BubbleStorm: Replication, Updates, and Consistency in Rendezvous Information Systems

BubbleStorm: Replikation, Updates und Konsistenz in Rendezvous-Informationssystemen

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte Dissertation zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)

von Dipl.-Inform. Christof Leng aus Friedberg (Hessen)

August 2012 — Darmstadt — D 17

BubbleStorm: Replication, Updates, and Consistency in Rendezvous Information Systems
BubbleStorm: Replikation, Updates und Konsistenz in Rendezvous-Informationssystemen

Genehmigte Dissertation von Dipl.-Inform. Christof Leng aus Friedberg (Hessen)

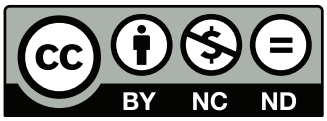
1. Gutachten: Prof. Alejandro P. Buchmann, PhD
2. Gutachten: Prof. Dr.-Inf. Bettina Kemme
3. Gutachten: Prof. Dr.-Ing. Klaus Wehrle

Tag der Einreichung: 03.07.2012
Tag der Prüfung: 22.08.2012

Darmstadt — D 17

Please cite this document as:
URN: urn:nbn:de:tuda-tuprints-urn:nbn:de:tuda-tuprints-30780
URL: http://tuprints.ulb.tu-darmstadt.de/30780

This document is made available by tuprints,
the e-publishing service of TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de



This work is licensed under the Creative Commons
Attribution – NonCommercial – NoDerivs 3.0 Germany License.
To view a copy of this license,
visit http://creativecommons.org/licenses/by-nc-nd/3.0/de/.

To Vera and Samson.

Abstract

As distributed systems are getting more and more complex, search facilities for finding services and data within the system become crucial. Users expect search engines to deal with complex query languages like keyword search, SQL, or XPath. At the same time, application developers cannot be expected to come up with distributed versions of those query languages from scratch. Rendezvous search systems are a very scalable solution to this problem. By separating the query processing from the network communication, existing libraries for query processing can be easily reused.

A wide range of rendezvous search systems for different scenarios has been proposed in the past. Their scalability and resilience make them an excellent choice for search in large-scale and dynamic peer-to-peer environments. The resilience stems mainly from the high number of replicas per datum, which however makes replica maintenance difficult. Unfortunately, most rendezvous search systems lack maintenance algorithms to sustain the desired replica count under node churn.

Replica maintenance is closely related to update mechanisms for mutable data. The highly distributed nature of peer-to-peer systems in general and the high replica count of rendezvous search systems in particular require carefully designed mechanisms for consistent updates with concurrent accesses.

In this thesis, replica maintenance and update mechanisms for the BubbleStorm peer-to-peer overlay and related rendezvous search systems are introduced. After analyzing the design space of replica maintenance for peer-to-peer systems, a complete solution covering all identified use cases is presented. This includes a maintainer-based mechanism for data managed by a single node and a collective mechanism for data that shall be persistent beyond any particular node's session time.

The algorithms are evaluated in BubbleStorm's sophisticated testbed, which allows prototype experiments and simulations with the same source code.

Zusammenfassung

Da verteilte Systeme immer komplexer werden, kommt Suchmechanismen zum Auffinden von Services und Daten eine stetig wachsende Bedeutung zu. Anwender erwarten, dass Suchmaschinen komplexe Anfragesprachen wie Volltextsuche, SQL oder XPath verarbeiten können. Gleichzeitig kann von Anwendungsentwicklern jedoch nicht erwartet werden, dass sie verteilte Versionen dieser Anfragesprachen von Grund auf selbst implementieren. Rendezvous-Suchsysteme stellen eine hochgradig skalierbare Lösung für dieses Problem dar. Durch die Trennung von Anfragebearbeitung und Netzwerkkommunikation können bestehende Implementierungen der Anfragesprachen leicht wiederverwendet werden.

Eine breite Palette an Rendezvous-Suchsytemen wurde bereits für verschiedene Szenarien vorgeschlagen. Ihre Skalierbarkeit und Robustheit macht sie zu einer ausgezeichneten Wahl für die Suche in großen und dynamischen Peer-to-Peer-Umgebungen. Diese Robustheit basiert zu großen Teilen auf der hohen Anzahl von Replikaten pro Datum, wodurch allerdings die Replikaverwaltung erschwert wird. Leider fehlt den meisten Rendezvous-Suchsystemen eine Replikaverwaltung, welche die gewünschte Anzahl der Replikate bei Veränderungen der Netzwerkzusammensetzung aufrecht erhält.

Die Replikaverwaltung ist eng verwandt mit Updatemechanismen für veränderliche Daten. Die verteilte und dezentrale Natur von Peer-to-Peer-Systemen im Allgemeinen und die hohe Anzahl von Replikaten in Rendezvous-Suchsystemen im Speziellen erfordern sorgfältig gestaltete Mechanismen für konsistente Updates bei konkurrierenden Zugriffen.

In dieser Dissertation werden Replikaverwaltung und Updatemechanismen für das Peer-to-Peer-Overlay BubbleStorm und verwandte Rendezvous-Suchsysteme vorgestellt. Nach Analyse des Entwurfraums für die Replikaverwaltung in Peer-to-Peer-Systemen wird eine vollständige Lösung für alle identifizierten Anwendungsfälle präsentiert. Dies beinhaltet einen verwalterbasierten Mechanismus für Daten, die von einem einzelnen Knoten verwaltet werden, und einen kollektiven Mechanismus für Daten, welche über die Onlinezeit jedes einzelnen Knotens hinaus verfügbar bleiben sollen.

Die Algorithmen werden mit BubbleStorms hoch entwickelter Testumgebung evaluiert, die es erlaubt, den selben Quellcode sowohl für Prototypenexperimente als auch für Simulationen zu verwenden.

Acknowledgments

When I started working on this PhD project, I had only a vague idea of where I was going. Looking back, I am astonished how much I have learned, and how big the whole BubbleStorm project got over the years. This would never been possible without all the people who accompanied me or who I met during this journey. Here, I want to express my gratitude to them.

First and foremost, my advisor Alejandro Buchmann, for his faith in my work and me. His advice and guidance were essential to this thesis and the whole process of becoming a researcher. He gave Wesley and me the freedom to pursue some of the most crazy research ideas we had over the years. My co-advisors Bettina Kemme and Klaus Wehrle, for their advice and feedback, which has been invaluable for the preparation of this thesis. I cannot imagine a better team of advisors for a work at the intersection of computer networking and management of data.

The whole DVS lab, for being a great team and an amazing place to work at. I will miss working with you. The DFG QuaP2P research group, for providing an unique environment for P2P research, in which I learned a lot about computer networking and conducting scientific projects. My BubbleStorm collaborators Wesley and Max, for being an inspiring team and for creating a cool project. Welsey, for always bringing up ambitious research goals and being keen enough to pursue them. Max, for being there whenever there is need and silently solving problems. All the undergraduate students I had the chance to work with and who contributed to the project. Marco, for designing the BubbleStorm logo, and Benjamin, for designing the cover picture of my thesis. Max, Ruth, Stefan, and Vera, for proof-reading and providing valuable feedback.

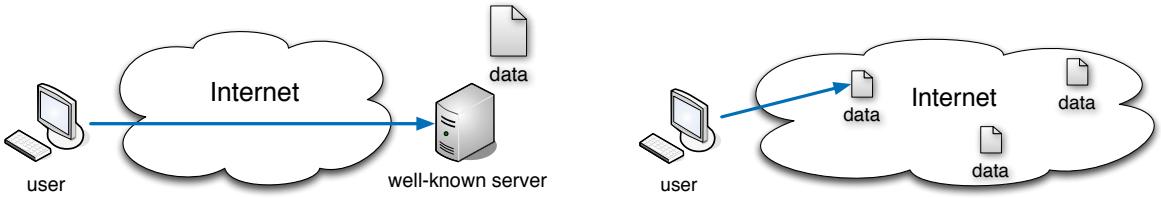
My parents, for making me who I am today, shaping my perspective of the world, supporting me through the years, and teaching me to always believe in myself. My siblings Stefan, Karena, and Benjamin, for all the inspiring discussions we had and all I learned from them. All of my family, Vera's family, and our friends for being with me during highs and lows.

Vera, for being the love of my life and for sharing our lives, whatever will come. Samson, who was born during my PhD, for always putting a smile on my face.

Contents

1 Introduction

In the past decade, the way the Internet is used has changed significantly. Traditionally, it was understood as a routing network, connecting clients to well-known and static servers, and was depicted as a cloud in schematic figures. Data was downloaded from or uploaded to the servers and thus passed through the network, but never stayed within the cloud (see Figure 1.1a). Albeit the technical principles of the Internet persist mostly unmodified, the user-level abstraction has changed fundamentally. Nowadays, more and more computing tasks and the storage of data are moved into the cloud, represented by anonymous servers in large data centers, dynamically assigned to paying customers. Users often do not know (and typically do not care) where exactly the data is stored, as long as it is available anytime and anywhere (see Figure 1.1b). To ensure availability, the data is often replicated among multiple servers or even data centers. Nonetheless, users expect to modify data in real-time and always get the latest version of requested data. Retrieving or modifying the distributed data in large data centers has to be coordinated by special servers with global knowledge of the multiple and dynamic locations of each data item.



(a) Traditional view **(b)** Cloud computing

Figure 1.1.: How cloud computing changed the abstraction of the Internet

The success of the Internet and the rampant growth of digital data has also changed the way of accessing data. Instead of manually navigating through hierarchical or otherwise structured sets of data, content-based search enables to find the desired information using a few keywords or attributes. The full-text search boxes on countless websites are as ubiquitous as the big search engines that make the world wide web usable. Their success is echoed by desktop search engines and improved search facilities in desktop applications. As it seems, the future of data is less structured than expected.

Therefore, two of the key requirements for many networked applications nowadays are maintaining the distributed data and supporting flexible content-based search on this data. In the realm of client-server computing, a versatile and field-proven toolbox of frameworks, libraries, and server suites for the development of such applications is available (like Apache Hadoop [148]) and makes the life of developers comparatively easy. One of the current major research challenges for those frameworks is scalability, i.e., to deal with large amounts of users accessing large sets of information, the so-called “big data”.

Scalability is an inherent feature of peer-to-peer computing. A peer-to-peer application benefits from the capacity that each peer brings to the system, which can counterbalance the additional load the peer induces. Even though servers can be made scalable by using large-scale clusters, the costs involved may be beyond the capabilities of many projects.

With all the peer-to-peer research of the past years and the enormous demand for scalable solutions, one would expect a boom of peer-to-peer applications. Unfortunately, the opposite is true. Very few new peer-to-peer applications are released, some providers of peer-to-peer-based services are switching to client-server infrastructures [4], and even prominent peer-to-peer software like BitTorrent [25] heavily relies on servers for coordination and search. Especially striking is the absence of sophisticated applications beyond the traditional peer-to-peer strong points like file sharing, instant messaging, and video streaming. Even those markets are dominated by companies exclusively focused on peer-to-peer software like Skype or PPlive.

The key to understanding the current lean time of peer-to-peer is to have a closer look at the outlined application requirements. Traditionally, application developers had to choose between unstructured peer-to-peer overlays, which offer flexible search capabilities but do not scale, and structured peer-to-peer overlays, which offer scalable key-value-lookups but are complicated when it comes to more sophisticated search methods. Additionally, the reliable long-term replication and consistency of updated data in peer-to-peer overlays might be an area which did not get the attention it deserves.

Putting together a working peer-to-peer system with full-text search and data replication from the current state of the art is an almost impossible task for a non-expert in peer-to-peer networking and challenging even for experts. This explains why peer-to-peer algorithms are not yet used widely in the software industry. Because search algorithms need to be adapted (read: reimplemented) for structured overlays, an application developer would not only need domain-specific knowledge but also expertise in both peer-to-peer networking and information retrieval.

To solve this problem, a proper abstraction for peer-to-peer frameworks is needed that not only enables application developers to treat the networking aspect as a black box, but also to re-use existing libraries for search algorithms. If it could solve the problem of data replication on top, such a framework would definitely help closing the gap between peer-to-peer and client-server computing.

In this thesis I present the BubbleStorm [143] peer-to-peer system, which is a powerful search framework that allows the re-use of search algorithms. Furthermore, I define replication modes that cover typical application scenarios. Each of the replication modes is implemented by a replication and update mechanism usable by BubbleStorm or comparable search overlays. The key insights of those mechanisms and the abstract replication modes advance the topic of replication in large-scale distributed systems in general.

BubbleStorm is an implementation of the rendezvous search concept, which is discussed in Chapter 2. Rendezvous search is a paradigm for distributed “blind search” (or arbitrary search) that is optimized for demanding query languages like keyword search. Although a noteworthy number of rendezvous search systems have been pro-

posed lately, there exists no survey of these approaches yet. Chapter 2 closes this gap by giving a comprehensive overview of the field.

Chapter 3 provides a closer look at the existing BubbleStorm infrastructure. This includes a brief introduction of the underlying theory, the overlay topology, the bubblecast search algorithm, and the gossip algorithm, which provides system-wide statistics for self-organization. Additionally, the CUSP transport protocol is introduced, which was developed in the context of BubbleStorm to overcome limitations of TCP in complex peer-to-peer environments. It is used as the basis for the implementation of BubbleStorm in this thesis.

Chapter 4 discusses the current state of the art of replication and updates in peer-to-peer search overlays. An analysis of the application requirements uncovers four fundamental replication modes for distributed systems. Depending on its persistence, lifetime, and ownership conditions, a data type can be classified as instant, fading, managed, or durable. Use-cases explain why each mode is necessary and a discussion of the related work shows, that mixing the modes in a replication algorithm is not recommendable.

Chapter 5 shows how the replication modes can be mapped to BubbleStorm. The data description primitives allow the definition of data types in the schema of a BubbleStorm application. Each bubble type can follow one of the four replication modes, and a fifth primitive is used to define the relationships between the bubble types. Instant and fading bubble types are naturally supported by bubblecast. For managed and durable types additional replication mechanisms are required. The chapter concludes with a discussion of how to use the primitives to implement different example applications.

Managed bubble types are implemented using maintainer-based replication, which is presented in Chapter 6. The algorithm enables maintenance and updates with eventual consistency on data items, which are controlled by a dedicated maintainer. Durable bubble types use the collective replication mechanism, which is presented in Chapter 7. Collective replication supports long-term persistence and concurrent updates in application scenarios where no dedicated maintainer for a certain data item is available. With a solution for each of the four replication modes and its powerful search capabilities, BubbleStorm offers a complete but very manageable framework for sophisticated peer-to-peer application developers.

Evaluating a distributed system at the scale and complexity of BubbleStorm is a challenging task on its own. The available evaluation methods are discussed in Chapter 8. The sophisticated evaluation framework for large-scale distributed systems developed for this work is presented in Chapter 9. It offers large-scale simulation, prototype experiments on network testbeds like PlanetLab, and building real-world libraries and applications, all without changing the application source code.

In Chapter 10, the presented algorithms are evaluated using the evaluation framework. The results prove the adherence of the given success guarantees, the low communication cost, and the extreme robustness against network disruptions. The system is highly self-adaptive and quickly adjusts to changing environments.

In summary, this thesis presents a novel taxonomy for replication algorithms in decentralized distributed systems, describes the peer-to-peer search overlay BubbleStorm, which is able to cover the complete design space of the taxonomy and yet provides a

simple abstraction to use the system in application development. Due to its powerful search and replication capabilities, BubbleStorm may enable a new generation of sophisticated peer-to-peer applications. Before discussing the problem statement and the contributions in detail, a short introduction to the terms and concepts used in this thesis is given.

1.1 Background

In a distributed system, multiple computers are connected through a communication network and appear as coherent system to the user [27, 138]. A distributed system provides services to the users. A *service* can range from a simple computation on a single machine to a complex state-changing operation involving many computers. Individual services are often combined into more complex and powerful composite services. A host providing a service is called a *server* and a host requesting and consuming a service a *client*.

1.1.1 Design Goals of Distributed Systems

Diverse goals are involved in the design of distributed systems [27, 138]. This work will focus on the goals most relevant to peer-to-peer systems: scalability, availability, performance, fault tolerance, and consistency [50, 58]. While security is also a very important topic for distributed systems, it is beyond the scope of this work. Since the definition of those terms vary in the literature, a brief discussion is given how the terms are understood in the context of this work.

- *Performance* is a measure for the service quality of a distributed system. The most important metrics for performance are throughput and response time. *Throughput* is the amount of requests or bytes served per unit of time, while the *response time* describes the time an individual request needs to complete (also called *latency*).
- *Efficiency* measures the ratio between performance and cost. Cost is typically expressed in terms of consumed bandwidth, number of messages exchanged, number of connections, memory usage, or CPU consumption, depending on the scenario. Cost can also be expressed as the monetary cost for the acquisition and operation of a distributed system.
- *Scalability* describes the ability of a distributed system to improve its performance by increasing the number of service-providing computers (the network size n). A scalable system is able to cope with a wide range of network sizes, from small to large.
- *Elasticity* means that the system is additionally able to adjust to changing network sizes during operation. Elasticity is expressed as absolute or relative size change per unit of time.
- *Fault tolerance* describes the ability of a distributed system to cope with failures of services, servers, or the communication infrastructure. In particular, a system is *partition tolerant* if it is able to deal with the temporary loss of connectivity between parts of the system. An unfortunate combination of message losses between any two participants can qualify as a partition.
- *Availability* is the probability that the system is operating at a specified time, where operating means that the system is processing service requests from users. This does not yet state anything about the quality of the result.

- *Consistency* helps defining the result quality. A consistency model defines the rules the responses of a system have to obey.

- *Strict consistency* means that every user operation (e.g., insert, update, delete) is reflected in the result in the exact order they were issued. Due to the limits in clock synchronization, this is practically impossible to achieve in a system connected over a communication network.

- *Sequential consistency* is a relaxed form of consistency, so that every result has to reflect the same order of operations, but not necessarily in the order the operations were issued.

- *Eventual consistency* does not guarantee immediate consistency, but that the system will become consistent after a sufficiently long period of time has passed.

Not all design goals can be achieved in the same system. The *CAP theorem* [17, 46] states that no system can provide (sequential) consistency, availability, and partition tolerance at the same time. When a partition (i.e., caused by arbitrary message loss) occurs, the system must either tolerate inconsistencies or stop servicing until the partition is healed. This insight has fundamentally affected the design of large-scale distributed systems. Since transient partitions are commonplace in the Internet and an unavailable system is not very useful, more and more system developers have embraced weaker consistency models such as eventual consistency.

1.1.2 Peer-to-Peer Overlays

Traditionally, the roles of servers (service providers) and clients (service consumers) have been fixed. In *client-server* computing, a small and fixed set of machines assumes the role of servers, and a much larger and dynamic set of machines are the clients operated by the users. This relatively simple model enables scalable and consistent systems with high performance, but the provisioning and maintenance of the servers result in a significant (monetary) cost overhead. If availability and fault tolerance are to be maximized, additional investment is required.

In *peer-to-peer* (P2P) computing, no strict distinction between clients and servers exists. Every host can be client, server, or both, and the roles can change dynamically. In a pure P2P environment, no dedicated server machines operated by system administrators exist, but every participating computer is provided by a user, who typically wants to use the services of the system. This radically reduces the cost of the system and thus makes P2P very attractive for scenarios in which there is no or only a low-profit business model. It is obvious that a system where every consumer not only induces load but also provides additional capacity can, by its nature, be very scalable. Indeed, some P2P systems have successfully scaled up to millions of online users in practice.

In a research area that is still evolving, many different definitions of P2P, which all describe the same phenomenon, can be found. A good characterization was given by Oram et al. [102] and refined by Steinmetz and Wehrle [133]:

> *"A Peer-to-Peer system is a self-organizing system of equal, autonomous entities (peers) which aims for the shared usage of distributed resources in a networked environment avoiding central services. In short, it is a system with completely decentralized self-organization and resource usage."*

Even though the underlying network architecture (the *underlay*) makes it possible for each peer to communicate with any other peer, not every peer knows all other peers in the system (nor is this desirable in most cases). Instead, the peers form an *overlay* network, where each peer is connected to a small set of *neighbors* and maintains a routing table of those neighbors. The *degree* of a peer is defined as its number of neighbors. Messages are routed through the overlay by forwarding them to the most appropriate neighbor (*recursive routing*) or repeatedly querying the current set of neighbors for more appropriate peers (*iterative routing*). Steinmetz and Wehrle provide a formal definition of overlay networks and their properties, which can be found in [133].

1.1.3 Churn and Open-Membership Systems

P2P technology can be deployed in a wide range of scenarios, from rather unstable wireless ad-hoc networks to relatively static data center environments. This work focuses on the traditional usage scenario of public Internet-based systems. Thus, it can be assumed that every peer has stable end-to-end network connectivity (i.e., Internet access). Furthermore, the membership in such a system is not controlled centrally. There might be an authority (or a federation of many authorities) which grants the credentials required to join the system, but this authority cannot force a peer to join. The size of the system in such an environment is beyond any central control and hard to predict. It might change or fluctuate significantly in short and long term. Such a system of autonomous peers will be called an *open-membership* system. The process of peers joining and leaving the system dynamically is called network *churn*.

In an open-membership system the set of available services changes dynamically. The services of peers that leave the system become unavailable. A P2P system needs a mechanism to locate the currently available services. Some applications cannot tolerate services to become unavailable and thus need to migrate their services between peers. Since no single peer can be assumed to be reliable and may fail at any moment without prior notice, failed services must be recovered or replicated proactively. The nomadic nature of such services aggravates the problem of how to locate them.

1.1.4 Self-X

The uncertainties of the open environment, the decentralized structure, and the sheer scale of P2P systems make them a natural application of *self-adaptive software* [123]. Self-adaptive systems use an automated feedback loop to continuously adjust themselves to a changing environment. The self-adaptation can aim at different goals. *Self-configuring* or *self-organizing* systems are able to operate without or with reduced manual configuration. *Self-healing* systems increase fault-tolerance by automatically discovering and counteracting failures or disruptions. *Self-optimizing* systems increase

efficiency through adjustment of system parameters. *Self-protecting* systems are able to detect and recover from security breaches. Optimally, a P2P system should support all kinds of self-adaptation, but at least self-organization is inherent to any P2P system.

1.1.5 Peer-to-Peer Search

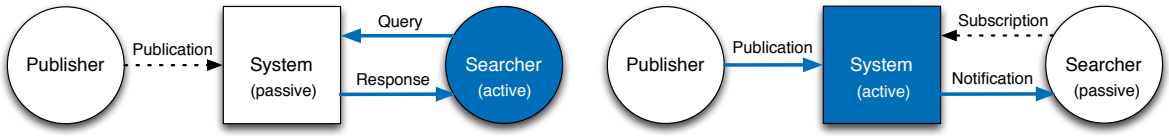
The problem of locating nomadic services is the archetypical example of self-organization in P2P networks. It has made search one of the core services of most P2P systems and *search overlays* a major building block in the area of P2P technology. A search overlay is a decentralized index structure to locate services, or put more generally: to query for data. Many different approaches to organize such an overlay exist [65, 115]. Normally, only the meta-data required to answer the queries, but not the service itself, is stored in the search overlay. This meta-data contains a reference to the location of the service. The service itself might provide data (e.g., a downloadable file in a file-sharing application), but in the following data will be used to refer to the (meta-)data stored in the search overlay. Data stored or provided by services outside of search overlays is beyond the scope of this work.

Search overlays are usually categorized into structured and unstructured overlays. In a *structured overlay*, the routing of inserts and search requests is based on the related data, normally by assigning a key to each item [29]. Such search overlays provide a *get* and *put* operation like a hash table and are therefore called *distributed hash tables* (DHT). Each peer in a DHT is responsible for a more or less randomly assigned range or set of keys and all requests related to a key should be routed to the responsible peer. Since the request popularity distribution in distributed systems is typically Zipf-like [16, 54], the (random) peers responsible for the most popular keys might become bottlenecks to the system. Furthermore, the limitation to key-value lookups makes more complex requests like keyword search [113], range queries [111], or XPath [13] a challenging task for a DHT. *Unstructured search overlays*, however, typically support arbitrary queries naturally, since the content of the requests is opaque to the overlay routing. In unstructured overlays, the requests are distributed by mechanisms like flooding [64] or random walk [20] and are processed by each receiving peer. Unfortunately, this approach has a message count of $O(n)$ for exhaustive searches and is not as scalable as DHTs which typically only require $O(\log n)$. Thus, peer-to-peer application developers have to choose between scalability or sophisticated search.

A new kind of search overlays tries to overcome this limitation. The *rendezvous search systems* put $O(\sqrt{n})$ copies of each data item and each query in the network, which makes them very scalable, especially in the context of complex queries which add additional overhead to DHTs [83, 157]. The placement is done in such a fashion that every query is evaluated against every data item somewhere in the overlay. A more formal definition will be given in Section 2.2. Since a query is executed against a complete copy of each data item, arbitrary queries can be used.

1.1.6 Query/Data vs. Publish/Subscribe

The term *search* normally describes a situation where data is stored in the system and a user issues a query to retrieve matching items. In contrast to this classic *query/data* approach, the concept of *publish/subscribe* [35] consists of a subscriber that stores subscriptions in the system, and a publisher that pushes data into the system. When a publication matches a subscription, the subscriber is notified by the system and receives the data. In the context of this work, both are classified as methods of search. Query/data is *query-triggered search* with an active searcher and a passive system that only reacts to search requests (see Figure 1.2a). Publish/subscribe is *publication-triggered search* where the system actively pushes notifications to passive subscribers (see Figure 1.2b).



(a) Query-triggered search **(b)** Publication-triggered search

Figure 1.2.: Query/data and publish/subscribe compared

A search overlay that is able to perform query-triggered searches can often be turned into a publish/subscribe system that performs publication-triggered searches [119, 142]. In the context of this work, search overlay denotes systems that can be used for query-triggered and publication-triggered search. Future applications might even combine both approaches and continuously update the initial result set of a query with a follow-up subscription.

1.1.7 Peer-to-Peer Replication

Replication plays a major role in the design of search overlays. Replication is the placement of copies (*replicas*) of a data item at different locations in a distributed system. Replication can be used to improve availability [150] and performance [151], both important aspects for large-scale, open-membership P2P networks. Since a peer may go offline immediately without prior notice, the replicas it stores can become unavailable to the system. Only by maintaining a sufficient number of replicas for each data item, it can be assured that neither churn nor large-scale failures can remove data permanently from the system. The placement of replicas can also help to improve search performance. A shorter path to the data can both reduce message cost and response time [26].

In a dynamic environment like P2P systems where the composition and organization of the overlays is constantly changing, the placement of replicas needs to be adjusted to ensure proper replication. The *replica maintenance* keeps track of the replicas and takes action if required. Depending on the algorithm and the situation, replicas need to be added, removed or moved from one peer to another. Replication is closely intertwined with updates on the replicated data. The replication algorithm is in control of the

replica placement. If updates are in-place, the update mechanism needs the placement information to modify the replicas. If a copy-on-write approach is taken, the update mechanism needs to issue new replicas through the replication mechanism.

The combination of mutable data and replication brings up consistency issues. As long as there is only one copy of each data item in the system, every request will return this version (if it is found). With multiple replicas and mutable data, individual copies might get out of sync if an update does not reach all replicas. A data item with differing replicas will be called incoherent. A data item with only identical replicas is *coherent*. An incoherency may lead to inconsistent results, because two different requests might access different replicas, but coherence is not necessary to ensure consistent responses. If it is guaranteed that all requests return the same version of the data item (e.g., by always accessing the same replica), incoherent data can be used for consistent responses.

Different approaches to replication in P2P overlays exist. The algorithms considered in this work put a pre-defined number of replicas on nodes selected by the algorithm. This ensures that every data item can be successfully retrieved. Other approaches [20, 26, 91], which shall be called *caching*, replicate data based on their request popularity. Since the replication is done by the requester and does not follow a pattern globally agreed upon, replicas in such systems can normally not be updated consistently since their location is unknown. Caching can be useful to improve load balance for skewed popularity distributions in potentially unbalanced systems like DHTs.

1.1.8 ACID vs. BASE

The insight of the CAP theorem sparked new research approaches in the area of client/server computing. The previously omnipresent concept of *ACID* transactions (atomicity, consistency, isolation, durability) [55] is now understood as a tradeoff that favors consistency over availability in the face of partial system failures. Its new counterpart *BASE* (basically available, soft state, eventual consistency) [105] tries to maximize availability and sacrifices strong consistency in failure situations. The need to scale systems up without compromising availability has made BASE popular among web application developers [105].

In an open-membership P2P environment, that is typically both of much larger scale and much more failure-prone than server clusters, this tradeoff is even more important. Since in a large P2P system failures are not the exception but the common case, any system with a useful availability will have to trade some consistency for availability. That said, giving up on strong consistency cannot be an excuse to ignore consistency issues completely. A proper search overlay and replication algorithm should try to maximize consistency under the availability constraint. When working with eventual consistency, the amount and duration of inconsistencies should be minimized.

1.2 Problem Statement

If P2P applications are meant to successfully compete with modern client/server applications, they need proper support for complex search and mutable data stored reliably in the system. The goal of this work is to describe such a search overlay for unstable open-membership environments. By building on joint work with Wesley Terpstra in the area of search [140, 143], this thesis focuses on the replication and update mechanisms required.

A system providing these services should be highly available, very scalable, and provide useful consistency guarantees. In an open-membership environment this can only be achieved by providing a high level of fault tolerance. Typical challenges are arbitrary communication failures and node crashes, a high level of node churn, abruptly changing network sizes, and dynamic workloads with heavily skewed popularity distributions.

In order to support a maximum of different P2P applications, the diverse use cases for replicated data and their requirements need to be identified, and corresponding replication modes have to be defined. Those use-cases vary widely in their requirements of persistence, lifetime, and mutability of the data to be distributed. For example, some information, like search requests, might be non-persistent, because it is consumed instantly, while other data, like documents, may require long-term persistence. Some persistent data, e.g., a wiki article, must stay available even though none of its authors is online anymore. On the other hand, the lifetime of data may be bound to the existence of a certain node, e.g., presence information in a chat system. Being able to modify distributed information is often required, but typically introduces a communication overhead, which is wasteful for immutable data. Currently, no taxonomy for P2P replication modes exist, and therefore the existing replication mechanisms have unclear design goals. A versatile search overlay should be able to cope with a wide range of application scenarios and thus needs a set of replication algorithms that support the requirements of the different replication modes.

To enable their widespread use, the proposed mechanisms must be understandable and easy to use for application developers, which are not always experts in P2P networking. This requires a clear separation of concerns between network communication, data management, and application code. The system must provide a comprehensible, yet flexible interface to the application developer.

Any scientifically valid proposal should be analyzed and evaluated thoroughly. The available methods include mathematical analysis, large-scale simulation, and prototype experiments. As none of the methods can provide exhaustive insight by itself, a combination of evaluation methods is required.

1.3 Contributions

The main contribution of this work is the description and evaluation of a search overlay that supports complex search and a versatile set of replication modes for the searchable data. This overlay is the rendezvous search system *BubbleStorm*, which is the result of a larger research project. Its search capabilities and basic organization have already been described in [140, 143].

This work completes the system by contributing replica maintenance and update algorithms. These capabilities allow the persistent storage and consistent modification of distributed data in an extremely dynamic environment. As this enables a much wider range of applications far beyond mere search, the system is better described as a *rendezvous information system*. To the best of the author's knowledge, no rendezvous information system for P2P environments has been presented before.

In detail, the contributions of this work include:

- **The first survey of rendezvous search systems in the area of P2P search overlays (Chapters 2 and 3).** Even though there are quite a few systems using rendezvous approaches, most of them seem to have been invented independently and even more recent publications about these systems seem to be unaware of most related work in the area. In this thesis, a comprehensive overview of the state of research is given, and the existing systems are classified into solutions for data centers, structured overlays, unstructured overlays, and semi-structured overlays.

- **A generic classification of replication modes for P2P search overlays (Chapter 4).** Different types of data and applications require different forms of replication. This work identifies four fundamental use cases for replication and proposes a taxonomy for replication mechanisms. The current state of the art in P2P replication is reviewed based on the classification. The replication modes are instant, fading, managed, and durable replication. Because the replication modes have conflicting requirements, no single replication algorithm can support all modes completely.

- **Data description primitives for BubbleStorm (Chapter 5).** The identification of replication modes makes it possible to define a schema model for rendezvous search systems, using BubbleStorm as a concrete example. Such a schema allows application developers to define their data model without in-depth knowledge about the underlying search overlay and enables portability of applications between different rendezvous search systems, similar to the SQL data definition language for relational databases. The compact scheme consists of four primitives to cover the four replication modes (instant, fading, managed, and durable) and a fifth primitive to define rendezvous matching constraints between data types. A complete application data model including distributed data, queries, subscriptions, and publications can be defined using only those five primitives.

- **A maintainer-based replication algorithm for managed data (Chapter 6).** Unlike instant and fading data, managed data needs a replica maintenance algorithm

to sustain availability. The algorithm presented here builds upon a dedicated maintainer for each managed data item and can be used with BubbleStorm and similar search overlays. It provides eventual consistency guarantees and avoids conflicting updates through serialization by the maintainer. It preserves the replica distribution in the overlay required by the rendezvous search system to meet its search success guarantees and is extremely robust against disruptive changes in the network.

- **A collective replication algorithm for durable data (Chapter 7).** This algorithm not only adds support for the last remaining replication mode to BubbleStorm and similar systems, but also provides efficient key-value lookups for unstructured rendezvous search systems. It provides eventual consistency and a fully automated resolution of conflicting updates. Like the maintainer-based replication, it preserves the replica placement invariants and matches the resilience of the other BubbleStorm components.

- **A state-of-the-art simulation and prototyping environment for the evaluation of P2P systems (Chapter 9).** Evaluation of large-scale distributed systems in highly dynamic environments is a challenging task. Even minor inaccuracies in the evaluation setup can lead to highly misleading results. The evaluation environment developed for this work has been carefully designed to avoid typical mistakes. By combining overlay simulation with automated prototype experiments, both synthetic large-scale simulations and very realistic real-world experiments can be conducted.

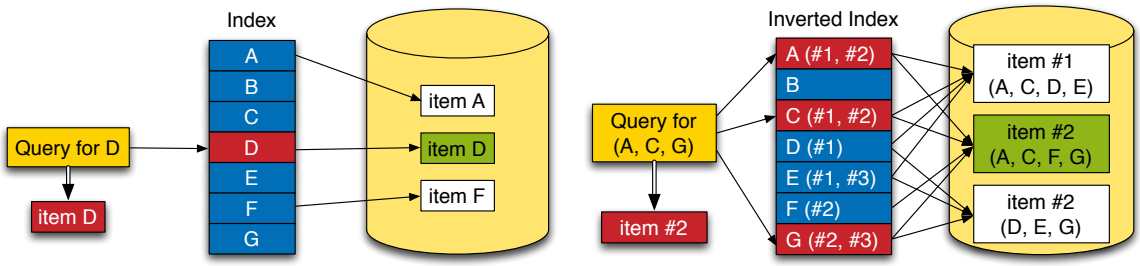
2 Rendezvous Search

This chapter gives an introduction to the concept of rendezvous search and an overview of the existing solutions for distributed rendezvous search. A brief version of this survey has been presented at a workshop [77], and a journal paper based on this chapter has been submitted for publication [78].

2.1 Index-Based Search

If each data item has a unique identifier, and all searches in a system only select items based on this primary key (a key-value lookup), it is obvious to organize items in the (distributed) data store by their identifier (*partition by key*) and thereby build an *index* as shown in Figure 2.1a. In a distributed environment each peer takes responsibility for a partition of the index, and the desired key can be found in logarithmic time through a DHT. Unfortunately, searches in reality are rarely that simple.

Often, data items consist of a number of attributes, and each attribute can have multiple values. There exists a wide range of tools for mapping such problems to indexes, like *inverted indexes* for keyword search [70]. This involves putting a reference to the item for each value (keyword) it contains. A query for multiple keywords would query each keyword separately and then intersect the result sets (see Figure 2.1b).



(a) Key-value lookup in an index **(b)** Content-based query in an inverted index

Figure 2.1.: Search with inverted indexes

While inverted indexes are a popular and efficient technique in standalone systems, the copying of references and intersecting of result sets becomes very inefficient when being mapped to a DHT in a P2P environment and can barely compete with the notoriously unscalable Gnutella [83].

Even worse is that partition-by-key requires a custom index and thus a specific P2P communication model for each query language. Inverted indexes are specific to keyword search and similar attribute-based queries. Queries on hierarchical data like XPath require a completely different approach [13]. Message routing and query evaluation are so tightly interwoven that existing libraries for standalone systems cannot be reused but

have to be reimplemented specifically for P2P search. Aside from the additional effort, this requires an engineer that is both an expert in P2P networking and the processing of the respective query language.

For these reasons, sophisticated search on DHTs has not been very successful. A different, more flexible approach is needed,

- that is easier to use with a wide range of query languages,
- that does not require the application developers to have expert-level knowledge of networking and query processing,
- and that is at least as scalable as DHTs.

Rendezvous search might be the perfect solution for this problem and thus has gained a growing popularity in the P2P community lately.

2.2 Concept

In rendezvous search the message routing is independent of the data and query language in use. The system ensures that a query meets every data item somewhere in the network, but otherwise serves as a *black box*. This is achieved by creating a set of replicas for each query and data item and distributing them in the overlay. Since rendezvous search systems treat query and data equally, in the following, *item* is used to refer to instances of query and data. This slight generalization can provide additional freedom in application design, as will be shown later.

Similar to the definition in [140], the correctness in rendezvous systems is defined as follows:

Definition 1 (The Rendezvous Problem)**.** *Let the function $R(x) \subseteq N$ denote the subset of nodes (out of all nodes N) who receive a replica of item x. For two given sets of items A and B find a replica dissemination function R which guarantees the existence of a* rendezvous peer *for each pair (a, b). $\forall (a, b) \in A \times B : R(a) \cap R(b) \neq \emptyset$*

There are numerous different ways to solve the rendezvous problem, each with its own advantages and disadvantages. All solutions share a number of properties which stem directly from the definition of the problem.

