ABSTRACT

CONTEXT-AWARE COLLABORATIVE STORAGE AND PROGRAMMING FOR MOBILE USERS

by
Mohammad A. Khan

Since people generate and access most digital content from mobile devices, novel innovative mobile apps and services are possible. Most people are interested in sharing this content with communities defined by friendship, similar interests, or geography in exchange for valuable services from these innovative apps. At the same time, they want to own and control their content. Collaborative mobile computing is an ideal choice for this situation. However, due to the distributed nature of this computing environment and the limited resources on mobile devices, maintaining content availability and storage fairness as well as providing efficient programming frameworks are challenging.

This dissertation explores several techniques to improve these shortcomings of collaborative mobile computing platforms. First, it proposes a medley of three techniques into one system, MobiStore, that offers content availability in mobile peer-to-peer networks: topology maintenance with robust connectivity, structural reorientation based on the current state of the network, and gossip-based hierarchical updates. Experimental results showed that MobiStore outperforms a state-of-the-art comparison system in terms of content availability and resource usage fairness.

Next, the dissertation explores the usage of social relationship properties (i.e., network centrality) to improve the fairness of resource allocation for collaborative computing in peer-to-peer online social networks. The challenge is how to provide fairness in content replication for P2P-OSN, given that the peers in these networks exchange information only with one-hop neighbors. The proposed solution provides fairness by selecting the peers to replicate content based on their potential to introduce the storage skewness, which is determined from their structural properties in the network. The proposed solution, Philia, achieves higher content availability and storage fairness than several comparison systems.

The dissertation concludes with a high-level distributed programming model, which efficiently uses computing resources on a cloud-assisted, collaborative mobile computing platform. This platform pairs mobile devices with virtual machines (VMs) in the cloud for increased execution performance and availability. On such a platform, two important challenges arise: first, pairing the two computing entities into a seamless computation, communication, and storage unit; and second, using the computing resources in a cost-effective way. This dissertation proposes Moitree, a distributed programming model and middleware that translates high-level programming constructs into events and provides the illusion of a single computing entity over the mobile-VM pairs. From programmers' viewpoint, the Moitree API models user collaborations into dynamic groups formed over location, time, or social hierarchies. Experimental results from a prototype implementation show that Moitree is scalable, suitable for real-time apps, and can improve the performance of collaborating apps regarding latency and energy consumption.

CONTEXT-AWARE COLLABORATIVE STORAGE AND PROGRAMMING FOR MOBILE USERS

by
Mohammad A. Khan

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

August 2016

APPROVAL PAGE

CONTEXT-AWARE COLLABORATIVE STORAGE AND PROGRAMMING FOR MOBILE USERS

Mohammad A. Khan

Cristian M. Borcea, PhD, Dissertation Advisor Date
Professor and Chair, Department of Computer Science, New Jersey Institute of Technology

---

Narain Gehani, PhD, Committee Member Date
Professor of Computer Science, New Jersey Institute of Technology

---

Reza Curtmola, PhD, Committee Member Date
Associate Professor of Computer Science, New Jersey Institute of Technology

---

Xiaoning Ding, PhD, Committee Member Date
Assistant Professor of Computer Science, New Jersey Institute of Technology

---

Adriana Iamnitchi, PhD, Committee Member Date
Associate Professor of Computer Science and Engineering, University of South Florida

BIOGRAPHICAL SKETCH

Author:	Mohammad A. Khan
Degree:	Doctor of Philosophy
Date:	August 2016

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, New Jersey, 2016

- Bachelor of Science in Computer Science & Engineering,
  Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, 2005

Major: Computer Science

Presentations and Publications:

Mohammad A. Khan, Hillol Debnath, Cristian Borcea, "Balanced Content Replication in Peer-to-Peer Online Social Networks", *The 9th IEEE International Conference on Social Computing and Networking (SocialCom 2016)*, October 2016

Mohammad A. Khan, Hillol Debnath, Nafize R. Paiker, Narain Gehani, Xiaoning Ding, Reza Curtmola, Cristian Borcea, "Moitree: A Middleware for Cloud-Assisted Mobile Distributed Apps",*The 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud 2016)* , March 2016

Mohammad A. Khan, Laurent Yeh, Karine Zeitouni, Cristian Borcea, "Efficient Mobile P2P Data Sharing in MobiStore", *Journal of Peer-to-Peer Networking and Applications, Springer, March 2016*

Cristian Borcea, Xiaoning Ding, Narain Gehani, Reza Curtmola, Mohammad A Khan, Hillol Debnath, "Avatar: Mobile Distributed Computing in the Cloud",*The 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud 2015)*, March 2015

Mohammad A. Khan, Laurent Yeh, Karine Zeitouni, and Cristian Borcea, "MobiStore: Achieving Availability and Load Balance in a Mobile P2P Data Store (poster paper)", *The 6th International Conference on Mobile Computing, Applications and Services (MobiCASE)*, November 2014

Susan Juan Pan, Mohammad A. Khan, Iulian Sandu Popa, Karine Zeitouni, and Cristian Borcea, “Proactive Vehicle Re-routing Strategies for Congestion Avoidance”, *The 8th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2012)*, May 2012

I dedicate this thesis to Kazi Razaul Haque, Mohammad Asaduzzaman Khan for all of their encouragements at the most difficult times and my wife, Humyra Reza for sharing all my difficulties together with love.

ACKNOWLEDGMENT

I would like to express my heartfelt gratitude to my Dissertation Advisor, Dr. Cristian Borcea. Without his continuous support with writing, thinking, idea forming, and motivation I would never be able to come this far. He is one of the most visionary, knowledgeable, and polite professors I have seen. His amazing personality left a permanent impression in my mind.

My most sincere thanks to Dr. Narain Gehani, Dr. Xiaoning Ding, and Dr. Reza Curtmola for their constant help with my research. Every week, in hours-long meetings, they endured my inabilities to form clear ideas and took their time and effort to clarify things to me. Especially, Dr. Narain Gehani spent lots of weekends and late nights to help me with my research. I am very grateful for all the time he took out of his busy schedule just for me.

Special thanks to Dr. Adriana Iamnitchi for agreeing to become a member of my Ph.D. committee despite her busy schedule. My visit to her research lab during the first year of the Ph.D. program and discussion of her research ideas left a significant influence on choosing my research topics.

Special thanks to my fellow labmates Hillol Debnath, Nafize Paiker, Pradyumna Neog and the others in the Networking Laboratory for the stimulating discussions and all their support in equipment setups and Android programming.

Most importantly, none of this would have been possible without the love and patience of all my family members.

TABLE OF CONTENTS

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

With mobile devices (e.g., smartphones and tablets) becoming the personal devices of choice for most people, the amount of mobile user-generated content and mobile sensing data are increasing daily and at a rapid rate. Innovative apps are using this data to provide new and rich experiences to the users. For example, users can run applications that tell them the restaurants visited by their friends in a particular city, search for a lost child face among the photos taken by people nearby, or find out about traffic congestions along their driving routes. Currently, these applications are implemented using the simple client-server model. Service providers offering these services have the ability to collect and analyze user behavior and content, which they can monetize by selling to the advertisers.

Instead of letting service providers collect user data, many users would like to own and control their data. They may, however, be willing to share the data with communities defined by friendship, similar interests, and geography. This scenario lends itself naturally to distributed mobile computing which enables direct collaboration among mobile users.

Designing an efficient collaborative mobile computing platform is challenging for two reasons. First, the computing resources are limited. Second, there is no central authority to maintain efficient and fair usage of the limited resources. Solutions to these issues have been proposed in the literature. However, recent advancements in mobile computing, social networking, and cloud computing extended the area of collaborative mobile computing and introduced new challenges. In this dissertation, we explore the new challenges associated with collaborative mobile computing and provide solutions for these challenges.

The rest of this chapter shows example applications, in Section 1.1, which can benefit from our research. Section 1.2 describes new issues and approaches for collaborative mobile data sharing. Section 1.3 discusses the usage of network centrality for designing collaborative storage for peer-to-peer online social networks. Finally, Section 1.4 shows programming abstractions for integrating cloud resources with mobile resources for cloud-assisted, collaborative mobile apps.

1.1 Example Applications

Many novel apps can take advantage of collaborative mobile computing blended with social networking and cloud computing. The following examples illustrate the valuable services people can get without sacrificing the control of their data to third parties.

Road Transportation. Users can share data about traffic such as speed, road condition, traffic congestion, etc., with other users. This data could be used for navigation. This type of data contains privacy sensitive information. People may feel comfortable in allowing the use of this data for collaborative route calculations, as opposed to giving the data to centralized authorities. People may choose to store the data stream in the cloud (paid by them) before sharing to minimize the data loss associated with ad-hoc/peer-to-peer networks and to perform efficient computation.

Mobile geo-social recommender systems. Recommender systems suggest local business/amenities to people based on their preferences and stored data about local businesses. A user may search for nearby restaurants or shops ranked high by other users or her friends. The required data is stored in the collaborative storage over time by the users. This type of data is tagged with location and time. Without a collaborative storage system, users would be forced to give up control and ownership of the data to the centralized service providers.

Collaborative mobile sensing and filtering. Mobile people-centric sensing is becoming widespread. For example, people can share ambient sounds picked up by cell phone microphones with friends and find quiet places in a city, restaurants with specific noise levels, or local businesses playing certain types of music [1]. These apps need data from sensors, which is privacy sensitive. Analyzing this data using collaborative storage and computation could be better than handing it over to central authorities.

Local real-time search. An app can find a user's nearby friends in real-time. Mobile users can update their location and perhaps direction/speed through a collaborative storage system and find each other in real-time. The collaborative storage can help minimize data loss for mobile users, which can happen in mobile ad-hoc network solutions.

