-subscriber of R (derived from column `$Inet`).

In order to calculate the overall bandwidth costs for our approaches, we need some additional information about the number of messages per collection interval and their size. Thus, for each publisher P we need to know not only the number of messages N^P_M published by P on topic T, but also the set of all individual messages $\mathbb{M}^P = \left\{M^P_1, M^P_2, \ldots, M^P_{N^P_M}\right\}$. Then, for each message M^P_j, we need to know its size $\Omega(M^P_j)$ in bytes.

With this, we can calculate the total costs Z_{Direct} for topic T using the direct delivery approach, given in equation 4.3.

$$Z_{Direct} = \sum_{k=1}^{N_P} \sum_{j=1}^{N_M^P} \sum_{i=1}^{N_R} N_S^{R_i} \times \Omega(M_j^{P_k}) \times \beta(R_i) \tag{4.3}$$

That is, for each publisher P, for each of its messages M_j^P and for each of the regions R_i serving topic T, the outgoing bandwidth is the size of the message multiplied by the number of subscribers using region R_i multiplied by the bandwidth costs per byte for this particular message.

The total costs Z_{Routed} for topic T using the routed delivery approach have to additionally consider that the region local to a publisher P forwards the message to all other regions serving topic T (of which there are $N_R - 1$):

$$Z_{Routed} = \sum_{k=1}^{N_P} \sum_{j=1}^{N_M^P} \sum_{i=1}^{N_R} N_S^{R_i} \times \Omega(M_j^{P_k}) \times \beta(R_i) + \sum_{k=1}^{N_P} \sum_{j=1}^{N_M^P} (N_R - 1) \times \Omega(M_j^{P_k}) \times \alpha(R^P) \tag{4.4}$$

4.5 Optimization Problem

For each topic T, the MultiPub controller determines on a regular basis the optimal configuration given the topic's delivery constraint $< ratio_T, max_T >$, the publishers and subscribers of T in the last observation interval, and the number and size of the messages sent by the publishers in that observation interval. To this aim, the controller has to consider every possible assignment (configuration) of the topic to the various regions. A configuration for T can be encoded as a bit vector: the bit for each region can either be set (topic assigned to the region) or not set. As this represents one row of the assignment matrix, we refer to it as assignment vector V for topic T. Thus, given that there are a total of N_R^{total} regions, then there are $2^{N_R^{total}} - 1$ possible assignments (it is not possible for all region bits to be zero since a region must be assigned to at least one region). Furthermore, for configurations where at least two bits are set, then the two delivery approaches described in section 4.2.2.3 become available: direct and routed delivery. Thus, there are a total of $2 \times (2^{N_R^{total}} - 1)$ -N_R^{total} possible configurations. Although this is exponential in the number of regions, the number of regions is typically fairly small.

For each of these configurations, MultiPub calculates what would have been the delivery times for

all messages sent in the last observation interval. From there, it calculates whether the configuration would have fulfilled the topic's delivery constraint $< ratio_T, max_T >$. If no configuration fulfills the constraint, then MultiPub chooses as next configuration the one that is closest to the requested constraint, that is the one which minimizes the lowest overall delivery time irrespective of the costs. If the delivery constraints can be fulfilled by one or more configurations, MultiPub chooses the one with the lowest costs. Thus, MultiPub will not always choose the most latency-optimal configuration; instead, it will choose *the most cost-effective configuration that respects the delivery constraint* if such a configuration exists.

Note that in the special case where no delivery constraints are given, then MultiPub chooses the configuration with the lowest costs irrespective of the delivery times. This case is useful if the application can tolerate high delivery times, for some topics. If there are several configurations with the same lowest costs, it chooses the one with the lowest overall delivery time. In the next subsections, we describe the individual optimization steps in more detail.

4.5.1 Checking for Delivery Constraint

This section describe precisely how MultiPub determines if a possible configuration G fulfills a given delivery constraint. Given configuration G for topic T (assignment vector V, and either direct or routed delivery), MultiPub first determines for each subscriber $S \in \mathbb{S}$ (and in case of routed delivery publisher $P \in \mathbb{P}$) the closest region R^S (R^P) for which the bit is set in V. With this, it calculates for each publisher P and subscriber S the latency $D(M^{PS})$ observed for sending a publication from P to S as expressed in equations 4.1 and 4.2 of section 4.4.3 for both direct and routed delivery approaches, respectively.

From there, MultiPub creates a list $\mathbb{D}_G$ which contains, for each publisher P and for each of P's messages sent in the last observation interval $\mathbb{M}^P = \left\{M_1^P, M_2^P, \ldots, M_{N_M^P}^P\right\}$, the delivery times: either $D_{Direct}(M_i^{PS})$ or $D_{Routed}(M_i^{PS})$ depending on the chosen approach. $\mathbb{D}_G$ is sorted by delivery time, with the shortest delivery time first. The cardinality $|\mathbb{D}_G|$ is the total number of messages exchanged between publishers and subscribers, i.e., the total sum of all messages sent by all publishers multiplied by the number of subscribers $N_S \times \sum_{k=1}^{N_P} N_M^{P_k}$. The delivery constraint for topic T requires that a percentile of $ratio_T$ messages be delivered in at most max_T time. This is equivalent to checking whether the n^T-ieth delivery time in the sorted list $\mathbb{D}_G$ is below or equal to max_T where:

$$n^T = \left\lceil \frac{ratio_T}{100} \times |\mathbb{D}_G| \right\rceil \tag{4.5}$$

We refer to this n^T-ieth delivery time as delivery time percentile $\mathring{D}_G$ for configuration G and it has to be lower or equal to max_T (equation 4.6). The following constraint has to be fulfilled for G to be further considered as a candidate configuration.

$$\mathring{D}_G = \mathbb{D}_G \left[n^T \right] \leqslant max_T \tag{4.6}$$

4.5.2 Bandwidth Costs for Topic T

Given a possible configuration G that fulfills the delivery constraint, the bandwidth cost Z_G is then calculated according to equations 4.3 and 4.4 of section 4.4.4, for each of the regions specified in the assignment vector V of G, as well as the delivery approach specified in G. The number of publishers, subscribers on topic T, as well as the set of publications issued by each publisher P ($\mathbb{M}^P$), are also taken from the last observation interval.

4.5.3 Determining the Optimal Solution

After having determined the delivery time percentile and bandwidth costs for each configuration G, we sort the configurations by costs. Then we take the configuration with the lowest cost that fulfills the delivery constraint. As mentioned previously, if there are several candidate configurations that have the same lowest cost, we choose the one that has the lowest delivery time percentile. If there are several configurations that also have the same delivery time percentile, then we choose the one that uses the least amount of regions. In the extremely unlikely event where multiple suitable configurations are found (same costs, same lowest delivery percentile and same number of regions), we choose a random one among them.

In contrast, if no configuration fulfills the delivery constraint, we take the one with the lowest delivery time percentile irrespective of its cost.

4.5.4 Independence of Topics

MultiPub minimizes costs for all topics, while respecting all constraints (whenever possible) for every topic. Minimizing the costs for every topic leads to a global minimization of the overall costs. Since there is no global constraint, or inter-topic constraints, all topics can then be considered as independent. Therefore, we have a distinct optimization problem for each topic. In other words, the outcome of optimizing one single topic will not impact the optimization of any other topic.

MultiPub does not consider the costs that are caused by changing the configuration, neither how it affects delivery times during reconfiguration, nor whether there are any cloud costs involved in the reconfiguration, as we do not expect changes to occur very often. Nevertheless, this might be an important consideration that is left for future work.

4.6 Experimental Validation

In order to evaluate the MultiPub model, we wanted to study its behavior and performance under a wide range of configurations. Unfortunately, running experiments across multiple cloud deployments is costly, as in addition to VM rental costs, outgoing bandwidth costs apply for data transferred towards clients and between cloud regions as listed in table 4.1. We therefore decided to simulate our large-scale experiments, which are described in subsection 4.6.3. To validate the accuracy of our simulations, we then built a full implementation of MultiPub that we deployed in the Amazon EC2 cloud. Using it, we ran a set of smaller-scale, real-world experiments in multiple regions to validate the trends observed in our simulation, while keeping the costs to a reasonable level. The real-world experiments are described in subsection 4.6.4.

4.6.1 Determining Latencies for Simulation

Since our formal modeling depends on the inter-cloud latency matrix L^R and on the client-to-cloud latency matrix L, we need to generate appropriate and realistic latency values. In the following subsections, we describe the methodology that we employed to generate reliable values for L and L^R.

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
R_1	-	33	42	41	45	74	92	109	115	60
R_2	34	-	11	74	84	52	68	88	79	93
R_3	42	11	-	70	79	45	60	81	81	92
R_4	41	74	70	-	10	107	122	105	155	96
R_5	45	84	79	11	-	117	135	126	161	99
R_6	73	52	45	107	117	-	17	38	52	128
R_7	89	68	60	122	133	17	-	34	67	144
R_8	109	88	81	97	126	38	34	-	88	164
R_9	115	79	81	155	162	52	67	88	-	171
R_{10}	60	93	91	96	100	129	144	165	171	-

Table 4.2: EC2 Inter-Cloud One-Way Latencies (in ms)

4.6.1.1 Inter-Cloud Latencies (L^R)

As explained previously, we have access to servers in all regions of the EC2 cloud (and we could have access to other clouds as well); therefore, we are able to measure latencies between machines in all pairs of regions and generate the latency matrix L^R.

We ran one $t2.micro$ virtual machine instance with Ubuntu in each of the Amazon EC2 cloud regions R_1 to R_{10} (the different regions are described in table 4.1). For each region R_i, we then measured the latencies using the Linux *ping* command to every other region R_j. We took 100 measurements, which we divided by 2, since *ping* yields the round-trip time. We repeated the experiment several times and observed that these measurements had only very small variations; since the inter-cloud EC2 latencies are stable, we simply use the average measured latencies for our simulation experiments as shown in Table 4.2.

4.6.1.2 Client to Cloud Region Latencies (L)

To determine realistic latency measurements between clients and cloud regions, we use the publicly available King dataset [57], which contains latency measurements between 1800 world-wide geo-distributed DNS servers.

Since the King dataset gives us a set of over 1800 IP addresses in different geographical regions, we attempted to measure the latency between each of the 10 cloud regions and each of the 1800

Figure 4.4: Geographical Distribution of the Live King Nodes

addresses. In order to reach that goal, we deployed a virtual machine in each of the 10 cloud regions, and performed a *ping* on each of the DNS servers in the King data set. From those 1800 machines, 700 responded. From those measurements, we built the latency matrix L. Since the King DNS servers are distributed across the globe (see figure 4.4), we can use their latency values to estimate client-cloud connection latencies from most places in the world.

4.6.2 MultiPubSimulator

We implemented a simulator named `MultiPubSimulator` in Python 3.5.1. MultiPubSimulator notably has access to the measured real-world latencies stored in the matrices L and L^R, as well as the outgoing bandwidth costs (towards the Internet and towards another EC2 region) for each of the EC2 regions (table 4.1).

`MultiPubSimulator` can run simulations with any number of topics. For each topic, the number of publishers and subscribers can be specified. Furthermore, for each publisher, a specific publication rate and publication size must be configured, as well as which of the Amazon EC2 regions the publisher machine should be geographically closest to. Using this latest criteria, MultiPub then draws

a set of appropriate publishers and subscribers from the pool of King nodes, whose values will be used to run the simulation.

Finally, for any given topic T, the upper bound in terms of maximum acceptable delivery time (max_T) and the ratio/percentile $ratio_T$ of all delivery time measurements that should be below max_T must be specified.

The software aspect of `MultiPubSimulator` is described in section 6.3. We invite the reader to read that section to get more details about the implementation of our tool.

4.6.3 Simulation Experiments

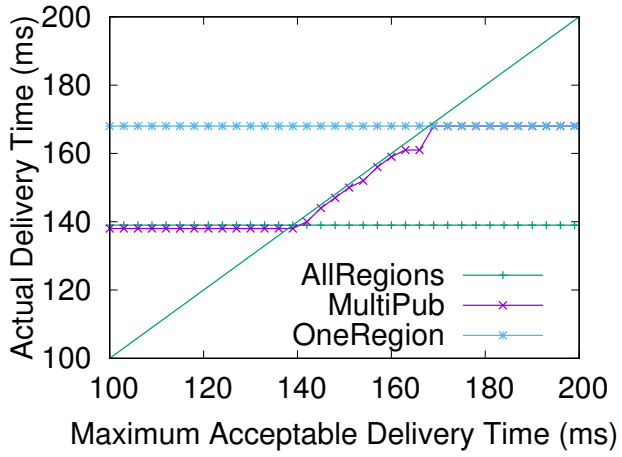
We used our MultiPubSimulator tool to run several simulation experiments in order to assess the correct working of MultiPub and to demonstrate that it can adapt to a wide range of scenarios, which are described in the next subsections.

4.6.3.1 Comparison MultiPub vs. Other Approaches

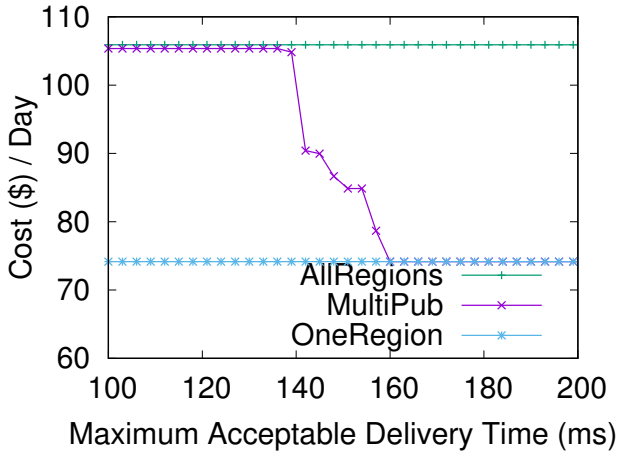
Experimental Setup The goal of this experiment is to compare how MultiPub compares to other approaches in a context where publishers and subscribers are distributed across the globe. We simulated one topic T with 100 publishers and 100 subscribers, where always 10 publishers and 10 subscribers are located close to each of the EC2 regions. Each publisher publishes on average once per second. Each message has 1 KByte

We compare MultiPub, where we vary the delivery time bound max_T between 100ms and 200ms, against a) the "All Regions (Routed Delivery)" model (see section 4.2.2.2), which should yield the fastest results because publishers and subscribers use the region that minimizes their delivery time, and b) the "One Region" model (see section 4.2.2.1), which should yield the most cost-effective results, because the publishers and subscribers choose the region that minimizes costs (and then delivery time). For all simulations, we set the delivery time guarantee ratio to 75%.

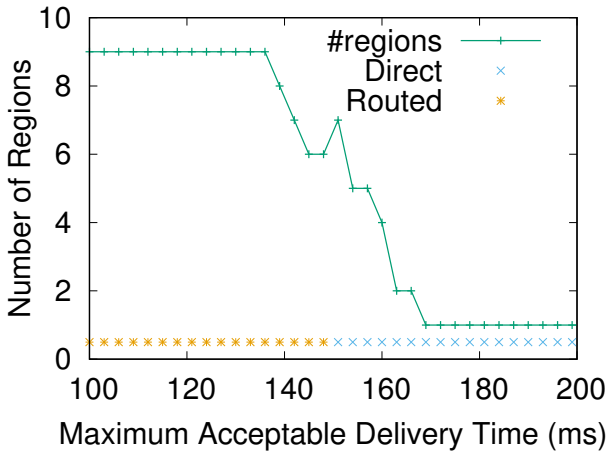
Results and Discussion The simulation results are shown in figure 4.5. We observe in 4.5(a) that the "All Region" approach is able to meet a delivery time bound of 140ms, while the "One Region" approach is able to meet a delivery time bound of 168ms. The MultiPub approach is capable of achieving the same, fast delivery time as the "All Regions" approach when $max_T \leqslant 140ms$. For higher



(a) Delivery Time



(b) Costs



(c) # Regions Used By MultiPub

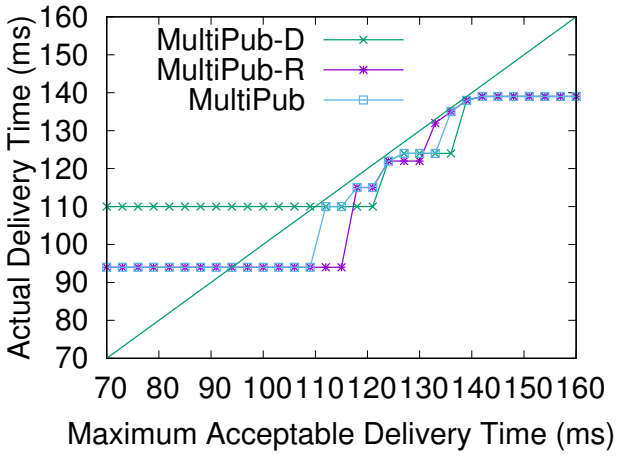
Figure 4.5: Comparison of MultiPub against other approaches - Global Scale

values of max_T, the actual delivery time increases, but remains under max_T until 168ms. Starting at 168ms, MultiPub aligns itself to the "One Region" approach. Figure 4.5(b) depicts the cloud cost calculated as if the test workload had run for a full day on the real cloud. We observe that the fast "All Region" approach is expensive ($107/day), whereas the "One Region" approach is 28% cheaper ($77/day). MultiPub selects the most cost-efficient approach that still meets the target delivery time. For $max_T \leqslant 148ms$, costs are as high as the "All Region" approach. Between 140ms and 168ms, MultiPub finds a wide range of intermediate configurations that meet the delivery time constraints but use less region servers. This is also visualized in figure 4.5(c), which plots the number of regions used by MultiPub, and whether publications are routed between cloud servers or only direct communication is used. Since inter-cloud links are generally faster (see measurements in subsection 4.2), MultiPub favors routed delivery even if it incurs additional forwarding costs. Eventually, MultiPub opts for direct delivery as it is less expensive. Finally, for $max_T \geqslant 168ms$, only one region is used, i.e., the cheapest one that minimizes delivery time.

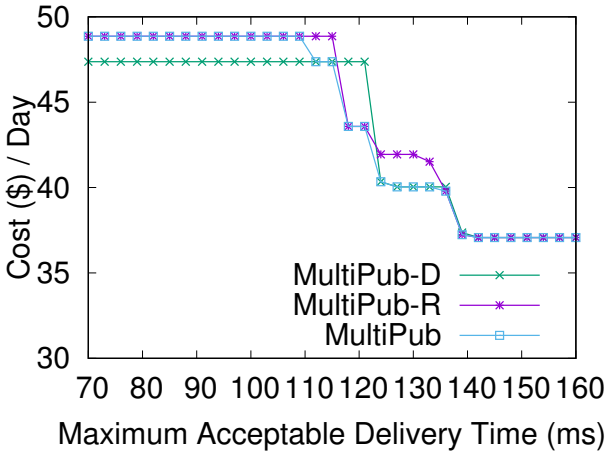
To summarize, this experiment demonstrates that MultiPub is able to generate costs savings, which in this specific experiment reached up to 28%, while still respecting delivery time constraints. Note that our cost numbers are for a single topic per day. Thus, overall gains over a longer period and more topics will be significant.

4.6.3.2 Comparison Direct vs. Routed Delivery

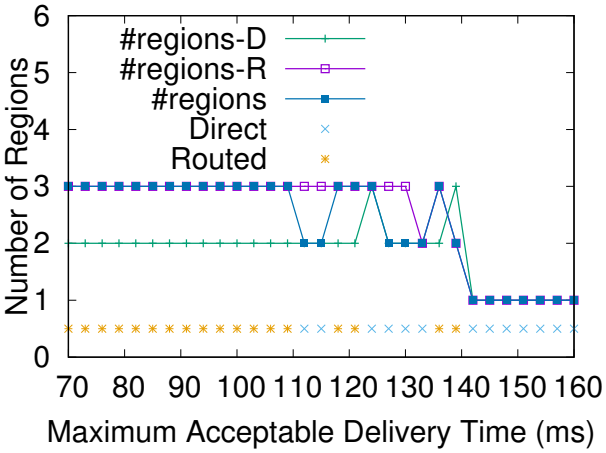
The goal of this experiment is to show that MultiPub is able to exploit both direct and routed delivery approaches in order to reduce delivery times and/or reduce costs. We deployed one topic T with 100 publishers and 25 subscribers in Asia, and 25 subscribers in the USA. cloud costs are again given for a 1-day period. The delivery time guarantee ratio was set to 75%. We ran 3 separate simulations, one with standard MultiPub, one where we allowed the solver to consider direct delivery only (MultiPub-D), and one where the solver had to use routed delivery (MultiPub-R). Figure 4.6a shows that the minimum reachable delivery time with "MultiPub-D" is 110ms, whereas it is 94ms with "MultiPub-R" due to the use of optimized inter-cloud links. Therefore, for $max_T \leqslant 110$ms, MultiPub uses routed delivery in order to meet the delivery time constraint, despite the costs being higher due to the extra inter-cloud communication (see figure 4.6b). For max_T values between 110ms and 138ms, MultiPub selects the best approach that minimizes costs depending on the desired delivery time bound. For values of $max_T \geqslant 138$ms, MultiPub chooses the direct delivery approach with only one server, located in the



(a) Delivery Time



(b) Costs



(c) Number of Regions and Delivery Scheme

Figure 4.6: Comparison of Direct and Routed Delivery

least expensive region that minimizes delivery time. Figure 4.6c plots the number of region servers used for all approaches, as well as symbols indicating when, for MultiPub, direct delivery and routed delivery is used.