First of all, the bandwidth cost of a rendezvous search is *independent of the complexity* of the query evaluation, which makes it especially competitive, as the bandwidth costs of other solutions grow due to the complexity. Therefore, they are an excellent choice for complex queries. The rendezvous problem can be solved with a complexity $O(\sqrt{n})$.

Secondly, rendezvous search systems can answer *arbitrary queries*. Let the function $M(a, B)$ denote the items matching the query a out of the set B of items stored at a given node. This *match function* can easily be implemented for a vast range of query languages. The advantage of rendezvous search systems is to always keep complete data items instead of breaking them up into distributed data snippets as, e.g., in inverted indexes. Rendezvous search systems *partition by document* instead of partitioning by key.

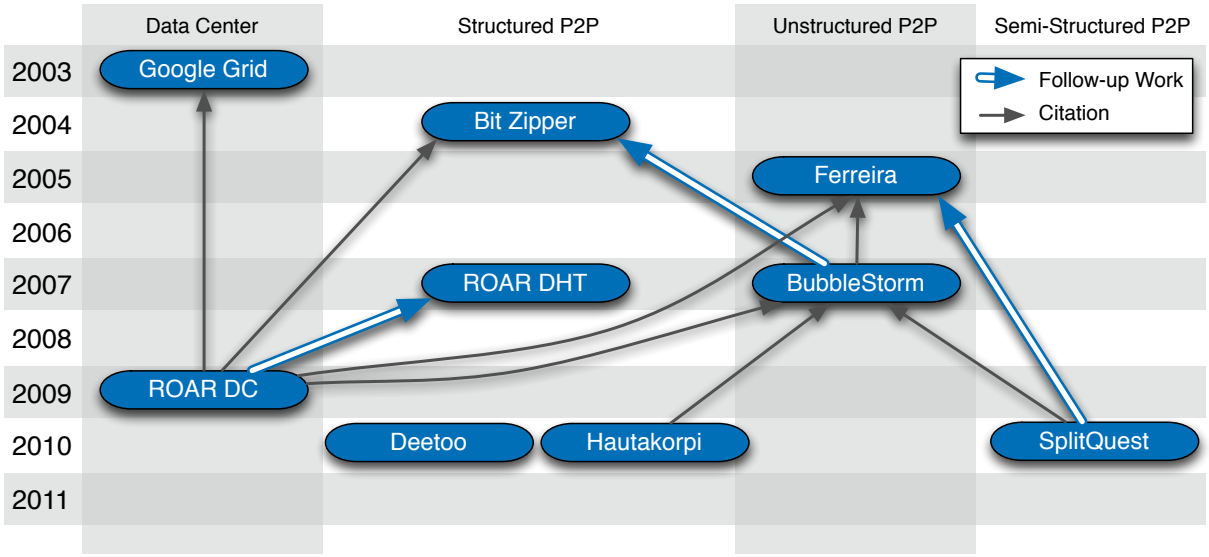


Figure 2.2.: The genealogy of rendezvous search systems: Google Grid [6], Bit Zipper [141], Ferreira [36], ROAR for DHTs [110], BubbleStorm [143, 145], ROAR for data centers [108, 109], Deetoo [22], Hautakorpi [56], SplitQuest [84]

The third important property shared by rendezvous search systems is the *separation of concerns*. The match function is completely independent of the replica dissemination mechanism, which is inside the black box and of no concern for the application developer. This is a huge improvement for application developers. Apart from completely hiding the highly complex network interactions, the separation of concerns allows the application developer to re-use off-the-shelf database systems or information retrieval libraries to implement the match function. By separating network communication from query matching, the application developer does not need in-depth expertise of either problem.

2.3 History and Classification

Concepts similar to rendezvous search systems like *sharding* for horizontal database partitioning [19] or *quorum* systems for consensus and locking [59] have been popular in the database community for a long time.

The most intuitive solution to the rendezvous problem is the grid approach as used by Google for their web search engine [6], which originally was published in the context of quorum systems [21]. The nodes are arranged in a grid, and each data item is copied to every node in a random row, while each query is copied to every node in a random column. The need for better search capabilities in P2P overlays led to a number of independent attempts to solve the rendezvous problem for highly dynamic, failure-prone, and self-organizing networks. In Figure 2.2 the genealogy of rendezvous search systems according to citations and follow-up projects is depicted. Surprisingly, even some of the newer systems seem to be completely unaware of the related work in the area. The wide range of solutions has created a quite diverse design space. In the

following, a survey of the existing rendezvous search overlays will be given and they are classified according to the following criteria.

Most fundamental to the system design is the *consistency model* for the rendezvous algorithm. Some systems are *deterministic*, i.e., they promise to produce all rendezvous pairs as long as the preconditions are fulfilled. *Probabilistic* systems do not promise all pairs, but guarantee minimum percentage of pairs produced. At first glance, deterministic approaches seem clearly superior to probabilistic ones, but this heavily depends on the environmental conditions. When link and node failures are common, the preconditions of a deterministic system are easily violated. In this case, they might be worse than probabilistic systems. The CAP theorem [17, 46] shows that there cannot be a system that provides consistency, availability, and partition tolerance at the same time. In other terms, no system can operate with deterministic success in the presence of failures. Therefore, deterministic approaches seem to be more suited to a relatively stable environment.

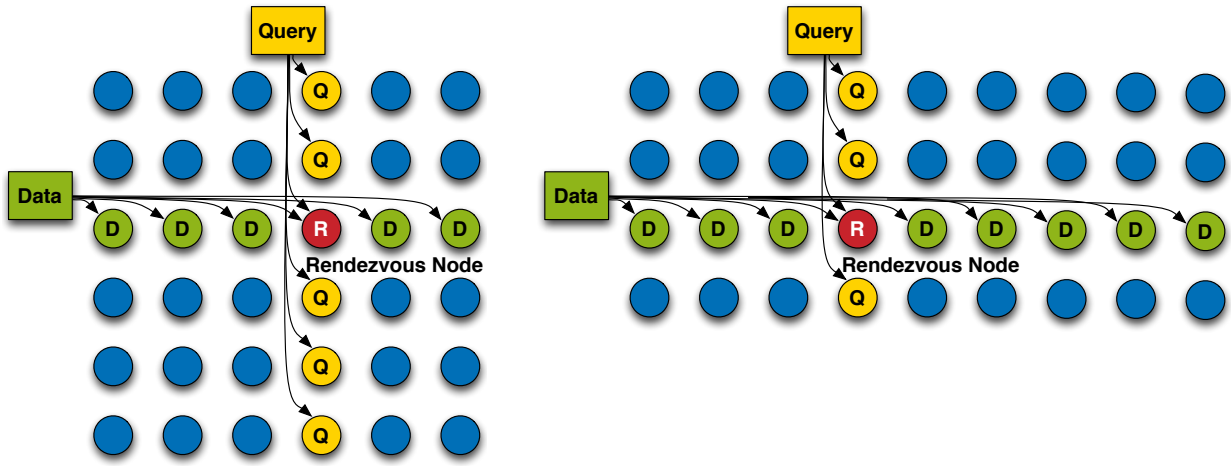
Most probabilistic rendezvous search systems build upon the theory of the generalized birthday problem with two mutually exclusive groups [156]. The collision probability of those two groups is $1-e^{-\lambda}$ for groups of size $O(\sqrt{\lambda n})$, which is surprisingly high and makes probabilistic systems competitive.

The second fundamental distinction is the *type of overlay* used. There are systems using *unstructured* and systems using *structured* overlays. While the unstructured approaches typically have to define their own topology, the structured systems simply build upon an existing DHT. Some approaches even build a structured routing on top of an unstructured topology. This concept will be called *semi-structured*. Finally, there are systems which do not use an overlay at all, but use global knowledge for direct connections. Such approaches are useful for *data center* environments. Even though they lack the self-organization of real P2P systems, they are very relevant to the evolution of their P2P relatives.

Another interesting characteristic is the replica *dissemination* algorithm for items. Data center solutions normally assume global topology knowledge and high bandwidth. Therefore, they can use *direct connections* to all receivers in parallel. In P2P topologies replicas can be placed one after another in a chain. This *walk* through the topology can be deterministic or random. Long walks do not only suffer from high latency, but are also fragile, because any dropped message loses the rest of the walk. Some systems therefore use multiple short walks in parallel for a constant improvement. Even better are *trees*, since they have logarithmic depth and a dropped message only causes loss of a small subtree.

Replicas lost to churn need to be replaced. Some systems already provide an algorithm for this *replica maintenance*. If replicas are not placed randomly, an *update* mechanism could change their content. In a large scale distributed system this should be done *consistently* even in the face of concurrent updates from multiple requesters.

The final attribute to be considered is the amount of *adaptivity* the rendezvous search system supports. The minimum adaptivity for a P2P system is a self-organizing *topology*. Not all systems specify their own topology mechanism but re-use existing overlays. Since the number of replicas in a rendezvous search system typically depends on the *network size* (n) or a similar topology attribute (e.g., sum of degrees), changes of such



(a) A square grid **(b)** A rectangular grid (query frequency > data frequency)

Figure 2.3.: Grid-based rendezvous search

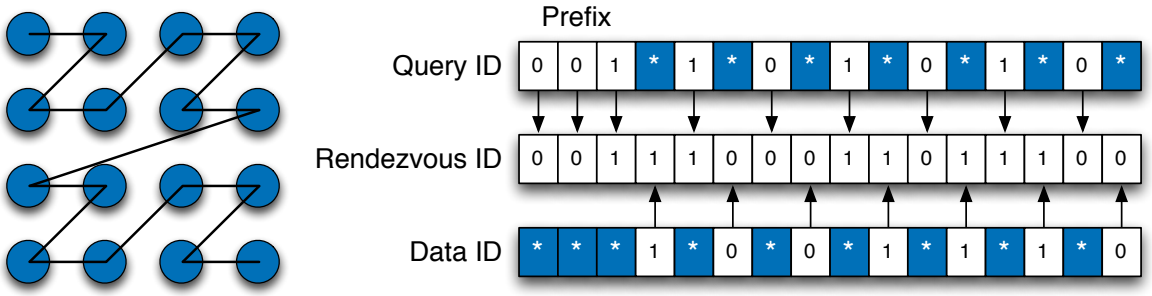
properties need to be detected and replica counts need to be adapted. In addition to network size adaptation, the replica counts can be adjusted to minimize network traffic or response times. A couple of rendezvous search systems support such online *traffic balance* adaptation. Some of the systems, especially those with a data center background, do not have complete self-organizing capabilities. Often at least some of those capabilities could be borrowed from other rendezvous search systems.

2.4 Grid

The most intuitive approach to solve the rendezvous problem is to arrange the nodes in a matrix- or grid-like fashion. Each item of one type (e.g., data) is replicated on every node in a randomly chosen row and each item of the corresponding type (e.g., query) is replicated on every node in a randomly chosen column (see Figure 2.3a). A node receiving a query matches it against all locally stored data and returns the result to the client that issued the query. Since every column intersects every row exactly once, the grid is a solution for the rendezvous problem.

This approach is used in the Google Web Search Engine [6]. Each row in the system is a shard of the total set of data, and each column is a complete copy. Each of the globally distributed Google data centers houses at least one complete column and thus can independently process any incoming query. Therefore, queries can simply be routed to the closest data center to reduce response times.

When choosing the same size for columns and rows (a square), the number of messages for queries and data is $O(\sqrt{n})$. This is optimal in terms of total message count, if both operations are equally common. If their frequencies differ, a rectangle shape can improve the system's total message count (see Figure 2.3b). More useful than message count is the notion of bandwidth cost. The ratio between rows and columns can be based on the bandwidth usage of queries and data to minimize the bottleneck bandwidth usage [139, 140, 108]. The grid approach is able to attain a lower bound on



(a) Z-order curve **(b)** Computing the rendezvous node

Figure 2.4.: Bit Zipper

bottleneck bandwidth usage and thus is—in terms of this metric—an optimal solution for the rendezvous problem.

Unfortunately, the grid is a very static design and thus not suited for P2P environments. The system does not take network size or traffic balance changes into account, and it is unclear how an online adaptation of size or traffic balance could work. Furthermore, failure tolerance is not an inherent part of the system design. Google solves the problem by keeping multiple replicas of each partition per data center, but this is only a feasible solution if node failures are rare rather than common. On the other hand, the grid is a very efficient and deterministic solution to the rendezvous problem. In terms of the CAP theorem the approach chooses consistency over availability and therefore is useful for data centers, but not for P2P environments. A P2P solution needs to be much more flexible and self-adaptive in respect to environment changes. Those shortcomings cannot be solved by adding internal redundancy (e.g., by having multiple nodes for each position in the grid), as they are fundamental to the design of the system.

2.5 Bit Zipper

Bit Zipper [141] was the first attempt to solve the rendezvous problem with a P2P system. It builds upon a DHT that supports the key-based-routing (KBR) interface [29]. The basic idea is to map the two-dimensional grid to the one-dimensional key space of the DHT. The solution chosen by Bit Zipper, even though not explicitly stated by its authors, is to use a space filling curve for this mapping (see Figure 2.4a). More precisely, it is a z-order curve [94]. The mapping is achieved by taking the row and column numbers of a rendezvous peer and interleaving their bits.

The dissemination algorithm of Bit Zipper works as follows. Each data item and query are assigned a (random) seed key. Then every even bit of the seed key for a data item and every odd bit of the seed key for a query are declared wildcards. An item is then sent to every possible key in the DHT that can be generated by assigning concrete values to the wildcards. This can be done in practice by splitting the message in both directions whenever a wildcard bit needs to be resolved during the key-based routing. The dissemination algorithm forms a tree in the DHT. The rendezvous peer of a given

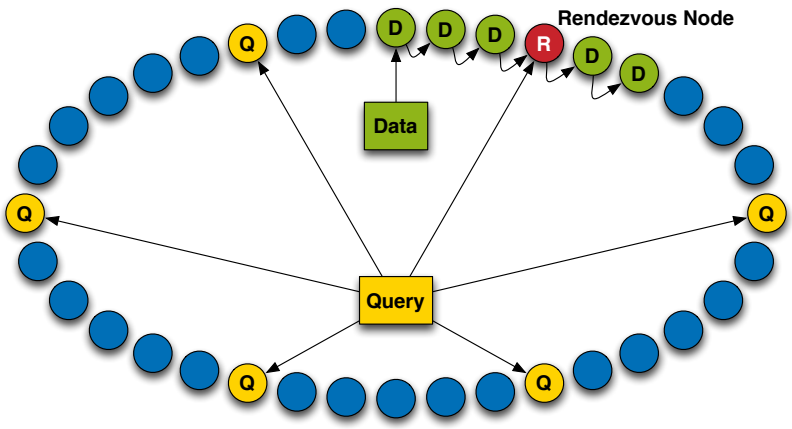


Figure 2.5.: ROAR (rendezvous on a ring)

query-data pair can be determined by "zipping" the bits of the query key together with the bits of the data key (see Figure 2.4b).

Bit Zipper is a deterministic approach that relies on the consistency and routing capabilities of the underlying DHT. Experiments on PlanetLab have shown that a DHT like Chord [136] can have more than 2% of failed lookups due to the usual network anomalies [47]. If a typical fraction of nodes in the overlay is behind a NAT, the amount of failed lookups rises beyond 5%. Such a high failure rate disqualifies DHT-based systems from most open-membership application scenarios for Internet-based P2P rendezvous search systems. In a more controlled environment, DHT-based systems can be useful nonetheless.

Bit Zipper uses $6\sqrt{n}$ messages to place $2\sqrt{n}$ replicas. That makes it a constant factor less efficient than the static grid solution. Unlike the grid, the system automatically scales with the number of nodes, due to the fractal nature of its dissemination mechanism. The authors do not specify any maintenance mechanism for the replicas. The normal mechanisms provided by DHTs do not seem helpful here, because they are not designed to deal with such a large number of replicas of the same item stored at different DHT key locations. Therefore, it is unclear how Bit Zipper would perform under churn.

Bit Zipper does support traffic balancing with a balance factor b. The first b bits are not used for wildcards in the item type with the higher load, and are all wildcards for the other type. This traffic balancing is unfortunately not dynamic but pre-defined during application design, which limits its usefulness.

The Bit Zipper paper is based on theoretical analysis only and lacks any type of experimental evaluation.

2.6 ROAR

ROAR (rendezvous on a ring) [110, 109, 108] takes an approach similar to Bit Zipper by mapping the grid to a circular key space. In contrast to Bit Zipper, ROAR maps rows to intervals in the key space. A data item is replicated to each node in a randomly

chosen interval. A query is sent to a randomly chosen node within each interval (see Figure 2.5).

The initial design of ROAR [110] defined that the system runs on top of a DHT like Chord [136]. The later publications [109, 108] assumed a data center scenario with global knowledge and one hop routing instead. ROAR is a deterministic system in both cases. Although the data center solution, of course, is more efficient, ROAR can also be compared with the other decentralized and self-organizing rendezvous search systems and thus both scenarios are considered here.

The data center solution requires $O(\sqrt{n})$ direct messages for the dissemination of an item. In the DHT scenario the data dissemination would require an $O(\log n)$ lookup for the first node in the interval followed by an $O(\sqrt{n})$ chain of replication messages within the interval, which is very competitive. Unfortunately, random walks of length $O(\sqrt{n})$ are very failure-prone in a dynamic P2P environment [143]. The query dissemination would require an $O(\log n)$ lookup for each of the $O(\sqrt{n})$ replicas, raising the total replication cost to a not very compelling $O(\sqrt{n}\log n)$. Fortunately, the lookup results can be cached. Thus, subsequent searches need only $O(\sqrt{n})$ messages as long as the target nodes stay online.

Unlike Bit Zipper, ROAR comes with a complete replica maintenance mechanism to mitigate the effects of churn. ROAR puts great emphasis on dynamic traffic balancing to optimize response times. It can change the replication levels of queries and data to adapt to changing workloads. It even supports heterogeneous node capacities.

ROAR has been evaluated mainly using numerical simulation and prototyping, but also includes some theoretical analysis.

2.7 Ferreira

Ferreira et al. [36] were the first to solve the rendezvous problem for unstructured P2P overlays. Due to the lack of global routing information, they chose a probabilistic approach that is based on the random placement of replicas. To disseminate an item, they first use an $O(\log n)$ random walk to pick a random node in the overlay [48]. They put the replica on this node and continue with an $\sqrt{\lambda n}$ random walk. In each of the following random walk steps they put a replica on the receiving node (see Figure 2.6). λ is a user-defined parameter that can be used to increase the success probability by putting more replicas into the system. The success probability is $1 - e^{-\lambda}$. Compared to Bit Zipper, this is quite efficient. With a total message cost equal to Bit Zipper's $6\sqrt{n}$, λ would be 36, yielding a success guarantee of 15 nines.

The symmetric nature of query and data items in this system opens up new opportunities. For example, application designs, where three item types rendezvous with each other in a triangular fashion, become possible. A type could even match with itself, so that an item of this type encounters every previously published item of its type. This could be used, e.g., to match buddy lists in an instant messenger system against each other to find buddies already online and publish the user's own presence information with a single operation. Another use-case is the implementation of self-joins for SQL-like query languages.

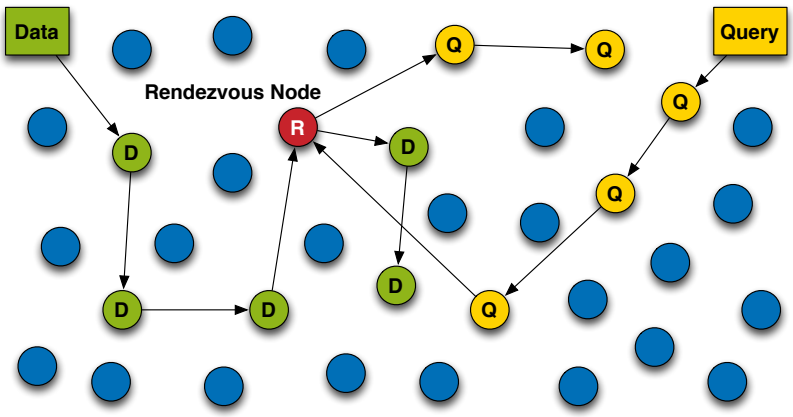


Figure 2.6.: Probabilistic rendezvous search with random walks by Ferreira et al.

Although the authors provide a brief theoretical analysis to prove their algorithm, they seem to misuse random walks. While it is correct that the short random walk selects a random starting node, the nodes chosen in the long random walk are depending on their predecessors. This is especially harmful since a random walk is likely to go back to the predecessor, thus putting multiple replicas on the same nodes, effectively reducing the replica count. Furthermore, long random walks are likely to get stopped prematurely by failing nodes. These two effects reduce the desired replication degree considerably, as shown in the experiments in [143].

The authors do not specify which kind of graph is suitable as a topology and how it can be constructed. It seems that any expander graph would be sufficient. In order to pick the correct length for the random walks, the system needs an estimate of the network size n. It is not clear what the authors would suggest as a solution, but there is a large body of work on how to solve this problem [144, 88, 96]. The system does not have a replica maintenance algorithm either. The random placement of replicas makes it hard to update already published items.

The system does take heterogeneous node degrees into account, but instead of using them to improve the efficiency like [143, 108], the authors use a Metropolis-Hastings approach to ensure uniform sampling independent of node degrees.

Unfortunately, Ferreira et al. only provide an incomplete analysis of their approach and then build their numerical simulation on the assumption that their analysis is correct.

2.8 BubbleStorm

Even though actually being the successor to Bit Zipper, the BubbleStorm system [143, 145] is somewhat similar to the Ferreira system. It places $O(\sqrt{\lambda n})$ replicas randomly in an unstructured P2P overlay and provides a probabilistic success guarantee of $1 - e^{-\lambda}$. Since BubbleStorm is symmetric as well, it also supports self-matching types.

In contrast to the Ferreira system, it comes with a random multigraph topology and the necessary join and leave algorithms to maintain the topology. The random topology makes the short random walk used by Ferreira et al. unnecessary. Any topology

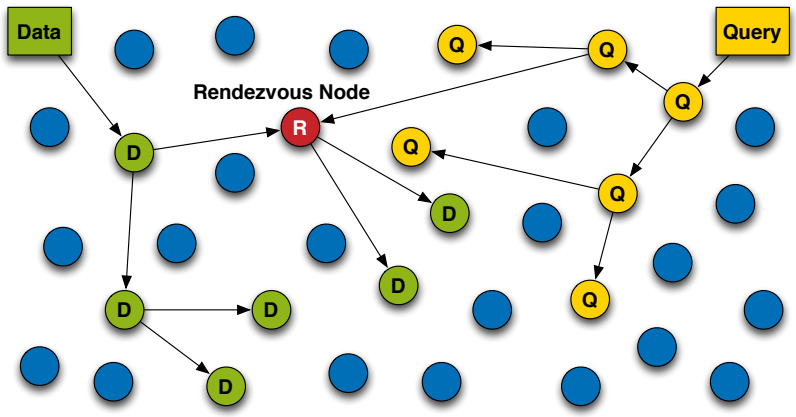


Figure 2.7.: Tree-based probabilistic rendezvous search with BubbleStorm

neighbor is already a random sample of all available nodes. Instead of using a fragile random walk of length $O(\sqrt{\lambda n})$, BubbleStorm uses a random walk that splits in every step, which makes it much more robust to message loss (see Figure 2.7).

BubbleStorm also provides a gossip algorithm [144] that can be used to periodically determine an accurate estimate of the network size n. The same algorithm can be used to estimate the traffic injected by queries and data respectively to dynamically adjust the replica counts and thus minimize the system's total bandwidth usage. The system can make use of heterogeneous node capacities to further reduce bandwidth requirements.

In its original form, BubbleStorm lacks a replica maintenance algorithm. It has the same problem of updating the randomly placed replicas as the Ferreira system.

BubbleStorm has been thoroughly evaluated using simulation [143] and analysis [145]. The simulation results assert BubbleStorm a high resilience against churn and catastrophic node failures (up to 50% simultaneous crashes). There also exists a prototype implementation [79].

2.9 SplitQuest

SplitQuest [84] is a follow-up work to the random walk system by Ferreira et al. The system uses a semi-structured approach. It builds upon any unstructured expander topology, but assigns key space responsibilities to nodes based on their node identifier as in a DHT. SplitQuest then uses an approach similar to ROAR to map data items to intervals in the key space, which are called *replication groups*. The routing to those groups is based on the unstructured topology and therefore probabilistic (see Figure 2.8).

To publish to a replication group, a node forwards the item to all neighbors with an identifier within the group. The receivers store the item and in turn forward it in a similar fashion, if they have not seen the item yet. A query carries a to-visit range, which is used to find useful receivers. Initially, the to-visit range covers all replication groups. After evaluating the query locally, a receiver removes its own group and selects all neighbors within the remaining range, but at most one node from every group. It splits the range between all receiving nodes. The search terminates when the to-visit range only covers the group of the receiver.

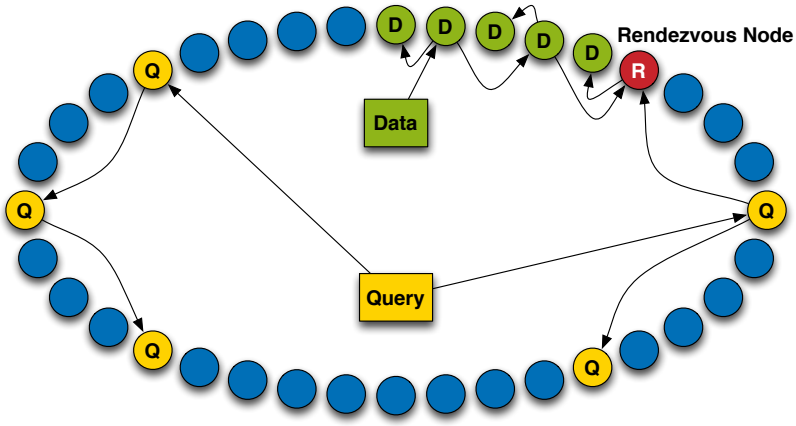


Figure 2.8.: Semi-structured probabilistic rendezvous search with SplitQuest

Unfortunately, the dissemination mechanisms for query and data are not symmetric, and thus SplitQuest does not offer the self-matching capabilities of its predecessor or BubbleStorm.

Lopes and Ferreira provide promising numerical simulation results, but the paper lacks theoretical analysis and real-world experimentation. The authors emphasize the system's resilience against churn, but do not provide measurements of catastrophic failures. Replica maintenance and updates are not covered in the publication.

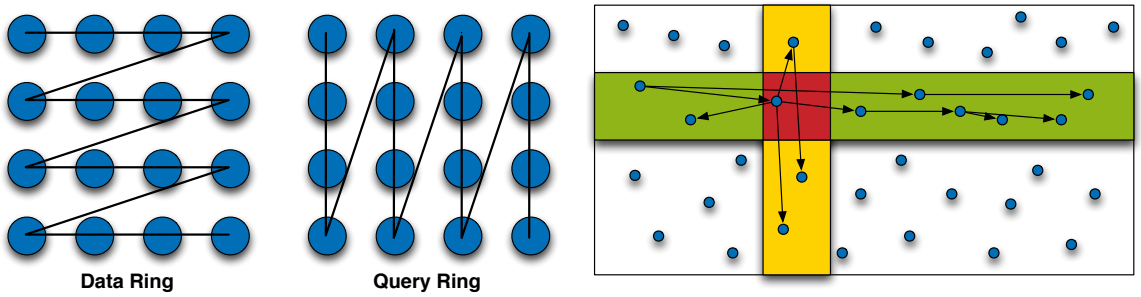
2.10 Deetoo

Deetoo [22] is a rather recent publication on the rendezvous problem in P2P overlays, but surprisingly, it completely lacks references to related work in the area. It is yet another attempt to map the grid to a DHT structure. Instead of using the same topology for queries and data like Bit Zipper and ROAR, Deetoo uses two separate Chord rings for queries and data. The address of a node in the data ring is the transposition of its address in the query ring. That means a node at position (x, y) in the grid gets the address $a = x + wy$ on the data ring and $b = wx + y$ on the query ring (w is the number of addresses per row or column). With this approach, nodes on the same column have adjacent addresses on the query ring and nodes on the same row have adjacent addresses on the data ring (see Figure 2.9a).

Because only a very small fraction of all available addresses are actually assigned to online nodes, Deetoo puts queries and data on a consecutive range of columns and rows to ensure a rendezvous point for each pair with high probability (see Figure 2.9b). The system uses a λ factor much like Ferreira and BubbleStorm to place $O(\sqrt{\lambda n})$ replicas for a success probability of $1 - e^{-\lambda}$.

To disseminate a query or a data item, Deetoo selects the respective ring and contacts the first node in the interval. This node stores the item and splits the remaining interval between those neighbors that are inside of the interval. They recursively proceed with this interval broadcast until all nodes in the interval have been reached.

To maintain replicas, joining nodes retrieve all items they are responsible for from their neighbors. Updates and deletions are possible through deterministic routing, but



(a) Topology mapping in Deetoo

(b) Selecting ranges of rows and columns in Deetoo

Figure 2.9.: Deetoo

concurrent operations from multiple nodes are not considered. The separation of items into query and data types makes self-matching types impossible. It is not explained how the system gets an estimate of the network size n, which is required to pick the correct replication degree.

Deetoo combines deterministic routing with probabilistic success guarantees, which seems to be the worst of both worlds. The deterministic routing relies on the correctness of the DHT's routing tables and can only keep its success guarantees as long as this assumption holds. But even then no deterministic success is guaranteed, since Deetoo ignores the traditional concept of responsibility ranges in DHTs and instead assigns the item only to nodes with an ID inside of the item's replication interval. Since the interval could contain zero nodes, dissemination could fail even with correct DHT routing.

Deetoo's response times, success probability, and replication cost have been analyzed mathematically and simulated numerically. A prototype evaluation is mentioned as future work.

2.11 Hautakorpi

Hautakorpi and Schultz [56] have presented the most recent rendezvous search system based on structured P2P overlays, which they falsely assume to be the first solution to this particular problem. Their system runs on top of the Chord [136] or Bamboo [114] DHTs. It is a probabilistic approach that places replicas randomly in the overlay and uses random walks as the dissemination strategy. Therefore, it resembles a structured topology version of the Ferreira system. The costs and guarantees are obviously the same.

To improve the response time of queries, the authors use multiple parallel and therefore shorter random walks instead of a single $O(\sqrt{\lambda n})$ long random walk. This constant improvement reduces the response latency, but is inferior to the tree-like strategies of Bit Zipper, Deetoo, and BubbleStorm. To avoid collisions between those random walks, the nodes in the overlay are partitioned by their ID and each random walk is only forwarded to nodes in a certain partition. This asymmetry makes self-matching types impossible.

The authors put much emphasis on the incremental deployment of their algorithm, i.e., how to use it on an existing DHT where not all nodes are aware of the algorithm

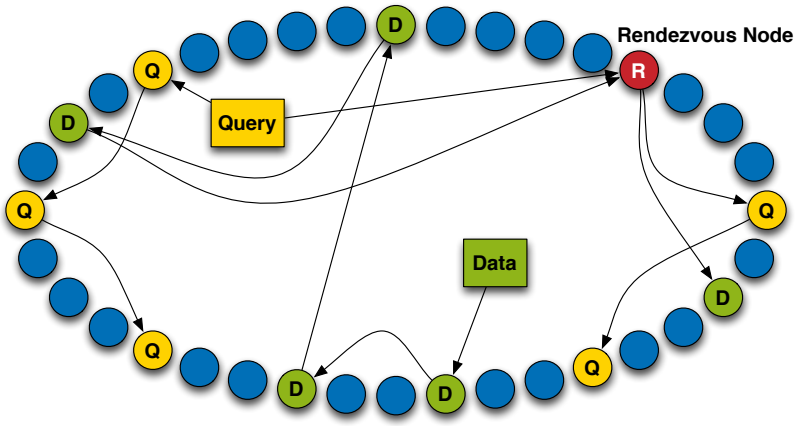


Figure 2.10.: Random walks on a DHT ring with Hautakorpi

yet. They achieve reasonable success rates, when 50-75% of the nodes support the algorithm (depending on the type of DHT).

Even though the system is designed to run on a DHT, there is no reason why it should not work on an unstructured expander graph topology, since it does not make any use of the DHT's routing metric.

The paper gives some basic mathematical background to the problem, but does not contain an actual analysis of the algorithm. The system has been implemented and simulated with Oversim [8]. Unfortunately, the simulation experiments do not contain any churn. The topics of replica maintenance or updates are not touched in the publication.

2.12 Related Systems

There are a couple of systems beyond those described above, which are related to rendezvous search systems. The Gnutella overlay [64] can be understood as a degenerated version of a rendezvous search system. It provides the same interfaces for issuing queries, publishing data, and matching them against each other. The publication does not replicate the data item beyond its publisher. Therefore, the query must be flooded to all nodes in the network, which leads to an unacceptable cost of $O(n)$ messages. Greatly simplified, Gnutella could be understood as a degenerated version of BubbleStorm with an extreme replication count imbalance.

The percolation-inspired search overlay from Sarshar et al. [125] can be used with a rendezvous search interface as well. The system is based on a power-law topology, which was believed to be natural for unstructured overlays—an assumption which has been disproved lately [137]. In contrast to Gnutella, it replicates its data items with a random walk of sub-linear length. Queries are also replicated with a similar random walk at first, but then a probabilistic dissemination scheme based on bond percolation is started from the nodes in the random walk. The system heavily depends on heterogeneous node degrees to perform competitively. It seems questionable if a maximum node degree of $O(\sqrt{n})$ or even $O(n)$ is the common case for typical P2P environments.

PathFinder [15, 72] is an unusual approach to structured P2P search. It uses a huge pre-generated random graph to construct a random topology and then computes routing

paths between the client and the node responsible for the requested key locally based on the pre-generated random graph. Although the algorithm seems a bit exotic compared to traditional DHTs, it is appealing that its topology can be used for expander graph based rendezvous search systems. The authors suggest to use BubbleStorm for arbitrary searches on the PathFinder topology and the PathFinder algorithm itself for key-value lookups. While this is a very useful combination of strategies, this approach unfortunately requires to maintain independent replication sets for both algorithms within the same topology. This does not only waste storage space, but might additionally lead to consistency issues with mutable data.

The content discovery system (CDS) from Gao and Steenkiste [41, 42, 43] was probably the first search overlay to apply the idea of the grid protocol from quorum systems to the P2P scenario. It is not a rendezvous search system, though. The CDS is an inverted index of attribute-value pairs stored in a DHT. The authors recognized the possible imbalance a DHT-based inverted index would impose on the nodes' workload. Therefore, they designed the system to replace overloaded nodes with a load balancing matrix (LBM) of multiple nodes, which processed the put and get requests to the given key with the classic grid protocol. Although being a pioneering work in the field, CDS had little influence on the development of those systems and is much less powerful than a real rendezvous search system.

2.13 Comparison & Review

Rendezvous search systems are a powerful, resilient, and flexible method of large-scale distributed search. This makes them a perfect building block for sophisticated P2P applications. An application developer can pick from a wide range of different solutions, selecting an approach that fits best into the existing infrastructure and fulfills the application's non-functional requirements. The similar interfaces of rendezvous search systems would make applications conceptually portable between different rendezvous search implementations, which is especially useful if the application requirements change.

There are multiple solutions for each environment: data center, structured P2P, and unstructured P2P. In the context of data centers, the classic grid approach is easy to implement and efficient. ROAR on the other hand is much more sophisticated and provides a high degree of self-adaptivity.

For structured overlays, the choice is much harder to make. Bit Zipper is fast, deterministic, and probably easy to implement. ROAR—running on a DHT—has rather slow and failure-prone sequential dissemination, but is feature-rich with replica maintenance and traffic balancing. Deetoo has fast dissemination and replica maintenance, but would need to borrow traffic balancing from ROAR. Unfortunately, it cannot guarantee deterministic success. It is also unclear how expensive the maintenance of two Chord rings is in practice. The system from Hautakorpi is not very compelling in comparison to the others. It is comparatively slow on dissemination, does not support replica maintenance or updates, and only has probabilistic guarantees. The possibility of an incremental deployment is an unique selling point, though.

In the area of unstructured overlays, the Ferreira system has pioneered the field, but is slow, fragile, and has problems with replica placement. BubbleStorm is definitely

	Google Grid	*Bit Zipper*	*Ferreira*	*BubbleStorm*	*ROAR DHT*	*ROAR DC*	*Deetoo*	*Hautakorpi*	*SplitQuest*
Publication	[6]	[141]	[36]	[143, 145]	[110]	[108, 109]	[22]	[56]	[84]
Year	2003	2004	2005	2007	2007	2009 2011	2010	2010	2010
Substrate									
data center	yes					yes			
structured		yes			yes		yes	yes	
unstructured			yes	yes				(yes)	
semi-structured									yes
Routing									
deterministic	yes	yes			yes	yes	yes		
probabilistic			yes	yes			yes	yes	yes
self-matching			yes	yes					
Replication									
data	direct	tree	walk	tree	walk	direct	tree	walk	tree
query	direct	tree	walk	tree	lookup	direct	tree	walks	trees
maintenance					yes	yes	yes		
Updates									
possible	yes	yes			yes	yes	yes		yes
consistent									
Self-Adaptivity									
network size		yes	yes	yes	yes	yes	yes	yes	yes
traffic balance				yes	yes	yes			

Table 2.1.: Comparison of rendezvous search systems

more sophisticated and provides traffic balancing and fast, reliable dissemination. The two systems are the only ones able to use self-matching types, but neither supports updates or offers a solution to replica maintenance. SplitQuest with its semi-structured concept is a very interesting new approach that might combine the resilience of unstructured systems with the efficiency of structured overlays. It is fast and supports updates, but does not have traffic balancing, replica maintenance, or self-matching types. The biggest downside is, that the theory behind SplitQuest is an open issue which makes it impossible to calculate the success guarantees of the system.