1.2 Collaborative Mobile Data Sharing

Collaborative data sharing has been an attractive research area for decades. Most existing research is focused toward mobile devices connected using ad-hoc or limited connectivity. Some research is focused toward reducing the computation and communication burden of the mobile devices. However, recent improvements in computing capabilities of the mobile devices and pervasive wireless access networks can be used for designing better collaborative mobile data sharing systems.

We explore a collaborative mobile data sharing system which assumes that mobile devices are connected over the Internet and are able to use more CPU cycles, memory, network bandwidth than earlier devices. New challenges come from two primary areas: (1) higher churn due to short wireless sessions, which are the results of mobility (e.g., connecting to different wireless networks while on the move, change of WiFi access points and IP address) or user choice (e.g., turning the device off to save battery power, turning the cellular data connection off to avoid usage above the contract limits, etc.), and (2) link quality variability as bandwidth and latency change drastically based on current network access point.

Despite all the advances in traditional collaborative/peer-to-peer networks, there is still a dearth of efficient solutions for collaborative mobile data sharing. For example, we are not aware of any implementation that works efficiently on smart phones and has good availability, scalability, and latency. Partially, this could be due to the lack of killer apps. However, this is also due to the fact that mobile devices are different from wired devices as collaborative platforms in the two primary aspects described in the previous paragraph.

Existing collaborative/peer-to-peer techniques for wired devices cannot work well in a mobile environment when used in designing storage systems. For example, structured solutions using Distributed Hash Tables (DHT) maintain rigorous geometric topologies for request routing resiliency. But these topologies require constant maintenance which substantially increases the overhead in the presence of high churn (join/leave of peers). Furthermore, churn can make routing tables inconsistent. This increases lookup latency and failure rates, and can even partition the network [2], [3], [4]. Existing structured solutions coping with churn use link delay estimations [2] and proximity. However, they cannot be used for collaborative mobile networks, as the values of these parameters change over time for mobile peers (unlike wired network devices). Also, unstructured solutions do not work in this type of environment due to their low efficiency.

Therefore, in this dissertation, we explore an efficient approach for designing a highly available and balanced collaborative mobile storage system.

1.3 Collaborative Social Content Replication

The widespread use of social networking has changed the landscape of collaborative mobile computing. Social relationships play a significant role in users' collaboration. Usually, people collaborate with social acquaintances for entertainment, search for services, recommendations, etc. These services require processing of the user-generated content as a result of social interactions. Collaborative mobile computing should take advantage of these new trends in computing.

Researchers addressed the need for social collaborative platforms with different peer-to-peer social network systems. For example, systems discussed in [5–9] are designed as an alternative to the existing online social networks and are used to share socially generated content and execute social applications over the shared content. The topology of these networked systems is defined by the social network graph of the participating users. These networks allow people to run collaborative applications with their friends in a decentralized fashion and, at the same time, prevent unauthorized accesses to their potentially private information. If designed and implemented properly, they could enable similar social applications with those currently provided by centralized platforms.

However, most of the existing works concentrate on maintaining the availability of social profile data, updating profile meta-data, privacy aspects, timely updating and propagation of short status messages, etc. Our focus, instead, is on socially-aware platforms that store and process user generated content. Specifically, we target collaborative content storage and content availability through efficient replication.

We explore a technique which uses social relationship structures to ensure fairness in distributing storage resources for content replication. We assume that users are willing to collaborate (i.e., sharing storage) for content replication with their 1-hop social connections (i.e.,friends) in the peer-to-peer network, but not with any other users. Replication has three benefits: (1) improving content availability in the face of typical churn, (2) enhancing data locality for data intensive programming, (3) accelerating the execution of social applications by running in parallel over different chunks of the replicated data.

Replicating user generated content efficiently over this social collaborative storage system is challenging for several reasons. First, for privacy reasons, users can only see their direct friends and no one can see the global social graph (i.e., the global network topology). As a result, users may not know one another, although they are sharing storage space. Therefore, they cannot measure the impact of using storage at other peers. It is not possible to coordinate the storage usage without being able to measure it. Second, the storage comes from resources donated by participating users and are limited. Therefore, storage space should be used efficiently. Third, there is no central authority to manage the reliability of the network. Therefore, it is a challenging problem to replicate data in a way that maximizes availability.

Existing solutions cannot address the above challenges. For example, researchers have studied the content placement problem extensively for centralized social networks [10, 11]. However, these solutions use the entire social graph for placing content, and thus do not work for collaborative content storage. Furthermore, they were designed for data center environments which are very different from collaborative environments regarding churn, latency, etc.

The above challenges cannot be addressed with straightforward techniques. For example, as members of collaborative social network use only 1-hop neighbors' storage and cannot see the entire social graph, a simple solution would give more replication storage to members with more neighbors; thus their local storage would quickly be allocated.

Subsequently, users with fewer friends would not be able to ensure the desired replication space as the space provided by their neighbors would be already full (the number of friends in social networks follows power law distributions).

In this dissertation, we design and evaluate an effective content replication technique that can allocate storage carefully and maintain uniform replication storage availability across the network over time. In the ideal case, the ratio of available local storage to the total network storage will be the same for every peer in the network at any time. This property ensures that members with many neighbors do not get overloaded, while members with few neighbors can replicate content with the same success rate as members with many neighbors.

1.4 Collaborative Mobile Computing with Cloud Support

Cloud computing has changed the landscape of mobile apps. All recent apps use cloud resources in some stage of their executions. Execution and communication offloading from mobile devices to their software surrogates in the cloud has proven to improve app response latency, reduce wireless communication overhead and energy consumption at the mobiles, and improve the availability of mobile apps [12–16]. These surrogates can be instantiated as virtual machines (VMs), containers, or even processes. Microprocessor manufacturers have recently started providing shielded application execution over untrusted cloud platforms [17], thus offering guarantees that the surrogates are truly personal and protected from the cloud providers [18]. Therefore, the converging model for mobile cloud computing assumes that each mobile device of a user is paired with a user-owned and controlled surrogate in the cloud.

This scenario lends itself naturally to mobile collaborative computing executed over sets of mobile/surrogate pairs. People collaborate within these apps by forming groups defined by friendship, common interests, geography, etc. Examples of distributed apps include discovering alternative routes to avoid traffic jam/congestion, finding people of interest in a crowd using face recognition techniques (e.g., a lost child), monitoring and stopping the spread of epidemic diseases, and mobile multi-player gaming.

The Avatar [19] platform provides support for such mobile collaborative apps assisted by the cloud. In Avatar, each mobile device is augmented with an avatar, which is a virtual machine (VM) in the cloud. Programming over mobile-avatar pairs is different than traditional mobile distributed programming. *First*, the end points in the computation are pairs of devices with different capabilities - mobile devices have different types of sensors and user interaction capabilities; avatars have more robust computation, storage, unlimited power, etc. Programmers need an easy way to use a common interface that leverages the capabilities of both devices. Apps need to read sensor/user inputs, but should offload most of the communication/computation to the avatars without introducing jitter in user interaction. *Second*, apps require user collaboration based on natural context such as location, time, social relationships, etc. Therefore, managing the collaboration in real time is important.

In this dissertation, we present a high-level distributed programming model that address these challenges through group-based abstractions and a middleware that hides the low-level system and networking details from the programmers.

1.5 Thesis

The thesis of this dissertation is that mobile collaborative applications can be effectively developed and deployed if they benefit from: (1) highly available and fair storage systems; and (2) simple, high-level distributed programming model.

1.6 Contribution of Dissertation

To demonstrate the thesis, this dissertation presents three main contributions:

- A mobile collaborative storage system (MobiStore), which provides high content availability, balanced workload distribution, and improved content retrieval latency for collaborative applications.

- A collaborative storage management system (Philia) that is designed for managing the replication of user-generated content in collaborative/peer-to-peer online social networks.

- A programming framework (Moitree) for cloud assisted collaborative mobile computing, which provides high-level abstractions that simplify app development and a middleware that utilizes efficiently mobile and cloud resources.

The **mobile collaborative storage system (MobiStore)** uses redundant peers and clustering to compensate for the side effects of the excessive churn and link level variability, such as low availability and skewed request load distribution. It structures the mobile network into clusters of redundant peers, each of which is called a Virtual Peer (VP). VPs maintain a network structure consisting of both algorithmically-defined and random edges to each other. The inter-VP routing information is updated using gossiping, as gossiping is simpler and more fault tolerant than fixed coordinated updates for collaborative systems. All VP members are fully connected as the cluster size is relatively small. MobiStore achieves $O(1)$ lookup operations with high probability, and its hierarchical structure makes the topology robust to churn.

The mobile peers in each VP replicate keys and data assigned to the VP. Thus, any peer can answer queries for the VP. A lazy update protocol is used to maintain weak consistency of the stored content among peers. As multiple peers can answer the same queries, the effect of individual churn is minimized. To simplify routing, VPs are assigned static IDs, managed by MobiStore seamlessly in a decentralized manner. Similarly, mobile peers are assigned static IDs at the time they first join the network, thus decoupling peer naming from IP addresses. In this way, rejoining the network incurs reduced overhead because under normal conditions mobile peers re-join the same VP they have previously left. To minimize the required bandwidth for topology management update propagation, we use a hierarchical update process. Load balancing is achieved through (1) storing data using consistent hashing over VPs, and (2) the load adaptive VP management which spreads the lookup requests randomly over the members of a VP and varies the number of peers in the VP based on the content popularity.

To manage the replication of user-generated content in a collaborative/peer-to-peer online social network (P2P-OSN), we developed **a replication storage management system (Philia)**. The main contribution of Philia is a replication method that prevents skewness in the availability of replication storage across P2P-OSN without relying on global knowledge of the network topology. The distributed replica placement algorithm ranks the neighboring peers based on a centrality metric and their currently available storage. The centrality metric provides insights about the network topology and helps measure the impact of replica placement decisions. Specifically, the centrality metric assigns a score to each peer based on its structural position in the network. Similarly, a score based on the currently available storage is assigned to each peer. The replicas for each piece of content are stored at the neighboring peers according to their ranks until all the desired replicas are stored or until no more peers are available.