4.6.3.3 Localized Pub/Sub Delivery across Different Regions

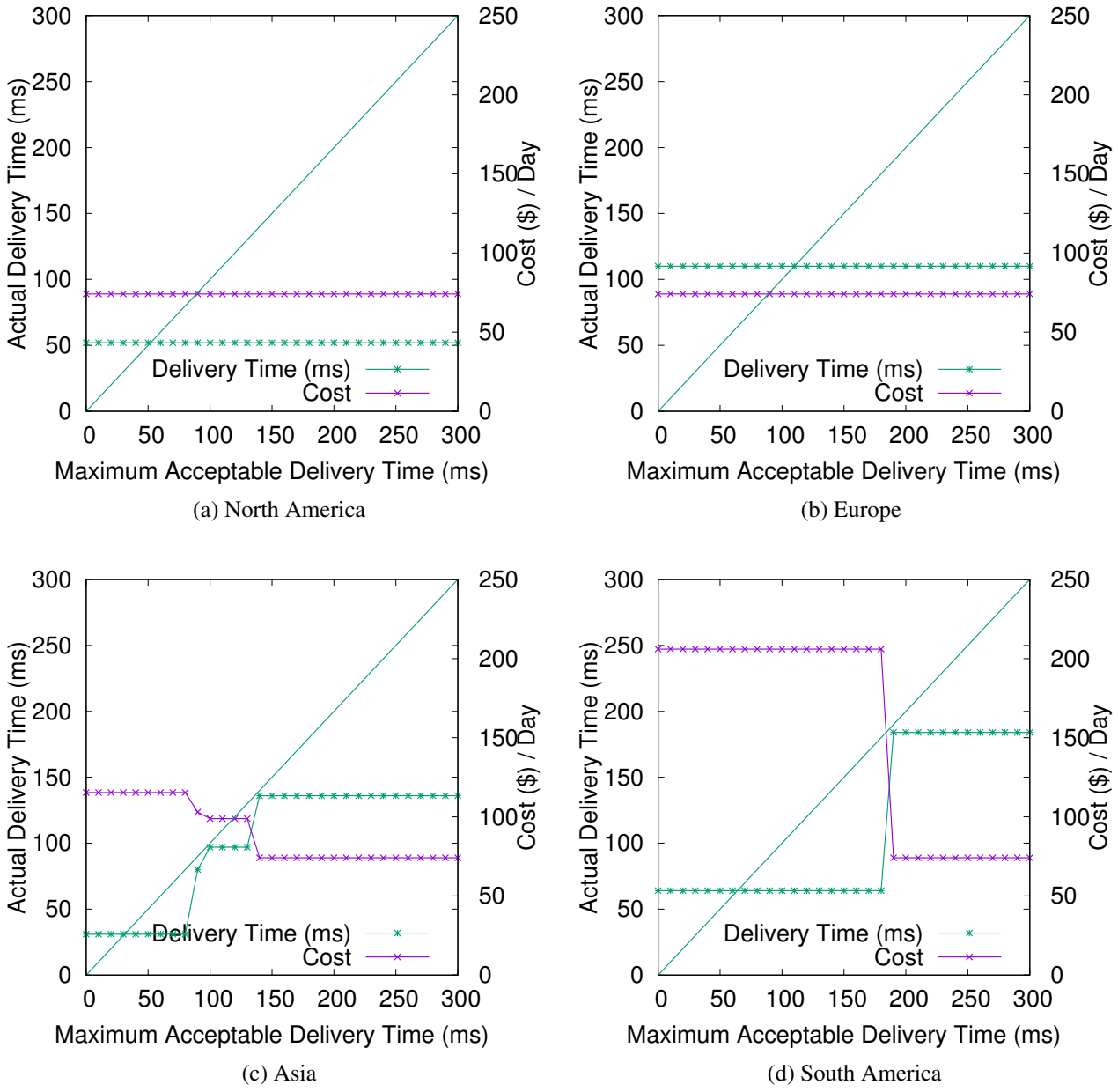
Experimental Setup In some contexts, publishers and subscribers are local to a region. For instance, in the context of a large-scale online game, players might decide to play against local players. Likewise, a country-wide weather alert system disseminates publications that are relevant only to subscribers that are located in the same country. In such a context, the straightforward approach is to deploy the relevant topics only in the local geographical region where the clients are located. However, such a configuration which would obviously yield the fastest delivery times, might not be the cheapest.

This experiment demonstrates that while MultiPub is designed with global-scale publish/subscribe systems in mind, it can also be useful in regional-scale scenarios to optimize costs. We ran the same experiment for four different regions: North America (EC2 region `us-east-1`), Europe (EC2 region `eu-west-1`), Asia (EC2 region `ap-northeast-1`) and South America (EC2 region `sa-east-1`). For each experiment on one of the regions R, 100 publishers and 100 subscribers were selected so that they were closest from a latency point of view to region R. The delivery time guarantee ratio was set to 95%.

Results and Discussion Figure 4.7 shows the results of the four experiments. For the North America and Europe experiments (figures 4.7a and 4.7b), the initial configuration never changes, since those regions have the lowest outgoing bandwidth costs (see table 4.1). For North America, delivery time constraints as low as 50ms can be met, whereas for Europe, only 110ms can be achieved.

In the Asia experiment (figure 4.7c), delivery times of 35ms can be achieved using the cloud region in Tokyo, but such a configuration incurs a high cost. With a relaxed delivery constraint of 80ms or more, MultiPub discovers cheaper solutions that use different clouds, until finally, for delivery time bounds of 145ms and above, MultiPub finds a configuration that uses a single European server (among the cheapest regions) to serve all clients in Asia. As a result, MultiPub significantly lowers the costs, achieving savings of 36% ($74 / day instead of $120/day).

In the South America experiment (figure 4.7d), the cost savings achieved by MultiPub are even



(a) North America

(b) Europe

(c) Asia

(d) South America

Figure 4.7: Localized Pub/Sub Delivery across different regions

more significant, since outgoing bandwidth costs in this EC2 region are the most expensive. MultiPub can meet a delivery constraint max_T of 60ms for 95% of the publication messages at a very high cost ($210 / day). For values of $max_T \geqslant 190$ms, MultiPub determines that an alternate configuration with servers in North America (Virginia) is suitable. In that case, costs for one day descend to $74, representing a saving of 65%.

4.6.4 Experiments in the Cloud

In order to run real-world cloud experiments, we implemented a prototype of MultiPub on top of Dynamoth [53]. The publish/subscribe interface is once again provided by the open-source Redis software [1]. Apart from demonstrating the feasibility of MultiPub, having a prototype makes it possible to run real-world experiments in the cloud to confirm that the simulation results correspond to reality.

As described in section 2.6, real-time multiplayer online games are perfect examples of distributed publish/subscribe systems in which message delivery time is critical to ensure fairness and playability, as well as provide a sense of immersion. We chose to reuse the RGame game prototype created as part of Dynamoth, which can be run with thousands of virtual players.

4.6.4.1 Experimental Setting

For logistic purposes and to limit incurred costs, we restricted our experiments to use the 3 following EC2 regions: `eu-central-1` (Frankfurt), `ap-southeast-1` (Singapore) and `ap-southeast-2` (Sydney). We deployed one pub/sub server (Redis) per region over one VM. Since we did not have access to client machines across the world, we had to deploy the subscribers and publishers in the cloud as well. We decided to distribute them over all EC2 regions on multiple VMs. In order to not get false latency measurements, we integrated latency emulation in the clients based on the King dataset, following an extension of the method described in Dynamoth (section 3.6.2).

On the client, the MultiPub library delays all received publications upon reception for a certain amount of time before delivering them to the application layer. For that to work, a virtual *king* node is assigned to every client, and the specific latency value between this virtual king node and every other client (also a virtual king node) in every region is taken into consideration. For communications in the same cloud, the delay is negligible, but for inter-cloud communications, a real physical delay applies, as shown in table 4.2. Thus, when it comes to emulating inter-cloud latencies, special care is taken to

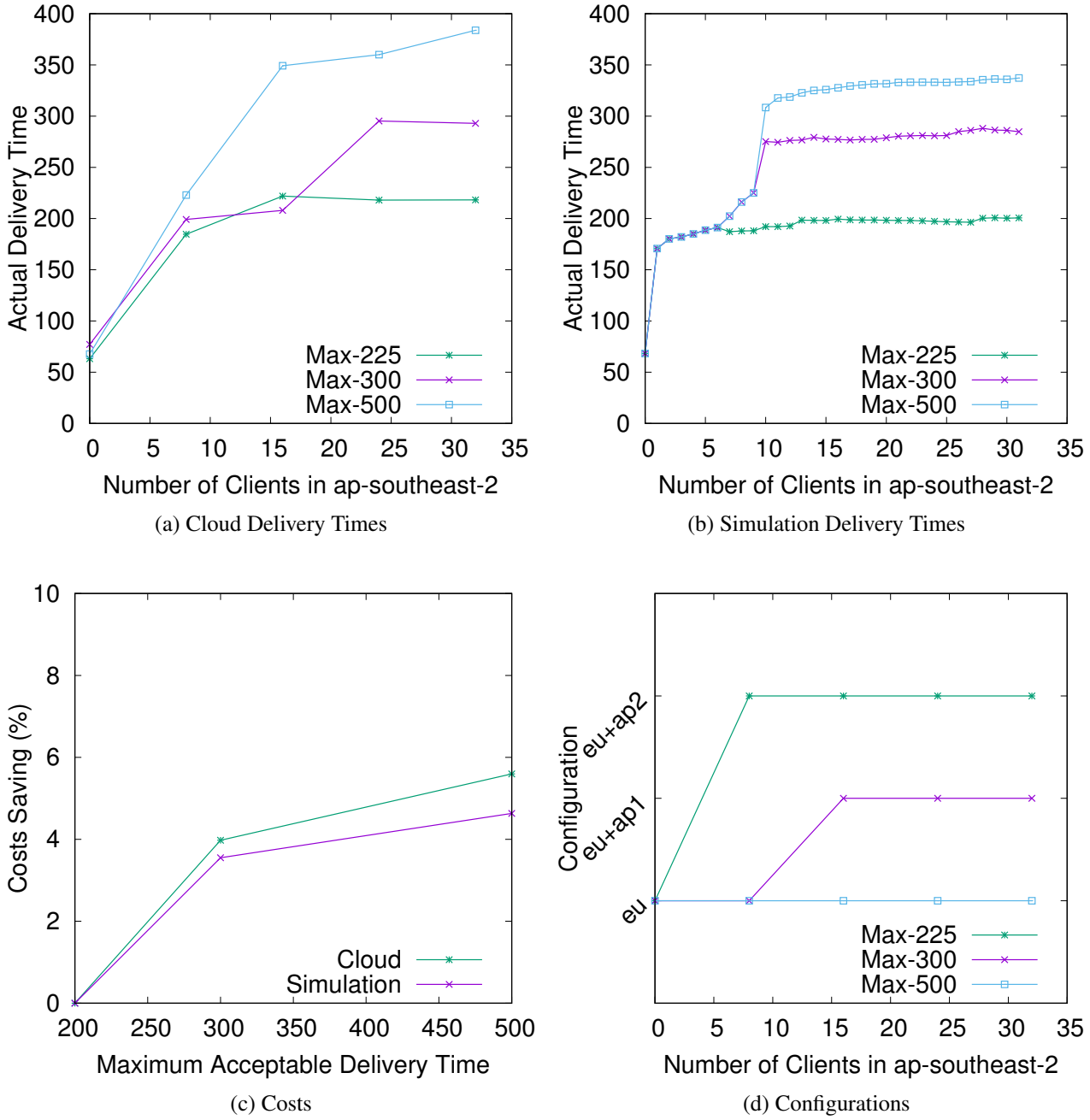
consider the real physical delay that may apply.

4.6.4.2 Medium-size Multiplayer Game

Experimental Setup The goal of this experiment is to assess the suitability of using MultiPub for update dissemination in a real-world, medium-size multiplayer game setting with players distributed over several locations. According to [40], first-person shooters and racing games require very low latency (<100ms), but 3rd person role playing games (RPG) or sport games, e.g., Madden NHL [81], can tolerate higher latency (<500ms). Finally, real-time strategy games (RTS) such as Warcraft can tolerate even higher latencies. Since it is currently not possible to achieve 100ms latencies world-wide, we set the parameters of the experiment to fit RPG, sport and RTS games.

Another important goal was to assess the capability of the MultiPub Controller to adjust dynamically to varying load conditions. Therefore, using a single topic $PositionUpdate$, we initially started RGame with 32 players simulated to be close to cloud region Frankfurt (region `eu-central-1`). For the needs of our experiments, those clients were therefore deployed on servers in the Frankfurt cloud. Players are both publishers and subscribers, as they publish position updates on average once per second, and they want to receive the position updates of the other players. Then, over time, 32 additional players located close to Sydney (region `ap-southeast-2`), and thus, installed on the Sydney cloud, join the game (8 at a time) to end up with a game of 64 players total. We ran the experiment 3 times with different maximum delivery time bounds: $max_T = 225ms$, $max_T = 300ms$ and $max_T = 500ms$. All experiments used a ratio of 95%. To validate our simulator, we also ran the same 3 experiments with MultiPubSimulator.

Results and Discussion Figures 4.8a and 4.8b compare the measured and simulated achieved delivery times, figure 4.8c plots the percentage of achieved cost savings when increasing maximum acceptable delivery time, and figure 4.8d shows which configurations are being used depending on the number of players close to Sydney. We observe that initially, for all configurations of max_T, MultiPub uses only the one region in Frankfurt, which is the least expensive (0.09$ per outgoing GB). For $max_T = 500$, MultiPub never needs to change the initial configuration, since the use of a server in cloud region Frankfurt allows for delivery times to remain below 500ms at all times. For $max_T = 300$, upon reaching 16 players in Sydney, MultiPub selects a configuration in which the game topic is handled by 2 cloud regions: Frankfurt and Singapore (`ap-southeast-1`). Even though the clients are



(a) Cloud Delivery Times

(b) Simulation Delivery Times

(c) Costs

(d) Configurations

Figure 4.8: Medium-size Multiplayer Game in the Cloud Results

located in Sydney, Singapore is chosen because it is cheaper (0.12$ per outgoing GB instead of 0.14$), and 95% of delivery times remain below 300ms. To guarantee $max_T = 225$ with already 8 players located in Sydney, MultiPub must opt for a configuration with regions Franfkurt and Sydney, at the price of additional costs.

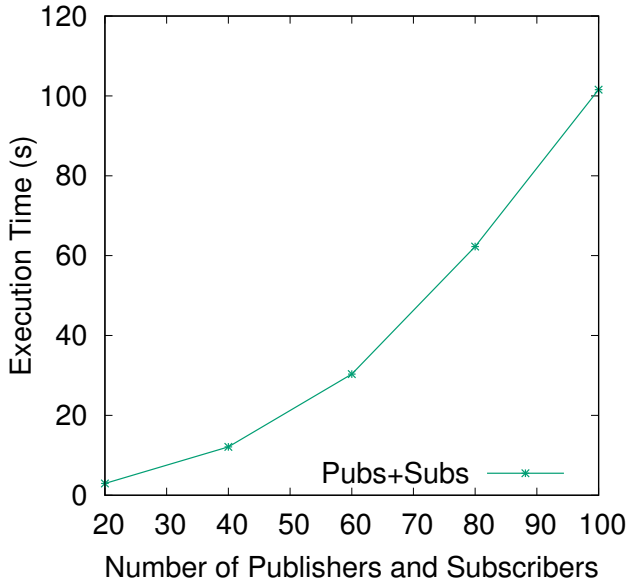
Note that the simulated delivery times in figure 4.8b correspond quite closely to the real-world delivery times that were collected from the cloud experiment runs. Also, the cost savings obtained in the cloud are slightly higher than the ones calculated in the simulation. This is due to the fact that in the cloud setting, additional costs were incurred because of the experimental data collection process, in order to obtain experimental results. In our implementation, we did not optimize this overhead and detailed information was sent from the region managers to the controller. Instead, the region managers can be optimized to send more aggregated information in order to reduce message overhead. Finally, the achieved savings do not appear to be very high in this specific experiment. This is due to the fact that most of the clients are in Frankfurt, which is the cheapest region. Other setups could lead to more important savings, as demonstrated by our simulation results.

4.6.5 Runtime Analysis

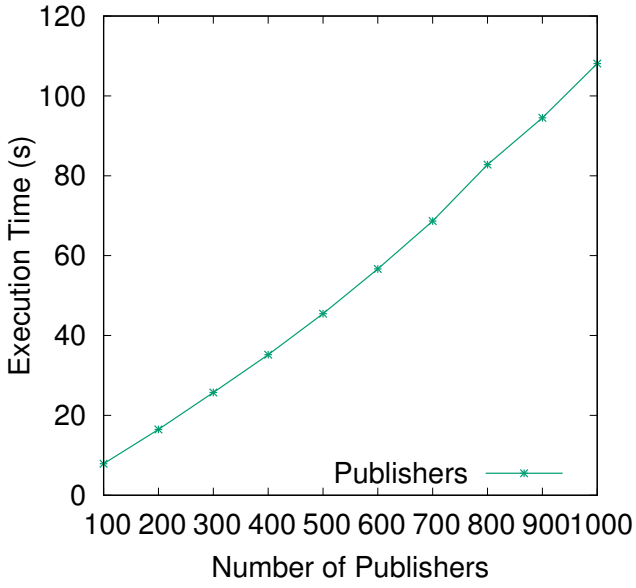
4.6.5.1 Experimental Setup

Determining the optimal solution is, as mentioned before, exponential with the number of servers. Furthermore, it is linear with the number of messages that have to be considered as we those messages have to be sorted, in order to determine the delivery percentile. In turn, the number of messages increases linear with the number of subscribers. Assuming that all publishers publish at the same rate, the number of messages is also linear with the number of publishers.

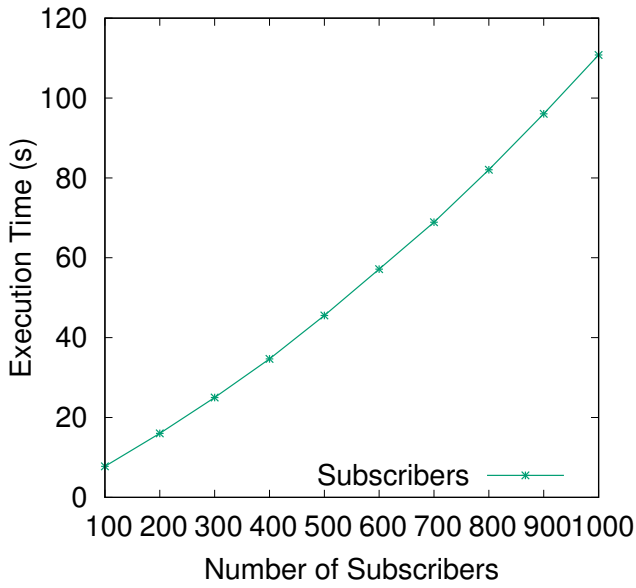
Figure 4.9 shows four experiments. The first experiment looks at the execution time for a 10-region system when both the number of publishers and subscribers increase to up to 100 (each publisher publishing once every second), showing that at 100 subscribers and 100 publishers, it takes a bit less than two minutes to determine the optimal configuration. Similar results are observed when we fix the number of subscribers to 10 and increase the number of publishers to 1000 (figure 4.9b), or when we fix the number of publishers to 10 and increase the number of subscribers to 1000 (figure 4.9c).



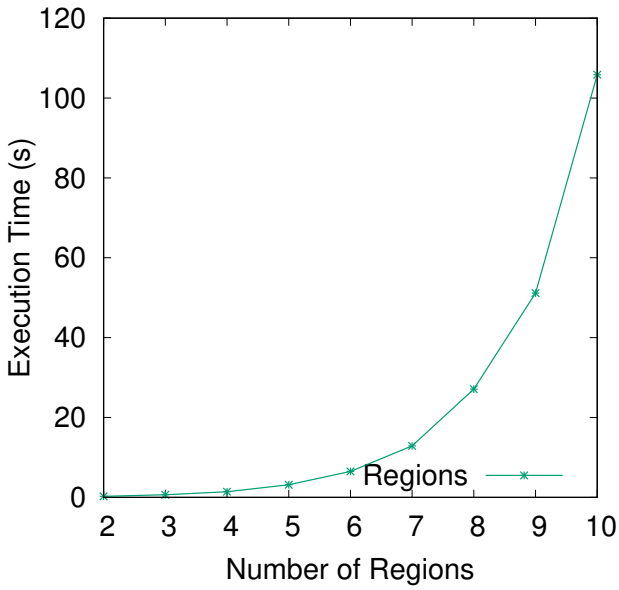
(a) Varying Number of Publishers and Subscribers



(b) Varying Number of Publishers



(c) Varying Number of Subscribers



(d) Varying Number of Regions

Figure 4.9: Runtime Analysis

4.6.5.2 Results and Discussion

Figure 4.9d shows the exponential influence of the number of regions by depicting the runtime of the solver for 100 subscribers and publishers when increasing the number of regions. With 5 available regions, it took the solver only 3 seconds to find the optimal configuration, that is, around 95% less time than with 10 regions. Considering only five regions, by extrapolation, a configuration for a system with 10 publishers and 35,000 subscribers could be determined in less than two minutes.

In summary, these performance measurements confirm that the optimization problem that Multi-Pub needs to solve can easily be solved for realistic settings. Our unoptimized, brute-force Python implementation can already handle a large amount of users, which is largely sufficient to adjust to load changes due to the arrival of new publishers and/or subscribers for any given topic.

4.6.5.3 Algorithmic Optimizations

To support larger scenarios, a more optimized implementation could be written. Also, different topics can be solved in parallel, as they are independent. To support extra-large scale settings, a defendable way to reduce solve time is to only consider a subset of cloud regions, which as we have seen, has an exponential impact on the search space. Simple pruning can remove expensive regions with no or very few subscribers, or regions very far from clients, for instance. Also, as the number of publishers and/or subscribers increases, clustering techniques could be employed in order to group clients that are close to each other, and replace them with a virtual client, in order to scale the problem to be considered.

However, past a certain point, clustering techniques can have their own scalability limits in the context of a linear optimization problem. Heuristic approaches could be designed to come up with a very good approximation of the best solution in a smaller amount of time. Another interesting optimization would be to follow a non-batch approach, such as an online algorithm, in which the solver would be able to generate solutions as it receives input (eg. changes to latency measurements or changes to the subscribers and publishers). Such approaches would definitely make sense in a highly dynamic context, such as in a game, as players frequently move around, thus modifying the set of subscribers and publishers for a large amount of topics on a recurrent basis. An online algorithm could then generate configurations faster after changes occur, and potentially without having to continuously reprocess the whole problem. Such algorithmic optimizations are definitely considered for future work.