The high replication degree of rendezvous search systems is both a blessing and a curse. A large number of replicas not only makes a system naturally resilient against data loss because of churn and catastrophic failures, but also allows a very even load distribution between nodes. Unfortunately, most of the systems lack mechanisms for replica maintenance and updates. Even those systems that are able to update data items do not deal with consistency problems on concurrent updates. The high replica count makes this especially challenging in the area of unstructured overlays, in which the whereabouts of replicas are hard to determine.

Since this work aims to solve replica maintenance in open-membership overlays, which operate in a highly unreliable Internet-based environment, data center and structured rendezvous search systems do not provide a useful basis. In the field of unstructured rendezvous search systems, BubbleStorm is the most evolved and reliable system. Therefore, it will serve as the basis for the replica maintenance algorithms presented in this work. Nonetheless, the algorithms can be applied to other unstructured rendezvous search systems and—to some extent—to their data center and structured cousins as well.

3 BubbleStorm

The replication mechanisms presented in this work are targeting BubbleStorm, because it currently is the most sophisticated of the unstructured rendezvous search systems. Unstructured systems seem to be the most resilient class of rendezvous search systems, and therefore the best choice for open-membership Internet-based P2P overlays. Other probabilistic systems like the random walk system by Ferreira et al. [36] or the DHT-based random walk system by Hautakorpi and Schultz [56] are similar enough for the mechanisms to be as easily implemented with these systems. However, BubbleStorm is the most mature solution and thus the natural choice.

The BubbleStorm system has been presented at ACM SIGCOMM 2007 [143]. Additional information like the proof of correctness can be found in the accompanying technical report [145]. The basic structure of the BubbleStorm system like topology, item dissemination, network size measurement, and traffic balancing are covered in the dissertation of Wesley Terpstra [140]. The implementation of BubbleStorm uses CUSP [146], which is a novel transport protocol designed for complex and dynamic distributed systems like P2P applications. It has been developed as part of the BubbleStorm research project.

For the reader's convenience, an overview of the existing BubbleStorm infrastructure is given here. The system builds upon a random multigraph topology protocol, which enables a random tree-based item dissemination mechanism known as bubblecast. In BubbleStorm, a set of replicas of a given item is called a bubble. The topology is also used by a gossip protocol which is able to compute network-wide sums, averages, minima, and maxima [144]. This is used to measure the network size, which is needed to determine bubble sizes. The system-wide traffic injection of each type of bubbles is also measured with the gossip protocol. This information is used by the bubble balancer to compute optimal bubble sizes with respect to bandwidth usage.

3.1 Concept

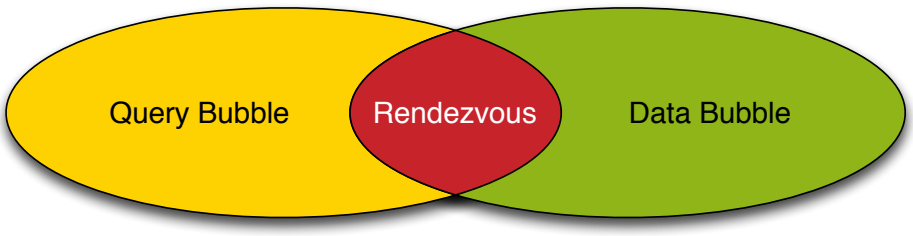


Figure 3.1.: Intersecting bubbles in BubbleStorm

BubbleStorm is a probabilistic and unstructured rendezvous search system. Each item (data or query) is replicated on a random set of nodes. This set of replicas specific to

certainty factor λ	1	4	9	16
traffic factor ($\sqrt{\lambda}$)	1	2	3	4
probability $1 - e^{-\lambda}$	63.21%	98.17%	99.99%	99.99999%

Table 3.1.: Intersection probability as a function of λ

the given item is called a *bubble*. Bubbles that carry the same type of item (e.g., a certain type of data records) are grouped together into a *bubble type*. BubbleStorm takes care that bubbles of bubble types which are supposed to intersect (like query and related data) do rendezvous with a certain probability. This probabilistic guarantee of $\mathbf{P}(\text{fail}) \leq e^{-\lambda}$ can be tuned by the application developer by setting the *certainty factor* λ of the intersection (see Table 3.1). The sizes of two intersecting bubble types are chosen by the system to suffice the following theorem and using the definition $g(z) := 1 - e^{-z}$.

Theorem 1. *Place x replicas of type X and y replicas of type Y uniformly at random on n nodes. Let M be the number of nodes receiving at least one replica of each type. Then, $\mathbf{P}(M = 0) \leq e^{-\lambda}$ whenever,*

$$g(\lambda/n) \leq g(x/n)g(y/n)$$

Theorem 1 is only valid for homogeneous networks with uniform node degrees. It can be generalized for heterogeneous networks by using a number of additional graph properties of the topology.

Definition 2. *Let $d(v)$ be the degree of node $v \in V$ in a graph (V, E).*

$$D_j := \sum_{i=1}^{n} d(i)^j$$

$$d_{max} := max\{d(v) | v \in V\}$$

In BubbleStorm, D_0, D_1, D_2, and d_{max} need to be monitored. D_0 is the network size ($D_0 = |V| = n$), D_1 is the sum of degrees, which is twice the number of edges ($D_1 = 2|E|$), D_2 is the sum of squared degrees, and d_{max} is the maximum degree in the network. The heterogeneous formulation of the bubble size equation is as follows:

Theorem 2 (The Bubble Size Equation)**.** *Place x replicas of type X and y replicas of type Y on the nodes V with probability proportional to their degree $d(v)$. Let M be the number of nodes receiving at least one replica of each type. Then $\mathbf{P}(M = 0) \leq e^{-\lambda}$ whenever,*

$$g\left(\lambda \frac{{d_{max}}^2}{D_2}\right) \leq \left(x \frac{d_{max}}{D_1}\right)\left(y \frac{d_{max}}{D_1}\right)$$

The bubble size equation ensures the probabilistic guarantees given by BubbleStorm. Proofs for Theorem 1 and Theorem 2 can be found in [140].

3.2 Topology

BubbleStorm's main concept—replicating bubbles to a random set of nodes—heavily depends on the overlay's topology. BubbleStorm uses a random multigraph topology, which means that each edge connects two randomly chosen nodes. It is a multigraph, because two nodes may be connected by more than one edge (a multi-edge) and an edge might even connect a node with itself (a self-loop). Multi-edges and self-loops are extremely rare in large networks and thus do not impact the performance, but are very useful for bootstrapping a small network and simplify the topology protocol.

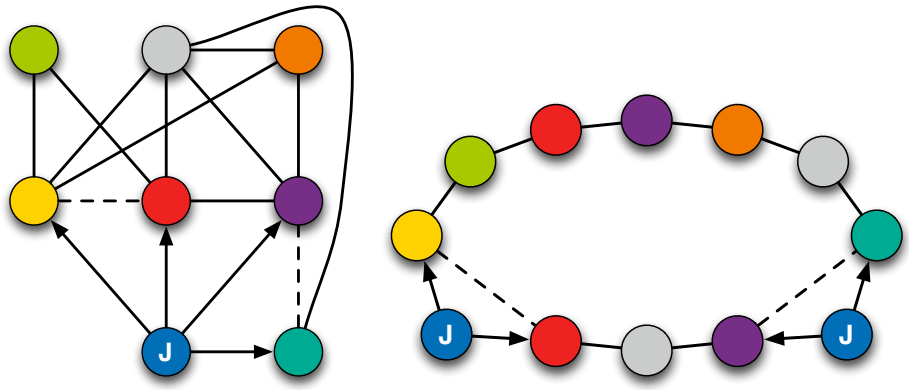
An overlay topology should avoid partitioning of the graph. Unfortunately, the basic definition of random graphs does not require the graph to be connected. Therefore, BubbleStorm's topology is based on an Eulerian circuit, which is a tour through all edges of the graph that visits every edge exactly once. In contrast to an Eulerian path, an Eulerian circuit ends at the starting node and thus forms a cycle. Every node appears $d(v)/2$ times in the circuit. This occurrence of the node is called a *location*.

In order to build a connected random multigraph based on an Eulerian circuit, every node picks a number of locations equal to its desired degree. These locations are then randomly permuted. The permutation defines the path of the Eulerian circuit and thus the edges the graph contains. Since every node with a non-zero desired degree is contained in the circuit, the graph is connected.

The notion of a *desired degree* is a useful concept for load balancing. Every topology edge carries the same expected load and therefore the expected load of a node scales linearly with its degree. In BubbleStorm, every node decides the capacity (bandwidth, CPU, storage) it wants to offer to the system, and the desired degree is derived from this capacity. The topology will take care that the node gets a degree equal or close to its desired degree. There is a *minimum degree*, which a node has to match to participate as a peer in the overlay. It is currently set to 16. Nodes with a capacity that does not suffice for a degree of 16 should use more powerful peers as a proxy to access the overlay. Such proxies are called *super nodes* in peer-to-peer networks and have been successfully used by unstructured systems like Gnutella [64], FastTrack [85], and Skype [53].

The main challenge is to define a protocol that creates and maintains the specified topology in a decentralized, self-organizing, and iterative fashion. The node that is creating a BubbleStorm network connects its locations in a daisy chain. All edges in the network are self-loops at this point. When a node wants to join the overlay, it contacts an arbitrary node in the overlay. It uses this bootstrap node to start a random walk of length $O(\log n)$ for each location. The random walk is long enough to select a node in the random graph with a probability proportional to its degree [12]. The receiving node then selects one of its edges at random. The two steps combined select a random edge with uniform probability.

This edge is then split. Instead of being connected to each other, the two nodes of the edge connect to the joining node. This leaves their degree unchanged and adds two neighbors to the joining node for each location. Therefore, each node can get to its desired degree (if it is even) without affecting the degree of any other node. This procedure is equivalent to adding the node's locations at random positions in the Eulerian circuit (see Figure 3.2).



(a) The nodes in the random graph

(b) The corresponding Eulerian circuit

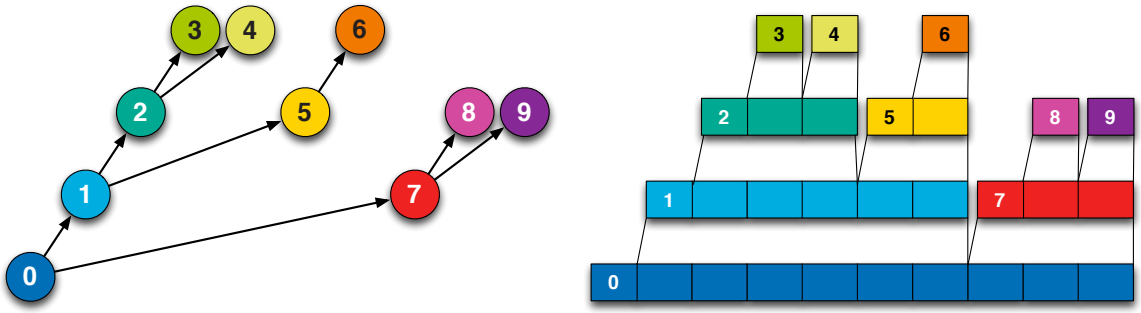
Figure 3.2.: A node with two locations joining the BubbleStorm overlay

When a node wants to leave, it simply connects the two neighbors of each location back together. Not all nodes leave the network orderly: some crash, some lose connectivity or power. In this case, the neighbors cannot be spliced together. They will detect the loss of a neighbor after a communication timeout and have to remove the connection from their neighbor table, which reduces their degree by one. This is called a *broken edge*. If the node's effective degree is at least two below its desired degree, it can create an additional location to find two additional neighbors with a random walk. Broken edges cannot be repaired, but if a node's location consists of two broken edges, this inoperable location can be deleted, which reduces the number of broken edges in the overlay.

Random graphs are extremely robust. Even with a large number of edges removed, the vast majority of nodes (the *giant component*) will stay connected [12]. This is not only important for resilience against individual or large scale crashes, but enables BubbleStorm's operation in spite of broken connections due to firewalls, NATs [132], and communication anomalies. In contrast, DHTs require connections between certain nodes to operate correctly, e.g., Chord [136] requires each node to be connected to its direct successor. Since such requirements do not hold on the Internet, Chord shows a significant and uncontrollable failure probability of several percent in real-world experiments [47].

3.3 Bubblecast

BubbleStorm uses trees to replicate items randomly in the network. The bubblecast mechanism combines the advantages of random walks and flooding. In flooding, the sender forwards the message to all its neighbors, which process it and forward it to all their neighbors (except the sender) in turn. This process is repeated recursively for a number of hops. Due to its large fan-out, flooding is fast and reliable, but the simple hop counter makes it impossible to select a precise number of receiver nodes. In a setting with heterogeneous node degrees, the edge load becomes unbalanced. A node with a



(a) The multicast tree with the node numbering **(b)** The splitting of intervals between the nodes

Figure 3.3.: An example bubblecast with a branch factor of 2

high degree will forward more messages to its neighbors, because it is receiving a lot. This makes degree-based load balancing impossible.

A random walk is the exact opposite. It is only ever forwarded to a single random neighbor. Therefore, its hop counter can precisely select the number of receivers. Random walks also induce even load on all links in the topology [12], which is perfect for degree-based load balancing. Unfortunately, random walks of length $O(\sqrt{n})$ are extremely slow and unreliable in large networks [143].

Bubblecast forwards messages to a fixed number of randomly selected neighbors in each step and splits the remaining number of nodes to reach between the receivers. Therefore, it shares the fast and reliable tree structure of flooding and the precise receiver count and edge load balance with random walks. Branching 4-way in every step has proved to provide excellent response times [140].

Essentially, bubblecast recursively chops up the random walk. The set of remaining nodes can be understood as an interval. Every receiving node removes the head element, because it processes the message itself. It then splits up the remaining interval equally between the nodes it forwards the bubblecast to (see Figure 3.3). Like in a random walk, this imposes a numbering on the nodes, which can be used by the replication mechanisms described later. The numbering happens in depth-first preorder.

Bubblecast, like flooding or a random walk, does not truly place items randomly in the network. All items of a bubble are inside of a connected subgraph. Since query and data are using the same topology, this means that there are fewer exterior edges to reach the data bubble than there should be with true random placement. Therefore, the success probability is less than in theory. BubbleStorm uses a *dependency correction factor*, that increases bubble sizes to compensate for the interdependency. This correction should also be used with related rendezvous search systems like Ferreira [36] and Hautakorpi [56]. The dependency correction factor F is defined as

$$F := \frac{D_2}{D_2 - 2D_1}$$

3.4 Bubble Balancer

The bubble balancer computes bubble sizes with respect to the bubble size equation to ensure the requested guarantees. It also applies traffic balancing to minimize the bandwidth usage of bubblecasting. Every type of bubbles $t \in T$ injects a certain amount of raw traffic S_t into the overlay. This traffic is the sum of the sizes of all items of that type being disseminated (which can be measured system-wide with BubbleStorm's gossip protocol). Let x_t be the bubble size of type t. The total bandwidth usage of a bubble type is $S_t x_t$.

The balancer seeks to minimize the total system traffic (subject to Theorem 2), which is

$$\sum_{t \in T} S_t x_t$$

This optimization problem can be formulated convex [140] and solved with a convex optimizer [14]. BubbleStorm monitors S_t for all bubble types and uses this information for a periodic recalculation of the bubble sizes. To avoid all too sudden jumps, the traffic data is smoothed with an exponentially weighted moving average. BubbleStorm is currently the only rendezvous search system that is able to monitor and optimize its bandwidth usage without needing any manual intervention.

3.5 Gossip Protocol

Topology, bubble balancer, and bubblecast depend on measuring global system attributes. The topology needs the network size n ($= D_0$) for computing the length of the $O(\log n)$ random walks for joining. The bubble balancer uses the topology properties D_1, D_2, and d_{max}, and traffic measurements S_t for the correct and traffic-optimal calculation of bubble sizes. Bubblecast needs D_1 and D_2 for calculating the dependency correction factor.

The measurements contain input from all nodes, and the results are required by each node. BubbleStorm therefore deploys a gossip protocol that allows collective calculation and system-wide dissemination of sums, averages, minima, and maxima of data provided by the participating nodes.

The gossip protocol is based on the push-sum algorithm [67], which uses mass conversation and density convergence to provide measurements after $O(\log n)$ message exchanges. In BubbleStorm, the algorithm has been adapted for asynchronous and completely decentralized communication [144]. The protocol is round-based and provides a new set of results after every measurement round. During a round, each node sends fractions of its current measurement values to its neighbors in a round-robin fashion. Received measurements are added to the local values. When the local measurement has been stable for a sufficient number of message exchanges, the node issues a round switch and notifies its neighbors, which propagate the round switch recursively.

The new round is started with fresh input values at each node. Naturally, the gossip protocol causes a delay between the state of the computed measurements and the actual state of the system. This delay consists of the measurement round needed for

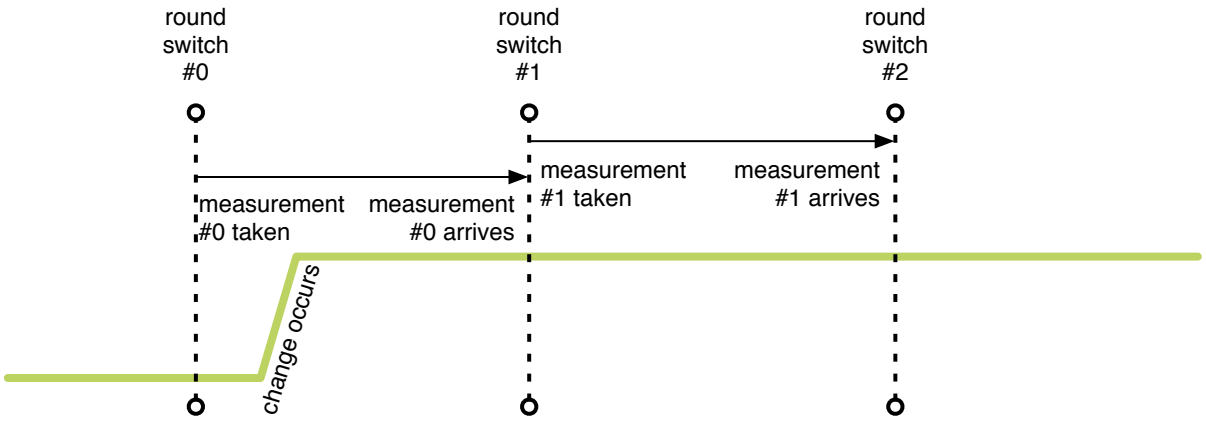


Figure 3.4.: Measurement delay of the gossip protocol

computation and the time since the last measurement round output. Therefore, the measurement delay is between one and two rounds (see Figure 3.4). This influences the time span BubbleStorm needs to adapt to changed environment parameters, which can be seen in simulations (see Section 10).

3.6 CUSP

Early implementations of BubbleStorm revealed the limitations of the traditional Internet transport protocols, TCP [104] and UDP [103]. TCP offers many powerful features like reliable in-order delivery, flow control, congestion control, and fragment reassembly, but is specifically designed for scenarios with a single application data stream per connection. In BubbleStorm, many concurrent messages for topology, gossip, and bubblecast need to be routed through the same connection. Since especially bubblecast messages can be larger than an IP packet, they are concurrent and independent data streams. An application using BubbleStorm for, e.g., video streaming or content distribution, would add long-lived bulk data streams. TCP would suffer from head-of-line blocking, which means that a lost packet in any stream stops progress in all other concurrent (and independent) streams. Using a separate TCP connection per stream is not a good solution, since it would require a slow three-way handshake each time.

UDP is not really an alternative. It is an extremely simple, unreliable datagram protocol, that can only be used as a platform for application-level protocols. Re-inventing transport level functionality like congestion and flow control at application level for each application protocol separately seems wasteful and shortsighted.

TCP and UDP do not even provide confidentiality and authenticity. Any application requiring a minimum of security would need to use TLS/SSL [32] on top of the transport protocol, adding message header overhead and additional handshake steps.

Therefore, we decided to design a novel transport protocol, the Channel-based Unidirectional Stream Protocol (CUSP) [146], which comprises the above mentioned features and is flexible enough to satisfy the many different requirements in complex application scenarios. While other modern transports like SCTP [134] or SST [39] have also tried to combine the advantages of TCP and UDP, CUSP overcomes their technical and concep-

tual shortcomings. Like SST, CUSP multiplexes dynamically created lightweight streams on a channel between two nodes. Low-level packet management like negotiation, cryptography, congestion control, and reliability are being managed by the channel. Streams ensure in-order delivery and can be created and destroyed without an expensive three-way handshake. Unlike in SCTP and SST, CUSP's streams are unidirectional, because many routing overlay scenarios do not need bidirectional streams. If bidirectional communication is required, two unidirectional streams (one in each direction) can be used as easily as bidirectional streams.

Because new protocols running directly on IP are hard to deploy, since they typically require kernel-level privileges and have to be supported by intermediate systems like firewalls and NAT routers, CUSP is encapsulated in UDP packets. This enables easy deployment and non-privileged userland implementations. Furthermore, existing UDP techniques for NAT traversal can be re-used [116].

Although originally designed for BubbleStorm, CUSP is a full-fledged transport protocol and could be used for any application that works with TCP, SCTP, or SST. It is especially useful for applications with many complex and concurrent operations. It has already been used for a real-time MMOG gaming application [74, 75].

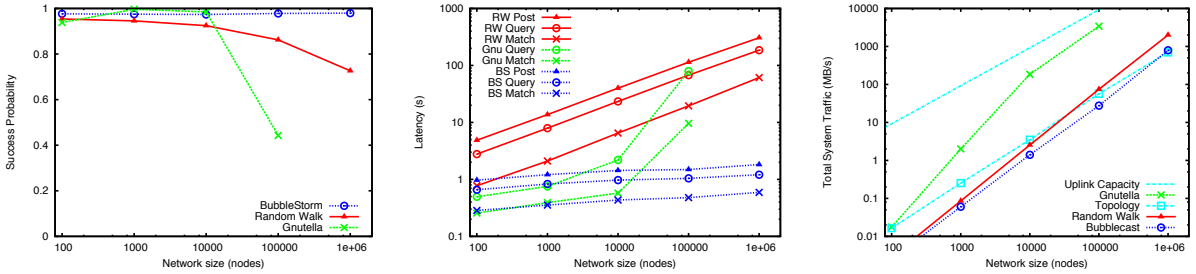
The problems tackled by CUSP are not unique to P2P overlays. HTTP [38] suffers from many problems like head-of-line blocking, connection setup latency (especially with encryption), and the lack of server-initiated messages. The SPDY protocol [49] tries to solve most of these problems by modifying HTTP, but keeps TCP for ease of deployment. By using (the also easy-to-deploy) CUSP instead of TCP, SPDY could achieve even more and with less effort.

3.7 Evaluation

For the original BubbleStorm conference publication [143], an early version of BubbleStorm was evaluated with a custom-built message-based simulator. The very space-efficient implementation in C allowed for experiments of up to one million nodes. It did not use CUSP, but a simple simulation model of TCP, and the protocols for topology maintenance and gossip. Furthermore, the traffic balance was static. Nonetheless, the results give a good impression of the capabilities of the lower layers of BubbleStorm, which are used as the basis for the replica maintenance and update mechanisms. An in-depth performance evaluation of the current BubbleStorm implementation can be found in the dissertation of Wesley Terpstra [140].

BubbleStorm is compared to the random walk system by Ferreira et al. [36] and the Gnutella flooding network [64]. All simulations were conducted with a λ of 2 (= 98.17% success probability). BubbleStorm provides the guaranteed success as specified (see Figure 3.5a). The Ferreira system drops in success because of the fragility of long random walks and the problem with collisions (see Section 2.7). Gnutella breaks down from overload at 100k nodes.

The response time plot shows a similar picture (see Figure 3.5b). Random walks cannot offer competitive latency. Gnutella performs great for small networks, but becomes quickly saturated.



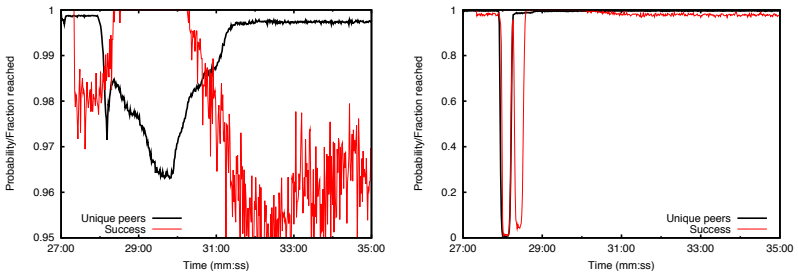
(a) Success probability **(b)** Response times **(c)** Bandwidth usage

Figure 3.5.: Performance evaluation of BubbleStorm, Ferreira, and Gnutella from [143]

The traffic analysis divides into topology and item dissemination traffic (see Figure 3.5c). The Ferreira system is run on the BubbleStorm topology. It uses significantly more traffic, because, unlike BubbleStorm, it cannot benefit from node heterogeneity. Gnutella is extremely inefficient and collapses at a network size of 100k. It dies of congestion even though it is using only ≈ 35% of the available bandwidth. This is caused by the load imbalance introduced by flooding in a heterogeneous network.

Figure 3.6 shows the success of BubbleStorm after 90% of all nodes leaving or 50% of all nodes crashing simultaneously. This is a very extreme stress test for any overlay network. BubbleStorm quickly recovers in both cases. In the crash scenario, the success rate drops temporarily, as messages are sent to crashed nodes before their disappearance is discovered by timeouts. In the leave scenario, a longer time disruption happens as the nodes wait to get their neighbors' edges spliced together. In both cases the success rate increases temporarily. This is caused by the delay of the gossip protocol. The measurements do not yet reflect the reduced network size and therefore the bubble sizes are too big. That the success rate does not reach the specified level again, is not caused by the leave event but is an artifact of the simulation setup. The network is growing rapidly after the leave event, because the global arrival rate is unchanged. In this case, the delay of the measurement protocol makes bubble sizes too small.

More details about the simulation setup, additional results, and an in-depth discussion of the observations can be found in the paper [143].



(a) 90% leave **(b)** 50% crash

Figure 3.6.: BubbleStorm under catastrophic failures (from [143])

3.8 Review

BubbleStorm is a versatile and sophisticated rendezvous search system, that has been designed and implemented with painstaking attention to correctness and performance. Based on a solid theoretic foundation, the system proved to be extremely resilient and very adaptive. Equipped with the versatile CUSP transport and the useful gossip protocol, BubbleStorm provides a good set of tools for the implementation of the replica maintenance and update mechanisms presented in the following chapters. These mechanisms are needed to make BubbleStorm a complete system, which can be used in practice.

4 Replication Modes in P2P Overlays

Replication is an essential feature for any P2P search overlay. Without replication, churn would cause search index structures to decay over time. Many search overlays and especially the rendezvous search systems use replication to improve scalability and search success rates. Not only data items are replicated, queries are often copied to many nodes too. Even though replication plays a major role in the design of search overlays, its importance is missed by many overlay developers, and the role of replication in P2P overlays has not been systematically examined yet.

Replication is the placement of information items on multiple distinct nodes in a distributed system. In the context of P2P search overlays, replication consists of the dissemination, maintenance, and update of items. The dissemination of items describes the original placement of replicas in the network. The maintenance of replicas ensures the desired replication degree in the face of churn. Update mechanisms manage the freshness and consistency of replicas when the item is updated. This should include resolving conflicts on concurrent update requests.

In this chapter, a taxonomy for P2P search overlay replication is presented. Four replication modes are identified and motivated: instant, fading, managed, and durable (see Figure 4.1). These four fundamentally different modes are compared and put into context of the current state of the art in P2P replication. The fundamental difference of the replication modes suggests that more than one replication algorithm is needed to cover all modes.

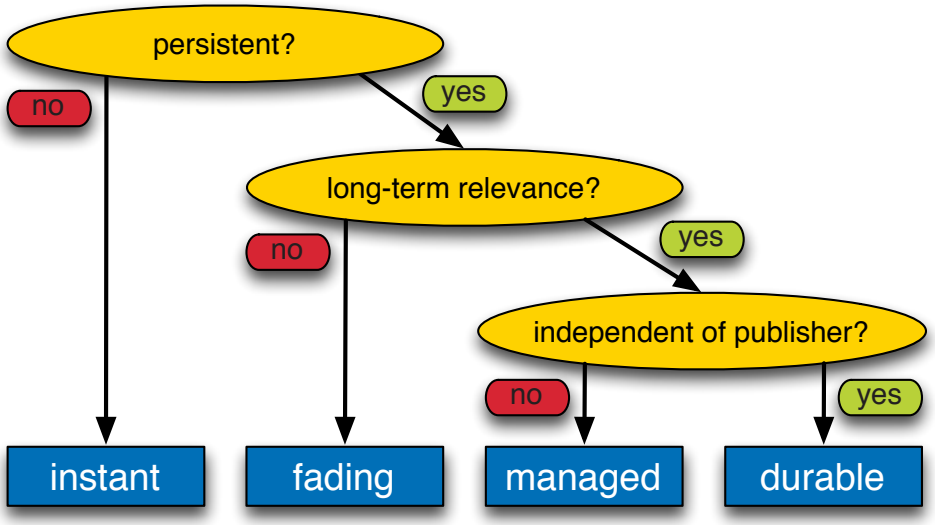


Figure 4.1.: Replication modes in P2P networking

4.1 Instant Replication

In order to categorize the different replication modes, the different requirements of the application scenarios need to be examined. The most essential question is, whether the items need to be stored. Quite a few item types are non-persistent, e.g., queries in query/data and publications in publish/subscribe. Such items are processed at the receivers and may trigger system reactions like responses or notifications, but are discarded immediately afterwards. This class of items does not need maintenance and can by its nature not be updated. I call this the *instant* replication mode.

4.2 Fading Replication

Even if the item is persistent, the overhead and complexity introduced by replica maintenance is not always justified. Short-lived information like position updates [86] or additional replicas to relieve hot-spot peers [91] are typical candidates for this replication mode, which will be called *fading*. If replicas are not maintained, the replication degree will shrink over time, because replica-holding peers leave the overlay. Sooner or later the replication will not be sufficient for the desired availability and search success anymore. Therefore, fading replication is often used for items with an expiration timer, which are deleted after a certain amount of time, because they are outdated anyway.

4.3 Managed Replication

If maintenance for long-term persistence is needed, the responsibility for the stored items needs to be determined. If there exists a node that can be made responsible for maintaining the replicas the replication mode is called *managed* replication. The maintainer node can be used to simplify the maintenance and update algorithms. The maintainer can monitor and adjust replication levels without complex coordination with other nodes. Furthermore, updates can be serialized through the maintainer, which greatly simplifies consistency management.

I distinguish between natural owners and dynamically assigned maintainers. A natural owner is a node that is inherently linked to the managed data. This relationship is typical for data that references a specific node. Examples are service offerings of a node (e.g., a list of shared files) or subscriptions (e.g., a list of buddies that a node is interested in). The lifetime of the data is bound to the maintainer node; if the node goes offline, the data should disappear.

Dynamically assigned maintainers require a slightly more complex model. There is no immanent relationship between data and maintainer, but data is assigned a maintainer by some selection algorithm. An intuitive approach is to use the responsibility ranges in DHTs to assign the maintainer. The node with the ID closest to the ID of the data becomes its maintainer [97, 98]. Since responsibility ranges in DHTs change under churn, the maintainer of a certain data item can change anytime. Routing inconsistencies in DHTs [47] can lead to ambiguities in the maintainer-data relationship. If two nodes act as maintainer for the same data and serialize update requests independently, inconsis-

tent states are a likely result. Dynamically assigned maintainers should therefore only be used in reliable environments and not in large-scale open-membership overlays.

4.4 Durable Replication

If the data to be replicated is not owned by any particular node, and the environment is too unstable to employ dynamically assigned maintainers, a collective approach is needed. A typical usage scenario would be articles in a P2P wiki. An article does not belong to any particular node and should stay available even if all of its authors have left the system. Without a dedicated maintainer, the responsibility for the data item becomes shared. This makes maintenance and update mechanisms especially challenging, because the decision making process needs to be distributed. Since this mode aims for long-term persistence without any lifetime constraints, it will be called *durable* replication.

The first challenge is to determine and contact the nodes responsible for storing a given item. DHT-based solutions [68, 2] typically use a set of hash functions and key lookups to find the responsible nodes. Unstructured systems can try to connect all replica holders of an item and flood the update requests through this specialized overlay [153].

When concurrent updates happen, a local conflict resolution has to decide which update takes effect and which update gets rejected. Since this is a difficult problem, most collective mechanisms rely upon external means like a reliable unique timestamp source [2] or by enforcing application-level conflict resolution through a reject-both strategy [153].

A notable side-effect of durable replication is the difficulty of deleting items. Only if all replicas have been purged from the system and will not be brought back by offline nodes re-joining the overlay, an item can be considered deleted. Otherwise, the remaining replicas might “re-infect” the system by being spread to previous replica holders again. A way to immunize the nodes against already deleted data is to replace replicas with so-called tombstones or death certificates instead of deleting them [122].

4.5 Related Work

The first pure P2P search overlay, Gnutella [64], kept data exclusively local, and therefore did not need any replication algorithm. With the introduction of super nodes in Gnutella and similar systems, a one-hop replication from the leaf node to the super node became necessary. This replication follows the managed mode, with the leaf node being the maintainer. In the next generation of unstructured overlays, Gia [20] extended this one-hop replication to all neighbors of every node, but the semantics remained unchanged. Both Gnutella and Gia use instant replication for queries. Gnutella uses hop-constrained flooding, while Gia employs short random walks.

Most of the better known DHT search overlays only touch replica dissemination, but mostly leave out updates and maintenance. Chord [136] suggests to replicate the data on a number of successors of the responsible node in the ring. Pastry [117] and Kademlia [91] store the replicas on the nodes closest to the key of the data, which is quite

similar to the Chord approach. CAN [112] and Tapestry [158], however, use a set of different hash functions to determine the responsible nodes for a given key. Tapestry additionally places a copy of the replica on each node along the routing paths to avoid hot-spots. CAN, Pastry, and Kademlia employ similar types of caching, but do this on top of the normal replica dissemination using separate mechanisms.

Chord leaves all replica maintenance and update handling to the application developer, thus only providing fading replication by itself. Tapestry and CAN use periodic republication and expiry of items, which is a form of managed replication. In Pastry and its storage system PAST [118], each peer monitors all other nodes responsible for the same data. When responsibilities change due to failures or joins, the replicas are exchanged among the affected nodes. This is an example of collective replication. CAN and Pastry additionally use caching to mitigate hot-spots. Additional replicas are placed along the routing paths as part of the lookup process or—in case of CAN only—actively by overloaded replica holders. Cached replicas expire or are evicted from the cache eventually. In both cases, the caching policy is separate and not coordinated with the replica managing peers. This has a negative impact on possible consistency guarantees for updates. Even worse, updates and consistency issues for items in the search overlay are not discussed by any of the mentioned DHTs.

Kademlia follows a "the more the merrier" strategy by combining all three replication modes for maintaining persistent data. The replica holding peers contact all other responsible peers every hour and exchange missing replicas with them, thereby implementing a durable mode. Additionally, the original publisher has to refresh an item every 24 hours or it expires, which makes it a managed replication. On top of that, requesters put a copy of the item on the last hop of the routing path to lessen hot-spots. These cached copies are not maintained and have an expiration counter like in fading replication. This combination of strategies does not only add a lot of complexity, but also limits the replication effects to the weakest of the used modes. Even though durable maintenance is used, the lifetime of an item is limited to the publisher's lifetime (plus one day at most). Although managed and durable replication would allow updates, the cached copies make consistent updates impossible. Kademlia is a good example why it is important to decide which replication mode is appropriate for a given type of data and not to mix them unnecessarily. A clear taxonomy makes this obvious, but without such a concept, it is hard to completely understand the consequences of the replication algorithm design.

The lack of consistent update mechanisms for data in DHTs has led to a number of add-on update mechanisms. Knežević et al. [68] propose a system based on multiple hash functions using durable replication. The approach solves conflicts by using version numbers and updating the replica holding nodes in a certain order. Every peer periodically queries the network for the newest version of the replicas it is responsible for. This mechanism ensures eventual consistency even when updates have been missed, but does not give an upper bound for convergence time.

The Update Management System [2] attempts to avoid update conflicts by using unique timestamps which can be obtained by a timestamp service. The authors propose to run a timestamp service for each key in a distributed fashion over the DHT. The peer responsible for a key's timestamps effectively serializes the update requests, which

makes this update scheme a managed replication with dynamically assigned maintainers. The replica maintenance, however, is done in a durable fashion, where a peer obtaining a new responsibility is pulling the replicas from the overlay without coordination from the maintainer.

Antony et al. [3] propose a replication and update mechanism for the P-Ring DHT [28], which provides transactional semantics. Replicas are stored on a successor chain similar to Chord. Updates are sent to the first node in the chain, which forwards the request to its successors. Each of the nodes only forwards the request, if it accepts the update. The last node sends an acknowledgement in the opposite direction. Upon receiving the acknowledgement, the nodes execute the update. To maintain the replicas, the head periodically sends information about items to replicas along the successor chain. The approach is a pure managed replication with dynamically assigned maintainers.

The P2P-based wiki Scalaris [127] uses the symmetric replication mechanism [45] and extends it with an adapted Paxos atomic commit protocol [95] for guaranteed consistency of concurrent updates. The symmetric replication partitions the identifier space of the DHT and every item is replicated to each partition. On responsibility changes, the previous replica holder transfers the replica to the newly responsible peer. If the previously responsible peer crashed, the receiving node fetches the replica from one of the other partitions. Since it lacks coordinators, symmetric replication can be categorized as durable mode replication.

All these approaches have in common, that they rely on the correctness and consistency of the underlying DHT. Routing anomalies and responsibility ambiguities can lead to unforeseen results. Routing table inconsistencies are quite common in DHTs [47].

The P-Grid overlay [1] uses an epidemic update mechanism, which consists of a push and a pull phase [30]. New updates are recursively pushed to responsible nodes, and newly or re-joined nodes pull replicas they have missed. The system does not deal with concurrent updates.