We define a new centrality metric for each peer, EasyRank, which is a function of the number of friends of its 1-hop neighbors. EasyRank assigns relatively higher scores to peers whose neighbors have few friends. These are the peers who finish their storage soon and introduce skewness in storage availability. Therefore, the EasyRank-based peer ranking has the potential to reduce the skewness in storage availability and improve replication fairness/success.

We designed a **programming framework (Moitree)** to seamlessly use cloud resources with mobile devices for mobile collaborative apps. Moitree[1] includes high-level programming abstractions and a middleware that facilitates the execution of collaborative apps within groups of mobile users, with each group being represented by a collection of mobile device/surrogate pairs. In addition to runtime support, Moitree provides an API and a set of libraries for developing cloud-assisted mobile distributed apps. While the concepts of Moitree are general and applicable to any distributed mobile cloud platform, we have designed it and implemented it for our Avatar platform [19]. An avatar is an instantiation of a surrogate in the cloud for a mobile device.

Moitree provides a unified view of the mobile/avatar pairs to the programmers, thus hiding the heterogeneity and complexity of the underlying system. Programmers can use the Moitree API to access resources, without any assumption of where the code is running (mobile or avatar). The Moitree communication API offloads the user-to-user communication from mobile devices to avatars.

User collaboration in Moitree is modeled using group semantics. Groups are formed based on factors such as location, time, and social connections. The key features of the user groups are: *hierarchy*, which allows programmers to naturally organize users into groups/subgroups and manage their collaborations within different scopes; *dynamic group membership*, which updates group membership based on the current context of users (e.g., a group for “visitors of the Statue of Liberty” changes dynamically over time); and

[1] The word “moitree” is taken from Bengali, which means alliance/collaboration.

communication channels, which facilitates the communication among the members in a group and offload the communication to the cloud.

It is possible to integrate all three systems together. Both MobiStore and Philia use storage donated by users. MobiStore assumes the storage is on mobile devices. However, the storage can come from the avatars in the cloud associated with the mobile devices. Philia assumes that storage comes from PCs and cloud virtual machines, and thus it fits well with the Avatar architecture. Therefore, Moitree could integrate MobiStore and Philia as storage systems. Furthermore, Moitree could use a storage system that incorporates the features of both MobiStore and Philia. The context-aware groups created by Moitree can be used to define the VPs in MobiStore (e.g., location-based) or the social graph in Philia. In this way, Moitree would provide both storage and programming support for mobile collaborative apps.

1.7 Contributor to this Dissertation

The Moitree platform was designed and implemented collaboratively with my colleague Hillol Debnath. My contribution is the programming model, the high level abstractions, and the group management system. Hillol designed and implemented the Moitree middleware architecture and API, with the exception of the group management system. In order to understand my contribution, the whole platform is presented in this dissertation, including Hillol's part.

1.8 Structure of Dissertation

The rest of this dissertation is structured as follows. Chapter 2 reviews related works. Chapter 3 describes MobiStore, a collaborative mobile data sharing system. Chapter 4 presents Philia, a collaborative replication storage system for content generated in P2P-OSN. Chapter 5 describes the Moitree programming abstractions and middleware for cloud assisted mobile collaborative apps. The dissertation concludes in Chapter 6.

CHAPTER 2

RELATED WORK

In this chapter, we shall discuss related works with regards to mobile collaborative computing and storage. Specifically, the first three sections discuss work related to MobiStore: Section 2.1 presents related work for traditional collaborative and peer-to-peer (P2P) systems connected by wired networks; Section 2.2 describes related work for collaborative computing in wireless systems; and Section 2.3 discusses related work on load balancing. We then present work related to Philia, as we describe social and socially-aware collaborative computing systems in Section 2.4. Finally, the last two sections focus mostly on programming aspects related Moitree and, to a less extent, MobiStore: Section 2.5 discusses collaborative computing and storage in ad-hoc networks; and Section 2.6 presents work on cloud support for mobile computing.

2.1 Wired Systems for Collaborative/Peer-to-Peer Computing

Most of the earlier collaborative computing platforms used P2P protocols designed for peers connected over wired networks. Their main goals were maintaining scalability and fault tolerance. A few well-known systems among them are: Chord [20], Pastry [21], CAN [22], and Tapestry [23]. These protocols support efficient content storage and retrieval and have robust routing mechanisms, but they have problems with high churn. It has been shown that these protocols increase the routing latency or even partition the network under churn [2, 3], which eventually leads to failures in collaborative computing.

To overcome the problems of excessive churn, [2] uses link delay estimations for predicting churn and takes pro-active steps. However, for mobile peers, the link delay is difficult to estimate as it changes dynamically with the point of attachment. In addition, this solution depends on fixed IP addresses to map content to peers. For mobile peers, the IP addresses change quite frequently. Therefore, this solution is inadequate for mobile collaborative computing.

Several P2P file systems, such as Past [24] and Oceanstore [25], have been built on top of DHT-based P2P protocols [21, 23]. These systems inherit the problems caused by churn that is associated with their routing protocols. In addition, the overhead of building complete file systems can be an overkill for typical mobile applications.

The principle of using multiple physical peers and keeping them as a group (similar to VPs in MobiStore) has been discussed in mDHT [26] and Kelips [27]. mDHT uses all the peers in a subnet as one super peer in the DHT ring. This facilities the use of the default Ethernet link-level multicast to optimize the network traffic. This idea is not suitable for mobile peers communicating over the Internet. Kelips's main goal is to make routing fast with O(1) complexity. The peers in Kelips store a large number of mappings between IP addresses and peers, as well as the mappings between files and IP addresses. While churn is not an issue for the entire group, it becomes a problem if a peer fails to store a file or disconnects abruptly. Unlike MobiStore, Kelips has no techniques to maintain content availability in the network.

2.2 Wireless Systems for Collaborative/Peer-to-Peer Computing

Mobile robust Chord (MR-Chord) [28], MChord [29], and opChord [30] are developed to solve the mobility-related issues. They design P2P networks to improve routing success and decrease lookup latency. MR-Chord [28] improves Chord's finger maintenance process by using failure statistics to pro-actively update finger entries in real-time. MChord [29], designed for mobile ad hoc network environments, employs several techniques, such as vigorous updates of the finger tables using information from every possible snooped message, exchanging entire finger tables, etc. opChord [30] maintains secondary sets of finger tables to point at additional peers and cover additional regions in the ring space, which are used for improving routing efficiency.

However, all these works still maintain multi-hop routing information. Our goal in MobiStore is to maintain 1-hop routing information, which has the potential to improve availability by minimizing the number of failures during request forwarding. MobiStore reduces the impact of the inconsistency due to high churn by using redundancy and randomization, while correcting the inconsistency periodically.

C-Chord [31] and Chordella [32] explore different aspects of P2P systems. C-Chord [31] is to localize P2P traffic among the users of the same base station to reduce network traffic. This system works only for cellular users. However, this solution is not general enough for a mobile P2P platform, because most users employ WiFi most of the time, especially when transferring large amounts of data. Chordella [32], on the other hand, uses PCs as super peers to maintain the DHT ring and use mobile devices to connect to super peers. Unlike Chordella, MobiStore provides a mobile-only solution that does not require the help from PCs, which introduce an extra complexity layer that may preclude simple deployment. In addition, unlike these systems, MobiStore improves the content availability and balances the load of the system.

Systems in [33, 34] assume that mobiles communicate with each other over the Internet and form P2P networks for data sharing and distributed computations. However, they have significant problems in larger scale networks, such as latency (due to cellular communication), availability (due to the change in IP addresses and resource limitations), and load balancing (generally not considered in the design). MobiStore solves these issues through a hierarchical and highly available network structure, which works well in the presence of short wireless sessions and resource limitations.

2.3 Load Balancing

Hierarchical P2P systems assign special roles to some peers. Authors in [35] proposed a load balance scheme for P2P file search using a three-level hierarchical P2P network. The lowest layer stores content and the top two layers store information about their bottom layers. The indirections in the top layers are exploited to balance the load. However, this type of architecture requires special maintenance for top-level peers. Hivory [36] uses a multilevel hierarchy designed as a tree of Voronoi planes to store multi-attribute information. Synapse [37] proposes protocols to interconnect and provide collaboration among heterogeneous overlay networks. In MobiStore, we exploit hierarchy to minimize the topology maintenance traffic, improve resiliency in routing, and better manage the redundancy. MobiStore does not assign special roles to any peers to make the solution more resilient to churn.

A few solutions discuss the load balancing problem for P2P networks. Chord [20] assigns several logical peers for one physical peer. This provides finer-grain control over the mapped key values, which can be exploited to balance the load. Further improvements of this idea using different methods of moving logical peers have been discussed in [38, 39]. These approaches have two problems. First, all the logical peers act as an actual peer and generate too much background traffic. Second, the effect of churn increases. Therefore, these solutions are not suitable for mobile peers.

2.4 Collaborative Social and Socially-Aware Computing

Authors in [6, 7] investigated the use of P2P platforms to manage the basic online social networks operations: profile meta-data storage and retrieval, privacy management, etc. Their main focus is to maintain availability and reduce access latency for the social profile metadata. These systems are optimized for managing a small amount of data, such as profile content, privacy rule metadata, etc. They are not suitable for efficient storage of user generated content, which is the focus of Philia.