4.7 MultiPub Conclusion

We presented MultiPub, a cost-minimizing topic-based pub/sub cloud middleware for latency-constrained applications with clients that are potentially distributed all over the globe that require strict delivery time bounds. Using the cloud brings many advantages, such as scalability and flexibility, but can also be costly due to the high outgoing bandwidth costs. MultiPub optimizes the routing of publications by taking advantage of the fact that cloud providers have resources in several geographical locations. Users of the service can define a delivery time constraint on any given topic, and MultiPub selects the optimal allocation of resources that respects the constraint while minimizing cost.

Part of our contributions consisted in defining a rich, realistic data model that notably takes into consideration the latencies between all cloud regions (on Amazon EC2) and between clients of the system (publishers and subscribers), as well as the bandwidth-related costs of all cloud regions. The model was generated using a combination of measurements and data from the open King dataset.

MultiPub makes use of this model to compute an optimal combination of cloud regions that meet the predefined delivery time constraints while minimizing cloud-related costs. More precisely, depending on the intrinsic characteristics of a given topic, MultiPub can assign the topic to one or multiple cloud regions, in order to reduce costs and delivery time. MultiPub also proposes two delivery methods and is able to select the most appropriate. MultiPub proposes a rich architectural model that integrates distributed data collection and analysis, as well as live reconfiguration, upon finding a more appropriate configuration, in order to adapt to highly-changing environments. The analysis and reconfiguration process is totally transparent to the user, as it was the case with Dynamoth. Furthermore, MultiPub runs on top of any existing unmodified topic-based pub/sub middleware, such as Redis, that we used in our experiments. As such, it can also be combined with load-balancing approaches that were proposed for addressing heavy load conditions in single-region pub/sub systems, e.g., Dynamoth [53].

In order to evaluate MultiPub, we built a full simulation package, as well as a real cloud implementation based on the Dynamoth platform. Both implementations make use of our data model. The simulator was used to evaluate our model under various conditions, while the cloud implementation was used to assess that MultiPub performs as expected in a real cloud setting on several regions of the Amazon EC2 cloud. Experiments revealed that MultiPub was able to significantly reduce cloud-related costs, while meeting delivery time bounds, compared to static pub/sub optimization approaches.

5

DynFilter: Limiting Bandwidth of Online Games using Adaptive Pub/Sub Message Filtering

Chapters 3 and 4 respectively presented our Dynamoth and MultiPub contributions to research in the field of scalable, reliable and optimized topic-based publish/subscribe systems. Dynamoth notably provided a scalable and reliable topic-based cloud platform, and MultiPub provided a cost-optimizing topic-based publish/subscribe system for latency-critical applications on a global scale.

In a similar spirit as Dynamoth and MultiPub, DynFilter [54] also proposes an optimizing publish/subscribe platform for the cloud, but tailored for the specific context of multiplayer games which typically require fast delivery times while generating high throughput and high publication frequency. While Dynamoth and MultiPub are more abstract and provide a general purpose publish-subscribe platform, their design was still motivated by the domain of online gaming, as many of the various experiments that were run on such systems were run in the context of a (massive) multiplayer game application (RGame). In addition, the fact that such systems were designed for latency-constrained applications in mind makes then perfect candidate for gaming-related applications.

DynFilter proposes a game-oriented topic-based publish/subscribe framework that aims at restricting bandwidth usage in such systems/games, in order to meet predefined bandwidth quotas, while minimizing the potential impacts on playability. This is accomplished by *filtering* (reducing the amount of) update messages exchanged between entities located far away from each other in the virtual game world. The rationale behind DynFilter is that it can be acceptable, considering a given player P, to

discard some of the state update messages transmitted from entities located away from P, without compromising playability from P's perspective. Our observations revealed that transparently discarding a portion of the state update messages can lead to significant reductions in bandwidth usage, and can lead to the following goals of DynFilter: (1) respecting a predefined cloud-based bandwidth-related budget and (2) preventing the game from becoming unplayable due to a bandwidth use that would be above the allocated resources.

DynFilter internally follows a tile-based interest management approach as described in section 2.6.2.2 and therefore maps game regions to tiles and players as publishers and subscribers to topics corresponding to relevant tiles (section 2.6.2.3). Our DynFilter implementation reuses Dynamoth internally (it is built as an extension on top of Dynamoth), and the game model is implemented as an extension to our RGame prototype game (DynGame) which was first developed for Dynamoth and later reused in MultiPub.

We consider that our work done on DynFilter and MultiPub is related and also complementary: while MultiPub aims at meeting latency constraints while optimizing costs, DynFilter aims at meeting bandwidth quotas, while ensuring acceptable performance/playability. Note that in the context of cloud systems, bandwidth usage correlates with cloud-incurred costs. Thus, both contributions have different optimization goals and strategies and one could see how both approaches could be used in different contexts, but nothing prevents one from integrating both approaches in the same context; ie. a game application could benefit at the same time from DynFilter's bandwidth reduction techniques while supporting global-scale applications and imposing delivery constraints on critical topics. Costs savings could then be combined: reduced bandwidth usage translates to less costs, combined with a strategical, cost-efficient assignation of workloads (topics) to cloud regions.

On the other hand, Dynamoth is also related as it can be leveraged to provide cloud scalability that applications using DynFilter and MultiPub might need.

5.1 DynFilter's Main Contributions

DynFilter notably provides the following contributions:

- Game operators can define a maximum target outgoing bandwidth that they are willing to allocate over a given window as well as a maximum filtering (degradation of quality) that is allowed for

each tile.

- Filtering is only applied to subscribers on remote tiles: state updates from players/entities located in the same tile (or group of tiles) are always delivered, which means that any given player will receive all state updates from nearby players.
- A *load analyzer* module continuously monitors the pub/sub server with minimal overhead and analyzes the bandwidth that has been used in the current window. If needed, the *load optimizer* applies adaptive message filtering to reduce the amount of messages that need to be disseminated, in order to stay below the target bandwidth.
- Our algorithmic model automatically adapts filtering for each game tile based on the number of subscribers in the tile. Filtering is continuously recomputed.
- DynFilter is completely transparent and non-obtrusive to game players.

5.2 DynFilter Architecture

5.2.1 Tile-based Area-of-Interest and Message Delivery

As mentioned, in DynFilter, the game world is divided into a set of interconnected square tiles[1]. Assuming a world grid made of X columns and Y rows, we have a total of XY tiles labeled as follows: $T_{x,y}$ where $x \in \{0, \ldots, X-1\}$ and $y \in \{0, \ldots, Y-1\}$. Considering that a given player P is located in one and only one tile T_{x_p,y_p} at any given time, we define the subscription range Z as how many tiles *around* the player's current tile P subscribes to in order to receive updates from other players and in-game entities. Formally, P receives updates in all surrounding tiles $T_{x,y}|x \in \{x_p - Z, \ldots, x_p + Z\}, y \in \{y_p - Z, \ldots, y_p + Z\}$ (within a distance of Z).

DynFilter makes sure that P always receives all state updates in it's own tile (T_{x_p,y_p}). For surrounding tiles, state updates can be filtered if needed. The impact of such filtering is greatly mitigated by the fact that players and entities located within these tiles are located farther apart from the player. Games typically make use of dead reckoning techniques [105, 83] to interpolate player positions between state updates. The inaccuracies in on-screen player positions (difference between dead-reckoned posi-

[1]Square tiles have been chosen because they simplify our spatial model. However, our model can easily be adapted to other tile configurations. For instance, if using triangular tiles, one could simply index each tile, and transform the two-dimensional tiling model proposed into a one-dimensional model.

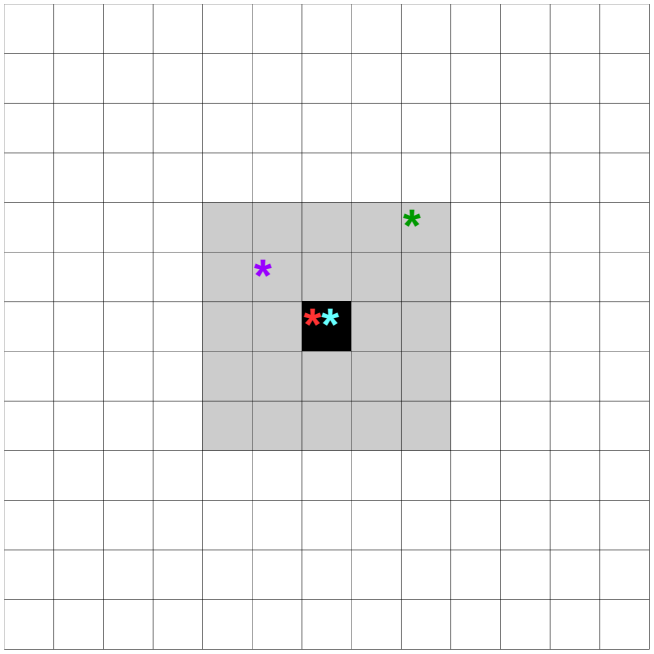


Figure 5.1: DynFilter Tiles Example

tion and real position) will appear smaller for entities located farther away. Figure 5.1 gives an example of which tiles a given player located in the dark tile will be subscribing to (in that case, $Z = 2$): the black tile represents a subscription to it's own tile (unfiltered) and the grey tiles represent a subscription to surrounding tiles (can be filtered). Players are denoted as small dots.

All game-related messages are delivered using a topic-based publish/subscribe middleware, such as Redis. In order to provide scalability, one could reuse Dynamoth as it also provides a topic-based pub/sub interface. Alternatively, if the game is run in a global-scale setting with players distributed throughout the world, then MultiPub would also be a good candidate.

5.2.2 Architectural Components

The DynFilter architecture is made of several distributed components. A high-level overview is presented in figure 5.2, in a cloud setting. A virtual machine (VM_Server) contains an instance of the pub/sub middleware, coupled with a data collector component whose goal is to collect real-time data about all topics that currently exist on the pub/sub server, in a non-obtrusive way. The data collector

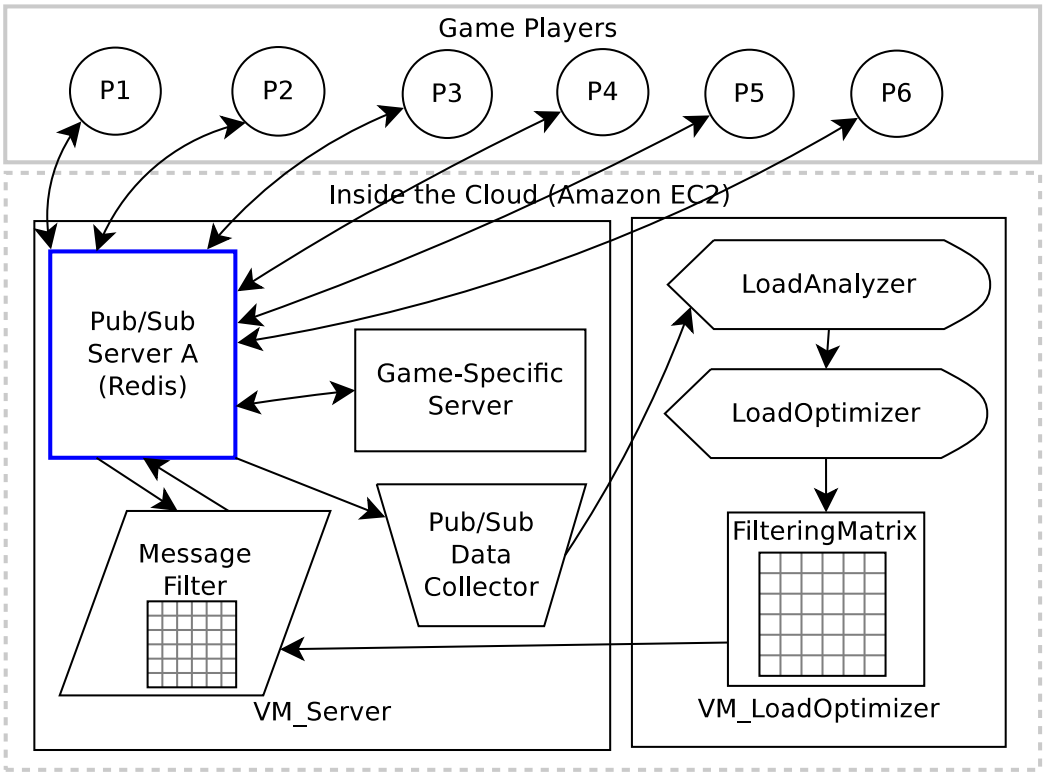


Figure 5.2: DynFilter Architecture Overview

component is conceptually similar to Dynamoth's local load analyzer (section 3.3.1) and MultiPub's region manager (section 4.3.3) components.

Aggregated data is periodically transmitted to the *load analyzer* module, which is located on a different VM, but in the same cloud to reduce bandwidth overhead and costs (VM_LoadOptimizing). The *load analyzer* module determines if the allocated bandwidth quota for the current time period will be respected, based on previous bandwidth use and based on aggregate data received by the *data collector* (please refer to section 5.3.1). A *load optimizer* module then computes a new *filtering matrix* (described in section 5.3.2), which is transmitted to the *message filter* component, located on the same VM as the pub/sub server. The filtering matrix, which will be discussed later, is used to inhibit the delivery of some of the publications.

5.2.3 Message Filtering

In order to implement the described filtering behavior, DynFilter generates two pub/sub topics on the pub/sub server for each tile $T_{x,y}$, as follows: (1) $T^H_{x,y}$ (high-frequency, without filtering) and $T^L_{x,y}$ (low-frequency, where filtering can occur). A given player P in tile T_{x_p,y_p} subscribes to topic $T^H_{x_p,y_p}$ (receive all state updates in it's own tile) and subscribes to topics $T^L_{x,y}|x \in \{x_p - Z, \ldots, x_p + Z\} \backslash x_p, y \in \{y_p - Z, \ldots, y_p + Z\} \backslash y_p$ (receive potentially filtered updates for all surrounding tiles). P always publishes to $T^H_{x_p,y_p}$. Therefore, no game entity or player directly publishes to any low-frequency $T^L_{x,y}$ topic.

DynFilter's message filter component is in charge of forwarding some or all of the publications from high-frequency topics ($T^H_{x,y}$) to low-frequency topics ($T^L_{x,y}$). It accomplishes that goal using the latest available filtering matrix $F_{x,y}$ that has been computed by the cost optimizer. In our model, $F_{x,y}$ is a 2-dimensional array that contains a filtering ratio for each tile $T_{x,y}$, between 0.0 (all messages are forwarded to $T^L_{x,y}$ - no filtering) and 1.0 (no message is delivered - this is not desirable so in practice, we provide an upper bound). The computation of $F_{x,y}$ is described in section 5.3.2.

The message filter component subscribes to all high-frequency tile topics ($T^H_{x,y}$), which does not incur additional network overhead since it is local; that is, on the same machine as the pub/sub server. For each high-frequency update at tile T_{x_0,y_0}, it does the following:

1. Obtain the filtering ratio F_{x_0,y_0} from the filtering matrix $F_{x,y}$;
2. Generate a random floating-point number between 0 and 1;
3. If the generated number is greater than F_{x_o,y_o} then
4. Forward the publication to $T^L_{x_0,y_0}$ (if the generated number is smaller than F_{x_o,y_0}, then the message is not forwarded; thus, subscribers of $T^L_{x_0,y_0}$ will not receive it).

5.2.4 N-Layered Message Filtering

The DynFilter architecture is two-layered: the first layer (high-frequency) ensures full delivery of all messages and the second layer might allow for partial delivery. While it is possible to alter the various parameters such as the subscription range Z and the size of the tiles, for some games requiring finer granularity, DynFilter could be extended to introduce additional layers of message filtering to allow for a partial degradation in the amount of state updates received as the distance grows. When playing

at very high resolutions on large displays, players might want to be able to see objects located very far away. N-Layered filtering could be used to allow players to view remote objects at a very low update frequency. Such an improvement is left as future work.

5.3 Cost Analysis and Optimization

The main goal of the DynFilter load optimization process is to make sure that the allocated bandwidth quota for the current time period is respected by filtering game update messages sent to subscribers of low-frequency tile topics. The following subsections describe in a detailed fashion how the current load is analyzed and how the filtering matrix $F_{x,y}$ is updated.

5.3.1 Load Model & Analyzing

DynFilter defines the concepts of a *time unit* and a *time period*. A bandwidth quota (B_{quota}) is allocated for a given time period (10 minutes in our experiments) and is made of $t_{\max}$ time units (20 seconds in our experiments). B_{quota} is defined by the game operator (possibly by taking into account the outgoing cloud bandwidth costs or the capabilities of it's current infrastructure). At every time unit t, the bandwidth that has been consumed since the beginning of the current period (B_{used}) is evaluated by the load analyzer. The load analyzer then computes the bandwidth that is *remaining* until the end of the period (equation 5.1).

$$B_{\text{remaining}} = B_{\text{quota}} - \mathrm{B}_{\text{used}} \tag{5.1}$$

From the remaining bandwidth, a *target* bandwidth allocation is then computed for the next unit, which is the average amount of bandwidth that the game should consume in all remaining time units throughout the end of the period, in order to respect B_{quota}. It is defined at equation 5.2.

$$B_{\text{target}} = B_{\text{remaining}}/(t_{\max} - t) \tag{5.2}$$

The next step is to determine if, by consuming bandwidth at the *current* rate, the game would go over B_{quota}. To do so, the load analyzer first considers the bandwidth that has been consumed in the

last unit (B_{prev})[2]. If $B_{\text{prev}} \leqslant B_{\text{target}}$, then filtering can be reduced or canceled if it is no longer needed, since we are *currently* using less bandwidth than allowed. However, if $B_{\text{prev}} > B_{\text{target}}$, then we are *currently* using too much bandwidth, and we need to lower bandwidth use. We define B_{remove} as the bandwidth that we need to remove in the next time unit as follows: $B_{\text{remove}} = B_{\text{prev}} - B_{\text{target}}$. By knowing how much bandwidth we need to remove, the load optimizer then computes an appropriate filtering matrix, as explained at the following section.

5.3.2 Load Optimization

The filtering ratio $F_{x,y}$ for tile $T_{x,y}$ was previously defined as the ratio of messages that should not be delivered to $T^L_{x,y}$. In this section, we describe how this ratio is computed, with two approaches: (1) trivial filtering, where the ratio is the same for all tiles and (2) DynFilter filtering, where a different ratio is computed for each tile. The latter takes into consideration the *density* (number of players) in the tile.

5.3.2.1 Trivial filtering

Assuming for simplicity that there was no filtering in the previous time unit, by knowing the number of bytes to remove (B_{remove}) as well as the number of bytes consumed in the previous time unit (B_{prev}), we can compute one global filtering ratio F for all tiles as follows: $F = \frac{B_{\text{remove}}}{B_{\text{prev}}}$. This holds if there was no prior filtering in place. If there was already filtering in place, then we need to compute an *extrapolation* of the bandwidth that would have been consumed in all low-frequency tiles over the last unit if no filtering was in place, by inverting the effects of the filtering already in place, following a similar process as described in equation 5.3. We would then obtain an extrapolated version of B_{remove} and B_{prev} which would yield an accurate computation of F.

We want to go beyond trivial filtering and consider the specifics of each tile as per the following rationale: filtering can be stronger on tiles with many players since the updates of any individual player will be less apparent in a crowd. In addition, players usually closely follow only a limited amount of players at the same time and pay less attention to the others [21, 104]. On the contrary, filtering should be lower on tiles with fewer players. Increasing the filtering ratio of dense tiles also has the

[2]We initially considered using the averaged bandwidth over all time units since the beginning of the period. We found out that this approach worked well only if the bandwidth did not vary too much. By taking the bandwidth over the last time unit only, we are able to react quickly to sudden variations.

added benefit of greater bandwidth reductions. DynFilter proposes an algorithm to compute a varying filtering ratio for each tile while still respecting bandwidth quotas.

5.3.2.2 DynFilter filtering

In order to obtain a filtering ratio for each tile $T_{x,y}$, we need to determine how many bytes we should save for every low-frequency $T^L_{x,y}$ tile topic. The idea is that the number of bytes that we should *remove* from each tile topic should be proportional to the total outgoing bytes of that tile for the previous unit. However, we multiply the number of bytes to be removed by a density factor $D_{x,y}$ that is logarithmic to the number of subscribers in that tile, to take the number of subscribers in the tile into account.

Let $S_{x,y}$ be the number of subscribers in $T_{x,y}$, $B^H_{x,y}$ the outgoing bandwidth (over the previous time unit) of topic $T^H_{x,y}$ and $B^L_{x,y}$ the outgoing bandwidth of $T^L_{x,y}$. Again, as it was the case with trivial filtering, $B^L_{x,y}$ depends on the filtering ratio $F_{x,y}$ used in the last time unit. In order to get accurate bandwidth computations, we define $B^{*L}_{x,y}$ as the *extrapolated* outgoing bandwidth, which is a projected value of $T^L_{x,y}$ without the effects of filtering (equation 5.3).

$$B^{*L}_{x,y} = \frac{B^L_{x,y}}{1 - F_{x,y}} \tag{5.3}$$

For every tile, we compute the density factor $D_{x,y}$ as follows: $D_{x,y} = \log_2 S_{x,y}$. We then compute the weight factor for tile $T_{x,y}$ by multiplying the total *extrapolated* bandwidth with the density factor (equation 5.4).

$$W_{x,y} = (B^H_{x,y} + B^{*L}_{x,y}) \cdot D_{x,y} \tag{5.4}$$

We define the sum of the weight factors as follows: $W_T = \sum W_{x,y}$. For each tile $T_{x,y}$, we can then compute how many bytes we need to remove from topic $T^L_{x,y}$ (equation 5.5).

$$Q_{x,y} = (W_{x,y}/W_T) \cdot B_{\text{remove}} \tag{5.5}$$

By knowing $Q_{x,y}$, we compute the filtering ratio for tile $T_{x,y}$ using equation 5.6 (ratio of bytes to remove to the number of *extrapolated* outgoing bytes that flowed through $T^L_{x,y}$ over the last time unit).

$$F_{x,y} = \frac{Q_{x,y}}{B^{*L}_{x,y}} \tag{5.6}$$

Note that a maximum value can be set for $F_{x,y}$ so that we can ensure that a minimal ratio of state update messages will be forwarded (in our experiments, it is set to 0.75) so at least 25% of the state update messages will be sent across all tiles.