A replication approach for unstructured systems by Wang et al. uses chains of peers for replication [153]. Under churn, peers originally located in the chain might become unavailable, and therefore each peer keeps a list of multiple predecessors and successors to bridge such gaps. Updates are propagated along the chain in both directions. Concurrent updates can be detected with a timestamped approach. In such a case, both updates are rejected and the requesting peers are required to jointly solve the conflict.

4.6 Review & Comparison

Even though replication is an essential feature to any P2P search overlay, the design space has not been analyzed systematically yet. The diverse types of data have varying, even conflicting requirements. In this chapter, the common use cases of replication in search overlays have been described, a taxonomy for replication mechanisms from them has been derived, and existing algorithms according have been classified to this schema. An overview of the identified replication modes can be found in Table 4.1.

Mode	Dissemination	Persistence	Maintenance	Updates	Conflict Resolution	Use-Case Example
Instant	yes	no	no	no	no	query
Fading	yes	limited	no	no	no	short-lived data, caching
Managed	yes	limited	yes	yes	n/a	list of shared files
Durable	yes	yes	yes	yes	yes	wiki article

Table 4.1.: Comparison of replication modes

Instant replication is mainly useful for queries or publication notifications, which are processed and forwarded, but never stored in the search overlay. This mode has at least the following requirements: no persistence, no maintenance, and no updates.

Fading replication is very similar, but the replicas are stored in the overlay, at least for a certain time. It is useful for temporary data or cached replicas. Fading replication cannot be used for data that needs long-term availability guarantees or consistent updates, but adds no maintenance overhead to the system.

Managed replication supports full-fledged maintenance and updates. It is especially useful for data that has a natural owner. The data can be maintained and updated by this maintainer. Since the maintainer can serialize update requests, this mode does not need a distributed conflict resolution. Most managed replication algorithms work with periodic keep-alive refresh messages. The variant with dynamically assigned maintainers even offers persistence beyond a single maintainer's lifetime. Unfortunately, it completely relies on the consistency of the leader election process and thus is not applicable to unstable environments.

Durable replication overcomes this and can be used for long-time persistence in Internet-based overlays. This collective approach is useful for shared data like wiki articles, but can also be applied to user-specific information, which should stay available when the owner is offline (e.g., account information). A collective of nodes responsible for the same data is typically connected directly or with a dedicated miniature overlay, which is used to exchange maintenance and update messages. Without a central point of control, concurrent updates must be resolved by a distributed conflict resolution or by using an external control source like an unique timestamp service.

The four replication modes are conceptually different and should not be mixed. If techniques from multiple modes are applied to the same data, persistence or consistency features can break and additional overhead is added. The taxonomy introduced here can give orientation during the design of replication algorithms to enable clean and focused designs.

5 Data Description Primitives for BubbleStorm

BubbleStorm is a flexible search overlay for large-scale open-membership systems in unreliable environments. It uses a large number of replicas for queries and data to achieve its probabilistic search guarantees. The flexibility and the extreme replication call for a very sophisticated data management.

In this chapter, the primitives for defining bubble types and their intersections are discussed. Four of the primitives enable the specification of bubble types of the four replication modes described in Chapter 4, and one primitive defines the intersections between types to enable reliable search guarantees. This primitive is called *match*, because it is typically used to facilitate the matching of queries against data or publications against subscriptions. When an application developer specifies that two bubble types must match, a *rendezvous function* must be given, which defines how matches are found and handled by the receiving node.

In the context of BubbleStorm, a *bubble* is the set of replicas of a single item. All bubbles of the same type of data are called a *bubble type*. A *bubble class* is the set of bubble types that are using the same replication mode.

The set of bubble types and matches between bubble types of a given application is called its *match graph*. The bubble types are the vertices and the matches are the edges. The matching graph contains descriptions for all types of data and their interactions and is a crucial characteristic of the application. This is comparable with relations and prepared statements in relational databases. The matching graph is the input for the bubble balancer, which periodically computes correct and optimal bubble sizes (see Section 3.4).

BubbleStorm itself does not provide a data storage component. The application provides BubbleStorm an adequate data store when a bubble type is defined. Thus, all incoming store, update, and delete requests are forwarded to this data store. This allows the application developer to re-use existing data storage libraries like relational databases or document repositories, while preserving maximum flexibility by separating the communication middleware from the storage backend.

5.1 Instant Bubbles

Instant bubbles are the most simple bubble class in BubbleStorm and implement the instant replication mode. They are non-persistent and thus do not feature replica maintenance or updates. Consequentially, an instant bubble type does not need a data store. Typical use-cases for instant bubbles are queries, pub/sub-style publications, notifications, and other process-and-forget types of data. The dissemination of instant bubbles is realized with bubblecast (see Section 3.3). Bubblecast's tree structure enables high

reliability and compelling response times, which makes it the optimal choice for interactive requests like queries. An instant bubble does nothing useful by itself. Only when the type is matched against a persistent bubble type, the rendezvous function will trigger the handling of the incoming data.

5.2 Fading Bubbles

Fading bubbles are the persistent counterpart of instant bubbles. The definition of a fading bubble requires a store function, with which incoming items can be added to the local data store of a node. Otherwise, fading bubbles work exactly like instant bubbles. They cannot be updated, are not maintained, and are disseminated via bubblecast. Accordingly, there is no maintenance overhead for fading bubbles. A typical use-case for a fading bubble type is short-lived data like position updates in online games [86].

5.3 Managed Bubbles

Managed bubbles implement the managed replication mode with fixed maintainers. The creator of a bubble automatically becomes its maintainer. Only this maintainer can update or delete the bubble, and if the maintainer leaves the network, the bubble will disappear too. The data store for a managed bubble type has to provide functionality to add, update, and delete items.

The underlying mechanism, which provides managed replication for BubbleStorm, will be described in Chapter 6. In short, every maintainer uses bubblecast to find a random set of so-called storage peers to hold his data. New storage peers also use bubblecast to offer their services to a random set of maintainers. Add, update, and delete requests are sent by the maintainer directly to the storage peers. A flush-and-reload mechanism guarantees that orphaned and inconsistent data is removed from the system eventually. Replica counts are automatically corrected to compensate against churn or to adapt to new bubble sizes.

5.4 Durable Bubbles

Durable bubbles provide long-term persistence for data without a dedicated maintainer. Durable bubbles can be created, updated, and deleted by any peer. A conflict resolution algorithm deals with concurrent updates from multiple users.

The responsibility for certain items is assigned based on the identifier of the node, similar to the approach of DHTs or semi-structured rendezvous search systems (see Section 2.9). As a handy side-effect, this enables very efficient key-value lookups for durable bubbles, like in a DHT. Every peer announces its responsibilities with a special managed bubble. The data from these bubbles is used to fill a key-based routing table at each peer. Through careful choice of parameters, every peer typically knows a few other nodes for each key, therefore forming a graph for each item, which can be used to efficiently and reliably distribute messages like update requests via flooding. Durable deletion of items is implemented with tombstones [122]. The data store for a durable

bubble type must support store, update, delete, and lookup requests. The details of the durable replication will be discussed in Chapter 7.

5.5 Match Constraints

A match is defined between a subject bubble type and an object bubble type. The subject is the bubble that triggers the rendezvous when it is received. The object is the bubble type against which the subject is matched. The received subject item is matched against all locally stored object items. Thus, the object type must be from a persistent class, since an instant bubble type does not have a data store to match against.

The matching is facilitated through an application-provided rendezvous function. In theory, the rendezvous function evaluates every pair of subject item and object item at the local node to find matches, but that would be rather inefficient in practice. Instead, the rendezvous function only receives the subject item and evaluates it against the local data store of the object type, typically using highly efficient index structures.

The effects of successful matches are defined within the rendezvous function. The application might want to return matching object items to the sender of the subject item (query/data model) or forward the subject item to the owner of the matching object item (publish/subscribe model), covering both active and passive search with the same mechanism (see Section 1.1.6). It is even possible to do both, issue messages to other parties, or trigger any other action required by the application. This concept enables BubbleStorm to preserve the full flexibility of the rendezvous search principle with a simple API. Convenience functionality for typical use-cases like query/data can be easily layered on top of this API.

Some search operations, like joins in relational databases, require matching more than two data types at the same time. In principle, BubbleStorm is capable of multi-edges in the matching graph, but this approach would not be scalable enough for most scenarios. Alternatively, database join algorithms like block nested loop join can be adapted to BubbleStorm's primitives. An in-depth discussion can be found in the paper on how to implement distributed SQL queries with BubbleStorm [80].

Since BubbleStorm is a probabilistic rendezvous search system, it gives probabilistic guarantees for search success. Those guarantees are tunable and controlled by the *lambda* parameter of the matching primitive. See Section 3.1 for details on how lambda works.

5.6 Comparison

BubbleStorm's data primitives cover a large area in the replication design space and support each of the four replication modes discussed in Section 4. Long-term and short-term data, active and passive search, node-specific and collectively-owned items, mutable and immutable data are all considered in a set of only five primitives. The richness of replication options puts BubbleStorm ahead of the related work in P2P search overlays.

The four bubble classes offer application developers an easy to understand and flexible tool set (see Table 5.1). Instant bubbles are the only non-persistent class and are

Bubble Class	Instant	Fading	Managed	Durable
Mechanism	bubblecast	bubblecast	maintained	collective
Discussed in	Section 3.3	Section 3.3	Chapter 6	Chapter 7
Store	no	yes	yes	yes
Update	no	no	yes	yes
Delete	no	no	yes	yes
Lookup	no	no	no	yes
Consistency	n/a	n/a	eventual	eventual
Implementation Complexity	low	low	medium	high
Maintenance Overhead	none	none	low	medium
Availability	n/a	short-term	high	high

Table 5.1.: Comparison of bubble classes

suited for queries and the like. Fading bubbles are similar but persistent and only useful for temporary data. Both classes have no replica maintenance overhead and are thus very efficient. By their nature, they cannot be updated or deleted (except by an expiration timeout). Managed and durable bubbles have higher overhead, because of the maintenance cost, but can be updated and deleted with eventual consistency. Managed bubbles are bound to the lifetime of their publisher and can only be modified by this node. Durable bubbles on the other hand are collectively managed and of permanent persistence.

5.7 Application Examples

For a better understanding of the data primitives, a few typical application examples and their respective matching graphs are discussed in the following. The graphical depiction of the matching graph consists of a circle for each bubble type. The intersections of bubble types illustrate match constraints between those types. The arrow points from the subject type to the object type. A self-match, in which the same bubble type is both subject and object, is depicted by a circular arrow.

5.7.1 Forum

An online forum is a structured discussion platform like the Usenet with its news groups. Nowadays, average users are most familiar with web-based forums. A forum typically consists of a hierarchical structure of (sub-)forums, each of them containing a number of conversation threads. Each thread consists of a number of individual postings. Those might be ordered linearly or in a tree-like fashion.

The example application (see Figure 5.1) uses two persistent bubble types: forums and posts. Both are durable, because users expect their postings to persist over time,

even when they leave the systems. Posts and forums also might need to be updated or deleted by an administrator or moderator. A post would consist of author, title, body, date, forum, thread ID, and position in the thread.

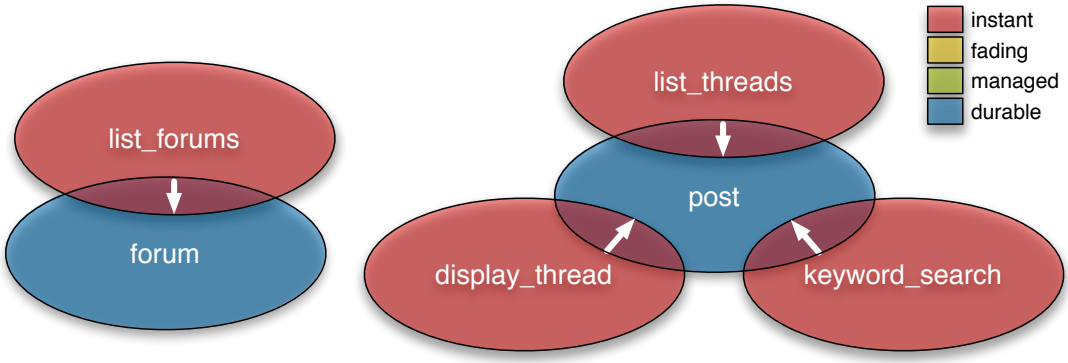


Figure 5.1.: Matching graph for a forum application

There are four instant types for search purposes. List_forums returns all forums in the system, so they can be displayed in the navigation section of the client application. In a very large forum structure, the search might be limited to a number of levels under a given root node to avoid unnecessary result traffic.

List_threads finds all posts in a given forum that have started a thread (i.e., they have a position of zero). This bubble is sent when a user clicks on a forum to display its contents. Display_thread finds all posts in a given thread (i.e., they match the given thread ID). This bubble is used when a thread is clicked by the user.

The last bubble type keyword_search illustrates the power of the rendezvous approach. It is used for full-text searches on all forum posts and its rendezvous function can be implemented easily by using an off-the-shelf keyword search engine like Apache Lucene [92] as the data store for the posts. An application developer could even easily provide advanced options like limiting the search to certain forums, authors, or date ranges.

5.7.2 Instant Messaging

In an instant messaging application users query central repositories to find the users in their contact list who are currently online. Any further communication like chat or voice messaging and monitoring of online contacts is normally done through direct point-to-point connections. A distributed version of the central contact repository can be implemented very easily with BubbleStorm.

A single bubble type called buddy_list is needed to publish the own online presence and to find the online contacts (see Figure 5.2). A buddy_list item contains the name of the publisher, its network address, and a list of all the user names the publisher has on its contact list. This bubble type is managed and self-matching. When a user comes online, it publishes its buddy_list. The rendezvous function matches the incoming contact list against the names of the owners of the locally stored buddy_lists. To ensure mutuality, the owner name of the incoming list also needs to be in the contact list of the matching

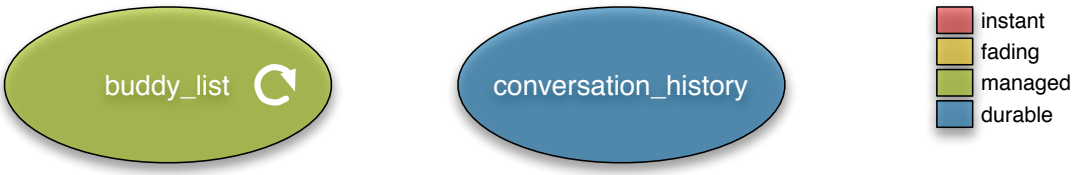


Figure 5.2.: Matching graph for an instant messaging application

buddy_list. On a match, the name and network address is forwarded to both nodes. When a node goes offline, it deletes its bubble automatically. If not, the replication mechanism takes care of garbage collection.

In the age of cloud computing, users expect their conversation histories to be stored in the network and automatically updated on a local device when it joins the system. This functionality can also be implemented with a single bubble type. The conversation_history bubble type is durable and stores all conversations of a certain user. For privacy reasons, such an item would typically be encrypted by the user. The durable class provides a key-value lookup mechanism, which can find a user's conversation history by using a hash of the user name as the item's identifier. Since the identifier of the item is well-known by the user, no separate search bubble is needed. The persistence guarantees of the durable replication ensure that the items are kept available, even when the user is offline for a longer period of time. This example shows that BubbleStorm can be easily used as a DHT with durable replication. Getting rid of separate structures for different search modes can reduce development complexity considerably.

5.7.3 MMOG

A central challenge in massively multiplayer online games (MMOGs) is interest management. Each player is interested in events occurring in a certain area around its avatar. In a P2P environment, a common solution is to establish direct connections between all nodes that are in each other's area of interest [126].

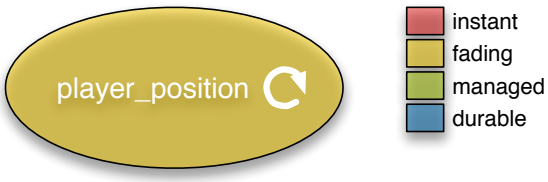


Figure 5.3.: Matching graph for an MMOG application

This can be implemented easily with BubbleStorm. Each player periodically publishes its own position and its area of interest, which is implicitly defined by the position. The player_position bubble type is self-matching (see Figure 5.3). Whenever a node receives a position update, it matches this position against the locally stored positions and notifies players that are within interest range. Since position bubbles are issued with a high frequency and are only of short-term interest, maintenance overhead should be

avoided. Therefore, fading bubbles with a few seconds timeout are a good choice. Instead of updating existing bubbles, every position update uses a new, short-lived bubble.

This communication model has already been successfully implemented for the Planet Pi4 online game with BubbleStorm as the rendezvous overlay [86].

5.7.4 File-Sharing

The classic application scenario for P2P is file-sharing, and search is a fundamental component of file-sharing: users need to find the content they want and the peers that offer this content. Surprisingly, the real-world file-sharing networks have not solved the problem of distributed search. The most popular services like the now defunct Napster [100] or the more up-to-date BitTorrent [25] heavily rely on central, well-known servers for metadata search and peer tracking. Other solutions like Gnutella [64] use simple flooding techniques for search and are therefore unable to achieve the same scalability and quality of service as their "cheating" competition.

With BubbleStorm, building a scalable and reliable search for file-sharing would not only be easy, but also could implement a superior set of features without needing central servers. The metadata description of files needed for search would be published with durable file_metadata bubbles. Any file in the system would have exactly one instance of this bubble, no matter how many peers hold a copy of the file. The identifier of the bubble could be computed from a hash of the file's content. This approach avoids duplicates in the search result list. Appropriate files to user queries are found with a find_file bubble, which is instant. The rendezvous function matching queries against metadata could be highly sophisticated, featuring range queries (e.g., file sizes, publishing dates), stemming and synonym support for keyword search (e.g., title, description), or even more advanced techniques like picture similarity search.

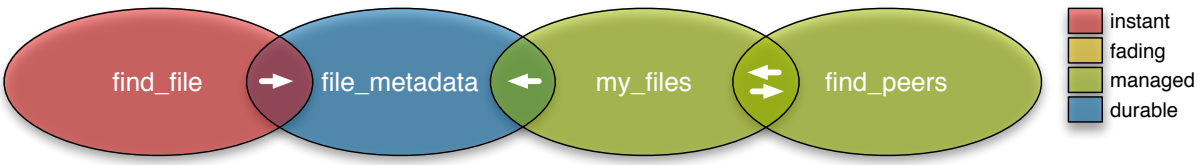


Figure 5.4.: Matching graph for a file-sharing application

The find_file search returns a list of matching file descriptions to the searching node. When the user has selected a file to download, a find_peers search with the file identifier is issued. Find_peers bubbles match my_files bubbles, which contain a list of file identifiers of files shared by a certain peer and the corresponding network address. The find_peers search returns the network addresses of peers that share the requested files. Those peers can then be contacted directly for downloads. Find_peers bubbles are managed and persistent. My_files also match find_peers, making it a mutual matching relationship. This means that an incoming my_files list also triggers stored find_peers requests. The find_peers request becomes a continuous query, which keeps updating its result list with new peers until it is deleted by the requester. That way, nodes get

informed about new download sources immediately and without the need for repeated polling.

My_files is also a managed bubble type, since it is bound to the node's lifetime. The file list is published when a node joins the network and can be updated when the shared files change. My_files matches file_metadata. When the list is published, metadata matching the file identifiers is returned to the publisher. The publisher can automatically detect metadata that is missing in the network or needs updating, because it is incomplete or conflicting with the local description. In an established network, a newly joined peer would not need to upload a huge metadata list, since almost all of the information is already in the network. Either it has been provided by other peers or by the node itself in a previous session. The file_metadata is of durable type. It can be edited by any peer that wants to improve the metadata and is not connected to any particular node.

A basic version of the BubbleStorm-based file-sharing search and tracking has already been implemented using tracker-less BitTorrent as the content distribution protocol [34, 147].

6 Maintainer-based Replication

The maintainer-based replication mechanism implements the replica management for managed bubble types in BubbleStorm (see Section 5.3). It follows the managed replication mode (see Section 4.3) with a natural owner—the publisher becomes the owner of the item. The algorithm has originally been published as a conference paper [81]. Since then, it has been extended and improved to reflect the concept of bubble types, save on routing state for multiple documents per publisher, and to react more robustly to environment changes. The algorithm has been designed with BubbleStorm in mind, but is fully compatible with other rendezvous search systems. Its basic principles can be applied to replication in any dynamic distributed system.

6.1 Model and Requirements

The node that publishes a certain item becomes the *maintainer* of this item. Replicas of the item are placed on a set of nodes, which are called the *storage peers* of the item. Every maintainer keeps a set of storage peers and selects from this set when a new item is published. The replicas should remain in the network as long as their maintainer is online. When the maintainer leaves or crashes, the items should disappear as quickly as possible. Orphaned replicas, which have not yet been purged from the network, are called *junk*. An item can only be changed or deleted by its maintainer.

The items are categorized into types. Each item of a certain type has the same desired replica count, and all items of this type share a common data store for storing the replicas at the storage peers. Those types correspond to the bubble types in BubbleStorm (see Section 5).

It is assumed that the underlying rendezvous search system provides certain functionality. First, an algorithm to find suitable storage peers is needed. Normally, the algorithm used to publish data items is suitable. In the case of BubbleStorm, bubblecast (see Section 3.3) is used for this purpose. Second, a service that regularly provides updated system statistics like network size, replication degrees, and total replica count is needed. BubbleStorm provides such a facility with its gossip protocol (see Section 3.5).

6.2 Overview

In this replication algorithm, each node serves both as a maintainer and a storage peer. A maintainer keeps a set of storage peers to store its publications. When a peer joins the network, it searches for an initial set of storage peers. Likewise, a node searches for maintainers it can offer its storage to, when it joins the network. A leaving maintainer deletes all stored items from its storage peers. A leaving storage peer silently removes the data it holds from the system. These operations ensure a stable density of replicas in the network in the face of churn.

Since a rendezvous search system requires replica counts proportional to the square-root of the network size, the maintainer needs to adjust the set of storage peers or the assignment of data to peers when the network size changes. Because the automatic load balancing can cause changes in bubble sizes even when the network size remains constant, similar adjustments are necessary in this case.

When a maintainer crashes without deleting its items from the network, junk is left behind. Instead of constantly checking the online status of its maintainers to determine the junk status of individual items, a storage peer simply deletes all items and searches for new maintainers when a heuristic overall junk threshold is reached.

The maintainer-based replication does not maintain exact replica counts per item, which would be impossible or at least overly expensive in a highly dynamic P2P system, but provides a binomial distribution of replica counts with the correct mean. This probabilistic guarantee still holds under extreme amounts of churn and is fully sufficient to support the probabilistic rendezvous search guarantees of BubbleStorm. The mathematical analysis of the algorithm, which proves that the system converges to a binomial distribution, can be found in Appendix A.

The storage operations between a maintainer and its storage nodes are implemented with point-to-point communication.

6.3 Maintainers

Every node that publishes at least one managed item becomes a maintainer. The maintainer keeps track of the storage peers that hold its items and adjusts the set of storage peers used when necessary. Since the maintainer knows the storage peers and their network addresses, storage operations can be issued with direct point-to-point connections. The maintainer node is the only peer that can modify the items it has published.

6.3.1 Joining the Overlay

Once a peer has successfully joined the BubbleStorm overlay, i.e., it has acquired topology neighbors and is able to issue bubblecasts, it searches for storage peers. Theoretically, this operation could be delayed until the first managed item is to be published, but this would significantly delay the publication of this item and it can be assumed that, when managed bubble types are present, sooner or later a managed item will be published. Especially in the case of collective replication (see Chapter 7), which uses managed items internally, the timely availability of storage peers is crucial.

Storage peers are found with a special bubblecast called find_storage. The size of this bubble type is the maximum of all managed bubble types registered, which ensures that the pool of storage peers acquired is big enough to accommodate every managed bubble type. This maximum relationship is supported by the bubble balancer (see Section 3.4).

Find_storage requests resemble fading bubble types, since they are not maintained, but are processed by a store function when they are received. In this function, the receiving peer becomes a storage peer for the requesting maintainer. It stores the requester in its list of maintainers and sends a response to the maintainer, which contains the necessary contact information to access its storage services.

The bubblecast algorithm implies a numbering of the receiving nodes (see Section 3.3), which is used by the maintainer-based replication to assign the items of different types to the stochastically correct number of storage peers. The replication algorithm naturally preserves a given replica density, which is the ratio of replicas to network size. Every storage peer retrieved by find_storage is assigned a *density position* p, which is the ratio of its position i in the bubblecast to the total network size n. The density d of a managed item is computed as the ratio of its bubble type's size b to network size n. A maintainer stores a managed item on all its storage peers with a density position equal to or smaller than the item's density (see Figure 6.1).

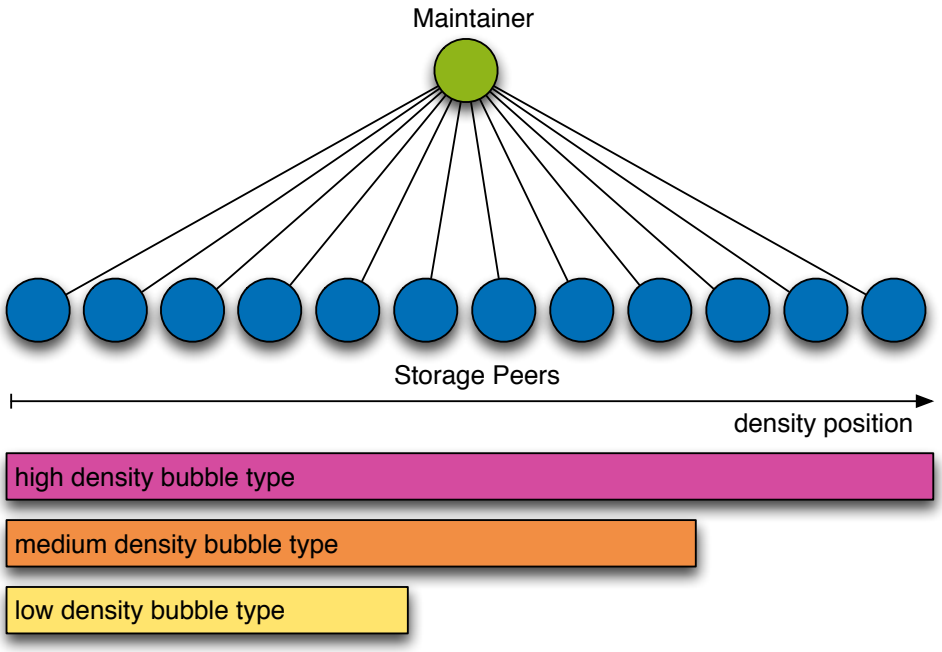


Figure 6.1.: The assignment of different bubble types to the storage pool

The internal calculations of the replication algorithm are based exclusively on density positions. A density position of a storage peer in a maintainer's pool does not change over time. The problem is that, at the time of a bubblecast, the actual network size is not known, because the gossip protocol has a measurement delay of up to two rounds (see Section 3.5). Therefore, the network size at the time of the bubblecast is only used as a first estimate. As a heuristic, especially in situations where the overlay grows quickly, a new density position is decreased accordingly, if a bigger network size is detected in one of the next two measurements that arrive.

6.3.2 Leaving the Overlay

When a maintainer leaves the network, it deletes all its managed items from the storage peers and does not accept requests from new nodes to become storage peers anymore. This approach follows the concept of managed replication: when the owner becomes unavailable, its items should disappear from the system.

If the maintainer crashes or becomes otherwise unable to communicate with its storage peers before (completely) deleting its items, orphaned data will be left behind on

the storage peers. This so-called junk accumulates in the system since storage peers never check the online status of their maintainers. Storage peers garbage-collect junk periodically to keep it below a configurable threshold (see Section 6.4.3).

6.3.3 Self-Adaptation

Changes of network size and bubble sizes make replication adjustments necessary. Since the maintainer-based replication approach sustains a constant replica density, the square-root-based replica counts of BubbleStorm will not be preserved without taking active measures upon size changes. Additionally, the bubble balancer (see Section 3.4) may change bubble sizes independently of network size to optimize bandwidth consumption.

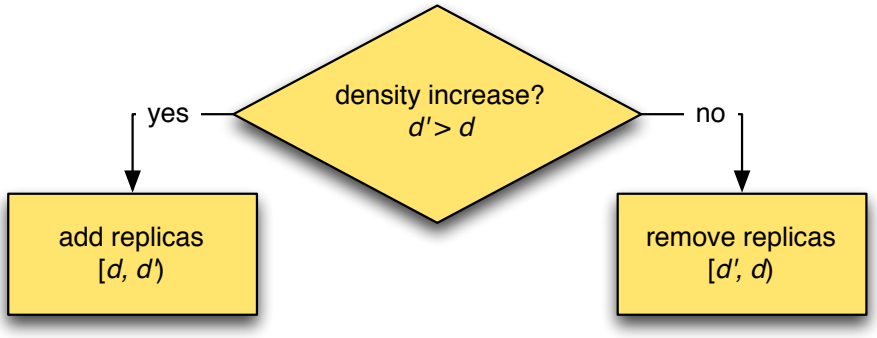


Figure 6.2.: Adjusting maintained bubbles on environment changes

Figure 6.2 explains how the maintainer-based replication achieves robustness despite these challenges. When the network size changes from n to n', and the bubble size changes from b to b', the target density of a bubble is adjusted from $d = b/n$ to $d' = b'/n'$. If the new bubble density d' is bigger than the previous density d, all storage peers in the interval $[d, d')$ are sent the maintainer's items of this bubble type. If the new bubble density d' is smaller than the previous density d, the maintainer's items of this bubble type are removed from all storage peers in the interval $[d', d)$. This approach sustains the invariant that a bubble type of a size b has an expected replica count of b.

If the highest bubble density is larger than it has been in the previous measurement round, additional storage peers need to be acquired. A bubblecast with the appropriate size is used to find those peers and assign them the correct density positions. If the largest bubble density is smaller than before, the unused peers can be removed from the maintainer's set of storage peers.

6.3.4 Storage Operations

A maintainer can execute the following operations on a storage peer: store, update, delete, and quit. A store request creates a new replica of an item on the storage peer. An update request updates the content of a previously stored item. A delete request removes an item from the storage peer. A quit request is used to notify the storage peer

that its services are no longer required, because densities have changed or because the maintainer is leaving the network. The quit request is only used internally and cannot be triggered directly by the user application.

When an operation is issued, not only all affected storage peers are contacted, but an internally kept local copy of the data is also updated. When a new storage peer is added to the storage pool later, the maintainer immediately provides it with the data from this local copy.

Every action, except quit, expects an acknowledgement. If no acknowledgment arrives within 60 seconds, the storage peer is considered offline and is removed from the storage pool. This reactive approach avoids keep-alive traffic at the cost of additional local state in the form of dead storage peer entries. Since bandwidth is typically scarce compared to local memory, this tradeoff appears to be the reasonable choice.

6.4 Storage Peers

Every node that is part of the search overlay offers its services as a storage peer. Nodes that are not willing or capable to contribute storage capacity to the system should not become peers in the overlay. Instead, they should become clients in a super node network and use the super nodes to access the search overlay. Otherwise, the bubblecast-based rendezvous search could not rely on a uniform probability distribution of items in the overlay and thus not provide any success guarantees.

6.4.1 Joining the Overlay

When a node has successfully joined the overlay, it starts a find_maintainers bubblecast. Find_maintainers has the same size as find_storage and is started in parallel. Nodes that receive this message add the new node to their storage pool and contact it to provide it with their managed items. The density position p of the storage peer is based on the position of the receiving maintainer in the find_maintainer bubblecast. The idea behind this inverse search is to find the maintainers that might have reached the storage peer with their find_storage requests, if the storage peer had been already online at the request time.

The expected density of existing items is preserved by this join operation. Maintainers are selected with uniform probability by the bubblecast and the added replicas exactly cancel the increased network size. In a network with “perfect” churn, where every leaving node is replaced by a new one, this approach would keep the size of storage pools balanced. Density and network size changes are handled by the maintainers as discussed in Section 6.3.3.

6.4.2 Leaving the Overlay

When a storage peer leaves the overlay, it simply deletes all items it stores. It is not necessary to notify the affected maintainers. Since maintainers are assigned their storage peers with uniform probability and a leaving storage peer also reduces the network size, the expected density of all items stays unchanged.

In the context of managed replication, it would not be helpful to keep stored items beyond sessions. If a storage peer brought back items from a previous session, multiple problems could occur. First of all, it is not unlikely that the maintainer of an item is not online anymore. Thus, the storage peer would bring junk into the system. Even if the maintainer is still online or comes online again, the storage peer and maintainer may be unable to locate each other, since their network addresses could have changed. This kind of junk will not only be orphaned, but will also miss updates from the maintainer, leading to inconsistent data in the overlay. Therefore, retrieving fresh data from the currently online maintainers upon join is a much safer approach from a consistency point of view.

6.4.3 Controlling Junk

When maintainers leave the network without being able to notify their storage peers, junk replicas will be left behind. This can happen when a maintainer crashes, loses power or network connectivity, or is unable to reach one or multiple of its storage peers due to network problems. Without countermeasures, this junk will stay in the system until the affected storage peers themselves leave and therefore delete the replicas they hold.

Junk not only unnecessarily uses up storage space, but since it is indistinguishable from valid data, it shows up in query results. This wastes bandwidth for the transfer to the requester and can also negatively affect user experience, if a large portion of the presented results is actually invalid.

A conventional eviction scheme might simply use keep-alive ping messages as a failure detection mechanism. Using a ping, a peer checks if a replica's maintainer is online and evicts the replica in case the ping does not return. In a P2P setting, however, such a failure detection mechanism is very unreliable. It would eventually bias the system towards preferentially storing replicas from either reliable or short-lived maintainers. The problem is that a transient network failure could cause the peer to incorrectly conclude that the maintainer is down. This false deduction will never be corrected, because once the replica is removed, neither party will place that replica onto the peer again. Temporary failures thus become permanent failures. For particularly long-lived peers, these errors will slowly accumulate, decreasing the replication degree and leading to a non-uniform replica distribution, which can negatively affect search success.

Instead, the maintainer-based replication algorithm does not try to identify individual junk replicas, but essentially simulates a leave-and-rejoin operation when appropriate. This is called the *flush and reload* approach. A storage peer deletes all stored items, searches for new maintainers with a find_maintainers request, receives their replicas, and does not accept any further storage operations from its old maintainers. With this approach, the issue of temporary failures becoming permanent is side-stepped, not by trying to make the system reliable, but by making the system forget. A transient failure could still cause the storage peer to miss a replica it should have stored, but on its next reload attempt it has a fresh chance to store it.

The main challenge is to find a trade-off between remaining junk in the system and the traffic overhead introduced by the reload operations. Fixed flush intervals are a bad

choice. If the number of maintainer crashes increases, the amount of junk can grow unbounded. Conversely, if no maintainers crash anymore, the reload traffic would be wasted. A reasonable solution needs to self-adapt automatically to environment changes to sustain the desired quality of service.

The maintainer-based replication provides a controllable probabilistic bound on the portion of junk in the system. Therefore, the *goodness factor* $g \in (0,1)$ is introduced. This parameter controls the desired percentage of valid replicas a peer stores on average. For instance, if $g = 0.8$, then on average 80% of replicas a peer stores are valid. When g is used as a system-wide parameter, then $1 - g$ is the expected percentage of junk responses to a query.

To determine the threshold for flushing, the system needs to know the sum of densities of all maintained items in the system $D = \sum_i d_i$, which equals the number of replicas each storage peer is expected to store. BubbleStorm's global value of D can be measured using the gossip protocol (see Section 3.5). Each maintainer provides the bubble size b_i for each item i published. This sum $B = \sum_i b_i$ is then divided by the network size n.

By comparing the number of replicas it actually stores with D, a storage peer can estimate the percentage of junk it has. To achieve a goodness g, a peer can tolerate up to $j = D(1/g - 1)$ junk on average. Since flushing reduces junk to zero, a peer should flush when it reaches $2j$ to get to an average of j. A naïve approach would now evict junk once the junk level exceeds the threshold $D + 2j$.

This is almost the approach taken, but there is a subtle condition for correctness. By reloading replicas after flushing, a replica of the item i with probability d is obtained. To keep the density unchanged, the peer must also have had a replica with probability d before flushing. This is only true if storage peer v's decision to flush is independent of the replicas it stores (*):

$$\begin{aligned} \mathbf{P}(i \text{ loses replica} \mid v \text{ flushes}) &= \mathbf{P}(v \text{ stores } i \mid v \text{ flushes}) \\ &\overset{*}{=} \mathbf{P}(v \text{ stores } i) = d \end{aligned}$$

Simply flushing when the threshold is reached is unsafe because peers with the most replicas flush first. So when v flushes, it is more likely to be storing a replica of i, and independence is lost.

For this reason, a two-bucket approach is used. Maintainers are randomly assigned either type #1 or #2 (e.g., using the last bit of a hash of their address). When a storage peer receives a replica from a maintainer of a certain type, it stores the replica in the respective bucket. When it has to delete a replica, it removes it from the corresponding bucket. This division makes it safe to use the traffic flow through bucket #1 in determining when to flush bucket #2 and vice versa. That is, when one bucket reaches the threshold, the other bucket is flushed.

Let r be the number of replicas in one bucket. The bucket contains the expected $D/2$ valid replicas plus the junk $j/2$, i.e., $r = D/2 + j/2$. The goodness factor requires that $g * r$, i.e., $g(D/2 + j/2)$ replicas are valid on average. As the bucket contains $D/2$ valid replicas, $D/2 = g(D/2 + j/2)$, which can be rewritten as $j/2 = D/2(1/g - 1)$. The threshold is reached when twice the desired junk has accumulated, i.e., j per bucket, so that the average will be correct. Thus, the threshold of an individual bucket is $r = D/2 + j = D/g - D/2$, i.e., half the threshold of a storage peer.