Authors in [5] use network centrality to efficiently project a social networking graph over a P2P network. They partition the social graph into small communities and assign them to physical peers. This work needs to analyze the entire social graph before mapping it to the peers, and it uses centrality to efficiently store the social graph. Philia, on the other hand, uses centrality to efficiently store user generated content.

Several works investigated another important operation of social networking platforms, timely and successful propagation of new content advertisement and profile meta data updates. SocialCDN [8] discusses several distributed caching mechanisms to efficiently distribute these updates for decentralized social networks. Mobiclique [40] and Contrail [41] optimize social update propagation for mobile environments. However, none of these research projects is connected with user generated content placement issues.

Placing the content over any trusted peer (not just friends) has been discussed in PrPl [42], Prometheus [9], Diaspora [43], Confidant [44] and Vis-a-Vis [45]. All these research projects assume that peers use storage from trusted peers and can retrieve the content. However, they do not describe how the content is placed, how to manage storage efficiently, or how to maintain content availability, which are essential aspects in Philia.

Using network centrality for replica placement in P2P networks has been discussed in [46]. This work uses peer ranks to place replicas for CDN and VoD systems. However, this solution targets peer-to-peer based CDN. It is not designed for mobile systems and socially generated content.

FriendStore [47] uses trusted peers for providing backup services. It uses social relationships as an incentive to encourage users to behave properly. A user selects a storage place based on the users that they trust and are using their storage most. Social-Cloud [48] uses social relationships to facilitate the sharing of storage resources. It uses a virtual credit model for regulating fair exchange of storage resources. Users set the price for their storage and use auction to trade storage with friends. This model helps achieve fairness in trading. Both [47] and [48] maintain fairness in storage donation and usage for the individual peers. Philia's goal is to achieve social fairness - everyone can store social content irrespective of how much they are storing for others. Therefore, Philia has to carefully place each replica such that everyone can store data, as long as there is storage space available in the network. In Philia, we want everyone to generate more content and receive novel services based on this content; eventually, this will encourage users to donate more storage. Another replica placement system for general P2P networks is Farsite [49], where peers can access any other peer in the network. In contrast, Philia targets a collaborative system where peers can access only their 1-hop neighbors.

Several projects discuss content placement for centralized social networks. The work in [10] discusses replica placement from a scalability perspective. This system minimizes the number of replicas and ensures that all the social networking operations access local disks to keep latency low. Another work in this area is S-Clone [11], which has slightly different goals - use a fixed amount of storage and place fewer replicas. Both of the above systems use the global social graph to find good partitions of the social graph before placing replicas. Therefore, these solutions are not feasible for a P2P-OSN, where peers do not have access to the global social graph.

2.5 Ad Hoc Mobile Collaborative Computing

In the past decade, there have been several attempts to leverage DHTs in mobile ad hoc networks and sensor networks [50–53]. Similarly, several projects have proposed data management and services over opportunistic mobile networks [54–57]. Due to the high volatility nature of these networks, the proposed solutions cannot acquire a global view of the network. Thus, their algorithms must be localized. This is good for scalability, but it degrades performance in terms of availability, latency, and load balancing. MobiStore solves these issues through its Internet-based mobile P2P solution, which allows peers to acquire a weakly consistent global view of the network.

Several works addressed mobile collaborative programming approaches over ad hoc/opportunistic mobile environments [58–61]. Among these works, LIME [61] and TMACS [58] propose group abstractions and middlewares for purely MANET environment. LIME [61] provides a framework in which mobile agents can form groups based on context-awareness. Our cloud assisted mobile collaborative programming framework (Moitree) has two main advantages over LIME: more flexible communication abstractions and the support for the middleware to perform transparent dynamic group management. TMACS [58] proposes an object-oriented distributed middleware framework for MANET. However, in Moitree, groups are defined based on users and their activities, rather than the types and scopes of objects as in TMACS. This makes mobile distributed programming simpler and more natural.

Melon [62] is a general purpose coordination language for MANET that supports asynchronous exchange of persistent messages. Although MELON provides an API similar to Moitree, it does not support group management or different types of communication between group members.

Pogo [63] and MobiSoC [64] are designed for specific areas of mobile computing. Pogo [63] proposes a middleware for distributed mobile phone sensing. Unlike Pogo which focuses on sensing, Moitree provides a general programming model for mobile collaborative computing. Furthermore, Pogo does not explicitly use group abstractions. Also, the assignment of mobile sensing devices to a particular researcher is done by an administrator in Pogo, while Moitree groups are handled dynamically by the middleware. MobiSoC [64] supports mobile social computing and provides a high-level API based on people and places, similar in nature with the one provided by Moitree. Both platforms use groups as main abstractions. But unlike MobiSoC which maintains global state about communities at the server-side, Moitree provides a distributed architecture in which apps work in peer-to-peer fashion. Furthermore, MobiSoC focuses on mobile social apps, while Moitree enables general-purpose mobile distributed apps.

2.6 Cloud Support For Mobile Computing

Assisting mobile devices with cloud resources is a very active research area [12–16]. These works offloads part of computation from the mobile devices to the cloud. Sapphire [16] is a distributed programming platform for mobile cloud applications that separates the application logic from the deployment logic. Thus, programmers can modify distributed application deployments without changing the application code (e.g., change the caching behavior).

Some researchers have investigated cloud support for mobile distributed computing [15, 16]. Clone2Clone [15] offloads peer-to-peer networking to the cloud to support efficient communication among mobile users.

Unlike these works, Moitree solves different problems - offloading both the computation and communication to the cloud and providing high level programming abstractions in order to build mobile distributed apps based on dynamic context such as social groups, time, and location.

2.7 Summary

This chapter has presented related work, focused on collaborative, peer-to-peer storage, and computing. We covered both traditional systems connected by wired networks and wireless systems. We illustrated the problems with existing systems and emphasized the novel contributions of MobiStore, Philia, and Moitree.

CHAPTER 3

MOBISTORE: P2P MOBILE DATA SHARING SYSTEM

In this chapter, we shall describe the design, implementation, and experimental evaluation of MobiStore. Section 3.1 describes the basic network architecture and assumptions. Section 3.2 presents the components of the design including the reasons for choosing this design. Section 3.3 shows the experimental results and provides insights based on these results.

3.1 Structure of MobiStore Network

Figure 3.1 shows the high level structure of MobiStore, in which the peers are divided into virtual peers (VPs) connected using a robust topology. A VP is a clique of peers (i.e., fully connected peers) which replicate the stored content of the VP and, thus, provide availability and load balance. The topology is formed by two types of interVP edges: (1) edges to maintain a Chord-like ring [20] for guaranteed topology update performance ($O(\log N)$), and (2) random edges to further improve the topology update performance. Unlike traditional P2P protocols, these edges are not used to route store/lookup requests; they are used just for topology maintenance. Each peer in a VP maintains this topology information in the VP finger table.

VPs periodically exchange their finger tables (i.e., done by the peers inside VPs), merge the information received from others, and form their aggregate routing tables. Therefore with high probability, these tables contain routing information to all the other VPs, thus making routing in MobiStore work in $O(1)$ with high probability. In rare situations, due

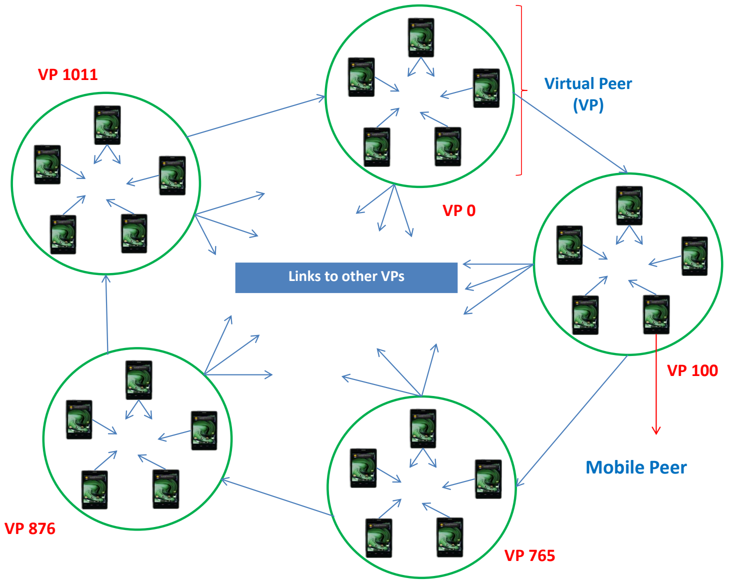
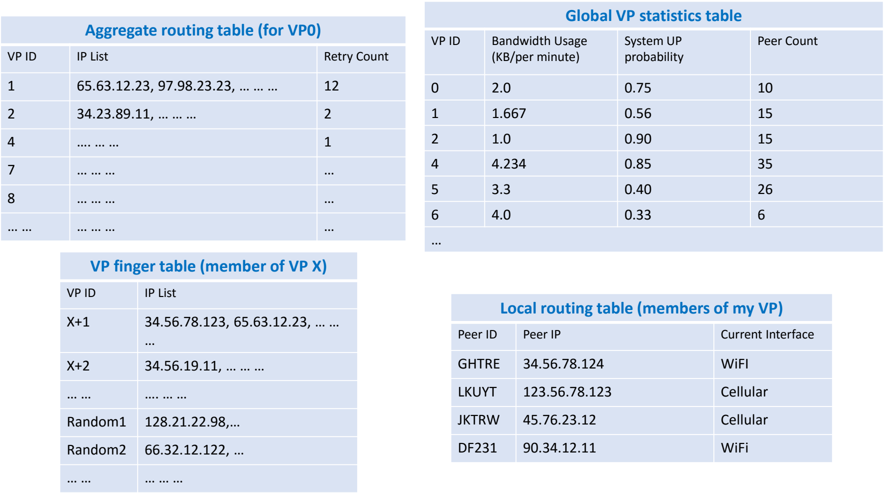


Figure 3.1 Structural overview of MobiStore.

to delays in topology update information synchronization, the routing will work in $O(\log N)$ using the ring. In this process, the peers minimize the routing latency and increase topology stability at the cost of slightly higher computation, communication and storage resources than existing P2P protocols.