The load optimizer then transmits the matrix of all filtering ratios to the message filter component that applies it.

5.4 Experiments

5.4.1 Implementation and Experimental Setup

We implemented DynFilter in Java on top of Dynamoth. Topic-based publish/subscribe is once again provided by unmodified Redis middleware. We ran our experiments on DynGame, which is based on the RGame prototype game skeleton developed for Dynamoth. As it was the case with RGame, DynGame supports a large amount of players that randomly move and uses square tiles. Publications and subscriptions are made according to the DynFilter model.

Experiments have been run in the cloud over a set of 20 Amazon EC2 instances: one *m3.medium* instance for the pub/sub server, data collector and message filter components; one *m3.medium* instance for the load analyzer and the load optimizer; one *t2.micro* instance for experimental data collection and 17 *t2.micro* instances to run our game clients. We determined that we were safely able to run 15 players per instance, up to a maximum of ~250 players. The decision to run server and clients components in the same cloud was motivated by the fact that intra-cloud bandwidth was free. While our implementation has been designed to support multiple pub/sub servers, we decided to limit our experiments to only one pub/sub server for simplicity reasons.

We considered a subscription range $Z = 2$ (players subscribe to 25 surrounding tiles, please refer to figure 5.1), except for players located near edges, who subscribed to less tiles. The subscription to the central tile is at high-frequency ($T^H_{x,y}$) (all messages are received) and the subscriptions to the other tiles is at low-frequency ($T^L_{x,y}$) (messages can be dropped).

5.4.2 Experiment 1: FPS Game / Scalability

5.4.2.1 Description

The goal of this experiment was to assess the scalability of DynFilter and its bandwidth-limiting capabilities in the context of a FPS-like game with many players. A typical FPS has a limited amount of players because of the high-bandwidth that is needed to support the high frequency of updates and the vision range for all players. For instance, WatchMen, which was based on a modified version of Quake 3, supported up to 48 players in the same game [104] (the original Quake 3 supported only 16 players).

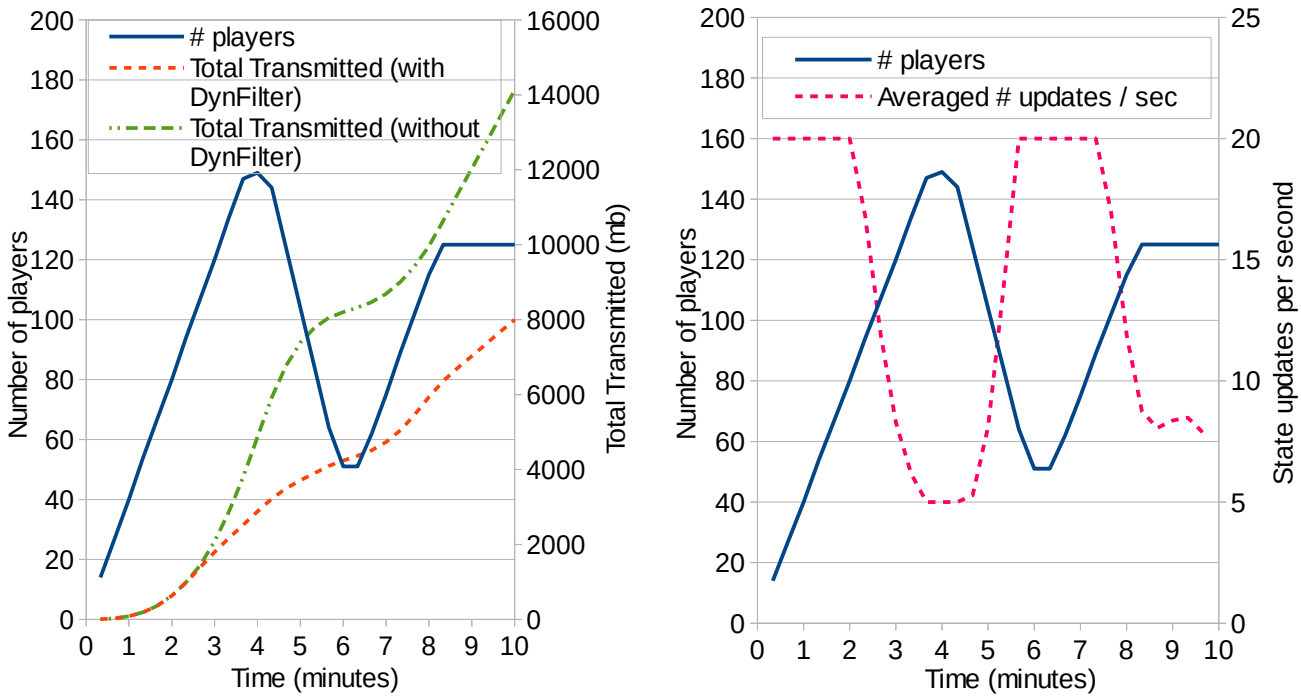
We configured a virtual map with up to 150 players and 100 tiles (10x10), with $Z = 2$. That means that any player would be able to view up to 25% of the map at any given time, which makes sense since FPS maps are generally small-scale compared to other types of games. As mentioned in the introduction, because of the fast-paced nature of such games, players optimally receive up to 20 state updates per second.

We progressively injected up to 150 players in the game, then reduced to 50 players, then increased again up to 125. We allocated a bandwidth threshold of 8000 Mb for the duration of the period (10 minutes), with units of 20 seconds (load analyzing and optimizing occurred every 20 seconds).

5.4.2.2 Results

Figure 5.3 details our results for the FPS experiment. On figures 5.3a and 5.3b, until about 3 minutes, we can see that no filtering occurred on low-frequency tile topics (all state updates were transmitted). Afterwards, due to the highly increasing load caused by the large amount of players, filtering starts to occur. The frequency of state updates received on low-frequency tile topics progressively drops until it reaches an average of 5 updates per second, which is the minimum frequency allowed for this experiment. Despite having a reduced frequency, we claim that playability was not sacrificed since updates were still received at least every 200 ms, and DynFilter takes special take to ensure that filtering only applies to players being located far apart (in different tiles). Thus, for players in the same tiles, the full frequency of updates (20 updates per second) was maintained. In addition, dead reckoning can compensate for some missing updates.

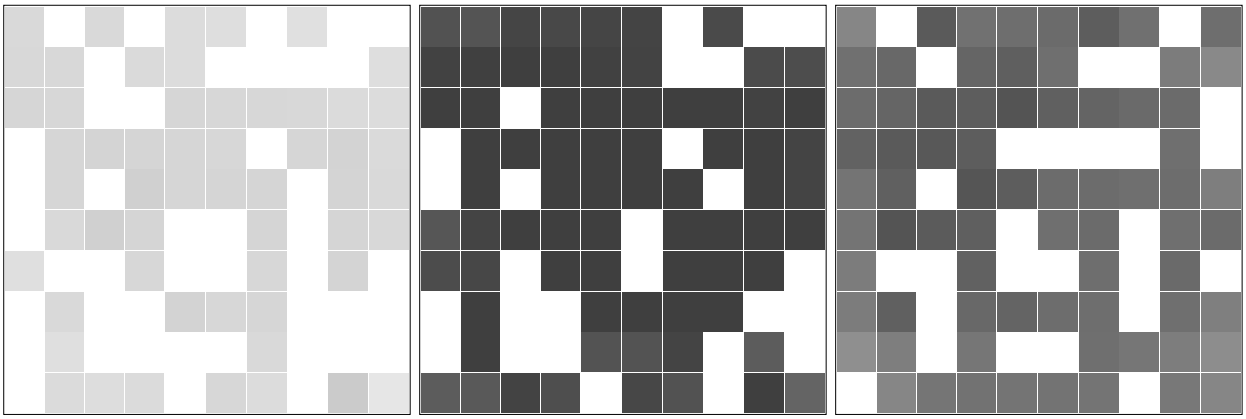
Afterwards, we observe that as the number of players starts to shrink (at about 4 minutes), the



(a) Number of Players and Total Outgoing Bandwidth

(b) State Updates per Second

Figure 5.3: FPS Game / Scalability Experiment Results



(a) $t = 2.3$; $\hat{F} = 0.14$ (b) $t = 4$; $\hat{F} = 0.73$ (c) $t = 8.7$; $\hat{F} = 0.54$

Figure 5.4: Filtering Ratio Heat Map / FPS Game

number of updates per second start to raise again until it reaches 20, which means that no filtering occurs once again - all state update messages are delivered to low-frequency tile topics. Then, as the number of players raise again above a certain threshold and up to 125, the number of updates per second starts to shrink again down to ~8 updates per second, which is a *best compromise* on degradation that will ultimately lead to a total bandwidth use of 8000 Mb at the end of our period; thus, respecting our predefined bandwidth quota. Overall, 8000 Mb have been used instead of 14000 Mb, thus representing a bandwidth saving of 43%.

Figure 5.4 illustrates the *averaged* filtering ratio ($\hat{F}$, which is the filtering ratio that would be equivalent to the current global reduction in bandwidth if all tiles had the same filtering ratio; that is, using the *trivial filtering* approach described in section 5.3.2.1) for all 100 tiles, at time snapshots $t = 2.3$, $t = 4$ and $t = 8.7$. White means that no filtering is in effect for a given tile (or no player is in that tile), dark grey means that filtering is at up to 75% and intermediate shades of grey illustrate an intermediate filtering ratio. We notice that as the number of players increases, the filtering ratios increase in roughly the same way across the whole game map (except for tiles with no players) since the density of players was roughly similar in this experiment.

5.4.3 Experiment 2: MMO Game with Flocking

5.4.3.1 Description

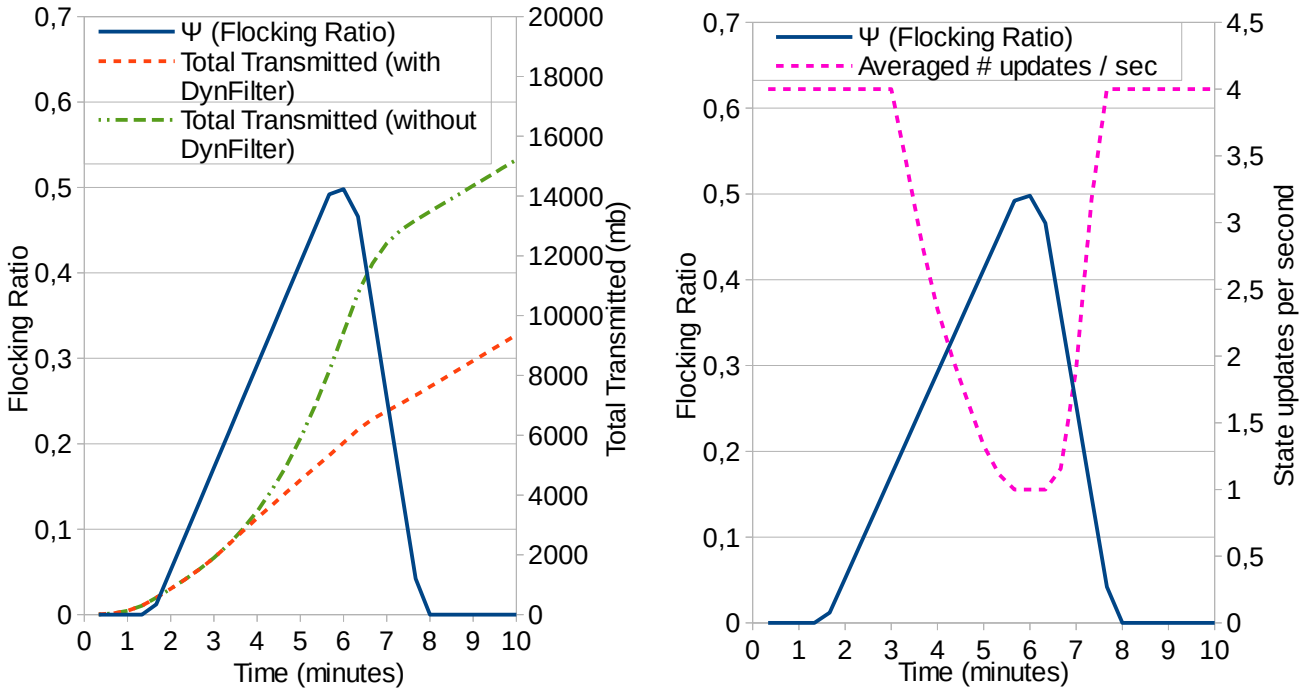
The goal of this experiment was to assess how DynFilter was able to handle the case of flocking within a medium-scale MMO game. Flocking refers to situations were many players gather towards the same location on the map, which can put a strain on the system since the number of state updates to be transmitted in those *hot spots* grows quadratically.

Since MMOs are slower-paced games compared to FPS games, we settled for an update rate of 4 updates per second, which is a realistic assumption based on an empirical observation of popular games. Also, since game worlds are much larger, we opted for 400 tiles (20x20). Thus, with a subscription span of 25 tiles, players only see a maximum of 6.2% of the map. We allocated a bandwidth quota of 10000 Mb for the 10-minutes period. We quickly injected 250 players in the map (which in itself would not go above the quota; thus, would not trigger the use of filtering). In this experiment, whenever a given player is *flocking*, it moves quickly towards the center 4x4 tiles of the map and remains within those tiles.

After injecting 250 players, we slowly increased the *flocking ratio* (ψ) from 0 to 0.5, which meant that up to 50% of the players were eventually located in the 16 centric tiles, thus greatly increasing player density and the number of messages that the pub/sub server had to deliver (near-quadratic growth).

5.4.3.2 Results

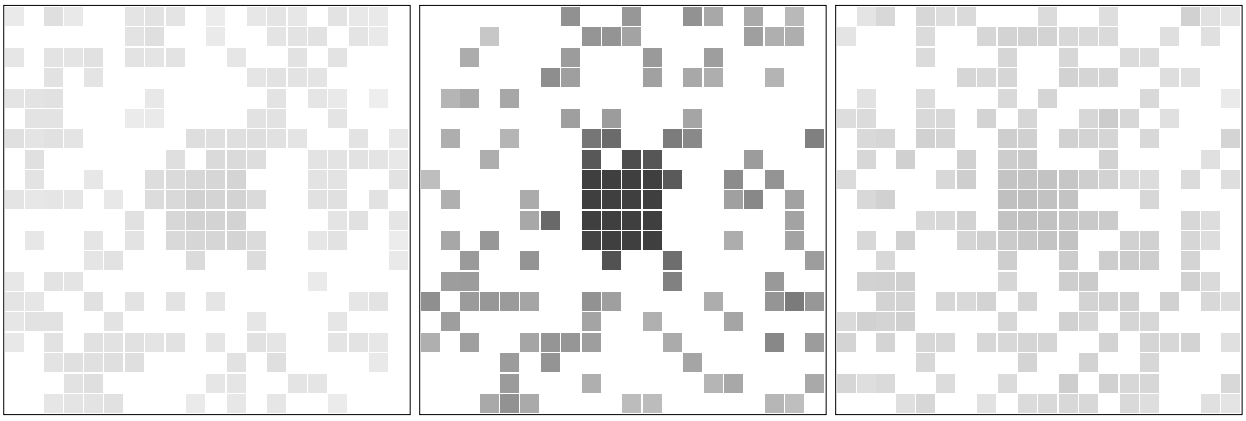
Figure 5.6 describe our results for the MMOG experiment. At time $t = 1$, ψ slowly starts to increase. After 3 minutes, when ψ reaches ~30%, DynFilter starts applying filtering in order to reduce the amount of messages that need to be transmitted and thus, the bandwidth use. In figure 5.5b, we observe that the average number of state updates per second for low-frequency tile topics starts to reduce until it reaches 1 (minimum allowed in this experiment), in order to compensate for the drastic increase of bandwidth. At $t = 6$, ψ slowly starts to decrease (players stop flocking and slowly move elsewhere to a random location anywhere in the map). In reaction to the reduction in bandwidth use, the average number of state updates per second starts growing again until it reaches 4 (low-frequency tile topic filtering disabled).



(a) Flocking Ratio (ψ) and Total Outgoing Bandwidth

(b) State Updates per Second

Figure 5.5: MMOG Game Experiment Results



(a) $t = 3.5$; $\psi = 0.21$

(b) $t = 6$; $\psi = 0.50$

(c) $t = 7.3$; $\psi = 0.15$

Figure 5.6: Filtering Ratio Heat Map / MMOG Game

At the end of the quota, a bit less than 10000 Mb were used since in the last minutes, we transmitted messages at the same rate as for high-frequency tile topics, which led to using less than the allowed quota. Overall, DynFilter was able to save 38% of the bandwidth.

Figure 5.6 shows a snapshot of the distribution of the filtering ratios across all tiles. At time $t = 3.5$, when flocking slowly starts to happen; the load optimizer starts increasing filtering ratios globally with a small emphasis on the center tiles. At time $t = 6$, filtering gets more important and really more concentrated in the centric of the map. At time $t = 7.3$, when flocking is being reduced, we observe that flocking ratios in the centric tiles gets less emphasized. This figure showed that DynFilter was able to adjust it's filtering based on the density of the tiles, in order to ensure that tiles with a lower amount of players would keep sending updates at a higher frequency despite the overall reduction in bandwidth, to account for the fact that players are more likely to notice individual players in lower-density tiles compared to higher-density tiles.

5.5 DynFilter Conclusions

In this chapter, we proposed DynFilter, a middleware designed to adaptively filter game state update messages in order to limit bandwidth use within a game to a predefined threshold. A major contribution is that our platform does per-tile filtering in order to adjust filtering levels to the number of players in each tile. We ran experiments in the context of FPS games with a high-frequency of updates and in the context of MMOGs with flocking. In both cases, DynFilter was correctly able to limit bandwidth use while maintaining the normal flow of the gameplay.

6

Dynamoth and Other Tools from a Software Engineering Perspective

The previous chapters described the more research-oriented parts of this thesis: namely our Dynamoth (chapter 3), MultiPub (chapter 4) and DynFilter (chapter 5) projects. These chapters described the theoretical and experimental aspects of our contributions. In order to run all the experiments described in this thesis, we developed solid implementations of the various models as well as appropriate infrastructure support tools, which proved to be challenging tasks. For that matter, we decided to describe these software systems from an implementation and software engineering perspective in the current chapter, as an additional contribution to this thesis.

The *Dynamoth Platform*, which is a full implementation of the Dynamoth system described in chapter 3, is a major contribution to this thesis. The implementation was designed to be faithful to the model, and flexible from a software engineering standpoint. As a result, it was very easy to extend the Dynamoth platform to support MultiPub and DynFilter. The Dynamoth platform was used to run experiments in cloud-like environments, such as on a pool of over 80 lab machines from the McGill School of Computer Science, as well as in real cloud environments, such as the public Amazon EC2 cloud. The Dynamoth platform is described in section 6.1.

In addition, running large-scale experiments such as our various game-related experiments that we ran in the context of our different projects in a cloud setting constituted major challenges. As an additional contribution to this thesis, I developed a set of tools to assist in running highly distributed

experiments with hundreds/thousands of nodes in the cloud or in cloud-like environments. These tools are described in section 6.2

Finally, the last section (6.3) describes in more details our `MultiPubSimulator` tool that we developed to evaluate our MultiPub system, in combination with our real implementation built on top of Dynamoth.

6.1 Dynamoth Platform

Our Dynamoth platform is inspired by and is built on top of the Mammoth networking infrastructure. In fact, it was originally built as a network engine, living within Mammoth, and called `RPub`. As such, it relies on a limited number of sub-components of Mammoth, but since Mammoth's network engines are pretty autonomous by design (they could be reused in other applications), the dependencies are quite minimal. Over the last year, Dynamoth has been separated from the main Mammoth project into a stand-alone project.

The goal of Dynamoth is to be a research tool that can be leveraged to run realistic, large-scale experiments, in the domains of topic-based publish/subscribe systems, e.g. for research on scalability and load balancing. It was however not designed for production-ready environments. We plan to open-source Dynamoth in a near future for other researchers to use.

The Dynamoth codebase is made of over 150 Java classes and over 15,000 lines of code. This section aims at presenting the core components, highlights and design choices of our Dynamoth implementation, from a software engineering perspective. It also describes the main ways in which Dynamoth could be extended. For the sake of brevity, it is unfortunately not possible to present every aspect of our implementation; as a result, many parts are not presented. Some areas of the source code are however well documented, which might ease its understanding.