6.4.4 Flush Cost Analysis

Flushing is surprisingly cheap. Consider counting every item transfer, which seems a reasonable metric. Any system which preserves the replication degree must create new replicas when storage peers join the system, and thus transfer at least as many replicas as the maintainer-based replication. The same argument applies when the density is increased. However, flushing and reloading adds additional transfers to recover flushed valid replicas.

To measure this cost, consider a system where only necessary transfers occur. Initially, storage peer v pulls an average $D/2$ replicas into bucket #1 with find_maintainers. Thereafter, v receives x replicas from new maintainers performing find_storage. Unfortunately, v now decides to flush, wasting transfers. For simplicity, assume v again pulls $D/2$ replicas and receives x pushes before its next flush. Before the Fth flush, peer v has performed $F(D/2+x)$ transfers, when only $D/2+xF$ were needed. Letting F grow, the percentage of wasted transfers is,

$$\frac{F(D/2+x)}{D/2+xF} - 1 = \frac{D(F-1)}{D+2xF} \approx \frac{D}{2x}, \text{ amortized for large } F$$

To make this equation useful, x needs to be found. Suppose that for every item created, another item is on average deleted. This is the case for a network in equilibrium. Storage peer v saw x pushes, so it should also have seen x deletes. However, if c is the percentage of maintainers which crash (and thus don't delete their replicas properly), storage peer v's bucket will have $j = cx$ junk replicas when it flushes. When s flushes, $j = D(1/g - 1)$. Solve for x to find the overhead which is,

$$\text{percentage of wasted transfers} \approx \frac{D}{2x} = \frac{c}{2/g-2}$$

For example, consider $g = 0.8$, requiring 80% of replicas to be valid. With no crashes ($c = 0$), there is no overhead. With 10% crashes, the overhead is 20% on the longest lived peers. In the real world, the overhead will be even smaller, because most peers have a much smaller F than infinity.

6.5 Extensions

Beyond the basic functionality already explained, the maintainer-based replication offers a couple of features that can further improve its usefulness and versatility: eventual consistency enabled by maintainer serialization, exploiting heterogeneous peer capacities, and uniform bubblecast for correct maintainer selection in heterogeneous environments.

6.5.1 Eventual Consistency

Given that each valid item has a single live maintainer, updates can be easily controlled by the maintainer. Deletion is considered a special case of an update. Although in

most applications, the maintainer will be the only node updating its items, others can be allowed to update by simply sending their update requests to the maintainer. By tagging each object with its maintainer's address, the maintainer of an object can be found easily.

As maintainers already keep a list storing a superset of their storage peers, this storage pool can be used for update management too. Whenever an update occurs, the maintainer sends the new version of the item (or the changed delta) to all peers in its storage pool. In principle, this updates all replicas in the system.

However, recall that maintainers remove non-responsive peers from their storage pool. If a transient network failure occurs, then the maintainer might incorrectly remove a peer from the storage pool and later fail to update it. This leaves temporarily inconsistent replicas in the network, a problem that cannot sensibly solved in an open-membership P2P overlay, since it would need to sacrifice either availability or partition tolerance [17, 46]. However, all peers eventually either leave the system or flush. Thus, inconsistent replicas will eventually be removed, guaranteeing eventual consistency. If a time limit on the potential inconsistency window is required, a maximum time between flushes could be set.

6.5.2 Heterogeneous Peer Capacities

Usually not all peers have the same capacity. Thus, storage peers should have the choice to only provide as much service as their capacities allow. In BubbleStorm, peers in the overlay can control their relative load. To support this, every peer v picks a capacity ℓ_v. If one peer has twice the capacity of another, it can store approximately twice as many replicas (and serve requests on these replicas). In BubbleStorm, the capacity ℓ_v determines the degree $d(v)$ of v in the topology. Since bubblecast follows every topology edge with uniform distribution, the load induced by instant and fading bubbles is proportional to the degree (see Section 3.3).

To be compatible with this approach, the maintainer-based replication requires relatively few changes. The network size D_0 and the sum over all capacities, $D_1 = \sum_{v \in V} \ell_v$, are already monitored by the gossip protocol (see Section 3.5). A peer v's relative capacity is thus $h = \ell_v D_0 / D_1$. When all peers have the same capacity, bubblecast chooses each peer with the same probability $1/n$. In a heterogeneous setting, in contrast, it chooses a peer v with relative capacity h with a probability of h/n. Therefore, the result set of a find_storage request will pick nodes with a probability proportional to their capacity, which in turn leads to the desired capacity-proportional load distribution.

However, the find_maintainers algorithm has to be adjusted. A joining storage peer must pull replicas with a bubble size scaled by h to receive the correct load.

Furthermore, maintainers must be selected uniformly at random. Therefore, bubblecasts for find_maintainers must not use the heterogeneous node degrees, but instead select nodes uniformly at random as in a homogeneous environment. Otherwise, the maintainers would get storage pool sizes depending on their capacity. This would lead to non-uniform bubble sizes for the same bubble type depending on the publisher, breaking the success guarantees of BubbleStorm. Bubblecasts with uniform selection probability in heterogeneous topologies are explained in the next section.

Finally, the flush threshold must substitute rh/g for r/g to reflect the individual capacity of a storage peer.

6.5.3 Uniform Bubblecast in Heterogeneous Topologies

Bubblecast selects nodes with a probability proportional to their degree. In a homogeneous topology where every node has the same degree, nodes are selected with uniform probability. Since nodes in BubbleStorm should select their degree proportional to their capacity, bubblecast selects nodes proportional to their capacity in a heterogeneous environment. For query and data replication, this is the desired and intended behaviour, because it enables a more efficient operation [140, 143]. In special situations, however, where nodes should be selected uniformly, e.g., when assigning maintainers to newly joined storage peers, the heterogeneity of the topology must be compensated.

There are multiple solutions to this problem. Ferreira et al. use a Metropolis-Hastings algorithm [5] for their random walk rendezvous search system [36]. Unfortunately, this algorithm is only applicable to random walks, but not to tree-like distribution schemes like bubblecast. The limited reliability and linear response times of random walks make them unsuitable for a large-scale open-membership P2P overlay.

The solution used for bubblecast is based on response probabilities. A node receiving a *uniform bubblecast* answers it with a probability proportional to its capacity. A node with the minimum capacity h_{min} answers the request with 100% likelihood. Nodes with a capacity $h > h_{min}$ answer with probability h_{min}/h. As the minimum capacity ℓ_{min} is a globally-known system parameter (in BubbleStorm it is the minimum degree), $h_{min} = \ell_{min} D_0/D_1$ can be calculated by any node.

Since a node receives a bubblecast message with a probability h/n proportional to its capacity, the combined probability of a node being selected and answering a bubblecast message is h_{min}/n. Not answering a bubblecast in this context means, that the receiver completely omits any local actions normally triggered by this type of message. Nonetheless, the bubblecast is split and forwarded normally in any case, including the removal of the head entry of the target interval (see Section 3.3).

In order to keep the expected number of answers b^* to a bubblecast correct, the bubble size b (i.e., the number of nodes selected) must be adjusted to reflect the reduced answer probabilities. The expected number of answers per node selected is $\sum_{i=1}^{n} \frac{h_{min}}{n} = h_{min}$. Therefore, $b = b^*/h_{min}$ to receive the correct number of messages. This algorithm is independent of the maintainer-based replication, and will be re-used for a similar problem in the collective replication (see Chapter 7).

7 Collective Replication

The collective replication mechanism implements the replica management for durable bubble types in BubbleStorm (see Section 5.4). It follows the durable replication mode (see Section 4.4) and thus implements replication for items that are not bound to any owner and should persist even when the participants of the overlay change completely. The collective replication has been designed with BubbleStorm in mind, but like the maintainer-based replication, it can be used for any unstructured rendezvous search system. Its principles may be useful to replication mechanisms for large-scale distributed systems in general.

The mechanism builds upon the infrastructure of managed bubbles provided by the maintainer-based replication (see Chapter 6) and introduces a responsibility concept to unstructured rendezvous search systems. It also allows for inexpensive key-value lookups with very good response times for items replicated with durable bubbles. Thus, it introduces a powerful feature to unstructured search overlays, which previously has been exclusive to structured systems like DHTs. The key-value lookups operate on the same replicas used for the usual rendezvous-based search, making redundant overlay structures for key-value lookups and complex search obsolete.

7.1 Model and Requirements

In the durable replication mode, an item is not bound to any particular node. An update-in-place model, which is favorable due to the storage requirements, suggests that a set of nodes is assigned responsibility for storing a certain durable item. These nodes are called the *responsibility set* of this item. All operations on the durable item should be routed to those responsible nodes. The responsibility set can change over time due to node churn. The state transfer to the newly joined nodes must be self-organized, since there is no dedicated authority for a durable item. In contrast to managed replication, in durable replication a node may keep stored replicas across sessions and bring them back into the system, because the data is not bound to an owner.

A system, in which data can be updated by multiple independent entities, needs support for concurrent updates with useful consistency guarantees, especially if the data is replicated widely like in a rendezvous search system. A replication mechanism for durable data is required to support insert, update, and delete operations concurrently from multiple nodes. The CAP theorem [17, 46] implies that only a weak consistency model like eventual consistency can be supported in a large-scale open-membership P2P overlay (see Sections 1.1.1 and 1.1.8).

It is assumed that the underlying rendezvous search system provides certain functionality. The collective replication requires existing mechanisms for instant replication (see Section 4.1) and managed replication (see Section 4.3). BubbleStorm provides those in the form of bubblecast (see Section 3.3) and the maintainer-based replication (see

Chapter 6). If the underlying rendezvous search system does not provide a managed replication mechanism, fading replication (see Section 4.2) with periodic re-publication could be used for environments with low and predictable churn.

In order to assign a node to a responsibility set, it needs some sort of permanent identifier, the node ID, which should stay unchanged even across sessions. Otherwise, the ability to bring replicas back into the system would be impaired. It is also assumed that each item has an immanent identifier, e.g., a hash of its name, which allows the system to assign it to a responsibility set. This identifier is called the item ID.

7.2 Overview

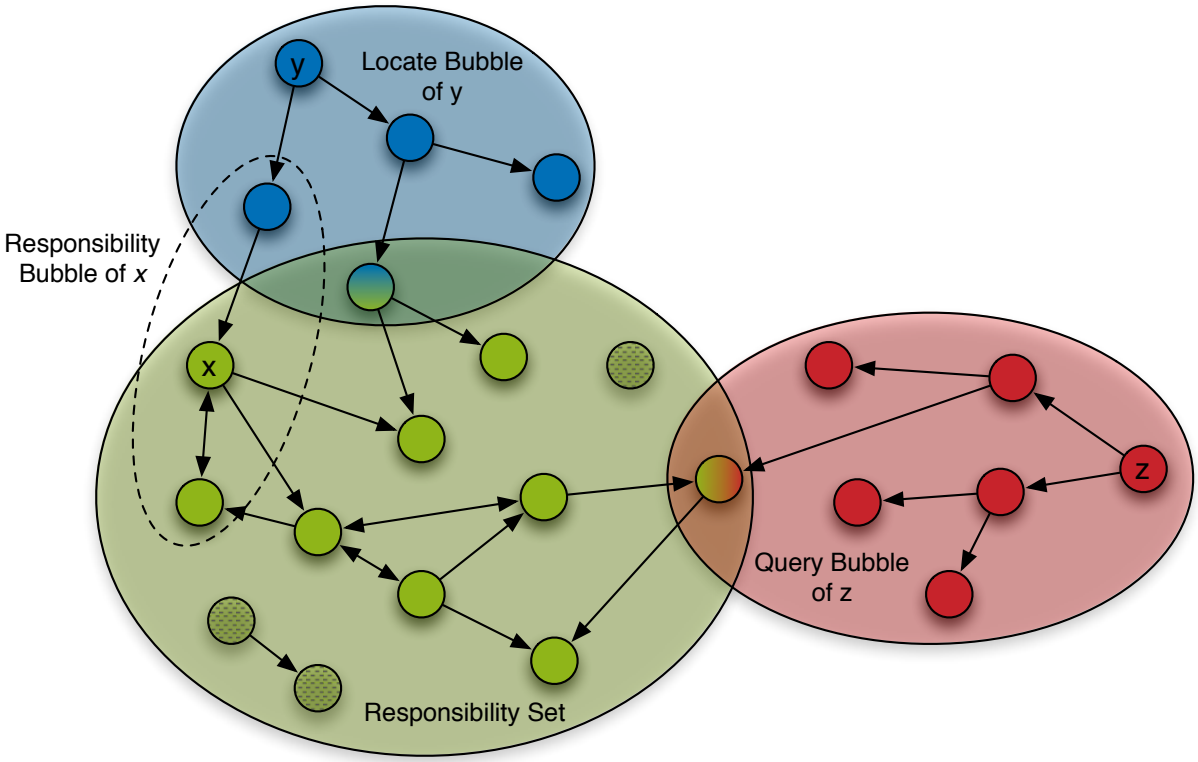


Figure 7.1.: Overview of the collective replication mechanism

In the collective replication, every node becomes part of a number of randomly assigned responsibility sets. Every node publishes a managed bubble, which contains its node ID (necessary to compute its responsibilities) and its contact information. This bubble is called the *responsibility bubble* (see Figure 7.1).

The nodes that receive the responsibility bubble, use this information to build a local routing table. They also provide their contact information to the sender to enable a bidirectional connection. The size of the responsibility bubble assures that the combined routing tables of a small set of nodes are sufficient to find one or multiple nodes in a given responsibility set with high probability.

When a node wants to contact the responsibility set of a given item, it issues a small *locate bubble*, which checks the routing tables of the receiving nodes for appropriate

contacts. If such entries are found, the request is forwarded directly to these nodes, which are called the *entry points*. A key-value lookup would simply be a locate bubble, which would return the replicas from the *entry points* directly to the requester.

When the item is to be inserted, updated, or deleted, the receiving responsible nodes not only process the request locally, but forward it to other responsible nodes they have in their own routing tables. The responsibility set is flooded with the request until all nodes reachable from the entry points have received the request. To support concurrent updates, each update is identified by a unique Lamport timestamp [73].

As the nodes of a responsibility set are randomly selected, a collectively maintained bubble type can be used for rendezvous-based search with any other bubble type, e.g., an instant query bubble type.

The routing tables create a meta-topology for each responsibility set, which is called the *responsibility graph* (the green bubble in Figure 7.1). The main challenge in this approach is to find the correct sizes for responsibility bubbles and locate bubbles to enable a high probability of success without unnecessary overhead. The responsibility bubbles have to assert that the routing tables create responsibility graphs in which most nodes are connected to each other (i.e., have a *giant component*), but without introducing too much redundancy, which would make flooding inefficient. A giant component connects a constant fraction of the vertices of a random graph, which is much larger than all other connected components of the graph. The locate bubbles on the other hand must assure that an entry point within the giant component is found.

The introduction of a key-based routing scheme on top of a random topology classifies the collective replication as a semi-structured solution for the rendezvous search problem, similar to the SplitQuest system (see Section 2.9). Additionally, the collective replication retains the symmetric matching capabilities of BubbleStorm, allowing a durable bubble type to have match constraint with itself. Furthermore, replication and updates are covered, and key-value lookups are added.

7.3 Responsibility

Every node that receives a responsibility bubble adds the information to its *routing table* for the collective replication. It also contacts the sending node, which can add the receiver to its routing table, enabling a bidirectional connection. The routing table is used to forward requests to insert, update, delete, or look up a durable bubble to the responsible nodes. The routing scheme is a one-hop algorithm. A receiving node forwards the request to all responsible nodes in its routing table and discards it when no responsible node is available. The node that sent the request is excluded from the routing step. The routing table must enable a node to determine which of the contained nodes is responsible for a given item. Assigning these responsibilities in a robust way is a key component to the replication mechanism.

Responsibility for item IDs can be assigned in many different ways. DHTs use different metrics like predecessor [136], numerical distance [117], or XOR [91], among others. These metrics compare the node ID with the item ID and assign responsibility to nodes with the minimal value, because a unique responsibility is required. Alternatively, any node under a given threshold could be used, if a set of responsible nodes is desired,

which is more in line with the requirements of a rendezvous search system. For the collective replication, the threshold is the responsibility probability. The classic metrics could be used for the collective replication, but the one-hop routing opens up additional design freedom to be exploited.

The downside of the discussed metrics is that nodes with similar node IDs share a large portion of responsibilities. Therefore, correlated node failures increase the chance of a range of item IDs becoming unavailable. Additionally, some item ID assignment schemes can lead to a biased ID distribution, which would result in hot-spots at the nodes close-by.

In the collective replication, responsibility is assigned with a pseudo-random algorithm. The assignment function takes node ID and item ID as parameters and returns a uniformly random value for the metric. If this random value is below the responsibility probability, the node is responsible. The probably most simple and fastest random function is the linear congruential method [69]. It multiplies an input value with a random seed and takes it modulo the output range. The random seed has to be odd. Otherwise, only even results would be generated.

The first 64 bits of the item ID are used as the input value and the the first 64 bits of the node ID as the random seed, adding 1 to make it odd, if necessary. Linear congruential generators are no high quality random number generators, especially when the seed is not carefully chosen. But since the responsibilities need to be calculated when requests are routed, they offer a good trade-off of computation speed and randomness in scenario of responsibility computation. Compared with a classic metric, the overlap of responsibilities between two nodes with similar node IDs is much smaller, which reduces the risk of losing multiple items on correlated node failures. In an environment where computation cost is critical or resilience is not an issue, any of the classic metrics could also be used.

7.4 Bubble Sizes

In order to achieve the desired replication degree for durable bubbles, the responsible set and the responsibility bubbles must have the correct sizes and the locate bubble must be big enough to find the giant component reliably. Fortunately, a sharp estimate on the volume of the giant component of random graphs with given degree sequence exists [23], which applies directly to the given problem. The lower bound given by Chung and Lu is summarized in the following.

In their random graph model, each node has an expected degree d. Edges are placed randomly between nodes, but with a probability proportional to the expected degree of the nodes. The expected volume for a subset S of vertices, $Vol(S)$, is defined as the sum of the expected degrees.

$$Vol(S) = \sum_{v_i \in S} d_i$$

Chung and Lu provide a lower bound on the volume of the giant component and an upper bound on the size of the second largest component.

Theorem 3. *Suppose that G is a random graph with n vertices and an expected average degree $\tilde{d}$. When $\tilde{d} \geq \frac{4}{e}$, almost surely the giant component of G has volume at least*

$$Vol(GCC) \geq cVol(G), where$$
$$c = \left(\frac{1}{2}(1+\sqrt{1-\frac{4}{\tilde{d}e}})+o(1)\right)$$

and the second largest component almost surely has size at most

$$1+o(1)\frac{\log n}{1+\log\tilde{d}-\log 4}$$

Theorem 3 proves that a random graph with a sufficiently large expected average degree is dominated by a giant connected component containing almost all nodes. In the context of the collective replication, the random graph is the responsibility graph, and its giant component is the durable bubble. Therefore, the size of the responsibility set r should be the durable bubble size $b_{durable}$ divided by the *connectivity factor* c from Theorem 3 to compensate for the nodes isolated from the giant component.

$$r = \frac{b_{durable}}{c}$$

Thus, a given node is responsible for a given item of bubble size $b_{durable}$ with the *responsibility probability*,

$$p = \frac{r}{D_0} = \frac{b_{durable}}{cD_0}.$$

The expected average degree $\tilde{d}$ of the responsibility graph depends on the size of the responsibility set r and the size b_{resp} of the responsibility bubbles. The number of peers a node knows for each responsibility set is the responsibility probability times the number of entries in its routing table, because each of the nodes it knows may be responsible for that item. Since every node publishes a responsibility bubble of size b_{resp} and each node receives them with uniform probability and then provides the sender with its responsibility information, the expected number of entries in the routing table is $2b_{resp}$.

It is to be noted that Theorem 3 only applies to undirected random graphs. The responsibility bubble only establishes a directed random graph. The undirected graph is created by the receiver contacting the sender, establishing a bidirectional edge. Nonetheless, preliminary simulations suggest that a directed graph may be sufficient. The current implementation nevertheless ensures bidirectional connections to be consistent with the mathematical theory.

The bubble size problem can be re-formulated as a rendezvous problem (see Theorem 1 in Section 3.1): r nodes are chosen randomly to be responsible for a given item and $2b_{resp}$ random nodes are known locally at each peer. Therefore, $\mathbf{P}(M=0) \leq e^{-\lambda}$, the probability of not having a responsible peer in the local routing table, whenever,

$$g(\lambda/n) \leq g(r/n)g(2b_{resp}/n)$$

As proven in [140], the distribution of how many matches are found is Poisson in the limit with a rate of λ. In practice, that means an expected number of λ responsible nodes are found in a routing table. This simplifies the determination of responsibility set and responsibility bubble size to a normal match constraint between these two bubble types with parameter $\lambda = \tilde{d}/2$ (see Section 5.5).

This expected average degree $\tilde{d}$ determines the connectivity of the responsibility graph and thus the size of the giant component (see Theorem 3). The connectivity factor c has to be high enough to ensure that the majority of all responsible nodes receives an update. For BubbleStorm, $\tilde{d} = 2.0$, resulting in a connectivity factor of $\approx 75\%$. Thus, typically a vast majority of at least $3/4$ of all responsible nodes receive an update, and it is highly unlikely to have a noteworthy population of any outdated version. Additionally, the communication overhead for flooding the responsibility graph is limited to two times the number of recipients, which is considered a sensible trade-off.

The last value to determine is the *locate bubble* size b_{locate}. It has to be big enough to find at least one entry point to the giant component with high probability. The probability that a peer has no responsible peer in its routing table is $e^{-\tilde{d}}$, and the probability that a responsible node is not in the giant component is $1-c$. Thus, a node is unable to provide an entry point to the giant component with probability $(1-c)e^{-\tilde{d}}$. The failure probability of a locate bubble of size b_{locate} is

$$\mathbf{P}(\text{fail to locate}) = \left((1-c)e^{-\tilde{d}}\right)^{b_{locate}}$$

With $\tilde{d} = 2.0$, a locate bubble size of 3 already provides a surprising >99.99% success probability, which seems more than sufficient for most use cases. It is to be noted that the locate bubble size is independent of the network size, enabling BubbleStorm to lookup durable bubbles with $O(1)$ cost and latency.

This performance advantage over the usual $O(\log n)$ of DHTs is enabled by a routing table size of $O(\sqrt{n})$, the size of the responsibility bubble, whereas most DHTs only pay $O(\log n)$. If only key-value lookup is desired, the log-based trade-off may be more appropriate for some scenarios. In the case of BubbleStorm, high-performance lookups are a nice side-effect of the collective replication, which can be exploited for application design.

7.5 Joining and Leaving the Network

When a node joins the network, it publishes its responsibility bubble containing the node identifier and the necessary contact information needed to send requests directly to the node as a managed bubble. When a node receives a new responsibility bubble, it offers the new neighbor all locally stored items the neighbor is responsible for. The neighbor is thus able to download all data it should store.

When a node leaves the network, its managed responsibility bubble is deleted automatically (or garbage-collected in the case of crashes) and hence purged from the routing tables. The node can keep the locally stored data items for use in a future session. Upon re-joining the network, it can determine which items should be deleted,

because it is no longer responsible for their storage, and is informed by the receivers of its responsibility bubble, which inserts and updates it has missed.

It is to be noted that a joining or leaving node does not affect the responsibilities of existing nodes directly, which is unlike traditional DHT algorithms, where the interdependencies can be the source of inconsistencies in an unreliable environment. Instead, only globally available network statistics from the gossip protocol are used, which is a much safer choice for an Internet-based P2P overlay with high churn and transient network failures.

7.6 Self-Adaptation

Overlay and workload dynamics can lead to changing network size and bubble sizes. BubbleStorm's gossip protocol periodically measures the relevant overlay characteristics and re-calculates the system parameters. In case of the collective replication, the responsibility of a node and the size of the responsibility bubbles may be affected.

The responsibility bubbles are managed and automatically adjusted to the new size by the maintainer-based replication (see Section 6.3.3). This may result in the deletion or insertion of routing table entries. The locally stored items of shared responsibility are offered to new neighbors as discussed in the previous section.

Changes in the responsibility of a node can have two consequences: losing responsibility for locally stored items and gaining responsibility for items not yet stored locally. If a node finds that it has lost responsibility, it simply deletes the local copy. Gaining responsibility is not detected by the node itself, but by the nodes that have received its responsibility bubble. If the density of a bubble type increases, a node checks the responsibility of all locally stored items of that type with all routing table entries and compares it with the previous responsibility. If a neighbor has become responsible for a certain item, it is offered to this neighbor. All existing items that a node has become responsible for are offered with the same probability as receiving the initial insert or update request, because the required responsibility graph properties are unaffected.

7.7 Key-Value Lookups

Any operation on an item of a durable type requires to locate the responsibility set. This is achieved by issuing a locate bubble for the item ID. With high probability, the locate bubble finds at least one node in the giant component of the responsibility set. A key-value lookup can be easily implemented by forwarding the request to the found nodes. Each of those nodes retrieves the locally stored item of the requested ID and returns it to the requester. The lookup request is not flooded in the responsibility graph, because the responses of the entry points are already sufficient.

As the locate bubble usually finds multiple responsible peers, and these may not have the same content for a given ID, a requester may get different versions of the requested item. The requester can filter the responses to use the highest version, which is the item that will remain when system-wide consistency is reached eventually.

By using the infrastructure of the responsibility bubbles with its $O(\sqrt{n})$ state, the locate bubbles can be used to enable $O(1)$ key-value lookups in BubbleStorm, because the

size of the locate bubbles is constant. The lookups operate on the normal durable data replicated in the system, which can be used to facilitate rendezvous search operations, and therefore does not require additional replicas or parallel structures for key-value lookups and advanced search.

7.8 Inserts and Updates

To store an item into a durable bubble, the requester uses a locate bubble to find entry points to the responsibility set, which then recursively forward the request to all responsible peers reachable. Because of the giant component of the responsibility set, which is found by almost all locate bubbles, the majority of responsible nodes will receive the request. If the rare event occurs that the giant component is not found, only a very small amount or even none of the responsible nodes will receive the update. A requester checks whether its request has been executed by issuing an independent key-value lookup some time after the store request. A lookup can be made independent by sending the locate bubble to a different set of nodes.

Unlike managed bubbles, there is no unique maintainer for a bubble, which can serialize store requests. Therefore, concurrent updates can happen and need to be resolved. When two nodes issue store requests for the same bubble simultaneously, both may have flooded some fraction of the responsibility graph before their requests meet. If the concurrency was ignored, both versions would retain a noteworthy fraction of the responsibility set.

In a large open-membership distributed system, it is practically impossible to achieve a total ordering of the requests. Instead, the system must rely on a partial ordering scheme like Lamport clocks [73] or vector clocks [37, 90]. For a network of x participants, vector clocks require a state of x entries, which needs to be transferred with requests. As the system contains $O(\sqrt{n})$ responsible nodes in each durable bubble, vector clocks limit scalability significantly by increasing the bandwidth cost from $O(\sqrt{n})$ to $O(n)$. Therefore, Lamport clocks are used for the collective replication.

Each request is tagged with a version number and the requester's node ID. The version number starts at 0 for a newly created item and is increased by 1 for each update. Inserting an item simply denotes a store request with version 0, whereas an update is a store request with a higher version number. To avoid version number overflow, the version field is 64 bits wide. A requester that wants to update an item, needs to retrieve the current version number via lookup before the update. Often, the update depends on the content of the current version anyway.

When a node receives a store request for an already existing item, it compares the local version number τ_l with the one of the request τ_r. If $\tau_l < \tau_r$, the update is accepted and overwrites the local version. The request is then forwarded to other responsible nodes. If $\tau_l > \tau_r$, the update is discarded and is not forwarded. If $\tau_l = \tau_r$, the requester IDs are compared analogously. If both attributes are identical, the request has already been received by this node, and thus is ignored, which stops the flooding of the responsibility graph.

Some P2P systems use mechanisms similar to two-phase-commit [51] or three-phase-commit [130] for consistent updates, e.g., [3, 97, 98]. Even though they can provide

strong consistency guarantees, such algorithms have the disadvantage that responsible peers leaving or crashing during an update cause an abort after a timeout, thus hurting availability. In a highly dynamic environment with many responsible peers, this will lead to many aborts and unacceptable availability. Therefore, it is necessary to trade consistency for availability for the unreliable large-scale systems targeted here.

The approach of the collective replication limits the expected amount of inconsistency in the network state, but tolerates inconsistent results (see previous section). The node churn leads to constant change of the responsibility graph, which helps propagating the newest version to previously disconnected peers. If no further updates happen, all responsible peers will eventually converge to the same version due to replica exchange of the self-adaptation mechanism.

7.9 Deletes

Deleting a collectively replicated item can be intricate. If a responsible node receiving a delete request completely purged the local state of the item, future encounters with nodes that missed the request would lead to a re-replication of the deleted item. The population of the almost extinct item will completely regenerate eventually, obliterating the delete request. Since it is to be expected that individual replicas that survive a delete, countermeasures must be taken.

The traditional method is to replace replicas with tombstones or death certificates instead of deleting them completely [122]. Delete is implemented as an update request, which removes all content of the item. This retains the version information of a deleted item. Surviving replicas of previous versions will be replaced by this tombstone through replication graph changes, preventing the old versions of slowly flooding the graph.

7.10 Heterogeneity

Like all other replication modes in BubbleStorm, the collective replication should exploit the heterogeneity of node capacities. This can be achieved by making the responsibility probability proportional to the node capacities,

$$p_v = \frac{b_{durable}\ell_v}{cD_1}$$

Responsibility bubbles additionally include the node's capacity ℓ_v to enable the receivers to calculate the adjusted responsibility.

The responsibility bubbles must be used with a homogeneous version of the managed replication. Otherwise, high capacity nodes will have large routing tables *and* increased responsibility leading to a disproportionately high routing load. Since the heterogeneous collective replication will be deployed together with a heterogeneous topology, a slightly modified version of the maintainer-based replication must be used. In order to ensure uniform routing table sizes, the find_storage requests of the maintainer-based replication for responsibility bubbles use uniform bubblecast (see Section 6.5.3), just like the find_maintainer requests.

8 Methodology

The evaluation of large-scale distributed systems is non-trivial and potentially error-prone. Network researchers have employed different evaluation methods, each of them with its own strengths and weaknesses. Gross and Güneş [52] name mathematical analysis, measurements, and computer simulation as the common evaluation techniques. Heckmann [57] additionally mentions practical experimentation. These four areas can be further divided into analysis, numerical simulation, message-based simulation, packet-level simulation, prototype experiments, emulation, real-world measurements, and benchmarking (see Figure 8.1).

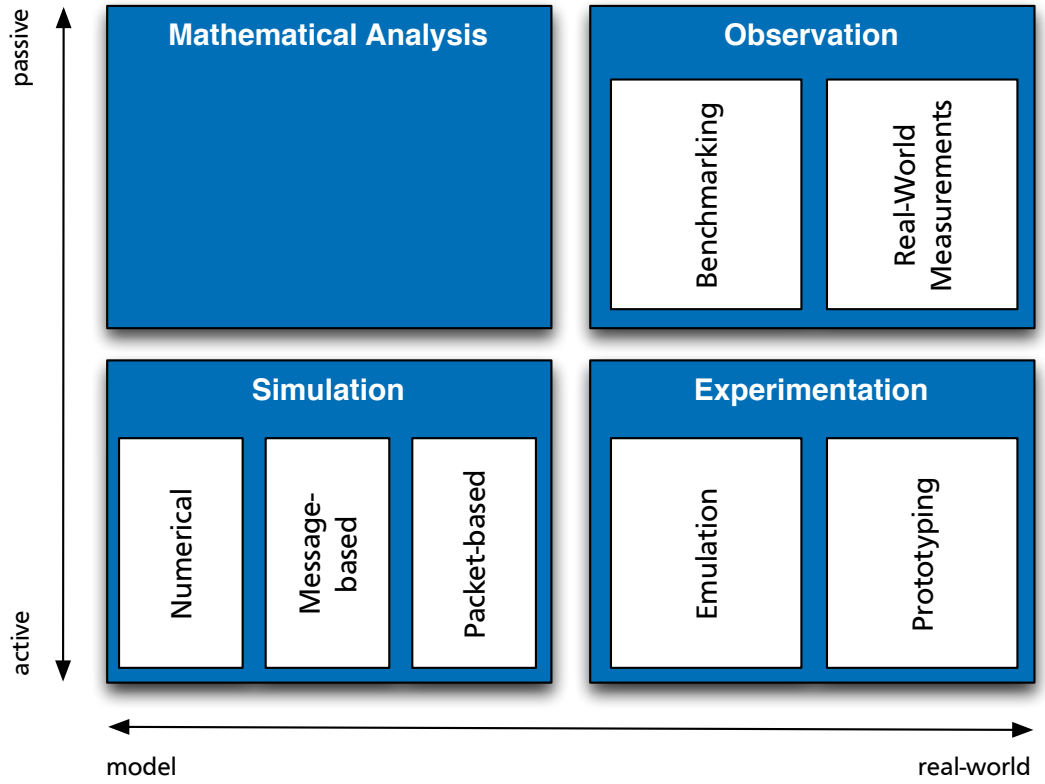


Figure 8.1.: Networking research methods (based on [57])

A perfect evaluation technique would be repeatable and would provide high performance even for large network sizes, while still modeling the environment realistically. Furthermore, it should enforce a correct isolation between the simulated entities, to prevent unrealistic side-channels that can lead to wrong results. Ultimately, an evaluation method should support fair and representative comparisons between different solutions for the same problem. Unfortunately, no single method fulfills all these requirements and thus cannot produce universally valid results. Therefore, it is best to combine multiple methods to double-check the results.

In the following, the strengths and weaknesses of different methods are discussed. In this work, a combination of analysis, different simulation techniques, and prototyping assures the validity of results. The simulator and prototyping environment used for this work is discussed in Section 9 and [79].

8.1 Analysis

Mathematical analysis uses a mathematical model to describe the system behavior by a set of equations. Normally, only an abstract representation of the system can be modeled mathematically, since the complexity of such models increases rapidly. This limits the understanding of the system behavior, but still provides fundamental insights like asymptotic behavior. Since the computational complexity of the mathematical model is typically independent of the parameters, very large networks can be evaluated as easily as small ones. The big O notation allows a very coarse grained comparison of the performance and cost of different algorithms.

8.2 Numerical Simulation

Computer simulations can be applied at many different levels of abstraction and with different techniques. *Discrete-event simulation* is the most common technique in the context of computer networks [52]. In this kind of simulation every state change is modeled as an event, and events are executed in the order of their schedule time. Sometimes *round-based simulation* is also used, but is only applicable to the more abstract simulation models, since relatively coarse rounds instead of fine-grained timestamps are used to model state changes.

The most abstract simulations for computer networks can be described as numerical simulations. A very simple model is used to simulate the fundamental system behavior. Neither the real-world limitations of computer networks like message delays, bandwidth limits, transient failures nor technical details like communication protocols are taken into account for the simulation. Instead of detailed message exchanges, the state of remote nodes is accessed and modified directly and often instantly during state changes. Even though the realism of such simulation results is relatively low, numerical simulation is highly useful as a proof-of-concept in an early design phase. The high level of abstraction can easily affect correctness, as the simulator works with a global view which provides no isolation. Since the computational complexity is very low, these simulations can easily scale up to millions of nodes. Numerical simulations are typically custom-built for a specific experiment. Their level of abstraction and the lack of standardization makes meaningful comparisons almost impossible.

8.3 Message-Based Simulation

Most P2P researchers use a more detailed form of simulation, which is called message-based. The system is modeled as a set of nodes, which can communicate by exchanging

messages. All state changes should be local to the affected node, modifications to remote nodes are only triggered by sending messages. Similarly, remote information should only be accessed through message exchanges, never directly.

Much emphasis in message-based P2P simulators is put on churn models, message delays, and workloads, but the network is only modeled as end-to-end connections. If at all, the underlying network protocols like TCP, UDP, IP, Ethernet, etc. are not simulated realistically.

This level of abstraction provides good performance of up to many thousands of nodes and can give rough estimates on service latency or bandwidth usage. Even though the approach seems to isolate nodes against each other, a small inattention, like accessing a remote node's internal data structures, might lead to side-channels between nodes that can invalidate the simulation results. Therefore, simulation experiments have to be implemented carefully to ensure correctness. There are both custom message-based simulators for specific experiments and generic simulator frameworks like PlanetSim [44], PeerFactSim.KOM [135], PeerSim [93], and ProtoPeer [40]. Comparisons between algorithms can be done using one of those simulators, but are limited by the (typically rather low) realism of the underlay and workload models used.

8.4 Packet-Level Simulation

Packet-level simulators provide maximal realism for network simulations. They simulate transport, routing, and even physical layers with high accuracy. This limits not only their capacity to a few hundred or thousand nodes, but also adds a lot of complexity to the simulator implementation. Due to their complexity, only a few well-known community projects like ns-3 [60] and OMNeT++ [152] and no relevant custom simulators exist. Oversim [7] provides a P2P-specific environment for OMNeT++. The configuration of an experiment for such a simulator is typically more complex than for more abstract simulators. The realism of the simulation results depends on a proper configuration and workload that models the real world accurately. If done properly, the results are much more precise than the measurements generated by a message-based simulation.

8.5 Prototyping

The ultimate proof-of-concept is to build the system and to run it on a real network. But implementing a sophisticated P2P system can be time-consuming by its own. Evaluating a prototype on the Internet is even more cumbersome. It involves the deployment of the prototype on a large number of hosts, which optimally are distributed globally. Most research projects do not have the resources to build and maintain such a testbed, but research initiatives exist that try to provide the necessary infrastructure to the research community. An example of such a global testbed is PlanetLab [131], that provides shared access to more than a thousand computers, which are made available by the participating research organizations. The shared access to machines and the cross traffic on the public Internet connections between the hosts can have a significant influence on the experimental results. Since these circumstances are beyond the control of the researcher that conducts the experiment, the repeatability of such real-world experiments

is limited. This also limits the ability to compare different prototypes. Nonetheless, a prototype that is deployed on a real network cannot break entity isolation with unrealistic side-channels.