Figure 3.2 shows the data structures (stored in each peer) used to maintain the topology and routing information: the *local routing table* is used for local routing inside VP, which includes the current type of Internet connection (WiFi or cellular); the *VP finger table* is the topology information table formed over VPs using the Chord-like ring and the randomized edges; and the *aggregate routing table* is formed by merging the VP finger tables received from other VPs. The retry count column in this table is used to asynchronously update the routing entries (without waiting for the update intervals). Every time a request sent to a VP fails, the associated retry count is incremented. If the retry count reaches a predefined threshold value, a request for the current local routing table is sent to all the members found



Aggregate routing table (for VP0)

VP ID	IP List	Retry Count
1	65.63.12.23, 97.98.23.23, … … …	12
2	34.23.89.11, … … …	2
4	…. … …	1
7	… … …	…
8	… … …	…
… …	… … …	…

Global VP statistics table

VP ID	Bandwidth Usage (KB/per minute)	System UP probability	Peer Count
0	2.0	0.75	10
1	1.667	0.56	15
2	1.0	0.90	15
4	4.234	0.85	35
5	3.3	0.40	26
6	4.0	0.33	6
…			

VP finger table (member of VP X)

VP ID	IP List
X+1	34.56.78.123, 65.63.12.23, … … …
X+2	34.56.19.11, … … …
… …	…. … …
Random1	128.21.22.98,…
Random2	66.32.12.122, …
… …	… … …

Local routing table (members of my VP)

Peer ID	Peer IP	Current Interface
GHTRE	34.56.78.124	WiFi
LKUYT	123.56.78.123	Cellular
JKTRW	45.76.23.12	Cellular
DF231	90.34.12.11	WiFi

Figure 3.2 Routing and management information maintained by the peers.

from that entry. The existing members will reply with the correct routing table while the non-existent members (who were responsible for the errors) will not answer, thus correcting the entry. Then, the entry is updated with the new IP addresses.

The *global VP statistics table* stores dynamic system statistics to maintain availability and load balance. The statistics include bandwidth usage over the VPs, average up-time of the VP members, and VP member counts. For example, when a new peer joins, it should be added to a VP with a lower number of peers or to a VP with low average up-time. The bandwidth column in this table is used for balancing the bandwidth usage load among the peers.

Inside VPs, peers use peer-to-peer rumor propagation to synchronize these tables before sending updates to other VPs. The rumor propagation reduces the amount of bandwidth used for synchronization and provides reliability to churn. Although VPs contain many peers, only a few peers from each VP send updates to other VPs (VP-VP communication) to maintain scalability and save bandwidth.

Having multiple mobile peers assigned to each VP allows MobiStore to provide high availability of stored content through replication. Key-values are mapped over the VP identifier space. All the members of a VP store the same key-value pairs. For example, in Figure 3.1, if key K_1 maps to VP 100, all the members of VP 100 will store the values of K_1. Thus, any VP member can answer queries for its VP.

Load balancing is achieved by spreading the requests uniformly over the peers of a VP. If the load on each peer of a VP becomes too high, MobiStore dynamically adds more peers in this VP; these peers are taken from lightly-loaded VPs. Load balancing is done without using centralized authority and is described in Sub-section 3.2.4.

3.2 Core Design Elements

This section discusses the core design elements of MobiStore: (1) how do peers join the network and what happens when they leave? (2) how to set the number of peers in a VP? (3) how to maintain the network topology and the routing tables at peers? (4) how to provide load balance? (5) how to store and retrieve content?

3.2.1 Peer Join/Leave

A peer needs to know at least one network member to join the system. A peer receiving a join request checks the total number of peers in its own VP and, in case the number is lower than a threshold value, the new peer is added to the same VP. Otherwise, the peer receiving the join request uses its global VP statistics table to find a suitable VP for the new peer. First, it attempts to assign the new peer to a VP which is suffering from low availability (found from the statistics update packets). If the availability is within limits, the new peer is assigned to the VP which is currently receiving the highest rate of lookup requests. In this way, MobiStore is able to use the new peers to improve availability and load balance.

The new peer receives the ID of its VP and the list of other VP members. Then, it sends a join request to any of these members. Finally, it creates its fixed random ID that decouples its naming from IP addressing and sends the ID to the other members of the VP. The new peer receives a copy of the routing table as well as the statistics table from the other peers.

The VP which added the new peer splits into two VPs if the joining of the new peer exceeds the maximum allowed number of peers in the VP. One of the newly created VPs remains in the same position of the Chord-like ring, and the other one randomly selects a position between the existing VP and its current successor. The two VPs and the current successor update their routing tables accordingly and send the information to everyone in the next global update interval. At the same time, the new VPs add a number of random links to other existing VPs.

The above mentioned steps work well for network formation. A new peer starts the network with just one VP and waits for new members. New peers are added to the same VP until the maximum threshold value is reached. After that, any new peer join request initiates a VP split and executes the steps mentioned in the previous paragraph. As more members join the network, these two VPs split again. The process continues over the lifetime of the system.

When a peer leaves the system gracefully, it sends a notification to the members of the VP. The members receiving the request remove its IP from their lists. It is assumed that a peer which changes its IP will inform the other members about the change. To collect system and network statistics, peers exchange updates including their IP addresses periodically.

Therefore, if a peer leaves the system suddenly without informing the other peers, these peers will notice after a while from the statistics; alternatively, they notice when a request to that peer fails.

3.2.2 Number of Peers per Virtual Peer

One of important question in our design is "how many peers should form a VP?". The answer depends on the designers goals. It is possible to set the peer count to satisfy certain availability and load balance criteria. In the following, we discuss two heuristics to determine the number of peers in a VP.

Satisfying the availability criterion: The number of peers per VP to maintain a certain availability of stored content can be found by using the following equations. Let, the probability of finding an individual peer up and running be up, and the probability of a successful retrieval at any time be P. Then

$$P = 1 - (1 - up)^m \tag{3.1}$$

where m is the number of peers in the VP. Thus:

$$m = \frac{\log(1 - P)}{\log(1 - up)} \tag{3.2}$$

Therefore, we can determine m, the number of peers in a VP, by fixing P to a value such as 0.99. The value for up can be found from system statistics.

Satisfying the load balance criterion: We fixed the peer resource utilization (computation, network bandwidth, etc.) and determine the number of peers per VP as a function of the offered load in order to maintain roughly the same load for each peer. A VP can be modeled as an M/M/m queue. The number of peers in the VP is m. Then, the peer usage is:

$$Util = \frac{\lambda}{up * m * \mu} \tag{3.3}$$

Here, λ is the incoming request rate for a VP, μ is the average serving rate for a peer, and up is the probability of a peer being up. It is possible to determine m by fixing the other parameters. For example, to find a system where every peer roughly receives the same workload, we determine m by fixing peer utilization, $Util$, to a predefined value and finding λ, μ, and up from the dynamic system statistics. To determine m in a system where everyone roughly spends the same fraction of bandwidth, we can fix $Util$ as the fraction of the total data capacity limit, μ as the serving rate of the bandwidth, and the other parameters the same as before.

In our implementation, we use the first heuristics (satisfy the availability criterion) and then dynamically change the peer count per VP to keep the average bandwidth usage rate for answering the queries constant. In a different implementation, one could also use the second heuristic to optimize for load balance.

3.2.3 Topology and Routing Table Maintenance

MobiStore maintains the topology, routing tables, and network statistics using hierarchical updates, which improve scalability and fault-tolerance. The data structures used in this process are shown in Figure 3.2. Updates are periodically exchanged and synchronized among the peers inside VPs. Then, VPs exchange the synchronized updates periodically as well. The update intervals for the peers inside VP (intraVP updating) have shorter duration than update intervals among the VPs (inter-VP updating). Therefore, peers inside the same VP can synchronize the information among them before sending it to other VPs.

In the following, we describe how the updates are disseminated, how the individual tables (Figure 3.2) are updated, and how the update process is made robust to churn.

Intra-VP Update Dissemination Intra-VP updates use a gossip-based dissemination protocol. Every peer sends updates to randomly selected $\log M + c$ peers periodically (i.e., every local-VP update interval). M is the number of peers inside the VP and c is a constant. Peers always send most up-to-date tables.

Inter-VP Update Dissemination Inter-VP updates are sent to $\log N + c$ other VPs periodically (i.e., every global update interval); $\log N$ are the Chord finger entries and c are the random edges. Let us emphasize that VP-to-VP communication means that a certain number of randomly selected peers from the sender VP (the number depends on the specific operation) communicate with a certain number of randomly selected peers from the destination VP.

Each peer sends an update with probability $\frac{p}{M}$, where p is a predefined constant and represents the number of peers from the VP that send updates at each interval, and M is the total number of peers in that VP. To maintain fault-tolerance and minimize the propagation latency, the peers send updates to q random members of each receiving VP. Selecting large values for p and q would increase the chance of successful propagation, but would use more bandwidth. Roughly, with this method, during each update interval, each VP sends $p \times q \times (\log N + c)$ packets, where N is the number of VPs and c is the number of random edges. The number of update packets is practical for current networks and devices: suppose a VP has 20 peers and $p = 4, q = 4, \log N = 15, c = 4$. Then, each VP sends 304 messages per update interval. This is roughly 15.2 messages per peer per VP, which is quite low considering the network size of 655,360 (20×2^{15}) peers. In addition, the size of update messages is always in the order of several KBs, and the global update intervals are on the order of minutes.