The following subsections describe the Dynamoth software system in detail. Section 6.1.1 gives an overview of Mammoth, section 6.1.2 gives an overview of the software architecture and the main packages of Dynamoth and section 6.1.3 describe the Dynamoth Client Library. Sections 6.1.4 and 6.1.5 describe the two core infrastructure components of Dynamoth, respectively the local load analyzing and dispatching framework, and the load balancing framework. Then, section 6.1.6 describes the implementation of our prototype game RGame over Dynamoth and section 6.1.7 describe how fault

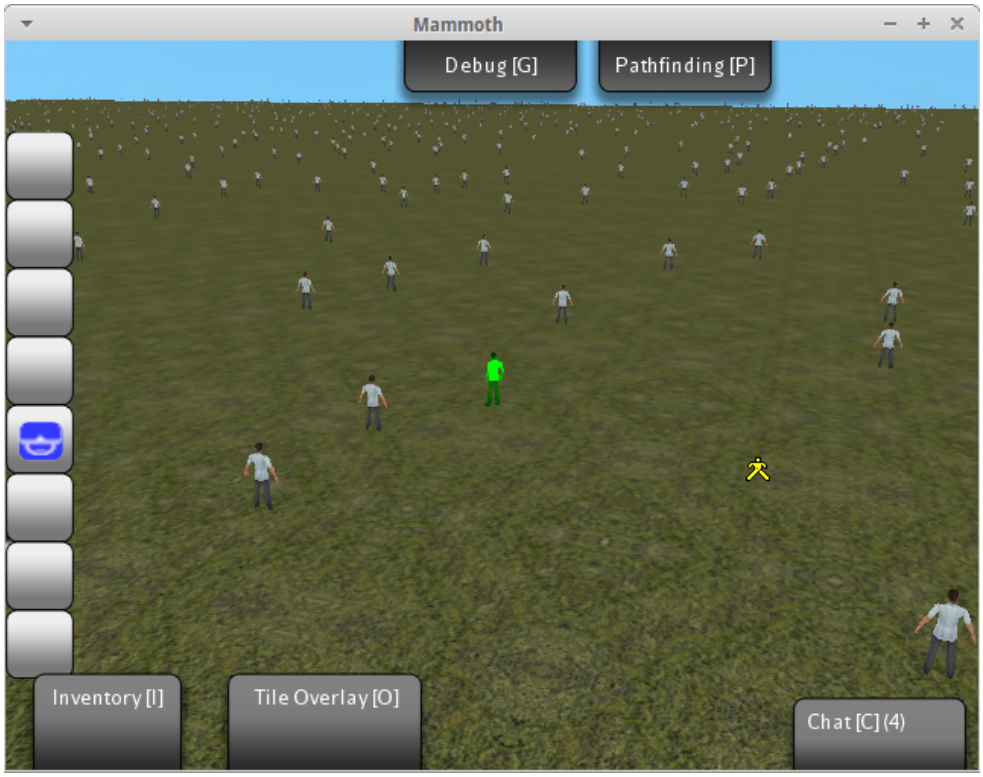


Figure 6.1: Mammoth Game Screenshot

tolerance was integrated within Dynamoth. Finally, sections 6.1.8 and 6.1.9 respectively describe how MultiPub and DynFilter were implemented over Dynamoth.

6.1.1 Overview of Mammoth

Mammoth is a massive multiplayer online game framework developed at McGill University. It was designed as a research tool aiming at studying and experimenting with the various aspects of large to massive-scale games. A screenshot of the Mammoth GUI Game Client is shown in figure 6.1. Mammoth is written in Java and is built on top of a modular architecture, where components loosely interact among themselves. Mammoth's main components notably include:

- a graphics (3D) engine, to manage the rendering of the game, using the JMonkey3 library;
- a world engine, to manage the game world and the various objects that it contains;

- a set of network engines, to manage the communications between all nodes (section 6.1.1.1);
- a persistence engine, to persist the game state;
- a game engine, to manage the actual working of the game;
- a sound engine, to manage in-game audio;
- an interest manager, to perform interest-management tasks as described in section 2.6.2.2;
- a replication engine, to manage the assigning of object replicas to in-game nodes (players and servers);
- a load-balancing engine, to balance the load between servers.

6.1.1.1 Mammoth Networking Infrastructure

A Mammoth network engine notably offers a topic-based publish/subscribe interface. Different protocols can be used to implement such an interface, and as mentioned previously, Dynamoth was originally built as a Mammoth network engine, where the core publish/subscribe engine was provided by Redis. Note that some network engines operate using different paradigms such as peer-to-peer, but they must still expose the topic-based publish/subscribe API calls: subscribe/unsubscribe and publish operations.

6.1.1.2 Mammoth Reactor

While it is possible to send any Java *serializable* object through the pub/sub interface, Mammoth also provides a higher-level component, namely the *reactor*, which allows any class to register interest in receiving incoming messages of a specific type (class). The reactor receives all network messages, and dispatches them to appropriate listeners depending on the type of the message. While developing Dynamoth, it made sense to reuse the Mammoth's reactor component since it was intuitive and easy to use.

As an example, our RGame clients periodically send state update messages of class `RGameMoveMessage`. A graphics engine could then register interest in receiving all such messages and could then update the graphical display upon reception of such messages.

In essence, the reactor simply provides an abstraction that eases message processing, but is not a mandatory component of the publish/subscribe interface.

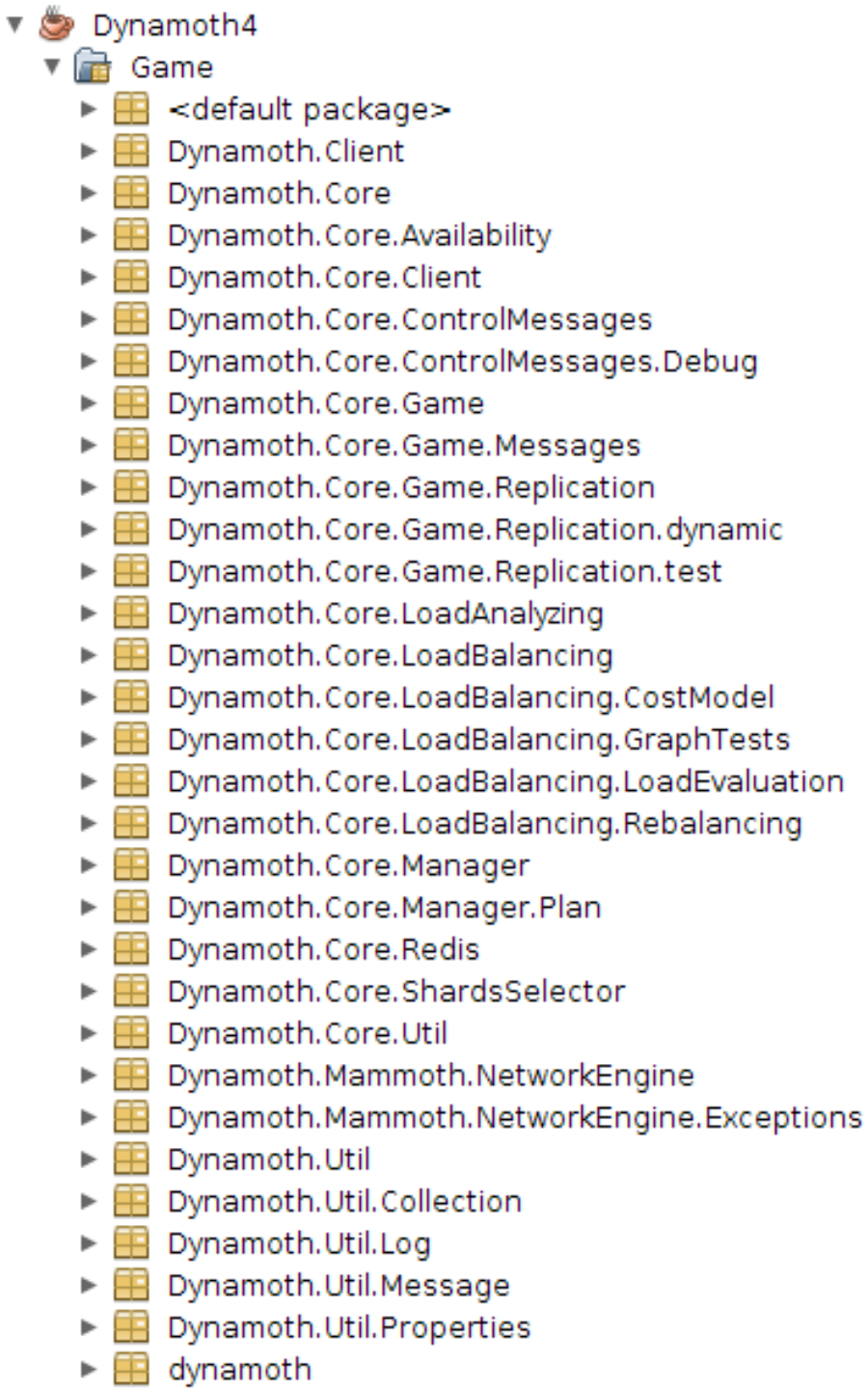


Figure 6.2: Main Packages of Dynamoth

6.1.2 Dynamoth Package Hierarchy

Figure 6.2 illustrates the package hierarchy of Dynamoth. The main classes of the network engine infrastructure of Mammoth have been placed in the `Dynamoth.Mammoth.NetworkEngine` package. Under `Dynamoth.Core`, the Dynamoth engine (called `RPubNetworkEngine`[1]) can be found and is the main class that developers wishing to use Dynamoth should use. Assuming that Dynamoth servers are running, one can create an instance of the network engine and connect to the Dynamoth infrastructure, as described at the next section (6.1.3), and perform publish/subscribe operations.

Presenting each package in a detailed fashion would be outside the scope of this thesis; as such, the next sections present the important parts of the Dynamoth API, both client-side and server-side. Section 6.1.3 discusses the Dynamoth Client Library, which is used mainly client-side, but also server-side, and sections 6.1.4 and 6.1.5 describe the local load analyzing and load balancing frameworks of Dynamoth, which are two key infrastructure components.

6.1.3 Dynamoth Client Library (DCL)

The Dynamoth Client Library (DCL) is the simple API that developers use to interact with a Dynamoth service, hosted in a cloud environment or not. This API is available at all nodes. It is assumed that the Dynamoth service (set of servers and other relevant components, e.g., the load balancer) are running.

Conceptually, from a developer's standpoint, the DCL is very simple: it simply exposes the core topic-based publish/subscribe primitives (publish, subscribe, unsubscribe), just like any other topic-based publish/subscribe platform. In addition, it also provides connect/disconnect operations in order to establish the connection to the Dynamoth service.

The DCL is also used server-side, as the different components that interact with the publish/subscribe servers also use the same API. The next subsections describe the main highlights of the DCL, as well as all the operations that the DCL manages behind the scene and that are transparent to consumers of the DCL.

[1] RPub refers to the previous name of the engine before Dynamoth. Further refactoring will eventually be done.

Listing 6.1: RPubClient Interface

```
1  public interface RPubClient {
2
3     void connect();
4     void disconnect();
5
6     boolean isConnected();
7
8     void createChannel(String channelName);
9
10    void publishToChannel(String channelName, RPubMessage message);
11    void publishToChannel(String channelName, String message);
12
13    void subscribeToChannel(final String... channelName);
14
15    void unsubscribeFromChannel(final String channelName);
16 }
```

6.1.3.1 Publish/Subscribe Interface

Dynamoth defines an abstract API that exposes the primitive topic-based publish/subscribe operations, as shown in listing 6.1, so that any pub/sub middleware can be used. This API, namely the `RPubClient` API, allows one to connect to and disconnect from a publish/subscribe server, as well as publish to, subscribe to and unsubscribe from topics on that server.

In our case, we used the Jedis library to work with Redis publish/subscribe servers. Therefore, we implemented a concrete class (`JedisRPubClient`) implementing our `RPubClient` interface. This class abstracts and conceals the Jedis API and offers an implementation of the `RPubClient` that can be used to perform all pub/sub operations through Redis publish/subscribe servers. While this implementation is used by default by Dynamoth, using an alternate publish/subscribe middleware would simply involve writing a new implementation of `RPubClient` that would communicate with this middleware to perform publish/subscribe operations.

6.1.3.2 Dynamoth Manager

We mentioned previously that Dynamoth employs a flat architecture between clients and servers (one hop to transmit any given publication or subscription request). For that principle to work, the Dynamoth clients maintain a connection to each server (thus, once instance of `RPubClient` for each server),

and they must be able to resolve by themselves to which server any given operation should be sent to.

To solve that challenge, Dynamoth exposes a `RPubManager` interface that allows one to specify to which RPubClient(s) any given publication or subscription should be issued[2] (package `Dynamoth.Core.Manager`). The ability to support multiple clients at the same time is needed to properly implement the Dynamoth replication feature.

In Lamoth [55], topics were statically assigned to servers, based on a hash of the topic name. This technique is also used in Dynamoth to map topics to servers, if no information is defined in the plan for such topics (see section 3.2.5.1). For Lamoth, we then defined a manager that used hashing to resolve topic-to-server assignations (`HashedRPubManager`).

On the other end, Dynamoth makes use of a plan (see section 3.2.4) to resolve to which server(s) a given topic T should map to. Thus, we created a `DynamothRPubManager` class which takes as input a `Plan` object to resolve mappings. The `Plan` object contains, for each topic T, the set of servers that should be used to process publications and subscriptions on T, as well as the replication strategy that is being used, if replication is active for that topic. The `Plan` object and some helper classes which are used to compute some operations on `Plan` objects, such the difference (delta) between two plans (mostly for load balancing purposes), are located in package `Dynamoth.Core.Manager.Plan`.

As described previously, the load balancer is responsible for generating new `Plan` objects, and is described in more details in section 6.1.5 below.

6.1.3.3 Transparent Reconfiguration

In addition to offering a topic-based publish/subscribe interface to the Dynamoth service, a key point of the Dynamoth Client Library is that it also transparently handles the internal reconfiguration mechanism of Dynamoth (described in section 3.4); that is, without users of the DCL being aware of the process. As such, when the load balancer component (described below in section 6.1.5) generates a new plan, it is propagated lazily to the various clients in the Dynamoth system (and eagerly to the local load analyzers). The mechanism by which publishers and subscribers are informed that they should switch servers for any given topic T or enable/disable replication, as well as the whole bootstrapping process, are all handled internally by the `DynamothRPubManager`.

[2] Simplified description. At a more abstract level, one can define the exact behavior that should trigger for publications and subscription API calls.

To perform these tasks, the `DynamothRPubManager` component transparently subscribes to special system topics, and registers itself to receive specific system control messages from the reactor. Upon receiving such messages, the `DynamothRPubManager` performs the necessary internal changes: updating the plan and then reestablishing appropriate subscriptions.

In more details, upon receiving a new "full" plan (`ChangePlanControlMessage`) (this can happen for local load analyzers, for instance, since they also reuse the DCL), the current local plan is replaced with the new plan. For non-infrastructure clients, as the reconfiguration process stipulates, clients are corrected whenever they hit the wrong server for publications and subscriptions. As such, when that happens, they receive from the local load analyzer a `ChangeChannelMapping-ControlMessage` which contains information only for the relevant topic. Upon receiving such messages, clients *patch* their local plan with the new partial information received. Appropriate subscriptions are then properly reestablished, and future publications are then sent to the new servers, according to the updated local plan.

Note that our `DynamothRPubManager` component implementation has some built-in optimizations when it comes to reestablishing subscriptions. For each topic T, the manager notably computes a "diff" in the set of servers that should be used to process publications and subscriptions on T, between the old and the new plan. It also handles the two replication models properly. More details are given in the `Dynamoth.Core.ShardsSelector` package.

Note that all Dynamoth internal system-related messages (referred to as control messages) can be found in package `Dynamoth.Core.ControlMessages`.

6.1.3.4 Latency Emulation

One feature of Dynamoth that was built for research purposes is that it allows publications to be arbitrarily delayed by the DCL before being delivered to the application layer. The delay can either be a fixed or a random amount sampled from a given dataset, e.g., as the King dataset [57] previously described. Since in our experiments we ran our clients in the cloud as well, we used this feature to emulate realistic latency values as if the clients were located outside the cloud.

6.1.4 Local Load Analyzing & Dispatching Framework

In addition to publish/subscribe servers, who make use of Redis in our implementation, a Dynamoth service includes several infrastructure components, as described in section 3.2.2 and illustrated in figure 3.1. This section discusses the local load analyzing and dispatching framework, and the next section (6.1.5) discusses the load balancing framework.

6.1.4.1 Local Load Analyzer

As described in section 3.3.1, next to each publish/subscribe server H sits a *local load analyzer* (LLA) component that is in charge of collecting detailed information on H. Our Dynamoth implementation notably collects over a given time interval $\Delta t = 1s$, for each topic T:

- The number and list of current subscribers to T;
- The number and list of current publishers to T (who published at least in the last interval Δt);
- For each publisher P, we store the number of publications transmitted, as well as the total number of incoming and outgoing bytes incurred due to the processing of all publications sent from P, over the last Δt, by H. Note that the number of bytes is computed based on the size of the publication messages.

The LLA also monitors the network interface on the machine and therefore collects the total amount of incoming and outgoing bandwidth (this is monitored and not computed). According to our empirical observations, notably by monitoring the CPU load of the local load analyzer, the data collection process had only negligible impact on the performance, as all communications are done through the local, loopback network interface.

All measurements collected by the LLA over the last interval Δt are periodically transmitted to the load balancer, on a dedicated topic that only the load balancer listens to and that is immutable (cannot be reassigned during load balancing).

6.1.4.2 Hooking the LLA to the Pub/Sub Server

The question arises on how to hook LLAs to publish/subscribe servers, considering that we claim that such servers are unmodified. Our current approach is for each LLA to subscribe to all topics on the cor-

responding pub/sub server. In our case, Redis allows *wildcard* subscriptions, which makes subscribing to all topics easy. However, as mentioned, Redis could be swapped with any other topic-based publish/-subscribe middleware and it is not guaranteed that such middleware will support wildcard subscriptions in the same manner as Redis.

As a fallback mechanism, we propose a scheme inspired by Mammoth [70], where subscribers send a special "subscription" publication message on a dedicated topic before actually subscribing to T, so that the local load analyzer can become aware of the new subscription and either start monitoring T if this is the first subscriber, and add this new subscriber to the set of subscribers for topic T. While heavier, this fallback mechanism has the advantage of broadening the compatibility with more pub/sub middlewares.

Note that Dynamoth supplies a launcher that allows for automatically spawning an instance of the Redis pub/sub middleware and launching the local load analyzer, and hooking that LLA instance to the newly launched Redis instance.

6.1.4.3 Dispatching

As described, in addition to the local load analyzer, Dynamoth servers also comprise a *dispatcher* component that is in charge of assisting the reconfiguration process (refer to section 3.4). The dispatcher is in charge of detecting publications and subscriptions addressed to the wrong server, and informing relevant clients of plan changes that they are not yet aware of. The dispatcher can properly carry out this task since just like the LLA component, since it has access to the full plan as transmitted by the load balancer.

The dispatcher also handles the temporary redirection of publications during the reconfiguration process, as explained in section 3.4.2. This is done both for non-replicated and replicated cases.

In our Dynamoth implementation, the tasks of the dispatcher are handled by the local load analyzer. Separating the tasks of the local load analyzer and the dispatcher would certainly be a good refactoring and is left as future work.

Note that all classes pertaining to the local load analyzing / dispatching framework can be found under package `Dynamoth.Core.LoadAnalyzing`.

6.1.5 Load Balancing Framework

Dynamoth proposes an elaborate and rich load balancing framework that is extensible, which allows Dynamoth developers to implement custom load balancing logic. Relevant classes are found in the `Dynamoth.Core.LoadBalancing` package and subpackages. At the heart of the Dynamoth load balancing framework is the `LoadBalancer` class, which is the entry point of a Dynamoth load balancer. It notably registers to receive all measurements from all local load analyzers (through the pub/sub interface itself), and performs aggregation of all measurements received from all LLAs. Such measurements are then exposed through a rich API, namely the `LoadEvaluation` API. From that, the rebalancing framework is invoked and determines on a periodic basis whether a new plan should be generated.

Note that for load balancing to be active, an instance of the load balancer must be launched. Upon launching, the load balancer automatically connects to the Dynamoth service and registers itself as the active load balancer. Remark that the Dynamoth load balancer is an optional component: without it, the Dynamoth system will still be functional, but no new plans will be generated. In the absence of the load balancer, the local load analyzers would then send transmit their measurements to a topic with no subscriber.

Listing 6.2: LoadEvaluation API

```
1  public interface LoadEvaluator extends Serializable {
2    Set<RPubClientId> getRPubClients();
3    Set<String> getClientChannels(RPubClientId client);
4    int getClientChannelSubscribers(RPubClientId client, String channel);
5    int getClientChannelPublishers(RPubClientId client, String channel);
6    Set<RPubNetworkID> getClientChannelSubscriberList(RPubClientId client,
       String channel);
7    Set<RPubNetworkID> getClientChannelPublisherList(RPubClientId client,
       String channel);
8    int getClientChannelPublisherPublications(RPubClientId client, String
       channel, RPubNetworkID publisher);
9    int getClientChannelPublications(RPubClientId client, String channel);
10   int getClientChannelSentMessages(RPubClientId client, String channel);
11   long getClientChannelComputedByteIn(RPubClientId client, String channel)
       ;
12   long getClientChannelComputedByteOut(RPubClientId client, String channel
       );
13   long getClientComputedByteIn(RPubClientId client);
14   long getClientComputedByteOut(RPubClientId client);
```

```
15    long getClientComputedCumulativeByteIn(RPubClientId client);
16    long getClientComputedCumulativeByteOut(RPubClientId client);
17    long getClientMeasuredByteIn(RPubClientId client);
18    long getClientMeasuredByteOut(RPubClientId client);
19    long getClientWastedByteIn(RPubClientId client);
20    long getClientWastedByteOut(RPubClientId client);
21    long getClientUnusedByteIn(RPubClientId client);
22    long getClientUnusedByteOut(RPubClientId client);
23    double getClientByteInRatio(RPubClientId client);
24    double getClientByteOutRatio(RPubClientId client);
25    long getClientMessageIn(RPubClientId client);
26    long getClientMessageOut(RPubClientId client);
27    RPubClientId getClientHighestByteOut();
28    RPubClientId getClientHighestByteOut(Set<RPubClientId> activeHosts);
29    RPubClientId getClientLowestByteOut();
30    RPubClientId getClientLowestByteOut(Set<RPubClientId> activeHosts);
31    String getClientChannelHighestByteOut(RPubClientId client);
32    String getClientChannelHighestByteOut(RPubClientId client, Set<String>
          ignoreChannels);
33    String getClientChannelHighestByteOut(RPubClientId client, Set<String>
          ignoreChannels, Plan plan);
34 }
```

6.1.5.1 LoadEvaluation API

The LoadEvaluation API (package `Dynamoth.Core.LoadBalancing.LoadEvaluation`, shown in Listing 6.2) allows rebalancers to access relevant load data for all pub/sub servers in an abstract form. The API allows one to access relevant metrics (number/list of publishers/subscribers, number of incoming/outgoing bytes, etc.) for any given topic T, in an aggregated form, for any given server H. One can also obtain detailed network bandwidth usage of each server H to determine to what resources are used and remaining on H, in order to take load balancing decisions. One such metric is the load ratio, which, as described in equation 3.1 of section 3.3.2, yields the ratio of the outgoing bandwidth compared to the bandwidth capacity of the server. Some other operations are helper functions which, e.g., determine the highest-loaded or the lowest-loaded server.