8.6 Emulation

Emulation is an alternative to real-world experiments. A prototype is deployed in a controlled lab environment that emulates the real world, e.g., by adding failures, delay, message loss, or bandwidth constraints. The realism of the emulated environment determines how close to real-world measurements an emulation experiment can get. An emulator can be understood as a simulator that executes prototypes instead of more-or-less abstract models. It offers the repeatability of simulators combined with the isolation and realism of prototypes. This provides a good starting point for comparisons, but still requires realistic workload and user behavior models to yield significant results. Executing a large number of prototype applications requires a large amount of computing power and heavily limits performance. Dedicated hardware testbeds like Emulab [63] provide the required environment, but are not widely available. The SliceTime project [155] combines the evaluation of one or a few prototypes with a large number of simulated nodes run by the ns-3 simulator. Unfortunately, this requires both a prototype and a simulator implementation.

8.7 Real-World Measurements

If the system prototype has been released to the public or is otherwise deployed operationally, it can be measured in the real world. This provides maximum realism, but can be hard to measure since the machines are typically beyond the control of the researcher. Nonetheless, such measurement studies have provided important insight into P2P systems in the past [54, 124]. Real-world measurements can only be conducted on real systems. Establishing a real-world community for a system prototype is beyond the scope of a typical research project and a complex endeavor by itself. Therefore, real-world measurements are most often used to derive models of user behavior from existing systems as a parameter to simulator experiments of new systems or to improve the understanding of system requirements. If two systems for the same purpose can be measured in the wild, this can serve as the basis for a very well-founded comparison.

8.8 Benchmarking

A benchmark is a set of standard tests to evaluate the relative performance of software and/or hardware components. Industry-standard benchmarks are widespread and well-known, e.g., for microprocessors [62], graphic cards [129], databases [82], and middleware [120, 121]. The development of a benchmark involves the definition of a representative workload for the benchmark scenario, which can be difficult to determine even for experts in the field. Benchmarks are traditionally conducted on real hard- and software that is under the complete control of the benchmark conductor.

This makes benchmarking incompatible with practically all other networking research methods. Running a P2P benchmark only on fully-controlled machines is not feasible in practice. Current research tries to apply the benchmarking approach to simulation and emulation experiments in P2P networking [76].

8.9 Review

Table 8.1 provides a comprehensive overview of the strengths and weaknesses of the different evaluation methods. None of them is sufficient alone. A method can provide either repeatability and performance or realistic environment models and isolation, but never both. Thus, a sound evaluation must use a multi-method approach to cover all desirable features. To maximize the benefit, the different methods need to be conducted independently to ensure that an error in one experiment does not impact other experiments using different methods.

Representative comparisons are best achieved with benchmarks or real-world measurements, but both tools are not applicable to most research projects. Either there is no recognized benchmark for the given scenario or it cannot be scaled to the typical P2P network sizes. Real-world measurements can only be done for applications deployed in the real world, which is rarely the case for research prototypes. Therefore, useful comparisons of P2P applications are currently a challenging problem.

Method	Repeatability	Performance	Comparability	Realistic environment	Entity isolation
Analysis	++	++	o	–	–
Numerical simulation	++	++	–	–	–
Message-based simulation	++	+	–	-	-
Packet-based simulation	++	o	-	o	o
Prototyping	-	-	-	+	++
Emulation	+	-	-	+	++
Real-world measurements	-	+	+	++	++
Benchmarking	o	–	++	+	++

Table 8.1.: Comparison of networking research methods

A promising approach is to combine existing methods to cover new positions in the design space. The evaluation method for this work, which is described in Chapter 9, combines prototyping with message-based or packet-based simulation. This multi-method approach aims to maximize the validity of the presented results.

9 Simulation and Prototyping Environment

The simulation and prototyping environment developed for this work is based on extensive experience gathered in a wide range of P2P research projects, ranging from message-based Gnutella simulations [87], custom-built message/packet simulations of BubbleStorm with up to one million nodes [143, 144], numerical simulations of the maintainer-based replication [81], and benchmarking concepts for networked virtual environments [71] to prototyping P2P wikis [99] and P2P gaming overlays [74, 75].

The combined knowledge led to the design of a novel evaluation framework for message/packet-based simulations, prototyping, and real-world experiments, generating all three modes from identical source code for the application under test. This approach combines the advantages of simulation and experimentation while avoiding most of their pitfalls. Together with the mathematical analysis presented in Chapters 6 and 7 and Appendix A, three of the four methods of distributed systems evaluation (see Chapter 8) are covered. Only observation is beyond the reach of this work, since real-world measurements can only be conducted with already deployed systems, and standardized benchmarks do not exist for P2P replication and updates.

An early version of the evaluation framework has been published [79], as well as the concepts of the packet delay model [66] and the session model [106, 107]. The same framework has been used to evaluate the CUSP transport protocol [146]. The system has recently been extended [11] to allow the execution of the experiments configured for the simulator in a real-world testbed like PlanetLab [131] or G-Lab [128].

9.1 Overview

The key to the evaluation framework design is the definition of a narrow system interface, which completely abstracts away the underlying runtime environment (see Figure 9.1). Applications building exclusively upon this interface can be executed in the simulator engine, in a distributed real-world testbed, and even as interactive standalone applications. Currently, a message-based overlay simulator and a real-world engine for standalone execution implement the system interface. An experimental implementation using the ns-3 packet-based simulator [60] exists, proving the portability of the system interface.

On top of the simple UDP interface provided by the system, a complete reference implementation of CUSP provides a full-fledged transport protocol for both simulation and real-world execution. This design decision unburdens the simulation engines from a lot of implementation overhead and guarantees an identical transport protocol behavior across all runtime engines. The combination of a message-based overlay simulator and a full transport protocol implementation places the simulation mode between the classic

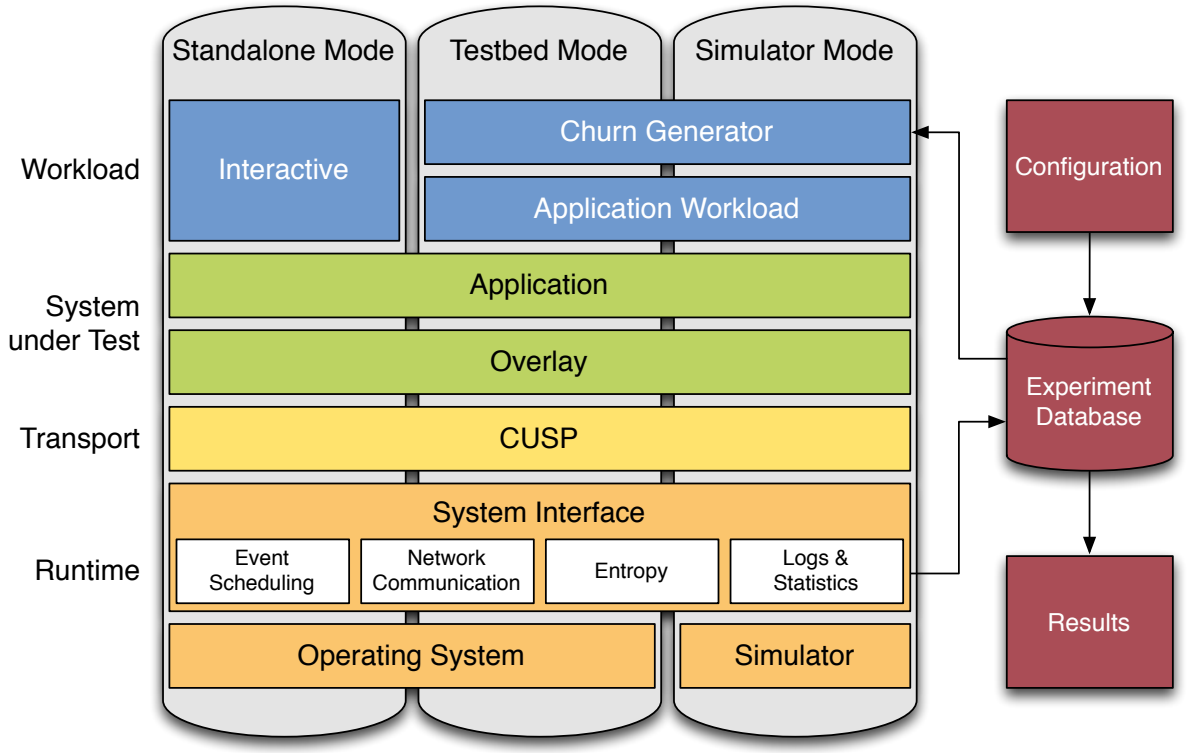


Figure 9.1.: The evaluation framework architecture

message-based and packet-based simulators. The user-space CUSP implementation also allows deploying BubbleStorm and other applications developed with the framework on operating systems unaware of the CUSP protocol.

Automated experiments for the simulator and the testbed mode are configured in a SQLite database [101]. In this experiment database, the nodes, their networking capabilities, the churn model, and the workload parameters are defined. The runtime engine reads this configuration upon startup and initializes the experiment accordingly. The experiment definition can contain correlated node events like catastrophic failures or simultaneous join events as well as notification events to application-specific workload generators. These events are implemented using standard POSIX signals.

The experiment database is also used to store the experiment output, simplifying the inspection of the configuration that produced the output. The output contains the data that was written to the log and statistics module of the system interface. The configurable log of all nodes in the experiment is very valuable for post-mortem debugging of the application. Step debugging of distributed environments is normally not possible, and thus a log is probably the best tool to trace problems, which often involve multiple nodes and a considerable timespan. The statistics contain aggregate state and performance information of individual nodes and the whole experiment. These statistics can be plotted directly from the experiment database with Gnuplot.

Using the same application code with different runtimes does not only eliminate redundant development effort, but additionally avoids individual shortcomings of each of the approaches. On the one hand, a simulated application can be tested and debugged much easier than a standalone prototype of a distributed application. A simulation

also offers perfect isolation from real-world anomalies, which are hard to separate from application code misbehavior. A real-world experiment is by its nature never fully reproducible. On the other hand, a simulation might abstract away important mechanisms of the underlying communication infrastructure or timing constraints that can significantly affect the application behavior. Furthermore, the lack of strict node isolation (see Chapter 8) may lead to communication side-channels in a simulation, which are impossible in a real-world deployment. By testing the application with both runtimes, it is possible to rigorously evaluate the correctness and performance of the system under test. The opportunity to build interactive prototype applications offers additional insight into the usefulness and relevancy of the developed algorithms in practice. Using the framework, a tracker-less BitTorrent client [34, 147] and a P2P gaming overlay [86] were built on top of BubbleStorm, among others.

The framework is implemented in the functional programming language Standard ML, using the whole-program optimizing MLton compiler [154]. For some performance-critical portions of the code, optimized C and assembler code is used. The rest of the chapter discusses the components of the evaluation environment in detail, starting with the system interface, which is the core of the framework, and then continuing from top to bottom in the component stack.

9.2 System Interface

The system interface, which decouples the evaluated system from the particular runtime environment, consists of four major components: scheduling, communication, entropy, and logging. These interfaces are implemented by each runtime. To minimize the implementation effort for the runtimes, the interfaces are narrow, but include all common services needed for the development of sophisticated network applications. Future revisions of the system interface may add a component for persistent data storage, like file system access and/or database connections. These have been unnecessary for the range of application prototypes implemented so far.

9.2.1 Event Scheduling

Applications in the evaluation framework are implemented in a pure event-based mode. Every action executed by the application is encapsulated into an event. Applications never actively sleep or wait, but instead schedule a continuation event, if an action needs to wait for whatever reason. Events can also be triggered externally, i.e., by incoming messages from the network.

The main advantage of this approach is the compatibility with discrete-event simulators. Event-based application code can be almost trivially mapped to the events of the simulator engine. The downside of the approach is the lack of explicit multi-threading, which is acceptable for dedicated network applications and even the gaming prototype [75]. Additionally, single-threaded applications are typically easier to implement and thus less error-prone than their multi-threaded equivalents.

The system interface provides the necessary facilities to create, schedule, and cancel events, and features a virtual system clock. Additionally, the application can register

handler functions for POSIX signals, which can be used to notify an application that it should shut down (SIG_INT) or to trigger application-specific events (SIG_USR1 and SIG_USR2). This interface provides a very portable method of controlling the application's behavior without the need of additional infrastructure.

9.2.2 Network Communication

The network communication API is essentially an asynchronous UDP interface. UDP sockets can be created, and UDP datagrams can be sent and received. UDP is a very simple transport protocol, which can be easily modeled by a simulator runtime. Accurately modeling a more sophisticated transport protocol like TCP would be much more challenging and thus would extremely increase the runtime implementation costs. With the CUSP implementation, a complete transport protocol is available inside the evaluation framework, offering a comprehensive feature set to the application developer. CUSP internally uses the UDP system interface for network communication.

The network communication interface furthermore includes an opaque network address structure. This address can be implemented as IPv4 or IPv6 addresses, allowing a smooth transition path between address types.

9.2.3 Entropy

The entropy interface provides random numbers to the application for cryptographic or stochastic operations and is used to seed the application's internal random number generators. In the real-network, it is implemented using operating system means such as /dev/random or /dev/urandom. In a simulation, it provides pseudo-random numbers from the simulator's random number generator and thus ensures repeatability of the experiments.

9.2.4 Logs and Statistics

The experiment output is passed to the log and statistics interfaces. The log interface can write short messages to a logging facility. Log entries are tagged with a severity level and a module name. Internally, the logging facility adds a timestamp and the node identifier. The typical logging backend is the experiment database. Logs in the database can be filtered by the tags through simple SQL statements.

The statistics interface can monitor performance metrics and system parameters measured by the application. Each statistic is accumulated over a configurable time span (typically one second) and then written to the logging backend. Each entry includes count, average, minimum, maximum, and standard deviation and is stored in a format suitable for aggregation.

9.3 Experiment Configuration

The composition and course of events of an experiment is defined in the experiment database. The same database layout can be used for the simulator and the testbed runtime. An experiment consists of node groups, which have configurable sizes, capabilities, and churn behavior.

The size of a node group consists of a fixed part and a proportional part. The fixed size is an absolute node count and the proportional size is the fraction of the total experiment size. The proportional size is calculated after the fixed sizes of all groups have been subtracted from the experiment size. Both sizes are added together to derive the total size of the node group. This approach allows the experiment conductor to easily scale the experiment size up and down without affecting the size of special node groups, e.g., a group consisting of a single bootstrap server, which initializes the overlay.

The capabilities of a node include bandwidth, buffer sizes, and location, among others. The churn behavior defines the session time distribution and special churn events, which trigger large-scale network size changes.

Every node group is assigned an application name and command line arguments, exactly like the the program would be called from the command line. The arguments can be used to configure the application and the application-specific workload generation. The argument parser of the churn generator supports a number of variables, which allow the experiment conductor to tell nodes their own IP and port, and the addresses of other node groups, which is necessary to discover bootstrap nodes and to join the overlay.

A detailed description of the experiment database layout can be found in [11].

9.4 Experiment Output

The runtime converts the information received through the log and statistics interface into the configured output format. Currently, the framework supports two formats: console and database. The console format is designed for standalone applications with very selective log filters, but can also be redirected into a text file for post-processing. The database format is designed for large-scale experiments and uses the experiment database for storage. Statistical data can be plotted with Gnuplot directly from the database (see Figure 9.2).

9.5 Churn Generation

The major difference of P2P systems compared to traditional distributed systems is the high level of node churn, which has to be considered in the experiment setup. Unlike application-specific workloads like search behavior, the churn generation is a generic model, which can be applied to any application under test. It is also impossible for an application to start itself. Therefore, the churn generation is implemented as part of the framework.

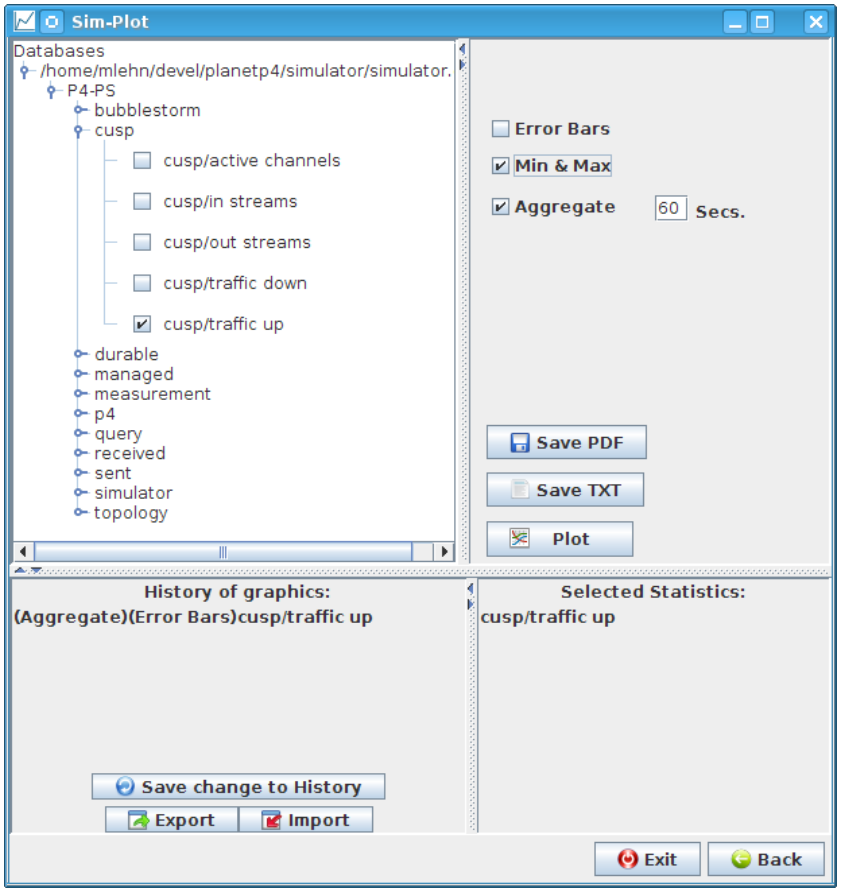


Figure 9.2.: Browser for selecting statistics from the database to be plotted with Gnuplot

Nodes can either orderly leave the network, e.g., by notifying their neighbors in the overlay or transferring their state or responsibility to a replacement node, which takes some time, or crash immediately without any cleanup.

9.5.1 Session Model

The session model used in the framework applies a session/intersession model [106, 107], which does not destroy nodes that leave the network when their session ends. Instead, the nodes are kept in an inactive state during the intersession time and re-join the system when their next session begins.

In addition to the background churn, which keeps the network in a steady state with approximately the same number of nodes joining and leaving in a given time interval, the experiment conductor may want to change the network size by slowly or abruptly adding or removing nodes from the network. Such large-scale changes are useful to simulate catastrophic events that test the system's elasticity and fault-tolerance. It is important that these events do not affect the background churn behavior, because otherwise unexpected side-effects may distort the steady state after the event.

The session model defines three attributes of a node's session state (see Figure 9.3):

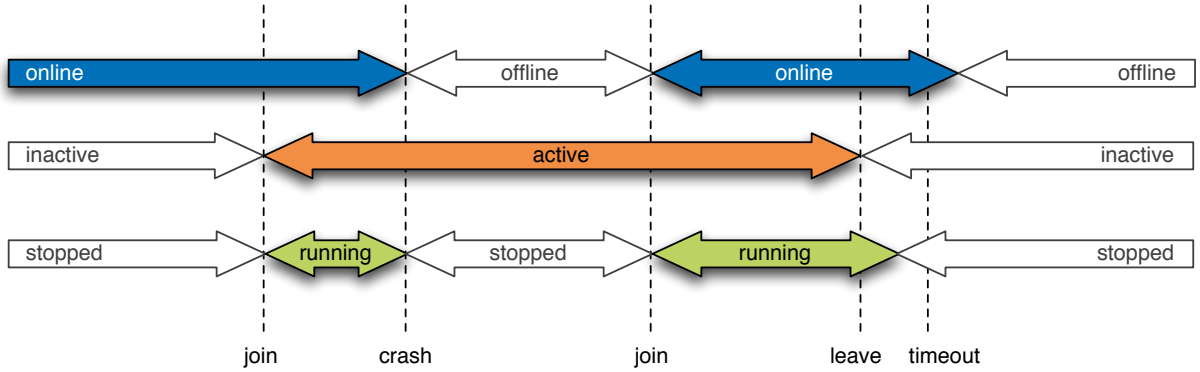


Figure 9.3.: The session model of the churn generator

- **online/offline:** This attribute is controlled by the background churn. When a session starts, it goes online. When the session ends, it goes offline. The churn behavior configured for the node calculates the session and intersession times. The churn behavior also defines a crash probability. Nodes that do not crash, execute the leave procedure of the application. The initial state of a node is picked randomly from the online/offline cycle of the used session/intersession duration distribution.
- **active/inactive:** An active node is considered available and may join the network. an inactive node does not participate in the network, even if it is online. The experiment conductor can define churn events that set a configurable fraction of the nodes active or inactive, either simultaneously or over time. These churn events are used to define catastrophic events, which are independent of the online state. If active/inactive and online/offline were combined into a single state, the nodes added by a massive join event could have a different mean residual lifetime as the existing nodes, which would have side-effects on the churn behavior in the subsequent stages of the experiment. Churn events can be defined independently for each node group and can trigger join, leave, and crash behavior.
- **running/stopped:** Only a node that has become both online and active joins the network. Such a node is called running. When a node crashes, it is set to stopped immediately. When it leaves, it is granted a five minute timeout to execute the leave procedure. If it is still running after the timeout, it is crashed by the churn generator.

The relationship of the session attributes can be summarized as follows:

$$\text{running} = (\text{online} \wedge \text{active}) \vee \text{shutting down}$$

9.5.2 POSIX Signals

While joining simply means starting a node's main function, leaving and crashing are more subtle. The framework design goal to be runtime agnostic and to only support

required functionality through a minimal system interface, contradicts the introduction of a specialized leave/crash interface, which would be hard to implement for the standalone mode.

Instead, the standard POSIX signals are used to trigger the node behavior. SIG_INT signals the node to leave. The application-specific leave procedure is implemented as a SIG_INT handler. SIG_KILL crashes the node immediately and cannot be handled by the application.

Additionally, the user signals SIG_USR1 and SIG_USR2 can be handled by the application to trigger custom behavior, like starting or stopping the workload. Sending user signals to node groups at specific points in time can be configured in the experiment database, similar to churn events.

More detailed workload generation models are very application-specific and beyond the scope of a general evaluation framework. Such a workload generator can be implemented in the application space, even coordinating multiple nodes, as described in the next chapter.

9.6 CUSP Implementation

The reference implementation of CUSP is implemented on top of the system interface of the evaluation framework. Thus, the same code basis can be used for validation in the simulator and real-world applications with the standalone mode. Additionally, the implementation overhead of the transport layer has been removed from both the runtime engines and the application code. This simplifies the runtime implementation and allows for a narrow system interface, yet still provides a full-fledged transport protocol to application developers. CUSP can be considered the thin waist of the evaluation framework.

9.7 Simulator Mode

The default simulator runtime is a message-based overlay simulator, which calculates end-to-end delays with a network coordinate delay model. The entropy interface is implemented with a Mersenne Twister random number generator [89]. The current implementation of the simulator engine is single-threaded, but an experimental prototype exists, which uses multiple processes; each being responsible for a set of statically assigned nodes. Only messages between nodes of different processes and synchronization messages are exchanged between the worker processes. This design does not only allow the use of multiple cores in a single computer, but the deployment of a simulation on a cluster of multiple computers.

9.7.1 Network Model

The coordinate-based delay model of the simulator runtime is based on the algorithm already presented in [66]. The mechanism uses real-world measurement data from the CAIDA [18] and PingER [149] projects, which employ long-term measurements

to derive base delays between host pairs and packet loss and jitter statistics between regions. The base delays are used to position the hosts in a low-dimensional Euclidean coordinate space. Nodes in the experiment are assigned to hosts from the measurement set and use their delay statistics. The experiment database allows the definition of the region of the hosts to be considered for selection for each group of experiment nodes. The model is more realistic than most comparable solutions [66], especially topology generators, yet it uses very little memory (a set of coordinates per node).

The original approach extracts jitter and loss data from the PingER measurements and then uses a stateless random loss and jitter calculation based on the expected values. This method is unrealistic, since the random jitter causes an extreme amount of packet reordering. Since the CUSP protocol, which runs atop the delay model, uses the TCP fast retransmit algorithm [61], any packet that is more than three packets late is considered to be lost by the congestion control. Therefore, the excessive jitter leads to a highly degraded congestion window and a correspondingly reduced throughput. The jitter model is thus excluded from the simulator, since it would heavily distort the application performance.

In addition to the base delay, which reflects the time a packet travels through the Internet backbone, a last hop delay is added for sender and receiver, which accounts for the significant delay that is added by certain access types like traditional ADSL or WiFi. The last hop delay is configured in the network capabilities of the node.

If the upstream of a node is busy, when a packet is scheduled to send, it is buffered in an output buffer for later transmission. If the buffer is too full to take the packet, it is dropped. Similarly, an input buffer exists, which stores incoming packets that cannot be downloaded yet. This design simulates the potential bandwidth bottleneck at the last hop. The bandwidths and the buffer sizes of upstream and downstream are part of the network capabilities.

9.8 Standalone Mode

As an alternative to the simulator runtime, the application can be compiled in the standalone mode, which uses the operating system services to implement the system interface. In this mode, the application can be executed as a native binary. Linux, Windows, and Mac OS X are supported as standard target platforms, other UNIX-compatible operating systems and processor architectures are supported by the compiler, but have not been tested. A standalone application can be used for interactive applications or for execution in the testbed environment.

9.9 Testbed Mode

The testbed mode shares the runtime with the standalone mode. A standalone application can be automatically deployed to a network evaluation testbed like PlanetLab [131] or G-Lab [128] and instrumented by the testbed churn generator. A central master script, started by the experiment conductor, connects to the available testbed hosts configured in the experiment database via SSH and copies all required executables and

configuration data to the selected machines. The master reads the experiment configuration from the database and assigns a number of experiment nodes to each testbed host.

A local management process on the testbed servers starts and stops the application nodes according to the churn model and sends them signals, if required. After the experiment run, the individual experiment databases of the application nodes are collected and aggregated into one result database. This result database has the same schema as the simulator database and can be used with exactly the same plotting tools.

9.10 Example Applications

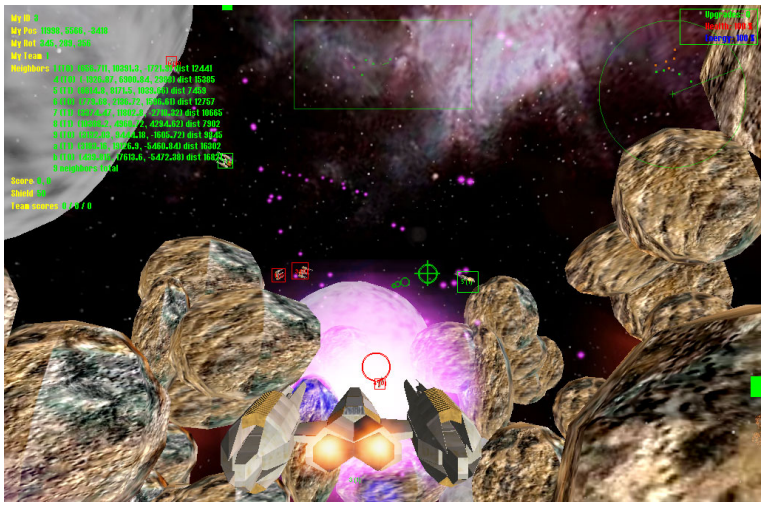


Figure 9.4.: The Planet Pi4 Massively Multiplayer Online Game [75]

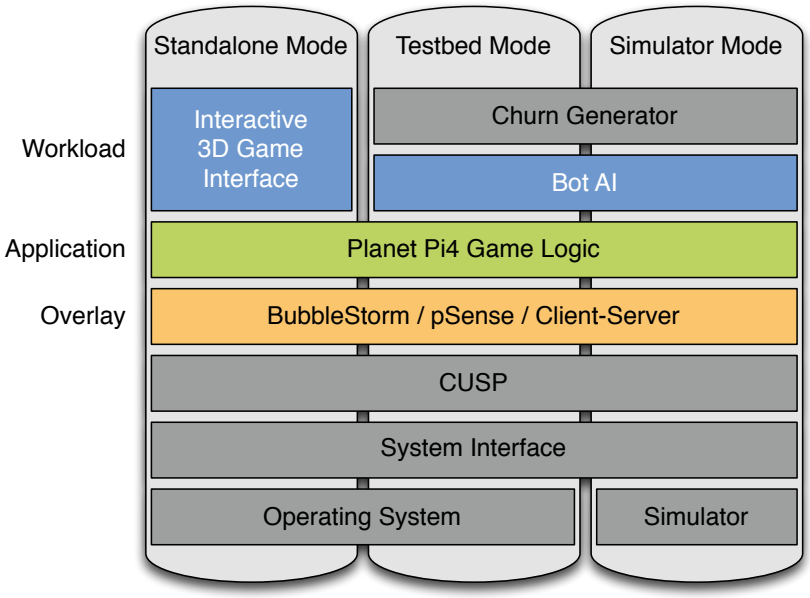


Figure 9.5.: Implementation of Planet Pi4 in the evaluation framework

The main purpose of the framework has been to debug and evaluate the implementation of CUSP and BubbleStorm. It is nonetheless a general-purpose simulation and prototyping framework, which can be used to implement a wide range of applications. The system has been successfully used to evaluate the sophisticated online gaming application Planet Pi4 [74] (see Figure 9.4).

The current implementation of Planet Pi4 does not only support the P2P overlays BubbleStorm and pSense [126] for in-game communication, but also a reference client-server implementation [75] (see Figure 9.5). The client and the server are two separate applications, which can be run in the same experiment, because the simulator allows running node-specific applications. The workload for Planet Pi4 can be either generated manually, using an interactive 3D game interface, or automatically by game bot artificial intelligence. The game is implemented in C++ and uses the C bindings of CUSP and BubbleStorm to use the framework. As it is using the framework's system interface exclusively and is single-threaded, it can be executed in the simulator.

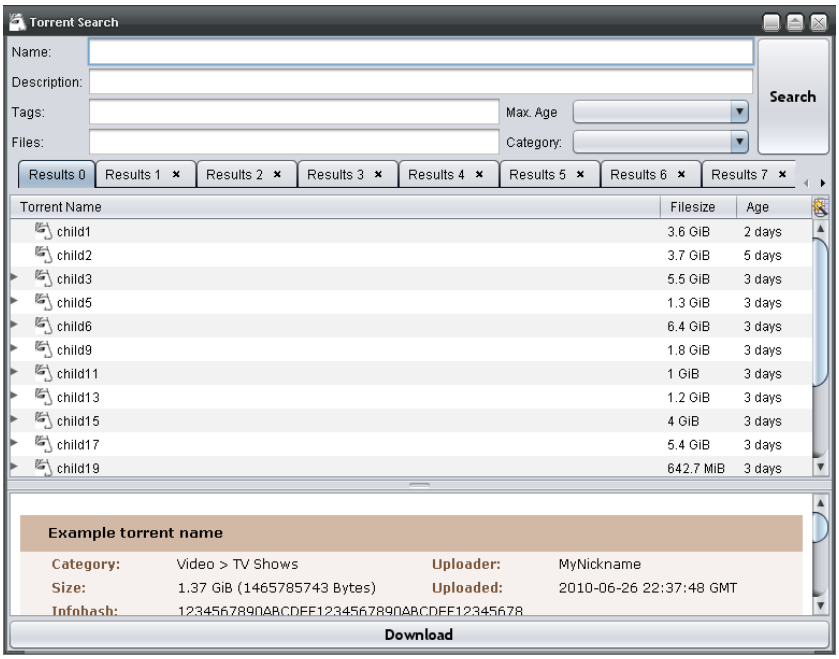


Figure 9.6.: Torrent search with BubbleStorm [34]

Another application, which has been built on top of the framework, is a BitTorrent client that uses BubbleStorm for decentralized torrent search and tracking [34, 147] (see Figure 9.6). The application is written in Java and uses the JNI bindings of BubbleStorm. Since the project is multi-threaded and uses TCP connections for BitTorrent-compatible swarming, it cannot be run in the simulator. However, this demonstrates that components that are built using the framework can be seamlessly integrated with components that are completely unaware of the framework, which makes it a versatile tool set for the development of network protocols and applications.

An implementation of the Kademlia DHT [91] is also available for the framework, which will be discussed in the evaluation chapter.

9.11 Review

The evaluation framework developed for this thesis is much more than a simple BubbleStorm simulator. The rigorous interface design, which allows moving an application from standalone mode into the simulator without changing a single line of code, is aiming to advance the understanding of network application simulation in general. In the process of implementing the framework, a number of shortcomings of commonly used techniques have been uncovered, like the random jitter problem or distorted node churn after large-scale simultaneous join events.

Even though the implementation of an application requires slightly more effort than in a traditional overlay simulator, the ability to deploy the system in three different modes easily compensates for that, since the parallel maintenance of independent code paths for prototype and simulation is no longer necessary. Additionally, the runtimes can be validated against each other to uncover artifacts in the results, which are caused by an unrealistic runtime. An application that is able to run in testbed or standalone mode is evidentially free of undesired side-channels, which can distort the system performance.

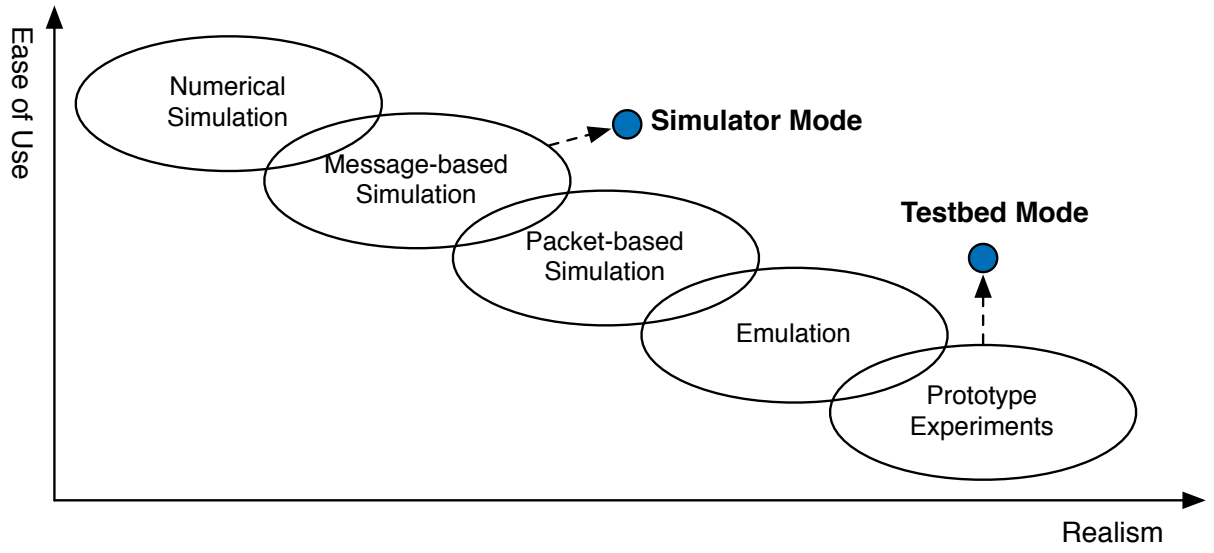


Figure 9.7.: Typical realism and ease of use of simulation and experimentation methods

A secondary objective of the framework design is to minimize the amount of work needed to implement additional runtimes, applications, and overlays, as well as preparing and interpreting experiments. The narrow system interface, the integrated transport protocol, and the central experiment database with the related tools are all helping to achieve this goal. In summary, the system provides a relatively realistic overlay simulator, which can be used to deploy real-world prototypes in fully automated experiments with very little additional effort.

Compared to the current state of the art (see Chapter 8), the evaluation framework offers a better combination of realism and effort (see Figure 9.7), especially when the different deployment options are combined, which achieves synergy both for the amount of work needed and validation of results. The simulator mode—although technically being message-based—benefits from the integrated transport protocol and the validation

with the testbed mode, putting it more in line with high-level packet-based simulators. The testbed mode on the other hand benefits from the automatic deployment, experiment database, and the possibility to debug the application in the simulator, which significantly reduces the effort to conduct prototype experiments.

10 Evaluation

The replication algorithms are evaluated using three different scenarios. The first scenario tests the long-term availability of data under churn and catastrophic events. The second scenario tests the consistency of search results when items are updated. The third scenario checks if items can be deleted from the network successfully.

For comparison, a basic BubbleStorm scenario with fading replication and a Kademlia overlay are used. Fading replication does not support updates or deletion and can therefore only be used in the replication scenario. Kademlia only supports expiration and is therefore not used in the deletion scenario.

10.1 Experiment Setup

The scenarios are item centric. An item is inserted, updated, or deleted by a randomly chosen node and then search requests for this item are issued by random nodes. Consequently, every search has exactly one correct result (the latest version of the item). This model is used to precisely measure the search success of BubbleStorm and make it comparable with Kademlia, which is constrained to key-value lookups.

Since rendezvous search decouples query matching from overlay communication, more advanced matching algorithms like keyword search would yield exactly the same search success and communication costs. In comparison, Kademlia would need to maintain an inverted index inducing significantly more traffic and possibly reducing search success. A meaningful keyword search benchmark comparing two fundamentally different overlay types is a complex task beyond the scope of this work. Especially, it is hard to define a realistic workload which does not penalize either overlay type, because the performance of each overlay depends on completely different workload parameters. Therefore, the scenarios only use key-value matching, even though this implies a clear disadvantage for BubbleStorm.

To measure the success, the standard information retrieval metrics precision and recall are used. Recall is defined as the fraction of relevant answers found. In this context, recall is the fraction of searches that retrieved the correct item. Precision is defined as the fraction of results that are relevant. In the update scenario, precision can be calculated as the fraction of searches with a result that returned the latest version.