Updating the VP Finger Table The *VP finger table* contains Chord-based finger entries for the VPs along with entries for some random edges. The finger entries are updated according to the original Chord protocol. The random edges are updated when failures are detected, and their number is predefined and does not grow with the network size. MobiStore uses this table for VP-to-VP communication and inter-VP synchronization.

Updating the Aggregate Routing Table and the Global VP Statistics Table The base routing information from the individual VP finger tables, which are exchanged among the VPs periodically, is used to compose the *aggregated routing table*. Eventually, this table will contain entries for all the VPs, not only for those existing in the local VP finger tables. Thus, it will enable $O(1)$ routing. Let us note that the aggregated routing tables of different VPs may sometimes be inconsistent, but this issue is solved through periodic inter-VP synchronization. This synchronization is done over the VP finger table entries. Each VP sends both the aggregated routing table and VP finger table even though sending only the former is enough. This is done to improve fault-tolerance and minimize the update propagation latency. In case a VP misses some updates, other VPs are going to send those updates in the aggregated tables. Therefore, VPs need not wait until the next interval to receive the information.

The *global VP statistics table* stores dynamic system statistics for load balancing. Peers inside a VP exchange information about bandwidth usage and up time. After receiving this information from all the members of the VP, any peer can calculate the averages for the VP and update the local copy of the global VP statistics table. The global VP statistics tabled are synchronized in the same way as the aggregated routing tables.

The inter-VP updates are disseminated to every VP in the system in $o(\log N)$ global update intervals. MobiStore uses the VP finger table to propagate the updates. In a Chord based system, the update propagation tree has a height of $\log N$. In addition to the finger entries, MobiStore uses the random edges which can reduce the number of intervals needed to send the updates to everyone. For example, the random edges could be pointing to the lower level of the propagation tree. Therefore, some peers will receive the updates in less than $\log N$ update intervals.

Updating the Local Routing Table The local routing table is used for local routing inside the VP, which is achieved in $O(1)$. The table is updated each time a peer changes its IP address, which results in an update message with the new IP to the rest of the members of the VP. Therefore, all members of the VP are always up-to-date. As the member count of a VP is low, the updates are not a significant burden. Nevertheless, a number of simple optimizations are done to improve the efficiency of this process: (1) wait for a couple of minutes to detect if the current IP is stable (when the peer is on the move), (2) send the new IP to a few peers who are using WiFi and they can broadcast it, thus reducing the effect of the cellular communication latency.

Robustness of the Update Process Both intra and inter VP update processes send updates to $\log(total) + C$ peers, where C is a constant. Intra-VP updating uses gossiping, and inter-VP updating uses gossiping along with fixed edges updates (dictated by the finger table). This process has been shown to succeed in reaching everyone with high probability [65]. For example, $C = 4$ can achieve 98.18% chance of success (which is expected to be higher for MobiStore due to aggregated updates).

3.2.4 Load Balancing

Load balancing is achieved through three methods: (1) consistent hashing to store data in VPs, (2) randomization for each operation to spread the load evenly (exploiting redundancy) and to limit the effect of temporary failures, and (3) load adaptive VP management which varies the number of peers in the VP proportional to the bandwidth needed to answer the queries. Let us note that MobiStore considers load balance over longer durations (e.g., do not consider flash crowds).

Uniformly Spreading the Requests In MobiStore, each peer keeps multiple pointers to different members of each VP. Therefore, peers send lookup/store requests to a random peer in a VP for each request. Furthermore, each peer knows how much data it served and how much is served by other members of the VP (from the statistics table). In case of an imbalance, the affected peer forwards the request to a less used peer.

This method works if the number of peers in a VP is proportional to the incoming request rates. If not, MobiStore can change the number of peers in a VP, create a new VP, or merge VPs.

Algorithm 1 Load adaptive member count for VP executed by peers.

```
Data structures
Some data structures are updated by statistics and routing modules
my_vp_members: IP list of the members of this VP
routing_table: aggregated routing table
finger_table: base routing finger table
my_stat_table: local peer statistics
global_stat_table: statistics for all peers
Constant Input parameters
MAX-BANDWIDTH-LIMIT: limit after which load balance starts
MAX-ALLOWED: maximum number of peers allowed in a VP
VP-CHANGE-INT: time interval before a peer can change its VP
DEF: probability a peer waits for another peer to start a new VP
Other variables set in previous iterations or by other modules
loose-peer ← amILoosePeer(my_stat_table)
min_band ← getMinBandwidthUsingVP(global_stat_table)
max_band ← getMaxBandwidthUsingVP(global_stat_table)
max_vp_count ← memberCountMaxBandwidthVP(global_stat_table)
new_vp ← idOfMaxBandwidthVP(global_stat_table)
my_band ← myVPBandwidthUsage(my_stat_table)
last_vp_change ← curTime() - lastTimeVPchange(my_stat_table)
update_variables(global_stat_table)
if $loose-peer\&my\_band == min\_band\&last\_vp\_change>VP\_CHANGE\_INT$ then
    if $max\_band>MAX-BANDWIDTH-LIMIT$ then
        if $max\_vp\_count < MAX\_ALLOWED$ then
            for p in $my\_vp\_members$ do
                send_permanent_leave(p,my_id)
            end for
            new_vp_members ← get_member_list(new_vp)
            for p in new_vp_members do
                send_load_bal_join(my_id,p)
                receive_routing_stat_update_from(p)
            end for
            prob ← uniform_real_prob(0,1)
            if (prob > DEF) then
                wait_for_another_moverequest_from_neighbors();
            else
                start_a_new_vp_between_max_and_neighbor()
            end if
        end if
    end if
end if
```

Changing the Number of Peers in a VP Algorithm 1, shows the logic used to change the number of peers inside the VPs. During peer join, some peers (with certain probability), are treated as loose peers. Only loose peers can move from one VP to another VP. They are always temporary members of the VPs. In this way, the system can maintain stability as most peers get permanent membership to VPs. The proportion of loose peers is a design trade off: a high number could result in an unstable system, while a low number could not balance the load well.

The algorithm shows the conditions of moving a peer: the peer must be a loose peer, must be the member of a VP which currently uses the least bandwidth to answer queries, and must have changed its VP a long time ago (to amortize the content movement cost). If these conditions are satisfied, the peer moves to the VP with the highest load, as measured by its average bandwidth utilization; this utilization is retrieved from the global VP statistics table. A VP has a limit on how many peers it can keep. Once a peer decides to move to another VP, it sends a peer leave message to all its current VP members, and then joins the new VP.

While one peer is moving, no other peer attempts to move. Peers exchange this information in the statistics update packets. Moved peers check for errors after a predefined time and can roll back if errors are detected.

Terminating Existing VPs An existing VP is terminated if it had redistributed all its loose peers and it still is the VP with the lowest incoming request rate. In this situation, all the permanent VP members are distributed to other VPs. To begin the process, the VP marks itself for re-distribution and sends this information at the next global update interval to the other VPs (piggybacked on the global statistics messages). After receiving

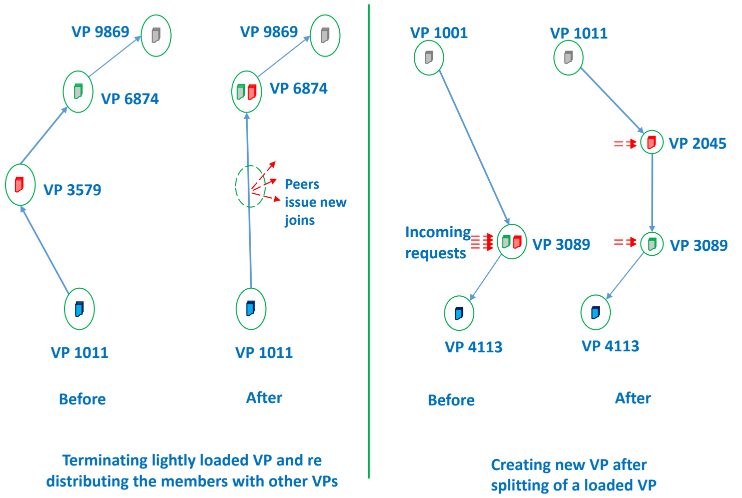


Figure 3.3 Load Adaptation in MobiStore: terminating VPs and splitting VPs.

acknowledgments from all the other VPs confirming that this marked VP was removed from the network, the peers from this VP send the stored content to the next clockwise VP and issue new join requests, as illustrated in the left part of Figure 3.3. By default, a peer submitting a join request is assigned to the VP with the maximum load. Due to inconsistent information, it is possible that more than one VP will attempt to remove itself from the network at the same time (i.e., the same global interval). In such a situation, the other VPs send negative acknowledgments and the marked VPs back-off randomly before attempting to remove them from the network again.

Creating New VPs If a VP has already added loose peers and has reached the maximum number of peers a VP can have, and it still experiencing a request rate higher than a certain threshold, it splits itself into two VPs. This process is depicted in the right part of Fig. 3. One of these VPs maintains the old VP ID, and the other takes an ID between the values of the old ID and one of its two direct neighbors on the virtual ring. The neighbor with the largest distance between the IDs is considered, and the new ID will be set to the half of this distance in the ID space. Each peer inside the splitting VP decides to join one of the two new VPs with probability 0.5. The new VP issues a join request to a virtual ring node (i.e., VP), and after the join is complete, it sends global updates to all the other VPs.

3.2.5 Content Storage and Retrieval

Consistent hashing together with the robust ring-like structure of the VPs ensure good load balance for data placement and scalability. Since recent mobile devices have large storage capacity, MobiStore assumes that peers in the network have enough storage for data sharing and does not deal with data eviction.