The load balancer performs load evaluation (generates a *new load evaluator*) at every update interval Δt so that the rebalancer always has access to the latest evaluation results.

Dynamoth supports multiple load evaluators and allows a developer to define new, custom, load evaluators for specific purposes. Dynamoth currently includes three built-in load evaluators:

- `DiscreteLoadEvaluator`, which yields measurements that correspond to the last observation interval Δt. As expected, such measurements are subject to high variation.
- `AveragedLoadEvaluator`, which averages the result of the last "DiscreteLoadEvaluators" over a larger window (typically 30 seconds), in order to smooth the measurements. This evaluator is used by the default Dynamoth rebalancer since it prevents sudden reconfigurations from taking place in reaction to sudden, short duration spikes.
- `NewPlanEstimatedLoadEvaluator`, which is a special evaluator that attempts to predict the outcome of a given new plan on the load of the various servers. This evaluator is also used by the default Dynamoth rebalancer in order to estimate the effects of a new plan, as part of our heuristic approach which iteratively generates a new plan until a suitable plan is generated. The estimated load evaluation process is done by taking into consideration the load of the topics that were subject to migration (or replication) between the old and the new proposed plan, and by translating the impact of these migrations on the global load. While our empirical observations showed that such estimations were close enough to the real behavior after applying a new proposed plan, we did not run formal experiments (this is left as future work). If conditions change after applying a new proposed plan, then a consequence is that the estimations will become less accurate. We think that this is acceptable since the Dynamoth load balancer can be reinvoked again to generate once again a better plan, in reaction to the *currently observed* conditions.

In addition to being fed to the rebalancing framework, the results of the load evaluation process are also published to a special internal topic so that a special *monitoring* client can capture them and generate experimental output files. One such client is the RServer component which is described later in section 6.1.6.

6.1.5.2 Rebalancing Framework

As mentioned, Dynamoth includes built-in rebalancers and supports developing arbitrary rebalancers to implement a specific rebalancing behavior. In this regard, it offers a rebalancing framework that can be extended (package `Dynamoth.Core.LoadBalancing.Rebalancing`).

Listing 6.3: `Rebalancer` Interface

```
1 public interface Rebalancer {
2
```

```
3   /**
4    * If this rebalancer decides that a new plan should be applied and a
         new plan is available
5    * return the new plan; otherwise, return null.
6    * Note: null will be returned if calling isNewPlanAvailable() returns
         false.
7    * @return New plan that should be applied or null if no new plan needs
         to/should be applied.
8    */
9   Plan getNewPlan();
10
11  /**
12   * Returns whether this rebalancer thinks that a new plan should be
         applied.
13   * @return True if a new plan should be applied; otherwise, false.
14   */
15  boolean isNewPlanAvailable();
16
17  /**
18   * Starts this rebalancer. A rebalancer will run in the background to
         continuously generate plans.
19   */
20  void start();
21
22  /**
23   * Instructs this rebalancer to stop.
24   */
25  void stop();
26
27  /**
28   * Is this rebalancer running
29   */
30  boolean isRunning();
31
32  /**
33   * Obtain the rebalancer's current plan
34   * @return Rebalancer's current plan
35   */
36  Plan getCurrentPlan();
37
38  /**
39   * Tells the rebalancer that the current plan has changed (and give the
         Rebalancer the current plan as input)
40   */
41  void setCurrentPlan(Plan plan);
42 }
```

At the base of the rebalancing framework is the `Rebalancer` interface (Listing 6.3), which rebalancers must implement. Rebalancers run in a different thread and periodically generate new plans, if needed, typically based on the current load, by querying the `LoadEvaluation` API described above. The load balancer periodically queries the active rebalancer to determine whether a new plan should be applied (`isNewPlanAvailable` function). In the affirmative, the new plan is obtained and applied, as described in section 3.4.

Dynamoth provides higher level abstractions in the form of base classes (`AbstractRebalancer` and `LoadBasedRebalancer`) that can be used to implement some common parts of the concrete rebalancers, such as threading aspects, so that Dynamoth developers can focus their time on implementing the rebalancing logic itself.

By default, Dynamoth includes several built-in rebalancers:

- `DynamothRebalancer`, which implements the system-level rebalancing algorithms of Dynamoth as described in section 3.3.2.2 (high load and low load);
- `DynamothReplicationRebalancer`, which implements the topic-level rebalancing algorithms of Dynamoth as described in section 3.3.2.1;
- `HierarchicalLoadBasedRebalancer`, which is a composite rebalancer (inspired by the composite design pattern) which supports chaining of multiple rebalancers, in order to implement the two-hierarchical load balancing model of Dynamoth (topic-level rebalancing followed by system-level rebalancing; thus, `DynamothReplicationRebalancer` followed by `DynamothRebalancer`);
- `MultiPubRebalancer`, which is a special rebalancer that uses the `MultiPubSimulator` tool (described later in section 6.3) to generate a proper MultiPub multi-cloud configuration/plan that takes into consideration the locality of the publishers and subscribers and the available cloud regions.

6.1.6 RGame Implementation

As described previously in section 3.6, `RGame` is a prototype game that abstracts a game world containing tiles and players. Each tile maps to a topic, players publish to the topic corresponding to the tile in which they are located, and also subscribe to their own tile, and possibly additional tiles within a certain

radius (optional). RGame is implemented within Dynamoth in the package `Dynamoth.Core.Game` and its subpackages. Eventually, RGame could be removed from the main Dynamoth tree in order to form a separate project, since it only depends on the Dynamoth Client Library (it could use that library externally).

Listing 6.4: RServer Sample Script file

```
% 0 players
+10
sleep 3000
+10
sleep 3000
+10
sleep 3000
+10
sleep 3000
+10
sleep 3000
+10
sleep 3000
+10
sleep 3000
+10
sleep 3000
+10
sleep 3000
+10
sleep 3000
% 100 players
```

RGame is launched through its main class `RMain`, which is used to launch RGame player instances. It takes as a parameter the number of virtual players that this instance will contain. Prior to launching players, one has to launch the lightweight RGame server (`RServer`), which controls the flow of player allocation, as well as a command-line interface to interact with the game, mainly for enabling/disabling players and making them *flock* to some location in the game world for *flocking* experiments. The command-line interface also supports a trivial scripting format so that commands can be batched and replayed to perform repeatable experiments. Listing 6.4 shows an example of a RServer script that enables 10 players every 3 seconds, until it reaches 100 players.

Upon launching, RGame connects to the Dynamoth service and establishes appropriate subscriptions to RGame-related system topics. Eventually, RGame is informed that it can spawn some or all

of its players. At that point, for each player, RGame performs a subscription to the appropriate tile topic(s), and starts publishing position updates on a regular basis (the interval is defined as a parameter) for that player as it moves within the virtual world. Subscriptions are reestablished whenever the player switches tiles. RGame defines custom network message classes (defined in `Dynamoth.Core.Game.Messages`) and makes use of the reactor to register to receive such messages at appropriate locations in the application layer.

Also, since RGame consumes the Dynamoth Client Library as would any other client using a Dynamoth service, it is completely agnostic to reconfigurations (plan changes), since such reconfigurations are handled internally within the DCL.

For experimental purposes, each RGame instance measures response times for all state update messages, and forwards such measurements to the RServer component, which generates CSV output files. The same RServer component also registers itself to the special Dynamoth topic that conveys load information and appends this to the aforementioned CSV output file.

6.1.7 Fault Tolerance Implementation within Dynamoth

Our original Dynamoth paper [53] did not include fault tolerance and availability. These aspects were added in a later iteration, which, combined with the original Dynamoth system, will constitute the basis of our upcoming journal publication. Nevertheless, these aspects are part of our thesis.

The original Dynamoth framework was not originally architected with these features in mind. However, given the flexibility of Dynamoth, we were able to add such features, but not in the most optimal way - some refactoring in this area would be interesting future work.

The availability aspect of Dynamoth is provided mainly through package `Dynamoth.Core.Availability`.

6.1.7.1 Failure Detection and Notification

In our Dynamoth availability model (section 3.5), all nodes share the responsibility of detecting failures of the various pub/sub servers. The availability package of Dynamoth provides a `FailureDetector` component, who is in charge of monitoring the state of all publish/subscribe servers. Therefore, if the availability module of Dynamoth is enabled, then all nodes must launch an instance of the failure

detector. Upon receiving a publication, the `DynamothRPubManager`, part of the Dynamoth Client Library at each node, notifies the failure detector so that it can properly track how much time elapsed since the last publication was processed by the pub/sub server and thus detect inactive servers.

Upon detecting a failure (unfortunately we did not have time to implement the failure suspicion mechanism described previously due to time constraints and the additional complexity involved), the failure detector notifies interested observers/listeners. This is done following the Observer design pattern. The availability package provides a `FailureListener` interface, and any instance of a class implementing the said interface can register with the failure detector component to be notified of server failures. In our current implementation, the `DynamothRPubManager` component registers itself to be notified of failures, so that it can properly reestablish subscriptions and replay missed publications.

6.1.7.2 Storing and Replaying Old Publications

Upon being notified of a failure, the `DynamothRPubManager` reestablishes subscriptions for all topics managed by the failing server, towards alternate servers, as described in section 3.5.4. Remember that all nodes in a Dynamoth system execute an instance of the `DynamothRPubManager`; therefore, all nodes, both client-based and infrastructure-based, perform the same actions.

If publication replaying is enabled, then the manager waits for a certain duration based on the timing protocol described in section 3.5.3, in order to make sure that all other nodes have properly detected the failure and reestablished subscriptions to alternate servers. In then proceeds to replay old publications. If FIFO ordering must be respected for a given topic T, then publications are replayed in the order that they were sent, while new publications are queued and sent only after all past publications have been replayed, as described in section 3.5.6. Otherwise, if FIFO is not required, then old publications are replayed in reverse order (freshest publications first), concurrently to any incoming new publication to be delivered.

As described previously, in order to support publication replaying, all outgoing publications must be remembered for a predetermined amount of time. Thus, the `DynamothRPubManager` also has the responsibility of enqueuing all relevant outgoing publications (if playback is enabled), for a duration that corresponds to the maximum amount of time that it takes to guarantee proper subscription reconfiguration at all nodes. Unfortunately, in our current implementation, we only support a global playback policy (ordered playback (FIFO), concurrent playback (no FIFO) and no playback) for all

topics and not on a per-topic basis as described in our model. As future work on the Dynamoth codebase, the code should be refactored to allow for playback policies to be defined on a per-topic basis.

6.1.8 Integration of MultiPub within Dynamoth

In our description of MultiPub, we explained how MultiPub and Dynamoth could be integrated: having Dynamoth deployments in each cloud region - in our model we abstracted each Dynamoth deployment as one server; and having MultiPub manage the mapping of topics between all the cloud regions.

6.1.8.1 Adaptation of Dynamoth to MultiPub

In our experimental implementation of MultiPub, we followed a slightly different, simpler model. Due to the flexibility of the Dynamoth system, it made sense to reuse and adapt it to the needs of MultiPub. In our model, Dynamoth is used to manage the global multi-cloud publish/subscribe service. In our experiments, we had 3 cloud regions, and one publish/subscribe server (and local load analyzer / dispatcher) per region. As mentioned in section 3.6.2, we deployed clients (we reused RGame) in up to two of the three regions, and we used appropriate scripts to control the proper deployment of such clients.

6.1.8.2 MultiPub Rebalancer

As mentioned previously, we launched the Dynamoth load balancer with the `MultiPubRebalancer`. This rebalancer periodically invoked the solver of our MultiPub simulator (briefly introduced in section 4.6.2 and described more thoroughly in section 6.3) in order to generate Dynamoth plans that allowed for a given latency constraint to be respected, while minimizing costs and while considering the current load conditions as exposed by the `LoadEvaluator` and considering the model of MultiPub. Therefore, in our MultiPub-over-Dynamoth implementation, load balancing was not done to manage bandwidth usage, but to handle delivery constraints and reduce costs. Upon a new plan being generated, the new plan was applied following the standard Dynamoth protocol.

6.1.8.3 Delivery Configurations

As described in section 4.2.2.3, MultiPub proposes two delivery configurations: direct delivery and routed delivery. Direct delivery can be viewed as a special case of Dynamoth's all-publishers approach,

where publishers publish towards all available servers/regions (one server per region in our implementation), and subscribers subscribe to only one server/region, which is the closest to them latency-wise.

In order to implement that behavior, we created an alternate replication scheme apart from the two offered by Dynamoth. This new scheme is based on the original all-publishers replication scheme, but it differs in that instead of subscriber S selecting a random server when subscribing on topic T among all servers handling T, S simply selects its closest server from a table (originally supplied to the `DynamothRPubManager`).

6.1.9 Integration of DynFilter within Dynamoth

DynFilter was also implemented over Dynamoth. Our experiments were run on a single-server configuration of Dynamoth. While our theoretical model allowed for multi-server setups, we nevertheless decided to run our experiments using only one server due to the additional complexity involved as well as timing constraints.

6.1.9.1 CostAnalyzer

DynFilter was built within the Dynamoth load balancer (`CostAnalyzer` class of the `Dynamoth.Core.LoadBalancing.CostModel` package), but was not integrated as a typical *rebalancer* within Dynamoth. In fact, due to the single-server configuration, we disabled the rebalancing aspect of the load balancer (a static plan is enforced at client-side through the `DynamothRPubManager`, since no plan is being generated by the load balancer). As a result, the load balancer still receives load information from the LLA, and computes appropriate load evaluators, which are used by the `CostAnalyzer` component of DynFilter to compute an appropriate filtering matrix (see section 5.3.2).

6.1.9.2 Transmitting the Filtering Matrix

Upon noticing that the `CostAnalyzer` generated a new filtering matrix, the load balancer publishes an appropriate control message that wraps the new matrix towards a special topic that local load analyzers subscribe to - once again reusing the internal pub/sub layer. Upon receiving the new matrix, the local load analyzers use it as their current matrix.

6.1.9.3 Forwarding Publications using the Filtering Matrix

As per the model of DynFilter (described in section 5.2.1), all players (once again RGame clients) publish towards the topic corresponding to the tile in which they are located (high-frequency topic), and subscribe to the same tile (high-frequency) plus surrounding tiles within a given radius (low-frequency). Recall that publications on low-frequency topics/tiles may be filtered/dropped. It is the responsibility of the infrastructure to forward messages from high frequency topics to low frequency topics, by taking the filtering matrix into consideration.

In our implementation, we modified the local load analyzer component so that upon receiving a publication (recall that local load analyzers receive a copy of all publications for monitoring purposes), they republish the same publication to the corresponding low-frequency topic with a probability computed from the filtering matrix, as described in section 5.2.3.

Note that an optimal implementation of DynFilter, perhaps through subsequent refactorings, would certainly involve having the CostAnalyzer code implemented as a typical Dynamoth rebalancer following the API described in section 6.1.5.2.

6.2 Tools for Running Large-Scale Experiments

Running large-scale experiments in cloud and cloud-like environments was no easy task. For instance, in the context of Dynamoth, we ran massive multiplayer game experiments with up to 1200 players. Simulation approaches are certainly easier, but they can sometimes lead to results that do not accurately reflect the real world, since they cannot always account for all factors [46]. Therefore, we decided to run real experiments in all of our projects (Dynamoth, MultiPub and DynFilter). For MultiPub, we conducted both simulated and real experiments.

In order to aid in running such large-scale experiments in an efficient and quick manner, we designed some software tools that we describe in the next sections: Distmoth (section 6.2.1) and MUDPLaunch (section 6.2.2).

6.2.1 Distmoth

Large-scale systems like Dynamoth comprise many components: a large number of game clients (hundreds to thousands), a game server, a set of publish/subscribe servers (Redis) with a local load analyzer, a load balancer. Such components of a highly distributed system need to be launched on a large set of different machines (in our experiments, we used up to 80 Linux machines of the McGill SOCS Labs). Another issue is that some components need to be started before others, so starting all components at the same time is not an option.

A trivial method is to establish SSH connections to all machines, and launch all components manually, but this is not a practical solution, especially when one needs to relaunch the system several times (it might take several rounds of the bug-fixing / relaunching cycle for an experiment to finally succeed).

After having spent considerable time on running experiments, I decided to develop a Python tool (Distmoth) to aid in launching such highly distributed experiments. The goal was to minimize the time to execute the launch sequence as much as possible. Another requirement of the tool was to make it flexible, so that it could be adapted to the needs of various distributed systems. Distmoth was used successfully for running all of our experiments and allowed us to save a significant amount of time. It is our plan to eventually release Distmoth as open source.

6.2.1.1 Overview

Listing 6.5: Fictious example of the a machine configuration

```
[servers]
server1.cloudprovider.com
server2.cloudprovider.com
server3.cloudprovider.com
[lab6]
lab6-1.cs.mcgill.ca
lab6-2.cs.mcgill.ca
lab6-3.cs.mcgill.ca
lab6-4.cs.mcgill.ca
lab6-5.cs.mcgill.ca
```

Distmoth comes in the form of a set of Python files. It manages the launching of a distributed system, using two specific configuration files:

- (1) `machinelist.conf`, which contains a pool of Linux machines that can be used, and that are reachable via the SSH protocol. Each machine can optionally be assigned to a category. Listing 6.5 shows a simplified example of such a configuration file, with 3 servers supposedly in the cloud, and 5 machines from McGill's SOCS lab #6;
- (2) `<configuration>.conf`, which specifies the different components of the distributed system and how they should be launched. More details are given in the next section (6.2.1.6).

Distmoth makes use of a pool of Linux machines that are available and that can be reached via the SSH protocol, as well as a list of commands to execute on these machines. A Dynamoth invocation is done in three steps, which are described in the next subsections: (1) finding available machines, (2) launching the components, (3) proposing a command-line interface and (4) killing all running components.

6.2.1.2 Finding Available Machines

The first step is for Distmoth to scan through all the machines defined in `machinelist.conf`. It then determines which machines are available and which machines are non-responsive. The latter are ignored, and Distmoth launches the components specified in the configuration file only on the available machines. Determining which machines are available can take non-negligible time, especially if the pool of machines to be used is large. This was the case when we put in the list all of the machines from the various McGill SOCS labs (~150 machines). As an optimization that we implemented, upon discovering available machines, Distmoth rewrites a more optimized version of it's machine listing: `machinelist_optimized.conf`. Upon relaunching, Distmoth can then load this optimized file instead, so that previously available machines are used first, before attempting to use machines that were unavailable in the previous invocation of Distmoth.

6.2.1.3 Launching the Components

The second step is for Distmoth to launch the components specified in the configuration file. Please refer to the syntax described in the next section (6.2.1.6). All components specified in the configuration file are launched in the order that they appear. Multiple instances of the sane component can be launched (for example, for players).

Prior to launching a given component instance, the next available machine is first drawn from the pool of available machines. The component instance is then launched by connecting via SSH to the

machine, and invoking the specified launch command. The connection is launched in a way so that it can be detached, and the process still runs in the background, in order to avoid having to maintain many open SSH connections. The output of the launched component can be redirected to a file for subsequent analysis, if needed.

In the event where all available machines have already been used, but component instances still need to be launched, then Distmoth starts reusing machines that already have component instances running (by going back to the top of the list).

Note that while the `machinelist.conf` file format allows for specifying machine groups, this feature is not yet implemented in Distmoth's execution engine.

6.2.1.4 Command-Line Interface

After all components have been launched, Distmoth enters a command-line mode where the user can perform additional tasks. Some built-in commands have been implemented (they can be found in the package `distmoth.commands`). For instance, one can launch additional instances of a given component (to launch 10 additional instances of the `player` component, one would type `launch player 10`), kill any running component instance (`kill` command), obtain the current list of all running components (`status` command), view the list of all active machines (`machines` command) as well as the `quit` command, which must be invoked when the experiment is done to kill all running component instances. Additional commands can be implemented by overriding the base class defined in `command.py`.

6.2.1.5 Killing all Running Components

A danger when running background tasks is that these tasks can still be active unknowingly, and can potentially consume large amounts of resources. Sometimes killing a parent process might not be sufficient as the parent might have spawned child processes that need to be terminated as well. Distmoth provides a mechanism to properly clean up tasks on active machines in the form of a virtual `reset` component (this component does not follow the regular execution flow of the configuration file).

The command corresponding to this reset command is the command that will be invoked to guarantee proper termination of all potentially running processes on any machine. Such a command will typically contain a combination of `kill`/`killall` shell commands under Linux.

The reset component is executed when one triggers the termination of one particular instance of a given component. It is also invoked on all currently active machines when one invokes the `quit` command to close all running components (freeing up all resources), which is in Distmoth terminology considered a *clean* shutdown. If the Distmoth process exits for a reason other than the invocation of the `quit` command, then it is not a *clean* shutdown and as a consequence, upon it's next invocation, Distmoth will trigger a reset on all previously active machines (the type of shutdown and the current list of active machines are persisted to disk). A similar scenario happens when some machines considered active cannot be reached during the invocation of a `quit` command: in this case, the machines for which the connection or the execution of the `quit` command failed are persisted so that the invocation of `quit` is rescheduled upon next starting Distmoth.