10.1.1 Coordinator

In order to calculate precision and recall, the current state of an item must be known. This requires an inter-node coordination between publishing and search nodes. In the simulator, this information can be kept in shared memory, establishing a clearly defined side-channel between nodes. Unfortunately, shared memory is not available when using the distributed testbed.

Therefore, the coordination of insert, update, delete, and search operations in the overlay is implemented as a separate application, which communicates with the nodes. The coordinator is a small server to which the peers connect, and which triggers operations at the nodes and provides them with the expected query result. In the testbed, the peers and the coordinator use CUSP to communicate. In the simulator, a shared memory queue is used to eliminate the additional coordination traffic from the simulation results.

Essentially, the coordinator implements the application workload in the evaluation framework (see Figure 9.1). Using the coordinator approach, the same experiments can be run in the simulator and the testbed.

10.1.2 Workload Generation

The items published in the experiments have a size of 2KB, a typical size for a small text document or media metadata object. Initially, every five minutes divided by the network size, the coordinator selects a random node to publish a new item with a random identifier. Thus, on average, each node publishes a new item every five minutes.

After a publication, the coordinator waits for a cooldown of 100ms and then starts 20 search requests for this item from random nodes. The search requests are distributed exponentially over one hour, which increases the measurement density at the beginning of the search cycle. A node waits 60 seconds for results after issuing a search and then evaluates precision and recall.

In the replication scenario, the item is deleted or expired after the completion of the search cycle if the replication algorithm under test does not support deletion. In the consistency scenario, a random node is selected to publish an update for the item instead, and a new search cycle is started. The updating and searching is repeated until the end of the experiment. In the deletion scenario, a random node is selected to delete the item. The following search cycle tests if undeleted replicas remain in the system. Again, this process is repeated until the end of the experiment.

10.1.3 BubbleStorm Prototypes

Three different setups of BubbleStorm are used in the experiment. The *fading prototype* uses a fading bubble type for the data and an instant bubble type for the queries. They match with a lambda of 4, resulting in an expected recall of $\approx 98.17\%$. The *managed prototype* is similar to the fading replication, but uses a managed bubble type for the data. The *durable prototype* uses a durable bubble type for the data with a minimum size of 20 replicas and uses the same instant search bubble type. All three prototypes are designed to evaluate the ability of the respective replication mechanism to maintain the replica distribution required for successful rendezvous search. The *lookup prototype* uses a durable bubble type identical to the durable prototype, but uses lookups instead of an instant bubble for search.

10.1.4 Kademlia Prototypes

The Kademlia implementation used in the experiments, kindly provided by Max Lehn, implements the Kademlia protocol as described in the original publication [91]. Using the default parameters, the replication factor k is set to 20 replicas and α equals three parallel lookups. As suggested by the authors, plain UDP is used as transport protocol. Since UDP does not support re-assembly of IP fragments, the maximum transfer unit is artificially increased from 1500 bytes to 2200 bytes in the Kademlia simulation runs. Otherwise, the transferred 2KB items would be fragmented during transmission and the store requests would fail.

A real Kademlia implementation must either use a reliable transport protocol like TCP or CUSP or implement its own fragment re-assembly protocol. The connection establishment and management of a reliable transport would introduce significant overhead. A simple re-assembly protocol, however, would suffer from the combined message loss probabilities of all fragments, which makes the transfer of larger items practically impossible. Neither of this is accounted for in the experiments at hand, granting Kademlia an additional advantage.

As explained in Section 4.5, Kademlia uses a mix of replication modes in its replica management. Replicas are autonomously exchanged between the responsible peers, following the durable scheme. Additionally, the original publisher must refresh the item every 24 hours or it is expired, which fits the managed scheme. This work only evaluates the durable part, since the experiments are not long enough to trigger the expiration. In the real world, this mechanism artificially limits the flexibility of the system.

Even worse, the caching mechanism additionally requires a searching node to put an additional replica on the last node without the data found along the routing path. These items have a decreased expiration time based on the number of nodes between the caching node and the ID of the cached item. This mechanism implements a fading replication, because the cached replicas are practically impossible to update. To illustrate the effects of this approach, the Kademlia experiments are run both with and without caching.

It is not the intent of this work to compare the performance of key-value search of structured vs. unstructured P2P overlays, but instead to highlight why a careful replication algorithm design is required to achieve maximum reliability and consistency. Kademlia has been chosen for comparison, because it mixes all three replication modes into a single replication algorithm.

10.1.5 Network Composition

Unless noted otherwise, all experiments have a network size of 1000 running nodes. Each node has an exponential lifetime with an expected average lifetime of 60 minutes and an online/offline ratio of 5%, resulting in a pool of 20000 total nodes to achieve the desired online count. Under churn, 90% of the peers leave the network using the correct shutdown procedure, while 10% crash and simply do not answer requests anymore.

Ten well-known nodes, which are always online, are used as bootstrap hosts to join the overlay.

In the first 45 minutes of the experiment, the nodes join the overlay with an exponential ramp-up. The overlay is granted another 15 minutes to stabilize before the actual measurement phase starts.

Every node has a bandwidth of 16MBit/s downstream and 1MBit/s upstream, a typical ADSL2 home user connection. The nodes are distributed globally using the delay model discussed in Section 9.7.1.

10.2 Replication Scenario

In the first experiment, the reliability of the replication and search algorithms is tested. Items are published and searched for, but not updated or deleted. After the first hour of network build-up, the search success in a stable network is measured for two hours. Then a number of catastrophic events are triggered (see Figure 10.1). At three hours into the experiment, 50% of all nodes leave the network simultaneously, using the orderly leave algorithm. At four hours, the network size is brought up to its previous size of 1000 nodes again, by simultaneously joining 500 nodes. At five hours, 50% of all nodes crash simultaneously, again reducing the network size to 500 nodes.

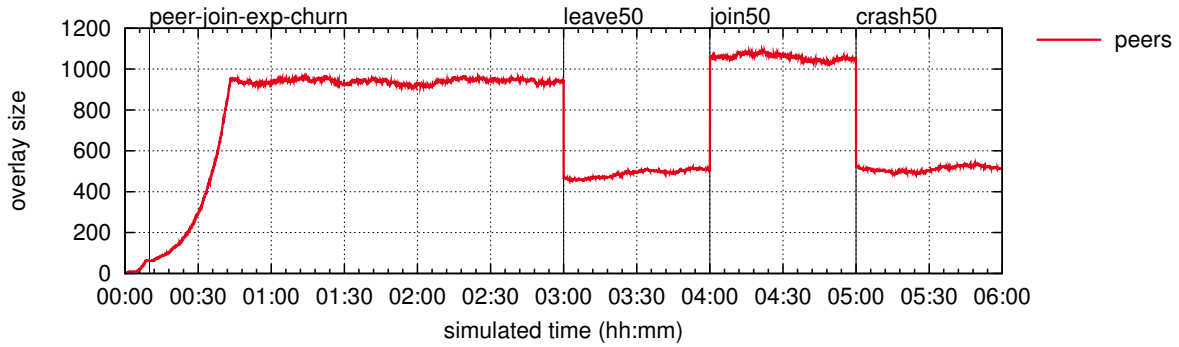


Figure 10.1.: The changing network size during the replication test

10.2.1 Replication Test

Figure 10.2 and Figure 10.3 show the recall in the replication test during the second and third hour of the experiment. The item age, which is the time span during the publication of the item and the beginning of the search request, is plotted on the x-axis. Figure 10.2 is plotted on a log scale for the item age to highlight the initial latency of replica distribution. Figure 10.3 is cropped to the relevant recall of 96%-100%.

The first thing to notice is the difference in distribution latency. Fading and durable replication are both extremely quick as they use constrained flooding mechanisms. Managed replication is slightly slower as the replicas are distributed by the maintainer exclusively. Nonetheless, it also gets to the desired replica degree in under one second. Kademlia is much slower, needing up to 30 seconds before an item is successfully published. Since Kademlia needs to ensure the k nodes closest to the item ID are selected

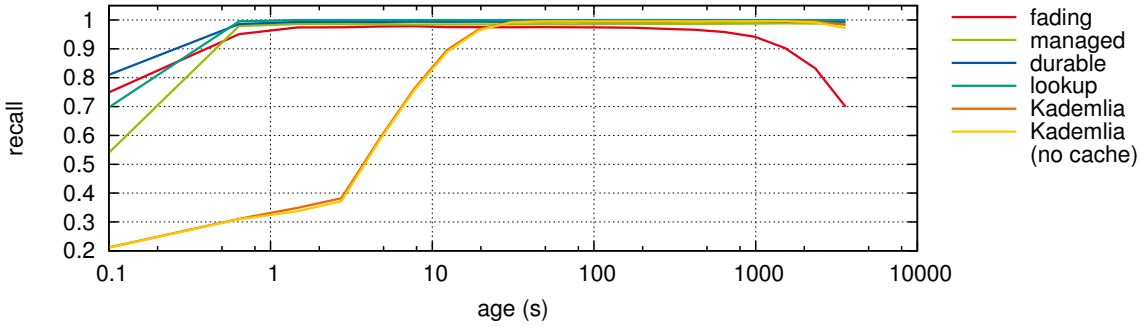


Figure 10.2.: Recall vs. item age (log scale for x-axis)

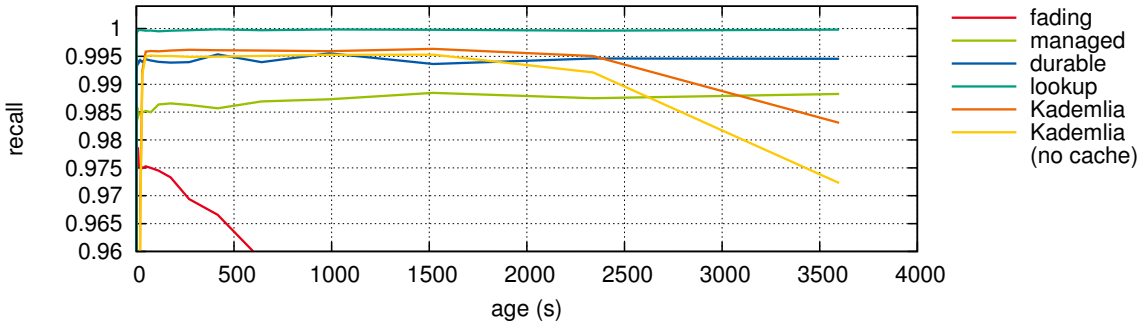


Figure 10.3.: Recall vs. item age (cropped)

and uses an iterative routing mechanism, communication timeouts can block the complete store process. These timeouts happen quite frequently, because nodes leave the Kademlia overlay without notifying their neighbors.

More interesting from a replication perspective is the long-term availability of items. The fading replication experiences the expected linear decay over time, since no counter-measures against lost replicas are taken. Kademlia is maintaining the high availability for at least 40 minutes before the item availability slightly decreases. It can be conjectured that this decrease is a result of Kademlia's reactive replica maintenance mechanism. Items are replicated from replica holders to the k closest nodes every hour, but if all of the replica holders either leave or are displaced from the set of k closest nodes due to churn, a lookup for the item may not find any replica. The caching of replicas lessens but does not eliminate the problem. Proactive replication mechanisms that exchange data immediately with newly discovered neighbors, like the managed and durable replication, do not exhibit this problem.

Kademlia also fails to achieve the 100% search success one would naïvely expect from the deterministic routing behavior. This is most likely to be attributed to routing table inconsistencies. However, the achieved 99.5% recall is already impressive, given the harsh conditions of a high churn overlay. The recall of the managed replication hovers around 98.5%, close to but safely above the guaranteed recall of 98.17%. The durable replication achieves a recall well above this mark, reaching up to the 99.5% of Kademlia. It must be assumed that the giant component of the responsibility graph is significantly higher than the lower bound (see Section 7.4), at least for the given

network size and minimum durable bubble size. This may be attributed to the $o(1)$ component in Theorem 3. The lookup mechanism is extremely reliable, providing a recall of 99.975% after completed replica distribution. Even though this is slightly below the specified 99.99%, probably because of node crashes, the recall is much better than the supposedly deterministic Kademlia.

In summary, both managed and durable replication achieve quick replica distribution and long-term availability.

10.2.2 Large-Scale Leave

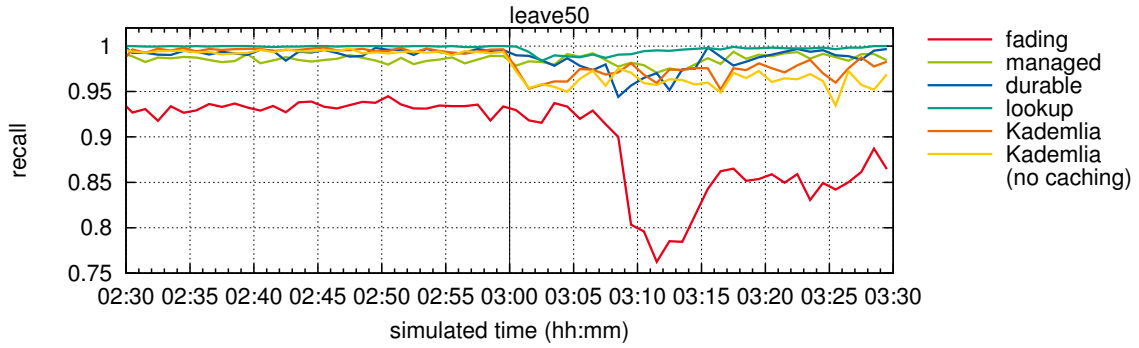


Figure 10.4.: Recall after a 50% leave event

After three hours, 50% of all nodes simultaneously leave the overlay. The effects on recall are depicted in Figure 10.4. The plot shows the combined recall of all search requests with an item age of at least 30 seconds, since this is the time required by the slowest replication algorithm to reach the desired replication degree. Fading replication is put at a disadvantage because it loses replicas over time, but the subject of discussion is the elasticity of the algorithms under stress and not a comparison of absolute recall between the algorithms. Each of the values in the plot is an average of 60 seconds of measurement, consisting of 800 to 2000 individual search requests.

After the leave, Kademlia drops to 95.3%, because some items lose most or even all of their replicas and an increased number of timeouts occurs. Afterwards, it slowly recovers to 98.3%, as caching replaces some of the lost replicas. Without caching, Kademlia keeps hovering around 96%.

Fading replication retains its search success right after the leave, even though it loses half of the 40 replicas per item on average. This is a simple side-effect of the search bubble size being calculated on the previous network size, and thus being too large for the current situation. When the new measurement from the gossip protocol arrives, the search bubble is shrunk from 120 to 80 and the recall drops accordingly. The only reason that the fading replication can recover is the publication of new items with the correct replica count. The items that existed before the event are permanently impaired.

The managed and durable replication are less affected by the massive leave. Like the fading replication, they benefit from unchanged bubble sizes in the beginning. When the new measurements arrive, the recall drops temporarily, as new replicas need to be distributed. After a short phase of reorganization, the search success recovers com-

pletely. The lookup prototype is similarly affected, since it depends on the durable data bubbles and managed responsibility bubbles.

10.2.3 Large-Scale Join

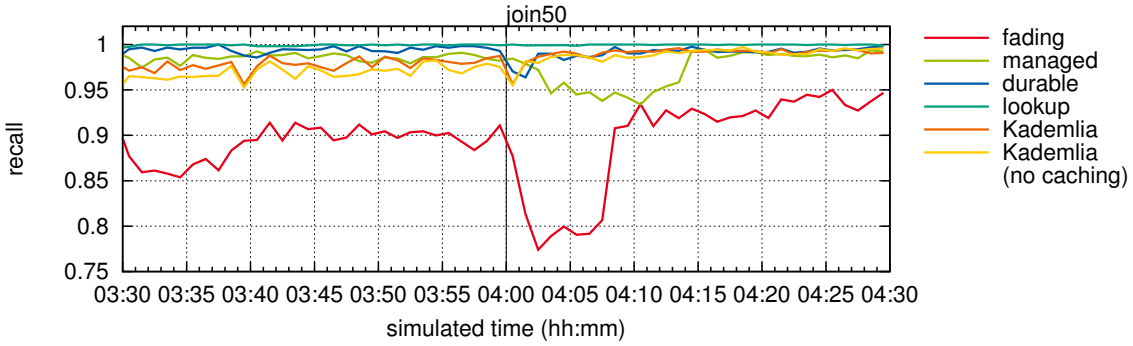


Figure 10.5.: Recall after a 50% join event

After four hours, the overlay is brought back to its initial size with a simultaneous join, doubling the network size. Figure 10.5 shows the effects on recall analogously to the previous section. Kademlia shows tremendous elasticity in this scenario. After only a very short drop, the recall goes up to the previous level as new nodes are integrated into the overlay very quickly. Caching has no significant impact in this situation.

BubbleStorm is in a much tougher position here. Before the new network size is discovered, the bubbles are too small to guarantee the desired recall. After the bubble size update, the recall in the fading replication jumps up to the previous level, because not only the new but also the old items benefit from the increased search bubble size.

Both the managed and durable replication need some additional time until the new replication level is set up, but then provide the usual high level of recall. The lookup mechanism suffers less, because initially the oversized responsibility bubbles compensate the missing durable replicas and when the responsibility gets reduced, the durable items have already been brought back to the correct replication level.

All of the examined replication algorithms show good stability in the case of large-scale join events, with Kademlia being much quicker on the recovery, since it does not have to rely on system-wide measurements.

10.2.4 Large-Scale Crash

After five hours, 50% of all nodes simultaneously crash without prior notice. Figure 10.6 documents how recall evolves during this event. Kademlia does not distinguish between a leave and a crash. A node that decides to quit the overlay simply closes its network socket. Therefore, the behavior of Kademlia during crash is practically identical to the leave scenario. Again, the failure probability after a crash is four times higher than before. Without caching, it is even increased seven-fold. The hourly republication may eventually fix this, but takes too long to be visible in the experiment.

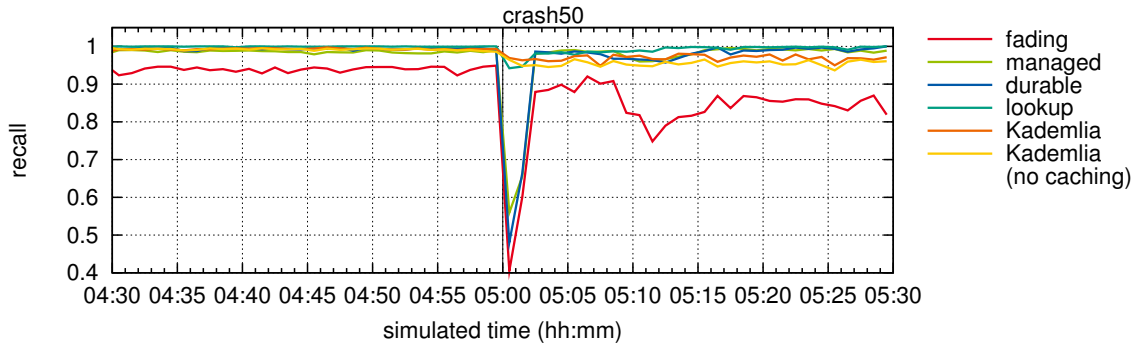


Figure 10.6.: Recall after a 50% crash event

BubbleStorm search suffers badly from the defunct overlay connection as most bubblecast messages are sent to non-existing neighbors. As topology timeouts discover the broken links and repair the overlay, the search success goes up again. Fading replication is unable to reach the previous level, because too many replicas have been lost. When the decreased bubble sizes take effect, the search success on previously published items drops again.

Managed and durable replication go through the same sequence of events, but are affected much less by the decreased bubble sizes and quickly restore replication of the existing items. Lookups are less impaired by timeouts, because they only use very small locate bubbles.

10.2.5 Query Response Times

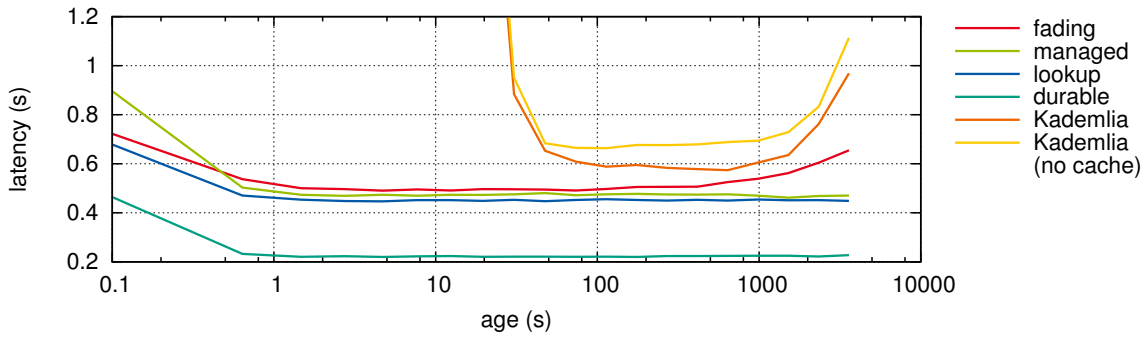


Figure 10.7.: Response time of searches

In Figure 10.7, the average response times of search requests are depicted. The measurements are taken from the stable phase of the replication test (see Section 10.2.1) and are plotted against the age of the item requested.

Kademlia has a rather unappealing search latency, starting at an extreme 11.3 seconds for items with an age of 100ms. This is about the time it takes to distribute the item, and an item cannot be found before it has been put into the overlay. As discussed in Section 10.2.1, most of these very early queries are left unanswered. Only the items suffering from a combination of timeouts have the chance of (late) success. However, even after replication is completed, the response time is only slightly below 700ms.

Most of the queries are answered much quicker, but in cases where multiple timeouts occur, the search may take several seconds. Caching improves the situation by reducing the number of outliers through early termination. When the replication starts dropping due to churn, the response times are affected too.

The BubbleStorm replication mechanisms provide almost constant response time of 450-500ms over the complete measurement cycle, excluding the initial replication phase, where the average latency is higher but already below one second. Only fading replication shows a slow increase in latency, which is caused by the decay of the replication degree. Durable lookups are much faster, achieving response times of 212ms on average, as only very few nodes need to be contacted for a lookup.

The results suggest that timeout-based behavior should be avoided in unstable P2P environments. This insight does not apply to search and replication in particular, but generally applies to P2P communication protocols.

10.2.6 Traffic Analysis

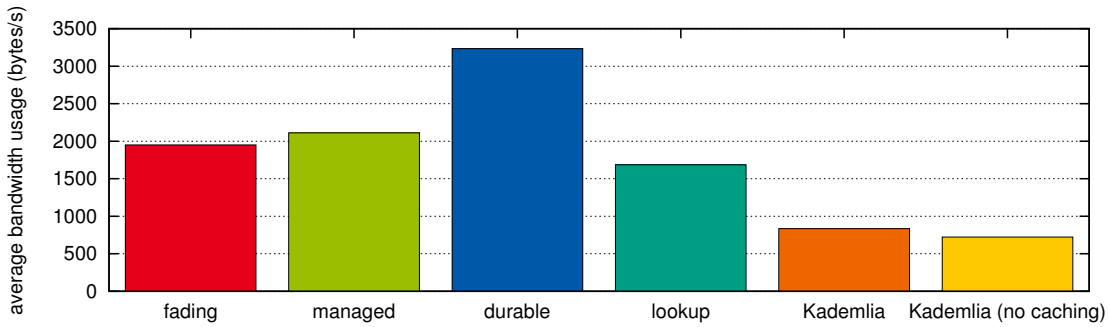


Figure 10.8.: Bandwidth usage

Figure 10.8 shows the average bandwidth usage of peers during the stable phase of the replication test. First thing to note is that all replication mechanisms are more than one order of magnitude below the upstream bandwidth limit of the peers in the scenario (128KB/s). Given the traffic complexity of $O(\sqrt{n})$ (BubbleStorm) or $O(\log n)$ (Kademlia), it is evident that there is enough headroom for much larger networks or even higher workloads. Unfortunately, the realistic evaluation of such large systems is beyond the computational power available for this thesis and would require a parallel simulator running on a large computing cluster.

Concerning the replication mechanisms, fading replication uses 1.95KB/s on average. Despite its management overhead, the managed replication needs only 2.11KB/s, which is just 8.3% more than the fading replication. The durable replication requires 3.23KB/s, which is 65.7% more than the base case. This increased traffic is caused by the inevitable transfer of replicas between peers that change responsibility and the redundancy of the responsibility graph. The excellent recall in the replication test with a failure rate less than a third of the probabilistic guarantee suggests that the trade-off between communication cost and reliability is not perfectly fine-tuned yet. However, both relative bandwidth overhead and recall may converge to a lower level when over-

lay grows to hundreds of thousands of nodes and beyond. Using lookups instead of rendezvous search for durable bubbles can reduce the bandwidth requirements significantly in the use-cases where the key of the item is already known. The lookup prototype consumes 1.79KB/s on average, which is only 55% of the durable replication using rendezvous search.

Despite BubbleStorm's bandwidth consumption being already very moderate, Kademlia of course is much more economical, as its $O(\log n)$ scalability already suggests. Without caching, Kademlia consumes 0.83KB/s and caching reduces this slighlty to 0.72KB/s. Larger network sizes would widen the gap between BubbleStorm and Kademlia even further, but one has to keep in mind that the two systems solve fundamentally different problems. Kademlia is limited to key-value lookups, and more sophisticated query languages like keyword search are a serious challenge for DHTs [83, 157], especially when it comes to bandwidth. BubbleStorm's traffic statistic would not change a single bit if the queries where keyword searches instead of key-value lookups.

10.3 Consistency Scenario

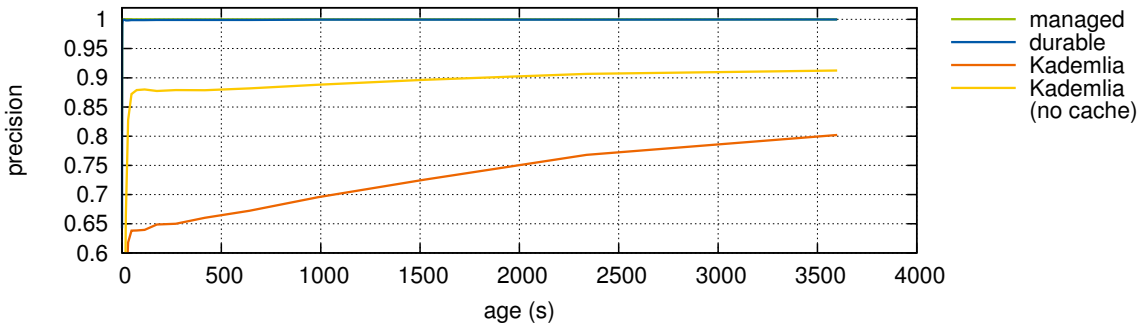


Figure 10.9.: Consistency of updates

In the consistency scenario, the overlay is grown analogously to the replication test, but then enters a three hour stable phase, in which only churn occurs, but no extreme events take place. During this time, items are searched for one hour as usual, but then a randomly selected peer sends an update. After that, a new one-hour search cycle begins, testing which version of the item is retrieved. This procedure is repeated until the end of the experiment. The precision of search results during the update phase is used as the key metric, since it measures how many queries receive the latest version of the item (see Figure 10.9).

Both BubbleStorm mechanisms show a very high level of consistency, with search result precision almost indistinguishable from 100%. This is a bit surprising, especially in the case of the managed replication, which is expected to accumulate a limited amount of junk due to crashes. There are two effects that promote higher than expected precision. Firstly, the junk level is much lower than the specified 80% goodness suggests. This network-wide limit is only ever reached when all online peers suffer from a long phase of maintainer crashes. In the experiment, most peers leave before reaching the junk threshold and new peers arrive in the network without any junk. Secondly, $\lambda = 4$ does

not only specify the average recall, but also the number of expected copies per search result. When the junk percentage is already low, the chances that all four answers are junk is extremely low. The same query behavior is responsible for the very good precision of the durable replication. Additionally, it benefits from the highly reliable distribution of replicas already discussed in the replication test.

The consistency test reveals the problems of the Kademlia replication. Even without caching, an update request is unable to displace the old version. Again, the reactive nature of the replication causes nodes that are no longer responsible for the item to keep their replicas until expiration. Since the default expiration in Kademlia is 24 hours, the old and new versions co-exist for a long time, causing a precision of only 88%, slowly rising to 91% as churn takes old replicas away.

With caching enabled, the situation is even worse. The initial precision is as low as 64%, i.e., one out of three queries does only return outdated results. Since cache replicas expire faster, the precision improves to 80% after one hour, which is still an unacceptable level for most applications.

Without immediate reaction to churn and a clear separation of replication modes, Kademlia is unable to achieve consistency for updates. Such a replication algorithm should only be used in environments where consistency does not matter or updates are not used at all.

10.4 Deletion Scenario

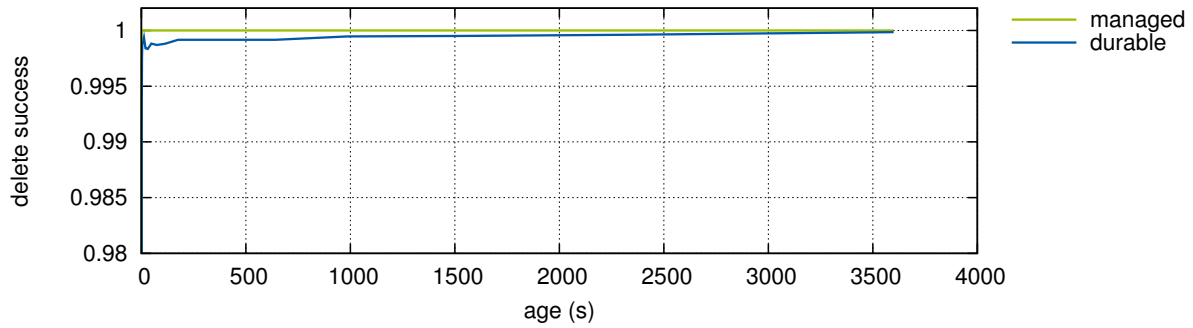


Figure 10.10.: Success of delete requests

The deletion scenario is almost identical to the consistency scenario, but instead of being updated, items are deleted after one hour. The delete success is depicted in Figure 10.10, which measures the percentage of queries that did not see the deleted item, i.e., how successful the deletion was. Since update and deletion are practically the same operation in the managed and durable replication, the behavior of the algorithms is identical. The chances of not detecting an outdated version of a deleted item are—at least in the given scenario—almost zero.

Kademlia is not included in the deletion test. The reactive approach and the inclusion of non-updatable fading replication leads to a highly inconsistent state. Without tombstones, deleted items will almost certainly be re-replicated over time, negating any deletion attempt.

10.5 Testbed Experiments

The evaluation framework used for the experiments can be used to generate prototypes and run them in a distributed real-world testbed (see Chapter 9). In order to validate the simulator and simulation results against real-world measurements, the replication experiment (see Section 10.2.1) was re-run on G-Lab [128], using ≈ 100 servers distributed across Germany. To avoid an overly bias due to too many peers sharing a single server, the network size was reduced to 400 peers. Kademlia had to be excluded from this test, because the simulator MTU hack to avoid UDP fragmentation cannot be used on the real Internet (see Section 10.1.4).

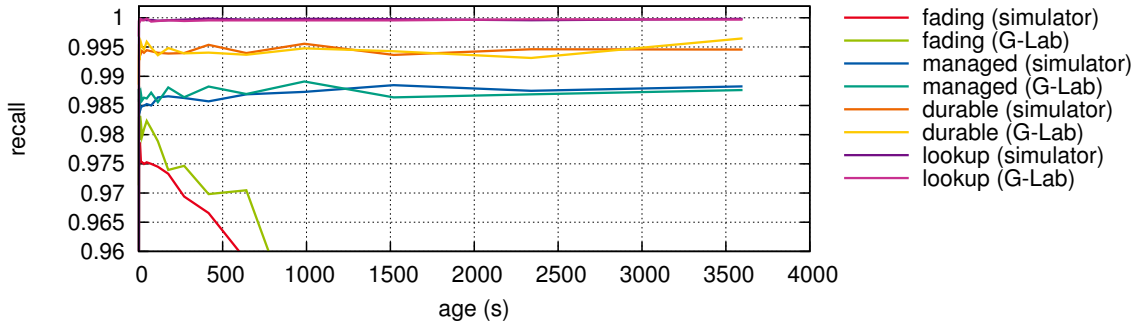


Figure 10.11.: Recall of simulator runs and G-Lab experiments compared

Figure 10.11 shows the recall results of the simulator experiments compared with their G-Lab counterparts. The differences between the simulator and the real world are mostly within the margin of measurement error, except for the fading replication, which even performs slightly better in the real world. The results convincingly show that the algorithms proposed in this work fulfill their specification in real-world environments.

10.6 Review

The simulation scenarios, which have been validated by real-world testbed experiments, have proven the reliability and consistency of the managed and durable replication mechanisms. The response times are comparable to the highly parallel bubblecast and the bandwidth overhead seems moderate. Update and delete requests are executed with very high consistency. The fading replication, which was previously BubbleStorm's only replication mechanism, is not only limited to immutable data, but also shows a constant decrease in search success over time. It is also not able to adjust replication after large-scale network changes. The lookups for durable bubbles provide an additional search algorithm, which is not only extremely reliable, but also less bandwidth-demanding than rendezvous search. A system that provides both mechanisms can let the application developer choose the optimal method for the problem at hand, even using both in parallel for different tasks.

Kademlia uses a slow but generally reliable replication scheme. Exchanging replicas between responsible peers once per hour was proven to be insufficient to cope with churn adequately. More serious is Kademlia's failure to achieve consistency after up-

dates, especially when using caching. The mixture of fading and durable replication modes limits the system's features to the far more constrained fading mode. The managed replication used for daily re-insertion of items, not simulated here, further limits the long-term availability of items. The taxonomy of replication modes given in Chapter 4 may help future replication algorithm designers to achieve a better separation of concerns.

11 Conclusion & Future Work

The BubbleStorm search overlay alone is already one of the most advanced P2P rendezvous search systems. But without long-term replication and update support, its applicability to real-world problems remains limited. The replication and update algorithms presented here close this gap and create a P2P overlay that provides much more than rendezvous search. The data management capabilities cover a wide range of use-cases with an easy-to-understand replication model and convenient application developer interface. The data management possible with this system is unparalleled by the existing rendezvous search system proposals. Therefore, we call BubbleStorm the first rendezvous information system.

Nonetheless, in every interesting research project, any achievement opens up new questions and research opportunities. The most relevant future work opportunities are summarized in the following, before a conclusion to this work is given.

11.1 Future Work

11.1.1 Data as a Signal

The persistent queries in BubbleStorm already permit to continuously receive additional results to a query or subscription, but this currently only covers newly inserted data. There is no technical reason for the current exclusion of updates and deletes and thereby limiting the observation of result set changes. The main challenge is to provide the application developer with a convenient and clear interface for a match function that is able to track all changes.

Data items in the result sets could be compared to a signal. They change from zero to a value when a new item is discovered, change their value on an update, and go back to zero when a delete happens. A user could send a query, which first matches the existing data to return an initial result set, and then is made persistent to track changes of the result set in real-time.

11.1.2 Additional Replication Mechanisms

The replication mechanisms provided for BubbleStorm cover all replication modes described in Chapter 4. The collective replication for durable data additionally provides key-value lookups, which would also be useful for data in the fading and managed replication modes. The lookups are enabled by the responsibility concept of the collective replication mechanism. It is certainly possible to build a managed replication mechanism on top of the responsibility routing and provide the same properties as with the maintainer-based replication. A fading replication using the responsibility would

be identical to durable data with the exception that it is never re-replicated to new neighbors.

In such a system, bubblecast would only be used for instant bubbles and to establish the maintainer-based responsibility routing tables. Fading, managed, and durable application data would all use the same (or at least similar) responsibility set flooding mechanism and all share the lookup capabilities. Nonetheless, rendezvous search including self-matching would be fully preserved.

11.1.3 Data Transfer

The origins of P2P computing lie in bulk data transfer and that is still one of its primary applications. Currently, the most popular system for swarming downloads is BitTorrent. BubbleStorm has already been employed as a distributed tracker and search engine for BitTorrent, but it seems desirable to integrate a bulk transfer mechanism directly into the BubbleStorm core.

Similar to blobs (binary large objects) in databases, sometimes an application needs to put large data chunks into the system, e.g., multimedia data. While BubbleStorm is content agnostic, transferring large items with bubblecast or one of the replication mechanisms would be slow and—due to the highly dynamic environment—potentially error-prone.

Instead of sending the data itself, a publisher could send the access information to a download swarm, which can be used to retrieve the data. In the collective replication, the same mechanism can be used to replicate data to new neighbors. In both cases, the parallel downloading from multiple sources can speed up data transfer and in the case of the maintainer-based replication reduce the load on the publisher.

If the binary data is not needed for query matching, some bandwidth can be saved by only distributing it to a small set of peers reliable enough to sustain the availability, possibly using the responsibility mechanism of the collective replication. The data bubble would then only contain a reference to the swarm for on-demand downloading. The collective replication could thus serve as a mechanism to permanently provide a set of seeders for a swarm.

11.1.4 Privacy

P2P systems are considered good substrates for privacy-preserving distributed systems, as implemented by Freenet [24] or GNUnet [9]. Unfortunately, these systems typically lack the support for complex queries, which is exactly what BubbleStorm provides. Integrating an anonymization mechanism like onion routing [33] into BubbleStorm or porting rendezvous search to one of the existing systems could provide a new level of privacy-preserving networking. The amount of self-organization and self-description possible with such a system would drastically reduce the need for external search engines, which may not be able to preserve the same level of privacy and resilience as the P2P overlay.

11.1.5 Cloud Computing

Google's grid-based search has served as the starting point for the discussion of rendezvous search. The harshness of P2P environments requires a high level of self-adaptivity that could also be useful in the ever-growing cloud data centers. Some of the results compiled and presented in this work may have their application in future generations of cloud computing.