In MobiStore, keys and VP IDs are mapped over the same address space. When a mobile peer performs a key lookup, it uses the aggregate routing table to find the 1-hop routing entry to the destination VP. In the unlikely case that such an entry does not exist yet, MobiStore falls back to using the finger entries. This case can happen for the time between joining the network and building the aggregate routing table (which requires several global update intervals). An entry in the aggregated routing table contains several IP addresses. The peer randomly chooses one to send the request. If the requested peer does not have the content yet (due to delayed synchronization within the VP) or it has already used more bandwidth than other peers within the VP, it forwards the lookup request to another peer. In our implementation, a request can be forwarded at most 3 times.

Once new content is stored at a peer, this peer uses a gossip based protocol to send the new content to everyone in the VP. To minimize the cellular network bandwidth requirements, only peers using WiFi take part in the content update process. The peers currently on cellular network, start getting updates when they switch to WiFi.

3.3 Evaluation

We used PeerSim, a P2P Java based simulator [66] to evaluate MobiStore. Our experiments compare MobiStore with two baseline data-stores built over MR-Chord (Mobile Robust Chord) and Chord. PeerSim provides the Chord implementation. Since we could not find any publicly available implementation of MR-Chord, we implement it based on its description [28].

MR-Chord improves the finger management process of the original Chord protocol. Each time a peer experiences a routing failure, it sends a failure message to the last successful hop (i.e., the one which provided the failed peer address). Upon receiving the failure message, the last hop tries to contact the failed finger entry, and if it fails again, it replaces the failed entry with the predecessor entry in the table. Furthermore, the peers maintain statistics about success and failures in the finger tables. If the failure count exceeds the success count by two or more, the corresponding entries are requested to check for failures and update if needed.

The two baseline stores, employing MR-Chord and Chord, use the same number of peers to store the content as MobiStore. They also use the same method to replicate the stored content. Furthermore, the lookup process retries three times before declaring a failure (i.e., the same with MobiStore). Therefore, the difference between MobiStore and MR-Chord or Chord comes only from the management and structure of the P2P network.

The evaluation has four goals. We quantify: (1) the resilience to churn and availability benefits; (2) the effects of scale on availability, latency, and overhead; (3) the benefits of load balance, and (4) the update latency, which can give application developers a better idea about the type of applications that are feasible over MobiStore.

Table 3.1 Simulation Parameters

Parameter	**Range**
Number of peers	10000
Peers per VP	5 - 25
Keys stored	2^{22}
Stored value size	10KB - 1MB
Lookup rate	2 - 3 per peer, per minute
Chord base	128
Chord finger update interval	4 minutes
Random links	10
Session time (ON)	2 - 60 minutes
Session time (OFF)	0 - 20 minutes
Network delay	2 - 41ms
Avg WiFi bandwidth	8 Mbps
Avg cellular bandwidth	500 Kbps
MobiStore global update interval	2 minutes
MobiStore local update interval	30 seconds
Load balance interval	20 minutes
Mobile peer speed	0.2 - 20 meters per second
Peers using cellular	30% peers

3.3.1 Simulation Setup

Peers are initially placed randomly in a square region 40KM by 40KM. The maximum number of peers in our simulation is 6,500 due to the computation/memory limitations of PeerSim. The peers leave the network at exponentially distributed intervals with a session time ranging from 2 minutes to 1 hour. We use this limit to mimic the short session times of mobile devices. After completing an active session, the peers leave the P2P network for periods ranging from 0 to 20 minutes.

We used the BonnMotion [67] tool to generate mobility traces using the Gauss-Markov mobility model. This model maintains temporal dependency while modeling velocities and directions. For each peer initial position, the velocity and direction are chosen uniformly distributed over the simulation region (40KM by 40KM, which is divided in a 200,000 by 200,000 grid). Then the speed and direction of each peer are changed after an interval uniformly distributed between 0 and 60 seconds. The speed of the peers varies randomly in the range 0.2-20 meters per second. Each peer, on average, remains mobile for 30% of the simulation time and uses the cellular network to connect to the Internet and P2P network. The rest of the time, the peers connect to the Internet/P2P network using WiFi. In 80% of the grid cells, the peers can always connect to the cellular network and have an average bandwidth of 10 Mbps. In the other 20% of cells, the peers can connect with a probability of 0.8 and with varying bandwidth in the range of 100Kbps-10Mbps (chosen uniformly). The average bandwidth for WiFi-connected peers is assumed to be 54 Mbps. Although in real world the bandwidth values change based on different technologies, the important factor is the bandwidth ratio between WiFi and cellular network, not the absolute values.

The propagation, processing, and queuing delay are modeled together as a function of the corresponding peer resources and the Euclidean distance between the communicating peers. This includes wireless communication (last hop) and wired communication (i.e., between base stations/access points). Overall, the delay varies between 2 and 41 ms. The corresponding transmission delay (function of the packet size and bandwidth) is added to this value. The rest of the simulation parameters are listed in Table 3.1.

Comparison with MR-Chord and Chord Baselines For this experiment, MobiStore has 930 VPs, each VP containing 7 peers. The Baselines have 6,510 peers (i.e., 930 x 7). The peers randomly generate and store 2^{22} key-value pairs before the simulation begins. Each peer issues lookups for random keys at intervals exponentially distributed with a mean of 20-30 seconds. For lookups, the peers only choose existing keys. The failures consist of routing failures, peer unavailability, or delayed replication.

Availability Availability Figure 3.4 shows the lookup success rate (which measures availability) for MobiStore vs. Baselines. MobiStore has a substantially higher availability than both MRChord and Chord Baselines, especially for shorter sessions. For example, MobiStore has 92% success rate for a 15-minutes session time, while MR-Chord has 47% and Chord has 37%. Also, MobiStore has high availability for sessions longer than 20

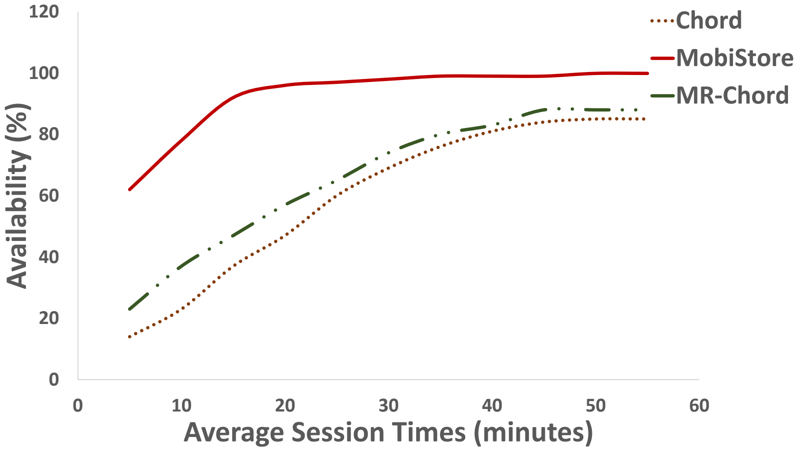


Figure 3.4 Availability of MobiStore vs. Baselines as measured by lookup success rate.

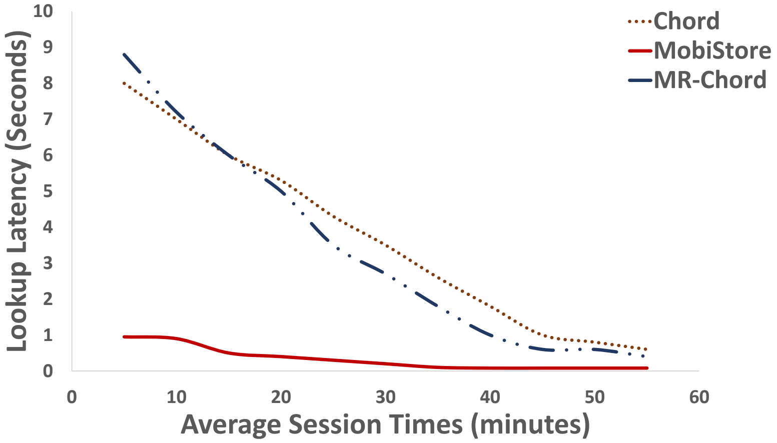


Figure 3.5 Lookup latency of MobiStore vs. Baselines.

minutes. The results corroborate what we described in the design sections: well managed redundancy improves significantly the availability of MobiStore. MobiStore ensures that the requests are routed with fault tolerance using multiple peers per VP and replicates the content among the members of the VPs. MR-Chord and Chord, on the other hand, suffer greatly from peer failures when it comes to propagation of routing information and content replication over multiple hops.

Latency Figure 3.5 shows the latency of the successful lookups. MobiStore achieves a latency as low as 9 times the latency of MR-Chord (for 5 minutes session time, MR-Chord

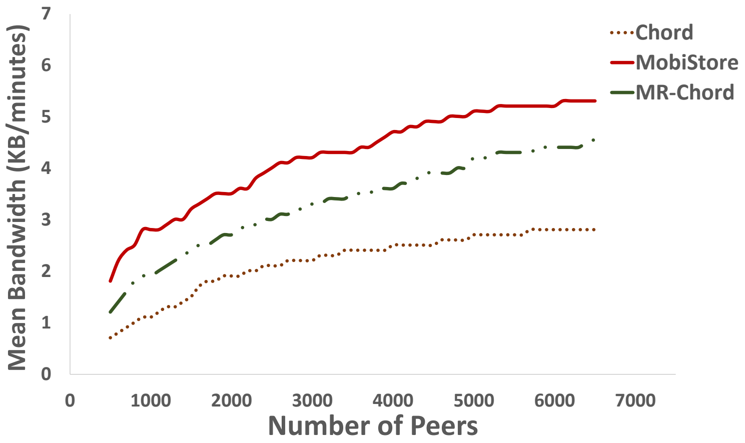


Figure 3.6 Per peer management overhead traffic for MobiStore vs. Baselines.

latency is 8.8 seconds and MobiStore latency is 0.95 seconds). This is due mostly to the 1-hop routing employed by MobiStore. We also observe that MobiStores latency is acceptable for most mobile applications. Even for an average session of 15 minutes, the latency is as low as half a second. This may not satisfy hard real-time constraints, but it is sufficient for most mobile applications.