6.2.1.6 Configuration File Syntax

Listing 6.6: Configuration File Syntax

```
1 [reset]
2 command=killall -9 redis-server; killall -9 java
3 autoclose=true
4 wait=10
5
6 [pubsubserver0]
7 count=1
8 directory=/tmp/${USER}_mammoth/mammoth2
9 command=java -Xmx1500M -jar mammoth-test.jar rpubhub 0
10 host=server1.cloudprovider.com
11 detach=true
12 localoutput=true
13 wait=15
14
15 [pubsubserver1]
16 count=1
17 directory=/tmp/${USER}_mammoth/mammoth2
18 commannd=java -Xmx1500M -jar mammoth-test.jar rpubhub 1
19 host=server2.cloudprovider.com
20 detach=true
21 localoutput=true
22 wait=15
23
24 [gameserver]
25 count=1
26 directory=/tmp/${USER}_mammoth/mammoth2
27 command=java -Xmx1500M -jar mammoth-test.jar master
28 detach=true
29 localoutput=true
30 wait=15
31
```

```
[player]
count=10
directory=/tmp/${USER}_mammoth/mammoth2
command=java -Xmx1500M -jar mammoth-test.jar npc 30
detach=true
localoutput=true
wait=6

[killswitch]
count=0
directory=~/Documents/mammoth2
command=touch "killswitch/%datetime%.ks"
autoclose=true
host=lab7-2.cs.mcgill.ca
wait=60
```

Listing 6.6 shows an example of a simplified Distmoth configuration file that was used to run the Mammoth distributed system. The file is divided in sections that contain the name of a component in brackets. For each component, one can specify the number of instances that should be deployed (`count` parameter), the directory and command to be executed (multiple commands separated by semicolons can be invoked), and a duration in seconds to wait before launching the next instance / component. One can decide to set `count=0` if the desired behavior is to not launch any instance of that component automatically, but still allow the command to be invoked manually through the command-line interface of Distmoth. One can also use the `host` parameter to force a specific command to be executed on a specific host. Some additional parameters (not described for brevity) are supported.

The execution sequence follows the flow of the configuration file. In our example, pub/sub servers are launched first, then the game server, then 10 (groups of 30) players. Note that the two pub/sub servers are forced to launch on specific dedicated cloud servers (`host` parameter).

6.2.2 MUDPLaunch

While the use of Distmoth allowed us to save a lot of time when running our experiments, establishing all SSH connections prior to deploying all instances was still time-consuming. In order to address this issue and speed up the deployment process, we designed the MUDPLaunch tool, which allows one to install a simple command-line listener on a large group of machines in order to quickly invoke a given process on the target machine.

MUDPLaunch provides two components: MUDPReceiver, which listens for simple shell-like commands on a given UDP port, and MUDPBroadcaster, which transmits commands to MUDPReceiver

instances, using UDP unicast and broadcast. MUDPLaunch is not a replacement for Distmoth: it is used in combination with Distmoth, whenever it makes sense to use it.

In the context of our projects, we created a special Distmoth launcher file that launched the MUDPReceiver tool instead of launching some of our components directly, and we used MUDPBroadcaster to send the proper startup and kill commands.

The most noticeable advantage of the use of the MUDPLaunch tool in combination with Distmoth is that when we made changes to the source code, terminating and restarting the processes on every machine involved in the experiment was very fast, compared to using only Distmoth where all the programs needed to be restarted through Distmoth.

6.3 MultiPubSimulator Implementation

As explained in section 4.6, as part of our work on MultiPub, we ran a set of simulated global-scale experiments in multiple regions of the EC2 cloud. Running such experiments in all of the 10 regions would have been very complex from a logistical standpoint, due to many factors. Notably, as each region is independent, one has to replicate the configurations and the virtual machine images across each individual region. Furthermore, costs can be significant, as inter-cloud costs apply for outgoing bandwidth transmitted between regions (figure 4.1). For Asian and South-American regions in particular, outbound costs are very high.

As we mentioned previously, because of all these considerations, we decided to run some of our experiments in a simulated environment. For that, we built the MultiPubSimulator tool, which implements a simulation of the MultiPub model and it's different configurations.

6.3.1 High-Level Description

As briefly introduced in section 4.6.2, `MultiPubSimulator` can simulate the execution of MultiPub with any number of topics, subscriptions and publications. Each topic has a set of publishers and subscribers. For each publisher, a specific publication rate and publication size, as well as an upper bound in terms of maximum acceptable delivery time (max_T) and the ratio/percentile $ratio_T$ of all delivery time measurements that should be below max_T must be specified.

When launched, MultiPubSimulator proceeds according to the following steps:

1. MultiPubSimulator determines the optimal configuration for every topic T given the current set of publishers and subscribers on T as well as the delivery time bounds imposed on T. In other words, it solves the optimization problem defined in section 4.5 using our brute-force approach. As we demonstrated in the *Runtime Analysis* section (4.6.5), our brute-force approach can scale to considerable numbers of publishers and subscribers.
2. MultiPubSimulator runs a set of experiments as specified in the configuration file.
3. MultiPubSimulator produces a set of output files that are used to produce graphs with *gnuplot* using a set of predefined *gnuplot* configuration files. These output files are very specific to the needs of this thesis and were used to produce the graphs shown throughout the chapters of this thesis. Therefore, this aspect of the tool is not described further here.

In short, upon launching, the tool determines the most optimal configuration given the contents of the configuration file provided as input, and then runs various experiments and outputs relevant experimental data. MultiPubSimulator is invoked through it's main class module `multipubsimulator`, and takes as parameter the path to a configuration file that it takes as input and the name of an experiment to run. The format of the configuration file is described in the next section.

6.3.2 Input and Configuration

Listing 6.7: XML Configuration File Example

```
<MultiPubExperiments>
  <Experiments>
    <Experiment name="Ex0">
      <Topics>
        <Topic name="T1">
          <Publishers>
            <Repeat count="100">
              <Publisher region="sa-east-1" publications="1" publicationSize
                  ="1024" />
              <Publisher region="ap-southeast-1" publications="1" publicationSize
                  ="1024" />
            </Repeat>
          </Publishers>
          <Subscribers>
            <Repeat count="10">
              <Subscriber region="ap-southeast-1" />
              <Subscriber region="sa-east-1" />
```

```
16              </Repeat>
17            </Subscribers>
18          </Topic>
19        </Topics>
20        <Optimizers>
21          <Optimizer topic="T1" percentile="75" bound="200" />
22        </Optimizers>
23      </Experiment>
24 </MultiPubExperiments>
```

All input data that is needed by MultiPubSimulator is defined in a simple XML configuration file. The XML format was designed to be as convenient and flexible as possible.

The file allows one to define a set of experiments, each identified with a unique name. Upon invoking the simulator, one specifies the configuration file and the name of the experiment to be considered. Each experiment contains a series of named topics. Listing 6.7 shows a sample configuration file that contains a single experiment (`Ex0`), which contains one single topic (`T1`).

Inside the section corresponding to each topic, one defines the set of publishers and subscribers. For each publisher and subscriber, the cloud region to which the publisher (subscriber) is closest must be specified. A trivial approach is to define one publisher (subscriber) per line. As an example, the line `<Subscriber region="us-east-1" />` would add a subscriber who is closest to region us-east-1 (Virginia) (it might also be *close* to other regions, but it is chosen in such a way that it is *closest* to the specified region among all regions). The definition of a publisher requires two additional parameters: the number of publications per second that this publisher produces, as well as the size in bytes of each individual publication. While our MultiPub formal model allows for different sizes for each publication, we decided to simplify the implementation model.

Defining each publisher and subscriber individually can be cumbersome. As a way to simplify, `MultiPubSimulator` proposes some syntactic sugar in the form of a `Repeat` definition that allows one to repeat the same publisher or subscriber definition(s) up to n times. In our sample configuration files, the repeat statement is used to produce 100 publishers in cloud region `sa-east-1` (Sao Paulo) and 100 publishers in cloud region `ap-southeast-1` (Singapore), using only two individual definitions. Likewise, the same shortcut is used to produce 20 subscribers in Sao Paulo and 20 subscribers in Singapore.

Following the definition of topics, one must specify an optimizer for each topic. The optimizer contains the delivery time percentile $ratio_T$ (75% in the sample configuration file), as well as the

delivery time bound max_T (200ms).

The simulator then takes as input the publisher and subscriber definitions for each topic, as well as the optimization criteria, and then solves the optimization problem (and, if enabled, generates some relevant experimental output files). Note that the various definitions parsed from the configuration file are encapsulated into a rich object-oriented model that corresponds to the format of the configuration and that exposes classes to represent the various domain concepts such as `topics`, `publishers` and `subscribers` deriving from a common ancestor `client`, etc. We think that one could adapt our MultiPubSimulator tool for other topic-based pub/sub simulation projects.

6.3.3 Latency Databases

In order to properly emulate latency values, we created two databases (CSV files) based on latency measurements made between nodes of the King dataset and servers in each of the 10 Amazon EC2 regions. The methodology was described in section 4.6.1. In the MultiPub terminology, these databases model the L and L^R latency matrices, as they contain latency measurements between approximately 700 King nodes and each of the 10 regions, and between each pair of regions.

The latency database files are read when the simulator starts. Relevant Python modules expose functions to query for specific database values. As an example, one can query to obtain a virtual client (King node) that is closest to a given region latency-wise. This specific function is notably called when the configuration file is read, in order to initialize the sets of virtual publishers and subscribers.

Alternatively, MultiPub also supports querying for a virtual node that is *geographically* closest to any other given node or EC2 region (note that that this doesn't necessarily imply that the node will also be closest from a latency point of view).

6.3.4 Solving

The solving process of the simulator closely follows the solving process of the optimization problem outlined in section 4.5. To accomplish that goal, MultiPubSimulator provides a `topicConfiguration` module/class that abstracts a given configuration (choice of regions and whether to use direct or routed delivery) for a given topic T. As such, the set of possible configurations depends on the number of regions and the delivery approach. Following the steps described previously, the `topicOptimizer`

module of MultiPubSimulator generates the set of all possible configuration using brute-force, and then determines which one is the most optimal.

Latency values between publishers and cloud regions, between cloud regions, and latency values between cloud regions and subscribers are taken from the latency databases described in section 6.3.3. More precisely, for each `topicConfiguration` in the configuration space, appropriate latencies are used depending on the location (latency-wise) of all publishers/subscribers, the delivery scheme and the set of *enabled* regions in that configuration. Likewise, costs are computed based on the configuration, as per equations 4.3 and 4.4 of section 4.4.4. A list of all publications sent by all publishers is also built, which is needed to assert whether the delivery constraint of the said configuration is respected (equations 4.5 and 4.6 of section 4.5.1).

After performing all computations, the solver returns the most suitable `topicConfiguration`, according to the steps outlined in section 4.5.3 (minimal costs if delivery constraint can be respected, otherwise minimal latency).

6.3.5 Invoking MultiPubSimulator through Dynamoth

As mentioned on several occasions, in addition to simulation experiments, we also ran some experiments in the real EC2 cloud, albeit with only 3 EC2 regions instead of 10. Since `MultiPubSimulator` already had a fully implemented solver, it made sense to reuse the solver component of the simulator (in Python), rather than reimplementing a new solver in Java.

Therefore, the MultiPub rebalancer component discussed in section 6.1.8.2 invokes the `MultiPubSimulator` solver to obtain the most suitable configuration. Every time the rebalancer is invoked, it outputs a specifically crafted `MultiPubSimulator` configuration file that reflects the exact current conditions as observed by Dynamoth's load balancer. It then executes a `MultiPubSimulator` solver instance (by invoking the Python interpreter), and captures the output, which contains the most optimal configuration reflecting the current conditions, and applies that new configuration. For more information, the reader is encouraged to refer to our relevant cloud-based experiments in section 4.6.4.

7

Final Conclusions & Future Work

7.1 Final Conclusions

In this thesis, we made several important contributions towards scaling topic-based publish/subscribe systems in a cloud setting.

Our first system, Dynamoth, proposes a cloud-based service that addresses the scalability and load balancing aspects of large-scale topic-based pub/sub systems, with a particular emphasis on the needs of latency-constrained applications, such as multiplayer online games. Dynamoth notably proposes a load monitoring model, combined with a full hierarchical load balancing architecture that takes the specific characteristics of each topic into consideration in order to accurately balance the load across different servers in the cloud. As part of its load balancing strategy, Dynamoth supports migrating topics between servers in a seamless manner, as well as adding new and removing unused servers. In addition, Dynamoth also includes mechanisms to balance the load of high-load topics featuring many subscribers and/or publishers across multiple servers. A key aspect of Dynamoth's load balancing approach is that rebalancings are propagated in a lazy way, as to minimize their impact on overall system stability. Moreover, Dynamoth also provides performance-driven availability and fault tolerance, as it can detect publish/subscribe server failures and quickly reconfigure itself to compensate for the server loss. In this regard, Dynamoth allows for potentially missed publications to be automatically replayed, and provides different guarantee levels regarding reliability and message ordering during recovery depending on the needs of the application. In order to evaluate Dynamoth, we implemented the full model that it proposes within our Dynamoth platform implementation and ran several cloud experiments.

Our second system, MultiPub, builds on top of Dynamoth to address a further set of challenges. It proposes a novel global-scale pub/sub service that takes into consideration the availability of cloud resources in different regions of the world, the locality of clients dispersed within and across these regions and the latency requirements of the application of MultiPub. As such, MultiPub allows one to impose delivery time constraints on different topics within the system. MultiPub has two main goals: (1) meeting the per-topic minimum latency constraints that the application can impose on publication delivery, and (2) minimizing cloud-incurred costs. The MultiPub rebalancing framework gathers extensive load and response time metrics from all clients in a distributed manner, and realizes it's two main goals by carefully mapping the different topics of the pub/sub system to servers located in different regions, considering that deploying a given topic to a server in a given region might yield (1) reduced delivery times for some clients and (2) reduced costs as cloud-incurred outgoing bandwidth prices vary between regions. Thus, determining the best allocation strategy is formulated as an optimization problem, where meeting delivery time bounds becomes a constraint, and where costs are the optimization criteria. The validation of MultiPub is based on a simulator as well as a full implementation of MultiPub on top of our Dynamoth platform. Extensive experiments were run: simulation experiments were run locally and live experiments were dynamically run in the cloud, and the results compared.

Multiplayer games and their relationship with the topic-based publish/subscribe model were yet another major aspect of this thesis. Our third system, DynFilter, proposes a scalable topic-based pub/sub service specifically tailored for the needs of large-scale games. The main goal of DynFilter lies in reducing bandwidth usage in games relying on a topic-based publish/subscribe paradigm, which, in a cloud setting, translates to reducing costs. As DynFilter is targeted at games, it exploits game-centric interest management concepts to transparently suppress a portion of the state update publications when it has to reduce bandwidth use. By exploiting the game semantics, DynFilter removes publications that are of lesser importance while insuring that the more important state update publications are delivered consistently. As a result, DynFilter performs adaptive filtering based on currently observed conditions, in order to not exceed the application-defined bandwidth quota over a given time window. Such filtering is done in a fine-grained manner and notably takes into consideration the dispersion of players within the game. DynFilter was also fully implemented within our Dynamoth platform, and experiments were run in the context of different types of multiplayer games for validation purpose.

Our final contribution consists in a collection of software tools that we implemented to properly evaluate our different systems. The first is the implementation of the Dynamoth platform itself, which

was designed to support running large-scale pub/sub experiments in a real cloud settings. The platform was built following state-of-the-art software engineering principles to make its code base highly reusable, customizable and extensible. As a result, the platform was successfully used to implement the three models (Dynamoth, MultiPub and DynFilter) presented in this thesis, and then successfully used to run validation experiments of the different models with over 1200 simultaneous clients. Besides the Dynamoth implementation platform, we further include a set of tools to aid in the setup and configuration of large-scale cloud experiments (Distmoth and MUDPLaunch), as well as a full implementation of a simulation package in order to run simulation experiments for our MultiPub project.

Overall, we think that we brought significant contributions towards scaling topic-based publish/-subscribe systems in a cloud setting. As such systems are used across a wide range of applications, we think that the approaches that we proposed could very well be implemented in commercial pub/sub systems in order to support the scalability and the globality needs of large-scale applications. In the context of this thesis, a particular emphasis was put on meeting the needs of latency-sensitive applications, such as games, which could, in our opinion, greatly benefit from our novel latency-minimizing approaches. In addition, we think that integrating some of our approaches could also yield the added benefit of reducing bandwidth-incurred costs in the cloud.

7.2 Future Work

Our work done in the area of scalable topic-based publish/subscribe systems opens up many potential research directions. This sections lists some of these promising orientations that we intend to explore.

Taking CPU Load into Consideration In the context of our different contributions, we considered the problem of scaling topic-based publish/subscribe systems in the cloud. CPU was not really a limiting factor, as the matching process of topic-based systems is lightweight, as opposed to content-based systems where the CPU can be a constrained resource. Nevertheless, as these systems scale, in some circumstances (for instance, if virtual CPUs are used in the cloud), then the CPU can become a bottleneck. An interesting research direction would be to adapt our approaches and models to take CPU load into consideration as part of our load balancing approaches.

Clustering and Heuristic Approaches for MultiPub Our results already demonstrated that our MultiPub system could scale to important figures in terms of publishers, publications and subscribers,

despite having to solve an optimization problem to generate the best configuration. Our runtime analysis of MultiPub (section 4.6.5) however revealed that there were limitations beyond which the solving process became too CPU/time consuming. Some of these limitations were mitigated by reducing the configuration space (such as having less regions), or by having a more optimized implementation. An additional way to reduce the configuration space that we aim at implementing involves clustering approaches, so that several nodes located in the same geographical area, that share similar latencies towards the different cloud regions can be abstracted as *a cluster of* nodes. In the same spirit, in the context of Dynamoth, for systems with a huge amount of topics, one could group *similar* topics (similar because of significant overlap of their sets of publishers/subscribers) into clusters of topics.

Heterogeneity of Cloud Resources Our MultiPub model description and experiments considered the 10 regions of the Amazon EC2 cloud. However, our model is not bound to using resources from a single provider only. An interesting area of future work would involve using cloud deployments from several providers. Such a scenario could possibly lead to further cost reductions and potentially more optimized latencies. MultiPub could also be extended to support selecting heterogeneous configurations, as cloud providers typically offer different sets of virtual machines with varying specifications (network bandwidth, CPU, RAM, etc.) at different prices. The latter would also be an interesting area of future work in the context of Dynamoth, as different virtual machine instances could be deployed when adding new publish/subscribe servers depending on the current scalability needs. The Dynamoth load balancer would then have to be adapted to consider heterogeneous cloud resources.

Publish/Subscribe in an IoT Context Internet of Things represent a promising and novel research orientation, in which a large amount of common devices are now inter-connected. An increasing amount of such devices are now part of our daily life, such as home appliances or other objects. IoT also enjoys widespread usage in various other devices, such as smart vehicles, or smart meters in the power grid that transmit power consumption, or all kinds of medical devices that transmit a wide range of health-related measurements. Publish/subscribe paradigms can certainly play a role in the IoT world. For instance, medical devices could transmit live measurements for a given patient to a given topic, that the medical staff could subscribe to. On the other hand, smart vehicles could transmit live traffic information that other drivers or that the authorities could subscribe to. Applying pub/sub paradigms to IoT devices brings interesting research challenges. From a privacy standpoint, in the context of medical devices, one might wish to restrict subscribing to sensitive topics carrying patient-

related information to authorized subscribers only [17]. From a scalability standpoint, in the context of smart cars, subscribing to receive live measurements from a large amount of individual cars could consume too many resources. A given user might be interested in receiving information at different levels of details: receiving aggregated traffic information at a higher level of details when requesting a global view, and receiving more detailed information about individual cars when requesting a more local view, in a similar spirit as our work in [69]. Inspired by some ideas proposed in DynFilter, the pub/sub layer could be adapted to aggregate and disseminate publications at varying levels of details according to the needs of the subscribers.

Object Caching and Disseminating Service Besides publish/subscribe, key-value stores represent another popular paradigm in which key-value pairs of arbitrarily complex data are stored for easier, more efficient retrieval. A good application of key-value stores is web applications, in which the results of frequently run database queries are cached, for efficiency purposes. Upon a client requesting the result of a given database query, the application first determines if an appropriate cached result already exists in the caching service. In the affirmative, the cached result is returned, which spares the execution of the query on the database server, which can ultimately lead to significant performance improvements.

In a game context, especially in massive multiplayer online games, virtual environments typically contain a colossal amount of virtual objects that are frequently accessed or altered by many players. In order to access these virtual objects, players need to obtain a copy of such objects, and therefore the typical scenario is for such clients to request replicas of these objects from the game server/service [70]. Our empirical observations on Mammoth revealed that having the server generate such replicas on a scale of a magnitude of a MMOG represented a heavy task. Therefore, we believe that games could benefit from a caching service. However, an important limitation of typical key-value caching services is that they usually don't provide update operations on values, as they trivially consider values as sequences of bytes [51]; therefore, values can only be overwritten instead of updated. As future work, we would like to explore the idea of designing an object-aware caching service that would not only provide a key-value storage, but that would also consider values as real *objects* (classes). This would allow for performing dynamic operations on objects instead of completely replacing these objects. In addition, upon a client fetching a given object, the caching service could support that client subscribing to receive updates performed on that object, by using a publish/subscribe middleware such

as Dynamoth. All clients subscribing to the objects that they are interested in would then always be updated automatically with the latest, freshest version of these objects, which we believe could lead to important resource savings.

List of Publications

Published:

- J. Gascon-Samson, F. P. Garcia, B. Kemme, and J. Kienzle. Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 486–496, June 2015 *(acceptance ratio: 12.8%)*.
- H. Khan, J. Gascon-Samson, J. Kienzle, and B. Kemme. Monitoring large-scale location-based information systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1171–1181, May 2015 *(acceptance ratio: 21%)*.
- J. Gascon-Samson, J. Kienzle, and B. Kemme. Dynfilter: Limiting bandwidth of online games using adaptive pub/sub message filtering. In *Network and Systems Support for Games (NetGames), 2015 International Workshop on*, pages 1–6, Dec 2015.
- A. Yahyavi, K. Huguenin, J. Gascon-Samson, J. Kienzle, and B. Kemme. Watchmen: Scalable cheat-resistant support for distributed multi-player online games. In *ICDCS 2013*, pages 134–144, July 2013 *(acceptance ratio: 13%)*.
- Julien Gascon-Samson, Bettina Kemme, and Jörg Kienzle. Lamoth: A message dissemination middleware for mmogs in the cloud. In *Proceedings of Annual Workshop on Network and Systems Support for Games, NetGames '13*, pages 9:1–9:2, Piscataway, NJ, USA, 2013. IEEE Press.