Essentially, rendezvous search is the P2P equivalent of map/reduce [31]. The query is distributed to a selection of computers that hold the desired data, they execute a map job to select the matches and return them to a node that combines (i.e., reduces) the matches into a coherent result set. This is exactly what a query bubble in BubbleStorm does. Admittedly, a P2P overlay consisting of end-user computers connected through relatively slow Internet connections cannot compete with a high-performance data center cluster. Yet, the wide range of map/reduce applications could shed light on how to use rendezvous search systems for highly advanced P2P applications.

11.2 Conclusion

This work presents BubbleStorm and the replication mechanisms needed to make it the first true P2P rendezvous information system. Besides presenting the first comprehensive literature study in the area of rendezvous search, a generic taxonomy of P2P replication mechanisms consisting of four fundamental replication modes has been given, which helps to cleanly separate the use-cases in replication algorithm design. These replication modes are used to define a simple set of five data description primitives sufficient to define the data scheme of a rendezvous search application.

In order to implement the missing replication modes in BubbleStorm, two novel replication mechanisms have been presented: the maintainer-based replication, which implements the managed replication mode by using the publisher of a data item as its maintainer, and the collective replication, which implements the durable replication mode by setting up a responsibility routing on top of BubbleStorm. The responsibility routing also adds extremely resilient $O(1)$ key-value lookups on durable data. All of the algorithms are based on the accurate mathematical analysis of the underlying stochastic processes and theory of random graphs.

For the evaluation of the system, a novel simulation and prototyping environment has been developed, which makes it possible to run the same source code in a simulator, in a distributed testbed like PlanetLab, or as a standalone application. Since the same experiments can be run in the simulator and the testbed, the experiment results can be validated against each other. The evaluation results prove the high availability and consistency of updates even under extreme conditions.

Already, BubbleStorm has been used as a substrate for a set of very different applications including a decentralized BitTorrent search engine and tracker, a P2P wiki, and a full-fledged 3D massively multiplayer online game. This versatility proves the power of the rendezvous search concept and its BubbleStorm implementation. Hopefully, it will be used to create a new generation of sophisticated P2P applications.

A Analysis of the Maintainer-based Replication

This section proves the convergence of the maintainer-based replication (see Chapter 6). It is taken from our original paper describing the algorithm [81], but is authored by Wilhelm Stannat and Wesley Terpstra. It is included here solely for the reader's convenience.

Taking the approach of the maintainer-based replication, any sequence of peer join and crash/leave events causes the replica distribution to converge to the binomial, $B_{n,p}$. Symmetrically, if all items share p, then any sequence of item creation and deletion events with d undeleted objects converges peer load distribution to $B_{d,p}$. We only prove the first claim as the second is analogous.

Let $n_t \geq 0$ be the network size for time $t \in \mathbb{Z}$. For simplicity, we require that every unit of time corresponds to exactly one event: join or leave. A joining peer causes $n_{t+1} = n_t + 1$, while a leaving peer causes $n_{t+1} = n_t - 1$. The sequence of joins and leaves is chosen by an adversary.

For an arbitrary item, consider the evolution of the number of replicas R_t over time. R_t is a random variable taking values in $[0, n_t]$. If $n_{t+1} = n_t + 1$, then a storage peer joined the system, increasing R_t by 1 with probability p;

$$\begin{aligned}\mathbf{P}(R_{t+1}=i) &= \mathbf{P}(R_{t+1}=i \,|\, R_t=i-1)\mathbf{P}(R_t=i-1) \\ &\quad + \mathbf{P}(R_{t+1}=i \,|\, R_t=i)\mathbf{P}(R_t=i) \\ &= p\mathbf{P}(R_t=i-1) + (1-p)\mathbf{P}(R_t=i)\end{aligned}$$

Let $r_t = (r_t(0), r_t(1), \ldots, r_t(n_t))$ be R_t's probability vector, where $r_t(i) = \mathbf{P}(R_t = i)$ for $i \in [0, n_t]$. We use the notation $r_t(i)$ to emphasize that r_t plays a dual role as both a vector and a function of i. Set J_n to the $[0, n] \times [0, n+1]$ join transition matrix where $r_{t+1} = r_t J_{n_t}$,

$$J_n = \begin{bmatrix} 1-p & p & 0 & \ldots & 0 \\ 0 & 1-p & p & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & 1-p & p \end{bmatrix}$$

When one storage peer out of n_t leaves (or crashes), it destroys a replica with probability R_t/n_t. Like J_n, define L_n as the $[0,n] \times [0,n-1]$ transition matrix mapping $r_{t+1} = r_t L_{n_t}$.

$$L_n = \frac{1}{n} \begin{bmatrix} n & 0 & \dots & 0 \\ 1 & n-1 & \ddots & \vdots \\ 0 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & 1 \\ 0 & \dots & 0 & n \end{bmatrix}$$

Using these two matrices, we can finally formulate the transition matrix $r_{t+1} = r_t T_t$ for all t,

$$T_t := \begin{cases} J_{n_t} & \text{if } n_{t+1} = n_t + 1 \\ L_{n_t} & \text{if } n_{t+1} = n_t - 1 \end{cases}$$

Theorem 4. *If there exists some upper-bound n_* on the network size, such that $n_* > n_x$ for all time x, then, for any initial replication distribution r and fixed t, the replication distribution r_t converges to $B_{n_t,p}$ as the mixing time grows;*

$$r_t = r \prod_{x=t_0}^{t-1} T_x \to B_{n_t,p} \text{ as } (t - t_0) \to \infty$$

where $B_{n,p} = (B_{n,p}(0), B_{n,p}(1), \dots, B_{n,p}(n))$ and

$$B_{n,p}(i) = \binom{n}{i} p^i (1-p)^{n-i}$$

We first prove three lemmas needed for this result. The first lemma explains why the binomial is the limit.

Lemma 1. *The binomial distribution is an invariant flow;*

$$B_{n_{t+1},p} = B_{n_t,p} T_t$$

Proof. The transition matrix is a join J_n or leave L_n:

$$\begin{aligned} (B_{n,p}J_n)(0) &= (1-p)B_{n,p}(0) = (1-p)(1-p)^n \\ &= B_{n+1,p}(0) \\ (B_{n,p}J_n)(i) &= pB_{n,p}(i-1) + (1-p)B_{n,p}(i) \\ &= p^i(1-p)^{n+1-i}\left[\binom{n}{i-1} + \binom{n}{i}\right] \\ &= B_{n+1,p}(i) \\ (B_{n,p}L_n)(i) &= \frac{n-i}{n} B_{n,p}(i) + \frac{i+1}{n} B_{n,p}(i+1) \\ &= B_{n-1,p}(i)\left[(1-p) + p\right] \end{aligned}$$

□

The next two lemmas show that the transition matrix forces any two states R_t, S_t towards each other. We will measure the distance between $\mathbf{E}(f(R_{t+1})\,|\,R_t)$ and $\mathbf{E}(f(S_{t+1})\,|\,S_t)$ for arbitrary function f using the Lipschitz norm,

$$||f||_\ell = \max_{i>j} \frac{|f(i)-f(j)|}{i-j}$$

Lemma 2. *For all time t and any $f : \mathbb{Z} \to \mathbb{R}$,*

$$||T_t f||_\ell \le ||f||_\ell \cdot \begin{cases} 1 & \text{if } n_{t+1} = n_t + 1 \\ 1 - \frac{1}{n_t} & \text{if } n_{t+1} = n_t - 1 \end{cases}$$

Proof. There are again two cases. Interpret function $f(i)$ as a column vector on $[0, n_t]$. Then, for all $i > j$,

$$\begin{aligned} & J_n f(i) = (1-p)f(i) + pf(i+1) \\ & |J_n f(i) - J_n f(j)| \\ & \quad \le \ (1-p)|f(i)-f(j)| + p|f(i+1)-f(j+1)| \\ & \quad \le \ (1-p)||f||_\ell (i-j) + p||f||_\ell (i-j) \\ & \quad = \ ||f||_\ell (i-j) \end{aligned}$$

By carefully regrouping terms,

$$\begin{aligned} & L_n f(i) = \tfrac{i}{n} f(i-1) + \tfrac{n-i}{n} f(i) \\ & |L_n f(i) - L_n f(j)| \\ & \quad \le \ \tfrac{j}{n}|f(i-1)-f(j-1)| \\ & \qquad + \tfrac{i-j}{n}|f(i-1)-f(j)| + \tfrac{n-i}{n}|f(i)-f(j)| \\ & \quad \le \ ||f||_\ell (i-j)\left[\tfrac{j}{n} + \tfrac{i-1-j}{n} + \tfrac{n-i}{n}\right] \\ & \quad = \ ||f||_\ell (i-j)\left(1 - \tfrac{1}{n}\right) \end{aligned}$$

□

Lemma 3. *If there exists some n_* such that $n_* > n_x$ for all time x, then for $f : \mathbb{Z} \to \mathbb{R}$ with $||f||_\ell < \infty$ and fixed t,*

$$\left|\left| \prod_{x=t_0}^{t-1} T_x f \right|\right|_\ell \to 0 \text{ as } (t - t_0) \to \infty$$

Proof. Reformulating the bound on network size,

$$n_x < n_* \implies \left(1 - \frac{1}{n_x}\right) < \left(1 - \frac{1}{n_*}\right)$$

There are at most n_t more joins than leaves, so there are infinitely many leaves. Repeatedly apply Lemma 2 to find,

$$\left|\left| \prod_{x=t_0}^{t-1} T_x f \right|\right|_\ell < ||f||_\ell \left(1 - \frac{1}{n_*}\right)^{[(t-t_0)-n_t]/2} \to 0$$

□

Proof of Theorem. R_{t_0} has initial probability distribution r. From the definition of expectation, $\mathbf{E}(f(R_{t_0})) = rf$ where f is a function interpreted as a column vector. Similarly, $\mathbf{E}(f(R_t)) = r\prod_{x=t_0}^{t-1} T_x f$. So, the Lipschitz term from Lemma 3 can be reformulated for all $i > j$ as,

$$\frac{|\mathbf{E}(f(R_t)\,|R_{t_0}=i) - \mathbf{E}(f(S_t)\,|S_{t_0}=j)|}{i-j} \leq \left|\left|\prod_{x=t_0}^{t-1} T_x f\right|\right|_\ell$$

Condition on events $E_i = (R_{t_0}=i)$ and $F_j = (S_{t_0}=j)$,

$$\begin{array}{rl} & |\mathbf{E}(f(R_t)) - \mathbf{E}(f(S_t))| \\ = & |\sum_i \mathbf{P}(E_i)\mathbf{E}(f(R_t)|E_i) - \sum_j \mathbf{P}(F_j)\mathbf{E}(f(S_t)|F_j)| \\ = & |\sum_{i,j} \mathbf{P}(E_i)\mathbf{P}(F_j)(\mathbf{E}(f(R_t)|E_i) - \mathbf{E}(f(S_t)|F_j))| \\ \leq & \sum_{i,j} \mathbf{P}(E_i)\mathbf{P}(F_j)|\mathbf{E}(f(R_t)|E_i) - \mathbf{E}(f(S_t)|F_j)| \\ \leq & \left|\left|\prod_{x=t_0}^{t-1} T_x f\right|\right|_\ell \sum_{i,j} \mathbf{P}(E_i)\mathbf{P}(F_j)|i-j| \\ \leq & \left|\left|\prod_{x=t_0}^{t-1} T_x f\right|\right|_\ell n_* \to 0 \end{array}$$

The Portmanteau theorem (Theorem 2.1 in [10]) proves convergence in distribution given this limit for the expectation. Alternately, for arbitrary i, set $f(k) = 1$ for $k = i$ and 0 otherwise. Now $\mathbf{E}(f(R_t)) = \mathbf{P}(R_t = i) = r_t(i)$. Give S_{t_0} binomial distribution. By Lemma 1, S_t also has binomial distribution, so $\mathbf{E}(f(S_t)) = B_{n_t,p} f = B_{n_t,p}(i)$.

$$|r_t(i) - B_{n_t,p}(i)| = |\mathbf{E}(f(R_t)) - \mathbf{E}(f(S_t))| \to 0$$

□

Index

Bibliography

[1] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-Grid: a Self-Organizing Structured P2P System. *SIGMOD Record*, 32:29–33, September 2003.

[2] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Data Currency in Replicated DHTs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*, pages 211–222, New York, NY, USA, 2007. ACM.

[3] Shyam Antony, Divyakant Agrawal, and Amr El Abbadi. P2P Systems with Transactional Semantics. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology (EDBT'08)*, pages 4–15, New York, NY, USA, 2008. ACM.

[4] Michael Arrington. Joost Just Gives Up On P2P Altogether. http://techcrunch.com/2008/12/17/joost-just-gives-up-on-p2p/.

[5] Asad Awan, Ronaldo A. Ferreira, Suresh Jagannathan, and Ananth Grama. Distributed Uniform Sampling in Unstructured Peer-to-Peer Networks. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, volume 9, pages 223c–223c. IEEE, 2006.

[6] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[7] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A Flexible Overlay Network Simulation Framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI'07)*, pages 79–84. IEEE, May 2007.

[8] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A Scalable and Flexible Overlay Framework for Simulation and Real Network Applications. In *Proceedings of the Ninth International Conference on Peer-to-Peer Computing (P2P'09)*, pages 87–88. IEEE, 2009.

[9] Krista Bennett, Christian Grothoff, Tzvetan Horozov, Ioana Patrascu, and Tiberiu Stef. GNUnet - A Truly Anonymous Networking Infrastructure. In *Proceedings of the Privacy Enhancing Technologies Workshop (PET*, 2002.

[10] Patrick Billingsley. *Convergence of Probability Measures*. Wiley-Interscience, second edition, July 1999.

[11] Marcel Blöcher. Kontroll- und Messumgebung für Netzwerkprototypen. BSc. Thesis, Technische Universität Darmstadt, April 2012. German.

[12] Béla Bollobás. *Random Graphs*. Cambridge University Press, 2nd edition, 2001.

[13] Angela Bonifati, Ugo Matrangolo, Alfredo Cuzzocrea, and Mayank Jain. XPath Lookup Queries in P2P Networks. In *Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management (WIDM'04)*, pages 48–55, New York, NY, USA, 2004. ACM Press.

[14] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[15] Dirk Bradler, Lachezar Krumov, Max Mühlhäuser, and Jussi Kangasharju. PathFinder: Efficient Lookups and Efficient Search in Peer-to-Peer Networks. In *Proceedings of the 12th International Conference on Distributed Computing and Networking (ICDCN'11)*, volume 6522, pages 77–82, July 2011.

[16] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-Like Distributions: Evidence and Implications. In *Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'99)*, volume 1, pages 126–134, mar 1999.

[17] Eric A. Brewer. Towards Robust Distributed Systems (Invited Talk). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*, pages 7–, New York, NY, USA, 2000. ACM.

[18] CAIDA. Macroscopic Topology Project. `http://www.caida.org/analysis/topology/macroscopic/`.

[19] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'82)*, pages 128–136, New York, NY, USA, 1982. ACM.

[20] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-Like P2P Systems Scalable. In *Proceedings of ACM SIGCOMM'03*, pages 407–418, New York, NY, USA, 2003. ACM Press.

[21] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data. *IEEE Transactions on Knowledge and Data Engeering*, 4:582–592, December 1992.

[22] Tae Woong Choi and P. Oscar Boykin. Deetoo: Scalable Unstructured Search Built on a Structured Overlay. In *Proceedings of the International Workshop on Hot Topics in Peer-to-Peer Systems (HOTP2P'10)*, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[23] Fan Chung and Linyuan Lu. The Volume of the Giant Component of a Random Graph with Given Expected Degrees. *SIAM Journal on Discrete Mathematics*, 20(2):395–411, 2007.

[24] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, pages 46–66. Springer Berlin / Heidelberg, 2001.

[25] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, volume 6, pages 68–72, 2003.

[26] Edith Cohen and Scott Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'02)*, pages 177–190, New York, NY, USA, 2002. ACM.

[27] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 4th edition, 2005.

[28] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. P-ring: an Efficient and Robust P2P Range Index Structure. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*, pages 223–234, New York, NY, USA, 2007. ACM.

[29] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In M. Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 33–44. Springer Berlin / Heidelberg, 2003.

[30] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, pages 76 – 85, May 2003.

[31] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[32] Tim Dierks and Eric Rescorla. The Transport Layer Security (TLS) Protocol (Version 1.2). RFC 5246, August 2008.

[33] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. Technical report, Naval Research Lab, Washington DC, 2004.

[34] Tilo Eckert. Verteilte Suche für BitTorrent-Netzwerke. BSc. Thesis, Technische Universität Darmstadt, September 2010.

[35] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR)*, 35:114–131, June 2003.

[36] Ronaldo A. Ferreira, Murali Krishna Ramanathan, Asad Awan, Ananth Grama, and Suresh Jagannathan. Search with Probabilistic Guarantees in Unstructured Peer-to-Peer Networks. In *Proceedings of P2P'05*, pages 165–172, Washington, DC, USA, 2005. IEEE Computer Society.

[37] Colin J. Fidge. Timestamps in Message-Passing systems that Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*, volume 10, pages 56–66, 1988.

[38] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henryk Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[39] Bryan Ford. Structured Streams: a New Transport Abstraction. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, New York, NY, USA, August 2007. ACM.

[40] Wojciech Galuba, Karl Aberer, Zoran Despotovic, and Wolfgang Kellerer. ProtoPeer: a P2P Toolkit Bridging the Gap Between Simulation and Live Deployement. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, 2009.

[41] Jun Gao. *A Distributed and Scalable Peer-to-Peer Content Discovery System Supporting Complex Queries*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, October 2004.

[42] Jun Gao and Peter Steenkiste. Rendezvous Points-Based Scalable Content Discovery with Load Balancing. In *Proceedings of the Fourth International COST264 Workshop on Networked Group Communication (NGC'02)*, pages 71–78. ACM, October 2002.

[43] Jun Gao and Peter Steenkiste. Design and Evaluation of a Distributed Scalable Content Discovery System. *IEEE Journal on Selected Areas in Communications*, 22(1):54–66, 2004.

[44] Pedro García, C. Pairot, Rubén Mondéjar, Jordi Pujol, Helio Tejedor, and Robert Rallo. Planetsim: A New Overlay Network Simulation Framework. *Software Engineering and Middleware*, pages 123–136, 2005.

[45] Ali Ghodsi, Luc Alima, and Seif Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. In *Proceedings of the International Workshop on Databases, Information Systems, and Peer-to-Peer Computing*, volume 4125 of *Lecture Notes in Computer Science*, pages 74–85. Springer Berlin / Heidelberg, 2007.

[46] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33:51–59, June 2002.

[47] Sarunas Girdzijauskas, Wojciech Galuba, Vasilios Darlagiannis, Anwitaman Datta, and Karl Aberer. Fuzzynet: Ringless Routing in a Ring-Like Structured Overlay. *Peer-to-Peer Networking and Applications*, 4(3):259–273, 2011.

[48] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random Walks in Peer-to-Peer Networks. In *Proceedings of the Twenty-third Annual Joint Conference of the*

IEEE Computer and Communications Societies (INFOCOM'04), volume 1, March 2004.

[49] Google. SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.

[50] Kalman Graffi, Aleksandra Kovacevic, Patrick Mukherjee, Michael Benz, Christof Leng, Dirk Bradler, Julian Schröder-Bernhardi, and Nicolas Liebau. Peer-to-Peer-Forschung - Überblick und Herausforderungen. *it - Information Technology (Methods and Applications of Informatics and Information Technology)*, 46(3), September 2007.

[51] Jim Gray. Notes on Database Operating Systems. *Operating Systems: An Advanced Course*, pages 393–481, 1978.

[52] James Gross and Mesut Güneş. Introduction. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*. Springer, 2010.

[53] Saikat Guha, Neil Daswani, and Ravi Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *Proceedings of The 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, pages 1 – 6, Santa Barbara, CA, USA, February 2006.

[54] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. *SIGOPS Operating System Review*, 37:314–329, October 2003.

[55] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys (CSUR)*, 15:287–317, December 1983.

[56] Jani Hautakorpi and Göran Schultz. A Feasibility Study of an Arbitrary Search in Structured Peer-to-Peer Networks. In *Proceedings of 19th International Conference on Computer Communications and Networks (ICCCN'10)*, pages 1–8. IEEE, August 2010.

[57] Oliver Heckmann. Improving the Scientific Methods of Networking Research. Research Demo Presentation, 2004.

[58] Oliver Heckmann, Ralf Steinmetz, Nicolas Liebau, Alejandro Buchmann, Claudia Eckert, Jussi Kangasharju, Max Mühlhäuser, and Andy Schürr. Qualitätsmerkmale von Peer-to-Peer-Systemen. Technical Report KOM-TR-2006-03, Multimedia Communications Lab, Technische Universität Darmstadt, May 2006.

[59] Abdelsalam A. Helal, Abdelsalam A. Heddaya, and Bhara B. Bhargava. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[60] Thomas R. Henderson, Sumit Roy, Sally Floyd, and George F. Riley. ns-3 Project Goals. In *Proceedings of the 2006 Workshop on ns-2: the IP Network Simulator (WNS2)*, pages 13–es, 2006.

[61] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. The NewReno modification to TCP's fast recovery algorithm. RFC 6582, April 2012.

[62] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34:1–17, September 2006.

[63] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-Scale Virtualization in the Emulab Network Testbed. In *Proceedings of the USENIX Annual Technical Conference*, 2008.

[64] Gene Kan. Gnutella. In Oram [102], pages 62–79.

[65] Cai Kang. Survey of Search and Optimization of P2P Networks. *Peer-to-Peer Networking and Applications*, 4:211–218, 2011.

[66] Sebastian Kaune, Konstantin Pussep, Christof Leng, Aleksandra Kovacevic, Gareth Tyson, and Ralf Steinmetz. Modelling the Internet Delay Space Based on Geographical Locations. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (ICPADS'09)*, pages 301 –310, February 2009.

[67] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-Based Computation of Aggregate Information. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03)*, pages 482–491, Washington, DC, USA, 2003. IEEE Computer Society.

[68] Predrag Knežević, Andreas Wombacher, and Thomas Risse. Enabling High Data Availability in a DHT. In *Proceedings of the Sixteenth International Workshop on Database and Expert Systems Applications*, pages 363 –367, aug. 2005.

[69] Donald E. Knuth. 3.2.1 The Linear Congruential Method. In *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, pages 10–26. Addison-Wesley, Reading, Massachusetts, USA, third edition, 1997.

[70] Donald E. Knuth. 6.5 Retrieval on Secondary Keys. In *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, USA, second edition, 1998.

[71] Kovacevic, Aleksandra and Graffi, Kalman and Kaune, Sebastian and Leng, Christof and Steinmetz, Ralf. Towards Benchmarking of Structured Peer-to-Peer Overlays for Network Virtual Environments. In *Proceedings of the 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS '08)*, pages 799 –804, December 2008.

[72] Lachezar Krumov. *Local Structures Determine Performance within Complex Networks*. PhD thesis, Technische Universität Darmstadt, November 2010.

[73] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[74] Max Lehn, , Tonio Triebel, Christof Leng, Alejandro Buchmann, and Wolfgang Effelsberg. Performance evaluation of peer-to-peer gaming overlays. In *Proceedings of the IEEE Tenth International Conference on Peer-to-Peer Computing (P2P'10)*, pages 1–2. IEEE, 2010. demo.

[75] Max Lehn, Christof Leng, Robert Rehner, Tonio Triebel, and Alejandro Buchmann. An online gaming testbed for peer-to-peer architectures. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 474–475. ACM, 2011. demo.

[76] Max Lehn, Tonio Triebel, Christian Gross, Dominik Stingl, Karsten Saller, Wolfgang Effelsberg, Alexandra Kovacevic, and Ralf Steinmetz. Designing Benchmarks for P2P Systems. In Kai Sachs, Ilia Petrov, and Pablo Guerrero, editors, *From Active Data Management to Event-Based Systems and More*, volume 6462 of *Lecture Notes in Computer Science*, pages 209–229. Springer, nov 2010.

[77] Christof Leng. Distributed Search with Rendezvous Search Systems. In *7th KuVS/ITG-Fachgespräch Future Internet*, Jan 2012.

[78] Christof Leng. Large Scale Arbitrary Search with Rendezvous Search Systems. *Peer-to-Peer Networks and Applications*, 2012. submitted for publication.

[79] Christof Leng, Max Lehn, Robert Rehner, and Alejandro Buchmann. Designing a Testbed for Large-scale Distributed Systems. In *Proceedings of ACM SIGCOMḾ11*, New York, NY, USA, August 2011. ACM. Poster.

[80] Christof Leng and Wesley W. Terpstra. Distributed SQL Queries with BubbleStorm. In Kai Sachs, Ilia Petrov, and Pablo Guerrero, editors, *From Active Data Management to Event-Based Systems and More*, volume 6462 of *Lecture Notes in Computer Science*, pages 230–241. Springer, nov 2010.

[81] Christof Leng, Wesley W. Terpstra, Bettina Kemme, Wilhelm Stannat, and Alejandro P. Buchmann. Maintaining Replicas in Unstructured P2P Systems. In *Proceedings of the ACM CoNEXT Conference (CoNEXT'08)*, pages 19:1–19:12, New York, NY, USA, 2008. ACM.

[82] Scott T. Leutenegger and Daniel Dias. A Modeling Study of the TPC-C Benchmark. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'93)*, pages 22–31, New York, NY, USA, 1993. ACM.

[83] Jinyang Li, Boon Loo, Joseph Hellerstein, M. Frans Kaashoek, David Karger, and Robert Morris. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.

[84] Pericles Lopes and Ronaldo A. Ferreira. SplitQuest: Controlled and Exhaustive Search in Peer-to-Peer Networks. In *Proceedings of the 9th International Conference on Peer-to-Peer Systems (IPTPS'10)*, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association.

[85] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Surveys and Tutorials*, 7(2):72–93, 2005.

[86] Marcel Lucas. Implementation of a Peer-to-Peer Spatial Publish/Subscribe System for Networked Virtual Environments Using BubbleStorm. MSc. Thesis, Technische Universität Darmstadt, May 2011.

[87] Ivan Martinovic, Christof Leng, Frank Zdarsky, Andreas Mauthe, Ralf Steinmetz, and Jens Schmitt. Self-protection in P2P Networks: Choosing the Right Neighbourhood. In Hermann de Meer and James Sterbenz, editors, *Self-Organizing Systems*, volume 4124 of *Lecture Notes in Computer Science*, pages 23–33. Springer Berlin / Heidelberg, 2006.

[88] Laurent Massoulié, Erwan Le Merrer, Anne-Marie Kermarrec, and Ayalvadi Ganesh. Peer Counting and Sampling in Overlay Networks: Random Walk Methods. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing (PODC'06)*, pages 123–132, New York, NY, USA, 2006. ACM.

[89] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: a 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[90] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[91] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems (IPTPS'01)*, pages 53–65, London, UK, 2002. Springer-Verlag.

[92] Erik McCandless, Michael Hatcher and Otis Gospodnetić. *Lucene in Action*. Manning Publications, 2010.

[93] Alberto Montresor and Márk Jelasity. PeerSim: A Scalable P2P Simulator. In *Proceedings of the Ninth IEEE International Conference on Peer-to-Peer Computing, 2009 (P2P'09)*, pages 99–100. IEEE, 2009.

[94] G. M. Morton. A Computer Oriented Geodetic Database and a New Technique in File Sequencing. Technical report, IBM, Ottawa, Canada, 1966.

[95] Monika Moser and Seif Haridi. Atomic Commitment in Transactional DHTs. In Thierry Priol and Marco Vanneschi, editors, *Towards Next Generation Grids*, pages 151–161. Springer US, 2007.

[96] Damon Mosk-Aoyama and Devavrat Shah. Computing Separable Functions via Gossip. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing (PODC'06)*, pages 113–122, New York, NY, USA, 2006. ACM.

[97] Patrick Mukherjee. *A fully Decentralized, Peer-to-Peer Based Version Control System*. PhD thesis, Technische Universität Darmstadt, Mar 2011.

[98] Patrick Mukherjee, Christof Leng, Wesley Terpstra, and Andy Schürr. Peer-to-Peer based Version Control. In *Proceedings of the 14th International Conference on Parallel and Distributed Systems (ICPADS 2008)*, pages 829–834. IEEE Computer Society Press, Dec 2008.

[99] Mukherjee, Patrick and Leng, Christof and Schürr, Andy. Piki - A Peer-to-Peer Based Wiki Engine. In *Proceedings of the Eighth International Conference on Peer-to-Peer Computing (P2P'08)*, pages 185 –186, September 2008.

[100] Napster. http://www.napster.com.

[101] Chris Newman. *SQLite (Developer's Library)*. Sams, Indianapolis, IN, USA, 2004.

[102] Andy Oram, editor. *Peer-to-Peer - Harnessing the Power of Disruptive Technologies*. OŔeilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2001.

[103] Jon Postel. User Datagram Protocol. RFC 768, August 1980.

[104] Jon Postel. Transmission Control Protocol. RFC 793, September 1981.

[105] Dan Pritchett. BASE: An Acid Alternative. *Queue*, 6:48–55, May 2008.

[106] Konstantin Pussep, Sebastian Kaune, Christof Leng, Aleksandra Kovacevic, and Ralf Steinmetz. Impact of User Behavior Modeling on Evaluation of Peer-to-Peer Systems. Technical Report KOM-TR-2008-07, Technische Universität Darmstadt, November 2008.

[107] Konstantin Pussep, Christof Leng, and Sebastian Kaune. Modeling User Behavior in P2P Systems. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*, pages 447–461. Springer Berlin / Heidelberg, 2010.

[108] Costin Raiciu. *ROAR: Increasing the Flexibility and Performance of Distributed Search*. PhD thesis, University College London, London, UK, 2011.

[109] Costin Raiciu, Felipe Huici, Mark Handley, and David S. Rosenblum. ROAR: Increasing the Flexibility and Performance of Distributed Search. In *Proceedings of SIGCOMM'09*, pages 291–302, New York, NY, USA, 2009. ACM.

[110] Costin Raiciu, David S. Rosenblum, and Mark Handley. Distributed Online Filtering. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'07)*, pages 15–16, New York, NY, USA, August 2007. ACM.

[111] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Brief Announcement: Prefix Hash Tree. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC'04)*, pages 368–368, New York, NY, USA, 2004. ACM.

[112] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. *SIGCOMM Computer Communication Review*, 31:161–172, August 2001.

[113] Patrick Reynolds and Amin Vahdat. Efficient Peer-to-Peer Keyword Searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware (Middleware'03)*, pages 21–40, New York, NY, USA, 2003. Springer-Verlag New York.

[114] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference (ATC'04)*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[115] John Risson and Tim Moors. Survey of Research Towards Robust Peer-to-Peer Networks: Search Methods. *Computer Networks*, 50(17):3485–3521, 2006.

[116] Jonathan Rosenberg, Rohan Mahy, Philip Matthews, and Dan Wing. Session Traversal Utilities for NAT (STUN). RFC 5389, October 2008.

[117] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of Middleware'01*, volume 2218, pages 329–350. Springer Berlin / Heidelberg, 2001.

[118] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. *SIGOPS Operating Systems Review*, 35:188–201, October 2001.

[119] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication (NGC'01)*, pages 30–43, London, UK, UK, 2001. Springer-Verlag.

[120] Kai Sachs, Samuel Kounev, Stefan Appel, and Alejandro Buchmann. Benchmarking of Message-Oriented Middleware. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems (DEBS'09)*, pages 44:1–44:2, New York, NY, USA, 2009. ACM.

[121] Kai Sachs, Samuel Kounev, Jean Bacon, and Alejandro Buchmann. Performance Evaluation of Message-Oriented Middleware using the SPECjms2007 Benchmark. *Performance Evaluation*, 66(8):410–434, Aug 2009.

[122] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Computing Surveys*, 37:42–81, March 2005.

[123] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4:14:1–14:42, May 2009.

[124] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking*, MMCN, page 152, 2002.

[125] Nima Sarshar, P. Oscar Boykin, and Vwani P. Roychowdhury. Percolation Search in Power Law Networks: Making Unstructured Peer-to-Peer Networks Scalable. In *Proceedings of IEEE P2P'04*, pages 2–9, Washington, DC, USA, 2004. IEEE Computer Society.

[126] Arne Schmieg, Michael Stieler, Sebastian Jeckel, Patric Kabus, Bettina Kemme, and Alejandro Buchmann. pSense - Maintaining a Dynamic Localized Peer-to-Peer Structure for Position Based Multicast in Games. In *Proceedings of the 8th International Conference on Peer-to-Peer Computing (P2P'08)*, September 2008.

[127] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: Reliable Transactional P2P Key/Value Store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG (ERLANG'08)*, pages 41–48, New York, NY, USA, 2008. ACM.

[128] Dennis Schwerdel, Daniel Günther, Robert Henjes, Bernd Reuther, and Paul Müller. German-Lab Experimental Facility. In Arne Berre, Asunción Gómez-Pérez, Kurt Tutschku, and Dieter Fensel, editors, *Future Internet - FIS 2010*, volume 6369 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin / Heidelberg, 2010.

[129] Fadi N. Sibai. Performance Analysis and Workload Characterization of the 3DMark05 Benchmark on Modern Parallel Computer Platforms. *SIGARCH Computer Architecture News*, 35:44–52, June 2007.

[130] Dale Skeen and Michael Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, pages 219–228, 1983.

[131] Neil Spring, Larry Peterson, Andy Bavier, and Vivek S. Pai. Using PlanetLab for Network Research: Myths, Realities, and Best Practices. *ACM SIGOPS Operating Systems Review*, 40(1):17–24, 2006.

[132] Pyda Srisuresh and Kjeld Borch Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, January 2001.

[133] Ralf Steinmetz and Klaus Wehrle. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer, Heidelberg, Germany, 2005.

[134] Randall Stewart. Stream Control Transmission Protocol. RFC 4960, 2007.

[135] Dominik Stingl, Christian Groß, Julius Rückert, Leonhard Nobach, Aleksandra Kovacevic, and Ralf Steinmetz. PeerfactSim.KOM: A Simulation Framework for Peer-to-Peer Systems. In Waleed W. Smari, editor, *Proceedings of the 2011 International Conference on High Performance Computing & Simulation (HPCS'11)*, pages 577–584. IEEE, July 2011.

[136] 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.

[137] Daniel Stutzbach, Reza Rejaie, and Subhabrata Sen. Characterizing Unstructured Overlay Topologies in Modern P2P File-Sharing Systems. *IEEE/ACM Transactions on Networking*, 16:267–280, April 2008.

[138] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Education, Inc., Upper Saddle River, NJ, USA, 2nd edition, 2007.

[139] Wesley W. Terpstra. Distributed Cartesian Product. Diploma Thesis, Technische Universität Darmstadt, May 2006.

[140] Wesley W. Terpstra. *BubbleStorm: Rendezvous Theory in Unstructured Peer-to-Peer Search*. PhD thesis, Technische Universität Darmstadt, 2011. under preparation.

[141] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Jussi Kangasharju, and Alejandro Buchmann. Bit Zipper Rendezvous—Optimal Data Placement for General P2P Queries. In *Proceedings of the EDBT 04 Workshop on Peer-to-Peer Computing & DataBases*, March 2004.

[142] 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)*, New York, NY, USA, June 2003. ACM.

[143] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. BubbleStorm: Resilient, Probabilistic, and Exhaustive Peer-to-Peer Search. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'07)*, pages 49–60, New York, NY, USA, August 2007. ACM.

[144] Wesley W. Terpstra, Christof Leng, and Alejandro P. Buchmann. Brief Announcement: Practical Summation via Gossip. In *Proceedings of the Twenty-Sixth Annual ACM Aymposium on Principles of Distributed Computing (PODC'07)*, pages 390–391, New York, NY, USA, August 2007. ACM.

[145] Wesley W. Terpstra, Christof Leng, and Alejandro P. Buchmann. BubbleStorm: Analysis of Probabilistic Exhaustive Search in a Heterogeneous Peer-to-Peer System. Technical Report TUD-CS-2007-2, Technische Universität Darmstadt, Fachbereich Informatik, Darmstadt, Germany, May 2007.

[146] Wesley W. Terpstra, Christof Leng, Max Lehn, and Alejandro P. Buchmann. Channel-based Unidirectional Stream Protocol (CUSP). In *Proceedings of the IEEE INFOCOM Mini Conference*, March 2010.

[147] Andreas Teuber. Soziale Komponenten für BitTorrent via BubbleStorm. BSc. Thesis, Technische Universität Darmstadt, July 2012.

[148] The Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org/.

[149] The PingER Project. `http://www-iepm.slac.stanford.edu/pinger/`.

[150] Robbert van Renesse and Rachid Guerraoui. Replication Techniques for Availability. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication*, volume 5959 of *Lecture Notes in Computer Science*, pages 19–40. Springer Berlin / Heidelberg, 2010.

[151] Maarten van Steen and Guillaume Pierre. Replicating for Performance: Case Studies. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication*, volume 5959 of *Lecture Notes in Computer Science*, pages 73–89. Springer Berlin / Heidelberg, 2010.

[152] András Varga. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference (ESM'01)*, pages 319–324, 2001.

[153] Zhijun Wang, Sajal K. Das, Mohan Kumar, and Huaping Shen. An Efficient Update Propagation Algorithm for P2P Systems. *Computer Communications*, 30(5):1106–1115, 2007.

[154] Stephen Weeks. Whole-Program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML (ML'06)*, pages 1–1, New York, NY, USA, 2006. ACM.

[155] Elias Weingärtner, Florian Schmidt, Hendrik Vom Lehn, Tobias Heer, and Klaus Wehrle. SliceTime: a platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*, Berkeley, CA, USA, 2011. USENIX Association.

[156] Michael C. Wendl. Collision Probability Between Sets of Random Variables. *Statistics & Probability Letters*, 64(3):249–254, 2003.

[157] Yong Yang, Rocky Dunlap, Michael Rexroad, and Brian F. Cooper. Performance of Full Text Search in Structured and Unstructured Peer-to-Peer Systems. In *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM'06)*, pages 1–12, April 2006.

[158] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2001.