To be submitted shortly:

- ICDCS 2017: Julien Gascon-Samson, Jörg Kienzle, and Bettina Kemme. MultiPub: Latency and Cost-Aware Global-Scale Cloud Publish/Subscribe.
- IEEE TPDS: J. Gascon-Samson, F. P. Garcia, B. Kemme, and J. Kienzle. Dynamoth: A Scalable and Available Pub/Sub Middleware for Latency-Constrained Applications in the Cloud.

Bibliography

[1] Redis website, 2013. 2.1.1, 3.2.2, 4.6.4

[2] Amazon SNS. http://aws.amazon.com/fr/sns/, 2014. 2.5

[3] Apache Hedwig Project. https://cwiki.apache.org/confluence/display/BOOKKEEPER/HedWig, 2014. 2.5

[4] Google Cloud Messaging. https://developer.android.com/google/gcm/index.html, 2014. 2.5

[5] Amazon Elastic Compute Cloud. https://aws.amazon.com/ec2/, 2016. 4.1

[6] Data Distribution Service 1.4. http://www.omg.org/spec/DDS/1.4/, 2016. 2.4.2

[7] Google Cloud Pub/Sub. https://cloud.google.com/pubsub/, 2016. 2.5

[8] Java Message Service Specifications v1.1. http://download.oracle.com/otndocs/jcp/7195-jms-1.1-fr-spec-oth-JSpec/, 2016. 2.4.2

[9] I. Aekaterinidis and P. Triantafillou. Pastrystrings: A comprehensive content-based publish/subscribe DHT network. In *Proc. 26th IEEE Int. Conf. Distributed Computing Systems (ICDCS'06)*, page 23, 2006. 2.3.2.3

[10] Norman Ahmed, Mark Linderman, and Jason Bryant. Papas: Peer assisted publish and subscribe. In *Workshop on Middleware for Next Generation Internet Computing (MW4NG)*, pages 7:1–7:6, 2012. 2.1.2, 2.3.2.3

[11] Kyoungho An, Aniruddha Gokhale, Sumant Tambe, and Takayuki Kuroda. Wide area network-scale discovery and data dissemination in data-centric publish/subscribe systems. In *Proceedings of the Posters and Demos Session of the 16th International Middleware Conference*, Middleware Posters and Demos '15, pages 6:1–6:2, New York, NY, USA, 2015. ACM. 2.4.2

[12] Kyoungho An, Aniruddha Gokhale, Sumant Tambe, and Takayuki Kuroda. Wide area network-scale discovery and data dissemination in data-centric publish/subscribe systems. Technical Report ISIS-15-120, EECS Department, Vanderbilt University, Nashville, Tennessee, 2015. 2.4.2

[13] E. Anceaume, M. Gradinariu, A. K. Datta, G. Simon, and A. Virgillito. A semantic overlay for self- peer-to-peer publish/subscribe. In *Proc. 26th IEEE Int. Conf. Distributed Computing Systems (ICDCS'06)*, page 22, 2006. 2.3.2.3

[14] Roberto Baldoni, Roberto Beraldi, Vivien Quema, Leonardo Querzoni, and Sara Tucci-Piergiovanni. Tera: Topic-based event routing for peer-to-peer architectures. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, DEBS '07, pages 2–13, New York, NY, USA, 2007. ACM. 2.3.2.2

[15] R. Barazzutti, T. Heinze, A. Martin, E. Onica, P. Felber, C. Fetzer, Z. Jerzak, M. Pasin, and E. Riviere. Elastic scaling of a high-throughput content-based publish/subscribe engine. In *ICDCS*, pages 567–576, 2014. 2.1.2, 2.3.3.3

[16] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert. Streamhub: A massively parallel architecture for high-performance content-based publish/subscribe. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 63–74, New York, NY, USA, 2013. ACM. 2.3.3.3

[17] Raphaël Barazzutti, Pascal Felber, Hugues Mercier, Emanuel Onica, and Etienne Rivière. Thrifty privacy: Efficient support for privacy-preserving publish/subscribe. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 225–236, New York, NY, USA, 2012. ACM. 7.2

[18] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003®. In *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '04, pages 144–151, New York, NY, USA, 2004. ACM. 2.6.1

[19] P. Bellavista, A. Corradi, and A. Reale. Quality of service in wide scale publish-subscribe systems. *IEEE Communications Surveys Tutorials*, 16(3):1591–1616, Third 2014. 2.4.2

[20] S. Benford and L. Fahlen. A spatial model of interaction in large virtual environments. pages 109 – 24, Dordrecht, Netherlands, 1993//. 2.6.2.2

[21] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *ACM SIGCOMM 2008 Conf. on Data Communication*, pages 389–400. 2.6, 2.6.2.1, 5.3.2.1

[22] Silvia Bianchi, Pascal Felber, and Maria Gradinariu. *Content-Based Publish/Subscribe Using Distributed R-Trees*, pages 537–548. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. 2.3.2.3

[23] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM. 2.6.2.2

[24] César Cañas, Eduardo Pacheco, Bettina Kemme, Jörg Kienzle, and Hans-Arno Jacobsen. Graps: A graph publish/subscribe middleware. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 1–12, New York, NY, USA, 2015. ACM. 2.1.3

[25] César Cañas, Kaiwen Zhang, Bettina Kemme, Jörg Kienzle, and Hans-Arno Jacobsen. Publish/subscribe network designs for multiplayer games. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 241–252, New York, NY, USA, 2014. ACM. 1.1

[26] César Cañas, Kaiwen Zhang, Bettina Kemme, Jörg Kienzle, and Hans-Arno Jacobsen. Publish/-subscribe network designs for multiplayer games. In *Middleware 2014*, pages 241–252, 2014. 2.6.2.3

[27] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and

Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM. 2.5

[28] F. Cao and J. P. Singh. Medym: match-early and dynamic multicast for content-based publish-subscribe service networks. In *25th IEEE International Conference on Distributed Computing Systems Workshops*, pages 370–376, June 2005. 2.3.1.2

[29] Fengyun Cao and J. P. Singh. Efficient event routing in content-based publish-subscribe service networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 929–940 vol.2, March 2004. 2.3.1.2

[30] N. Carvalho, F. Araujo, and L. Rodrigues. Scalable qos-based event routing in publish-subscribe systems. In *Fourth IEEE International Symposium on Network Computing and Applications*, pages 101–108, July 2005. 2.4.2

[31] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, August 2001. 2.1

[32] M. Castro, P. Druschel, A.-M. Kermarrec, and A.I.T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, 2002. 2.3.2.2, 2.4.1.2

[33] Raphaël Chand and Pascal Felber. *Semantic Peer-to-Peer Overlays for Publish/Subscribe Networks*, pages 1194–1204. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. 2.3.2.3

[34] T. Chang and H. Meling. Byzantine fault-tolerant publish/subscribe: A cloud computing infrastructure. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 454–456, Oct 2012. 2.4.1, 2.4.1.3

[35] Jin Chen, Baohua Wu, Margaret Delap, Björn Knutsson, Honghui Lu, and Cristiana Amza. Locality aware dynamic load management for massively multiplayer games. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 289–300, New York, NY, USA, 2005. ACM. 2.6.2

[36] Alex King Yeung Cheung and Hans-Arno Jacobsen. *Dynamic Load Balancing in Distributed Content-Based Publish/Subscribe*, pages 141–161. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. 2.3.1.2

[37] Alex King Yeung Cheung and Hans-Arno Jacobsen. Load balancing content-based publish/subscribe systems. *ACM Trans. Comput. Syst.*, 28(4):9:1–9:55, December 2010. 2.3.1.2

[38] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Constructing scalable overlays for pub-sub with many topics. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 109–118, 2007. 2.3.2.2

[39] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Spidercast: A scalable interest-aware overlay for topic-based pub/sub communication. In *DEBS*, pages 14–25, 2007. 2.3.2.2

[40] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):45, 2006. 2.6, 2.6.1, 4.6.4.2

[41] M. Claypool, D. Finkel, A Grant, and M. Solano. Thin to win? network performance analysis of the onlive thin client game system. In *Network and Systems Support for Games (NetGames), 2012 11th Annual Workshop on*, pages 1–6, Nov 2012. 2.6.3

[42] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. 2.3.3.1

[43] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007. 2.3.3.1

[44] Stephen E Deering and David R Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Transactions on Computer Systems (TOCS)*, 8(2):85–110, 1990. 2.1.1

[45] A. Denault and J. Kienzle. Journey: A massively multiplayer online game middleware. *IEEE Software*, 28(5):38–44, Sept 2011. 1.1, 2.6.2, 2.6.2.2

[46] Alexandre Denault and Jörg Kienzle. The perils of using simulations to evaluate massively multiplayer online game performance. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, pages 4:1–4:8, ICST, Brussels, Belgium, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 3.1, 6.2

[47] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003. 2.1, 2.1.1, 2.1.2

[48] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 115–126, New York, NY, USA, 2001. ACM. 2.1

[49] W. Fan, Y. Liu, and B. Tang. Gem: An analytic geometrical approach to fast event matching for multi-dimensional content-based publish/subscribe services. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016. 2.3.1.2

[50] Eli Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. The padres distributed publish/subscribe system. In *FIW*, pages 12–30. Citeseer, 2005. 2.1.2, 2.3.1.2, 2.3.3.3, 2.5

[51] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004. 7.2

[52] Franz-Philippe Garcia. Channel replication in dynamoth: Implementation and analysis. Technical Report COMP400-F14-FPG, Department of Computer Science, McGill University, 2014. (document), 3, 3.4.1.4

[53] J. Gascon-Samson, F. P. Garcia, B. Kemme, and J. Kienzle. Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 486–496, June 2015. (document), 3, 4, 4.1, 4.3.1, 4.6.4, 4.7, 6.1.7

[54] J. Gascon-Samson, J. Kienzle, and B. Kemme. Dynfilter: Limiting bandwidth of online games using adaptive pub/sub message filtering. In *Network and Systems Support for Games (NetGames), 2015 International Workshop on*, pages 1–6, Dec 2015. (document), 5

[55] Julien Gascon-Samson, Bettina Kemme, and Jörg Kienzle. Lamoth: A message dissemination middleware for mmogs in the cloud. In *Proceedings of Annual Workshop on Network and Systems Support for Games*, NetGames '13, pages 9:1–9:2, Piscataway, NJ, USA, 2013. IEEE Press. (document), 6.1.3.2

[56] F. Glinka, A. Ploss, S. Gorlatch, and J. Muller-Iden. High-level development of multiserver online games. *International Journal of Computer Games Technology*, 2008:16 pp., 2008. 2.6.2

[57] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *ACM SIGCOMM Workshop on Internet Measurment (IMW)*, pages 5–18, 2002. 3.6.2, 4.6.1.2, 6.1.3.4

[58] S. Guo, K. Karenos, M. Kim, H. Lei, and J. Reason. Delay-cognizant reliable delivery for publish/subscribe overlay networks. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 403–412, June 2011. 2.4.2

[59] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: Content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '04, pages 254–273, New York, NY, USA, 2004. Springer-Verlag New York, Inc. 2.3.2.3

[60] Thomas Heinze, Patrick Meyer, Zbigniew Jerzak, and Christof Fetzer. Demo: Measuring and estimating monetary cost for cloud-based data stream processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 333–334, New York, NY, USA, 2013. ACM. 2.4.3

[61] R. Hong, S. Shin, Y. Yoon, A. Laxmankatole, and H. Woo. Global-scale event dissemination on mobile social channeling platform. In *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on*, pages 210–219, April 2014. 2.4.2

[62] Shun-Yun Hu. Spatial publish subscribe. In *The 2nd International Workshop on Massively Multiuser Virtual Environments at IEEE Virtual Reality 2009*, 2009. 2.1.3

[63] Shun-Yun Hu and Kuan-Ta Chen. Vso: Self-organizing spatial publish subscribe. In *Self-Adaptive and Self-Organizing Systems (SASO)*, pages 21–30, 2011. 2.6.2.3

[64] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010. 2.3.3.3, 2.4.2

[65] Beihong Jin, Xinchao Zhao, Zhenyue Long, Fengliang Qi, and Shuang Yu. Effective and efficient event dissemination for rfid applications. *The Computer Journal*, 2009. 2.3.3.3

[66] G. Jung, M. A. Hiltunen, K. R. Joshi, R. D. Schlichting, and C. Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 62–73, June 2010. 2.3.3.1

[67] R. S. Kazemzadeh and H. A. Jacobsen. Reliable and highly available distributed publish/subscribe service. In *Reliable Distributed Systems, 2009. SRDS '09. 28th IEEE International Symposium on*, pages 41–50, Sept 2009. 2.4.1, 2.4.1.2

[68] Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. Introducing publiy: A multi-purpose distributed publish/subscribe system. In *Proceedings of the Posters and Demo Track*, Middleware '12, pages 1:1–1:2, New York, NY, USA, 2012. ACM. 2.4.1, 2.4.1.2

[69] H. Khan, J. Gascon-Samson, J. Kienzle, and B. Kemme. Monitoring large-scale location-based information systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1171–1181, May 2015. 2.6.2.1, 7.2

[70] Jörg Kienzle, Clark Verbrugge, Bettina Kemme, Alexandre Denault, and Michael Hawker. Mammoth: a massively multiplayer game research framework. In *Foundations of Digital Games (FDG)*, pages 308–315, 2009. 1.1, 2.6.2, 2.6.2.2, 2.6.2.3, 3.6.1, 6.1.4.2, 7.2

[71] J Kreps, N. Narkhede, and Jun. Rao. Kafka, a distributed messaging system for log processing. In *NetDB'11*, 2011. 2.5

[72] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. 2.3.3.3, 2.4.1

[73] Ming Li, Fan Ye, Minkyong Kim, Han Chen, and Hui Lei. A scalable and elastic publish/subscribe service. In *IPDPS*, pages 1254–1265, 2011. 2.1.2, 2.1.2.1, 2.3.3.3

[74] W. Li and S. Vuong. Towards a scalable content-based publish/subscribe service over dht. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–6, Dec 2010. 2.3.2.3

[75] X. Ma, Y. Wang, and X. Pei. A scalable and reliable matching service for content-based publish/subscribe systems. *IEEE Transactions on Cloud Computing*, PP(99):1–1, 2014. 2.3.3.3

[76] Xingkong Ma, Yijie Wang, Qing Qiu, Weidong Sun, and Xiaoqiang Pei. Scalable and elastic event matching for attribute-based publish/subscribe systems. *Future Generation Computer Systems*, 36:102 – 119, 2014. 2.3.3.3

[77] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: providing consistency for replicated continuous applications. *Multimedia, IEEE Transactions on*, 6(1):47 – 57, feb. 2004. 2.6.2.1

[78] Tova Milo, Tal Zur, and Elad Verbin. Boosting topic-based publish-subscribe systems with dynamic clustering. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 749–760, New York, NY, USA, 2007. ACM. 2.3.2.2

[79] Jens Müller, Andreas Gössling, and Sergei Gorlatch. On correctness of scalable multi-server state replication in online games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM. 2.6.2

[80] Mahdi Tayarani Najaran, Shun-Yun Hu, and Norman C. Hutchinson. Spex: Scalable spatial publish/subscribe for distributed virtual worlds without borders. In *ACM Multimedia Systems Conference (MMSys)*, pages 127–138, 2014. 2.6, 2.6.1, 2.6.2.3

[81] J. Nichols and M. Claypool. The effects of latency on online madden NFL football. In *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 146–151. ACM New York, NY, USA, 2004. 4.6.4.2

[82] Lothar Pantel and Lars C. Wolf. On the impact of delay on real-time multiplayer games. In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '02, pages 23–29, New York, NY, USA, 2002. ACM. 2.6

[83] Lothar Pantel and Lars C. Wolf. On the suitability of dead reckoning schemes for games. In *NetGames 2002*, pages 79–84, 2002. 2.6.2.1, 5.2.1

[84] P.R. Pietzuch and J.M. Bacon. Hermes: a distributed event-based middleware architecture. In *ICDCS Workshops*, pages 611–618, 2002. 2.1.2, 2.3.2.2

[85] Daniel Pittman and Chris GauthierDickey. A measurement study of virtual populations in massively multiplayer online games. In *NetGames 2007*, pages 25–30, 2007. 2.6.2

[86] W. Rao, L. Chen, P. Hui, and S. Tarkoma. Move: A large scale keyword-based content filtering and dissemination system. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 445–454, June 2012. 2.3.1.2

[87] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM. 2.3.2.3

[88] David S. Rosenblum and Alexander L. Wolf. A design framework for internet-scale event observation and notification. In *ESEC*, pages 344–360, 1997. 2.1.2

[89] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag. 2.3.1, 2.3.2, 2.3.2.1, 2.3.2.3

[90] Pooya Salehi, Christoph Doblander, and Hans-Arno Jacobsen. Highly-available content-based publish/subscribe via gossiping. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 93–104, New York, NY, USA, 2016. ACM. 2.4.1, 2.4.1.2

[91] Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content based routing with elvin4. In *Proceedings AUUG2k*, 2000. 2.1.2.1

[92] V. Setty, G. Kreitz, G. Urdaneta, R. Vitenberg, and M. van Steen. Maximizing the number of satisfied subscribers in pub/sub systems under capacity constraints. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 2580–2588, April 2014. 2.4.2

[93] V. Setty, R. Vitenberg, G. Kreitz, G. Urdaneta, and M. van Steen. Cost-effective resource allocation for deploying pub/sub on cloud. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 555–566, June 2014. 2.4.3, 4.2

[94] Vinay Setty, Maarten Van Steen, Roman Vitenberg, and Spyros Voulgaris. Poldercast: Fast, robust, and scalable architecture for p2p topic-based pub/sub. In *International Middleware Conference (Middleware)*, pages 271–291, 2012. 2.3.2.2

[95] R. Shea and Jiangchuan Liu. On gpu pass-through performance for cloud gaming: Experiments and analysis. In *Network and Systems Support for Games (NetGames), 2013 12th Annual Workshop on*, pages 1–6, Dec 2013. 2.6.3

[96] R. Shea, Jiangchuan Liu, E.C.-H. Ngai, and Yong Cui. Cloud gaming: architecture and performance. *Network, IEEE*, 27(4):16–21, July 2013. 2.6.3

[97] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM. 2.3.1, 2.3.1.1, 2.3.2, 2.3.2.3

[98] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, February 2003. 2.3.1, 2.3.1.1, 2.3.2, 2.3.2.3

[99] Christof Strauch. Nosql databases. Technical report, Hochschule der Medien, Stuttgart (Stuttgart Media University), 2011. 2.3.3.1

[100] David Tam, Reza Azimi, and Hans-Arno Jacobsen. *Building Content-Based Publish/Subscribe Systems with Distributed Hash Tables*, pages 138–152. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. 2.3.2.3

[101] Pin-Yun Tarng, Kuan-Ta Chen, and Polly Huang. An analysis of wow players' game hours. In *NetGames 2008*, pages 47–52, 2008. 2.6.2

[102] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *Proceedings of the 2Nd International Workshop on Distributed Event-based Systems*, DEBS '03, pages 1–8, New York, NY, USA, 2003. ACM. 2.3.2.3

[103] Spyros Voulgaris, Etienne Riviere, Anne-Marie Kermarrec, Maarten Van Steen, et al. Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. In *IPTPS*, 2006. 2.3.2.3

[104] A. Yahyavi, K. Huguenin, J. Gascon-Samson, J. Kienzle, and B. Kemme. Watchmen: Scalable cheat-resistant support for distributed multi-player online games. In *ICDCS 2013*, pages 134–144, July 2013. 2.6, 2.6.1, 2.6.2.1, 5.3.2.1, 5.4.2.1

[105] Amir Yahyavi, KÃ©vin Huguenin, and Bettina Kemme. Interest modeling in games: The case of dead reckoning. *Multimedia Syst.*, 19(3):255–270, June 2013. 5.2.1

[106] Amir Yahyavi and Bettina Kemme. Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Comput. Surv.*, 46(1):p1–51, 2013. 2.6.2.1

[107] Y. Yoon, V. Muthusamy, and H. A. Jacobsen. Foundations for highly available content-based publish/subscribe overlays. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 800–811, June 2011. 2.4.1, 2.4.1.1, 2.4.1.2

[108] Biao Zhang, Beihong Jin, Haibiao Chen, and Ziyuan Qin. Empirical evaluation of content-based pub/sub systems over cloud infrastructure. In *Intl. Conference on Embedded and Ubiquitous Computing (EUC)*, pages 81–88, 2010. 2.1.2, 2.3.3.3

[109] Chi Zhang, Arvind Krishnamurthy, Randolph Y Wang, and Jaswinder Pal Singh. Combining flexibility and scalability in a peer-to-peer publish/subscribe system. In *Proceedings of the*

ACM/IFIP/USENIX 2005 International Conference on Middleware, pages 102–123. Springer-Verlag New York, Inc., 2005. 2.3.2.3

[110] Ye Zhao, Kyungbaek Kim, and Nalini Venkatasubramanian. Dynatops: A dynamic topic-based publish/subscribe architecture. In *DEBS*, pages 75–86, 2013. 2.3.1.1, 2.4.1.2

[111] Suiping Zhou, Wentong Cai, Bu-Sung Lee, and Stephen J. Turner. Time-space consistency in large-scale distributed virtual environments. *ACM Trans. Model. Comput. Simul.*, 14(1):31–47, January 2004. 2.6.2.1

Acronyms

AOI	Area of interest
API	Application programming interface
CPU	Central processing unit
DDS	Data distribution service
DHT	Distributed hash table
FPS	First person shooter
IaaS	Infrastructure as a service
IoT	Internet of things
LB	Load balancer (specific to Dynamoth)
LLA	Local load analyzer (specific to Dynamoth)
MMO	Massive multiplayer online
MMOG	Massive multiplayer online game
MMORPG	Massive multiplayer online role-playing game
PaaS	Platform as a service
P2P	Peer to peer
QoS	Quality of service
SaaS	Software as a service
VM	Virtual machine