We see that the lookup latency is higher for MR-Chord Baseline than for the Chord Baseline. The reason is MRChord increases the hop count during excessive churn while coping with failures. Therefore, for small session times, MR-Chord performs worse than Chord in terms of latency, but it increases the availability as found from Figure 3.4.

Overhead The availability and latency benefits provided by MobiStore come at the expense of extra-overhead to maintain the network structure. MobiStore propagates aggregated routing tables which consume more bandwidth. For this experiment, MobiStore used a combination of full and“dif” updates of the aggregated routing tables. Every fourth update transmits the full table to help with consistency; in between, only the modified entries are transmitted. Figure 3.6 shows that MobiStore and MR-Chord have similar growth

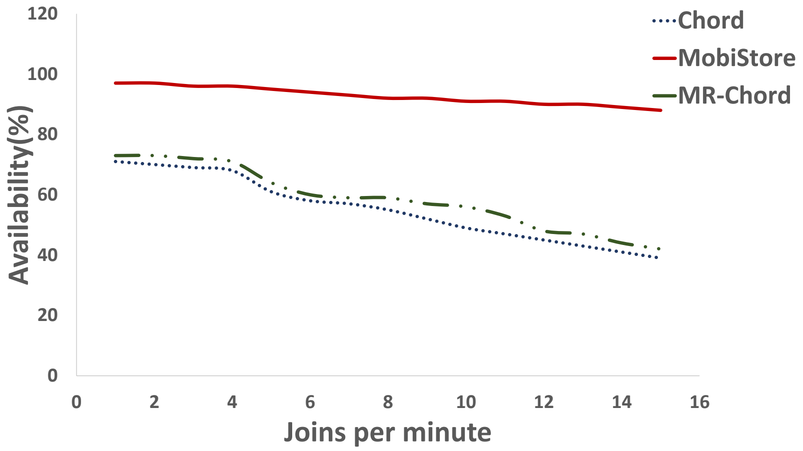


Figure 3.7 Availability (measured by lookup success rate) as a function of parallel joins for MobiStore vs. Baselines.

patterns for network management overhead. However, the absolute value of the overhead is low, each peer sending around 5KB per minute for a network of 6,500 peers. MobiStore uses around 1KB/minute more bandwidth than MR-Chord, but with this little extra overhead MobiStore can improve availability and decrease latency, which has the potential to reduce the number of data packet retransmissions and save bandwidth usage in future.

Scalability Figure 3.6 also demonstrates that the growth in the overhead with the increase of peer count is between sub-linear and logarithmic. Since the update process is hierarchical and VPs have a limit on how many peer they can have, we conclude that MobiStore scales well with the increase in the number of peers: the more peers in the network, the closer to a logarithmic function the overhead is. Furthermore, Figures 3.7 and 3.8 show that MobiStore is able to maintain the availability benefits almost constant with the increase in new peer joins (figure 3.7) or network size (Fig. 8). On the other hand, availability decreases more rapidly with parallel joins for MR-Chord and Chord Baselines. These results hold even for high rates of parallel joins or larger network size. As we stated earlier, if a peer cannot find data in its own storage, it silently forwards the request to a

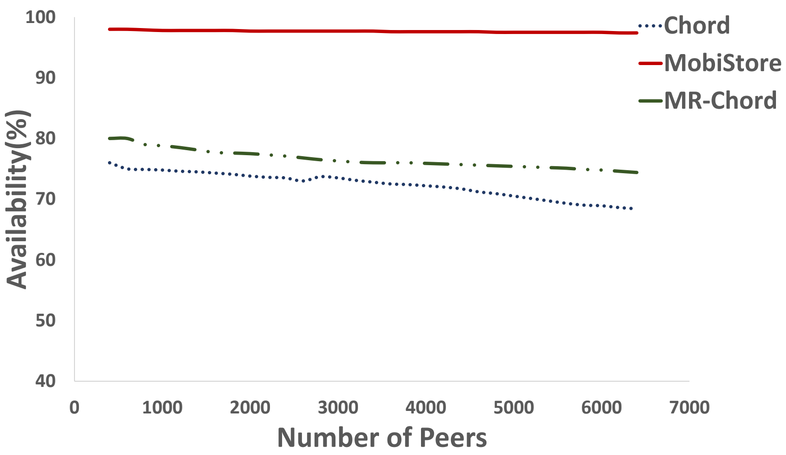


Figure 3.8 Availability (measured by lookup success rate) as a function of network size for MobiStore.

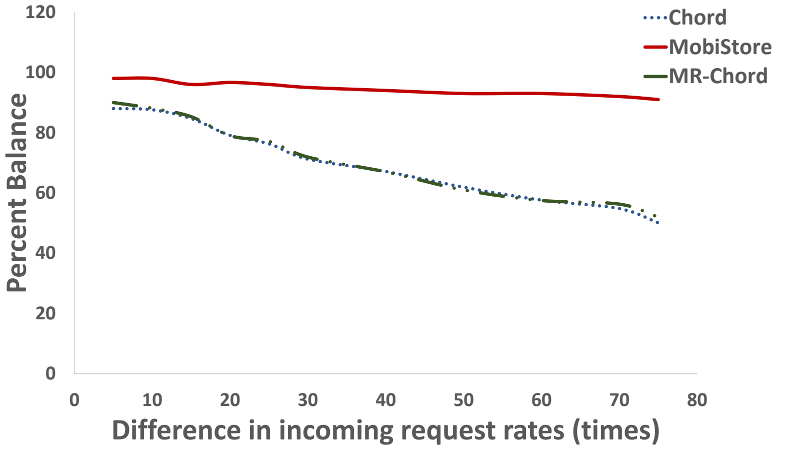


Figure 3.9 Fraction of peers receiving almost the same load ($\pm 10\%$ mean-load).

fellow peer in the same VP. Therefore, the lookup succeeds with high probability if data is present in any peer in the network for MobiStore. MR-Chord and Chord on the other hand, need the entire request forwarding path (all the routing hops) to work properly; this is affected adversely if new peers keep joining or the network size increases. Thus, based on our results, we conclude that MobiStore scales well to moderately large size networks.

3.3.2 Load Balance

For this experiment, we changed the content popularity for different stored values. The most popular content receives 5-75 times more requests than the least popular content. We used 325 VPs, with 20 peers each. The maximum number of peers per VP is 40 (after the VPs acquire additional peers), and the minimum peer number is fixed at 10. Therefore, 10 peers in each VP are loose peers (as described in the load balance discussion of Sub-section 3.2.4). For MR-Chord and Chord, we used the average balance framework present in PeerSim [66]. The average balance methods tries to keep the load equal to average values over the peers. MobiStore uses the methods described in Sub-section 3.2.2.

Figure 3.9 presents the achieved load balance of MobiStore. The fraction of peers receiving mean load 10% is higher than 91%in the worst case for MobiStore although the content popularity varies substantially. For a similar load imbalance, MR-Chord or Chord Baselines can maintain only 50% peers receiving mean load 10 almost half of the peers receive significantly more load than others. This result demonstrates the benefits of integrated load balance techniques in MobiStore. All three techniques, namely, random request distribution, request forwarding, and changing the peer count in VPs are working to achieve this balance.

Figure 3.10 demonstrate the load balance process in more details as it shows the change in VP count and per-VP peer count as the incoming request rate increases. After applying the load balance techniques, we see that the peer count difference between VPs grows up to 30 (maximum possible in this setup). We also see from the figure that the load balance effort changed the number of VPs in the system from 325 to 225. The eliminated VPs are those which received the fewest requests, and they were merged with clockwise

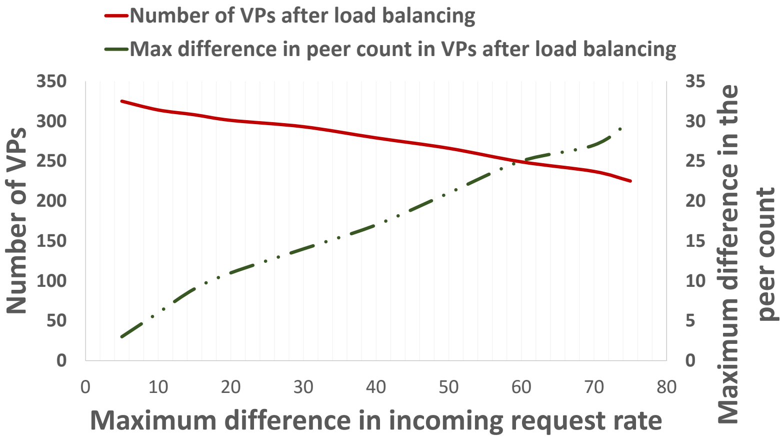


Figure 3.10 Effect of Load Balancing on the network (number of VPs and per-VP peer counts) as a function of request rate.

next VPs. These results show that our load balancing techniques can quickly adapt to load variations.

Finally, Figure 3.11 presents the effect of load balance on availability over time. The availability fluctuates, but it is not grossly affected. Ideally, the load balance should not have any effect on availability because it does not affect the stored content. Failures occur, however, due the temporary inconsistencies of the routing entries caused by moving peers from one VP to another. Nevertheless, the routing tables become consistent quickly and the availability improves.

3.3.3 Update Latency

These experiments assess both management update latency and content update latency. For both experiments, we randomly selected 10%peers to generate management data/new content. The selected peers post management data/new content in parallel. The size of management data depends on the routing table size, and the size of the content comes from

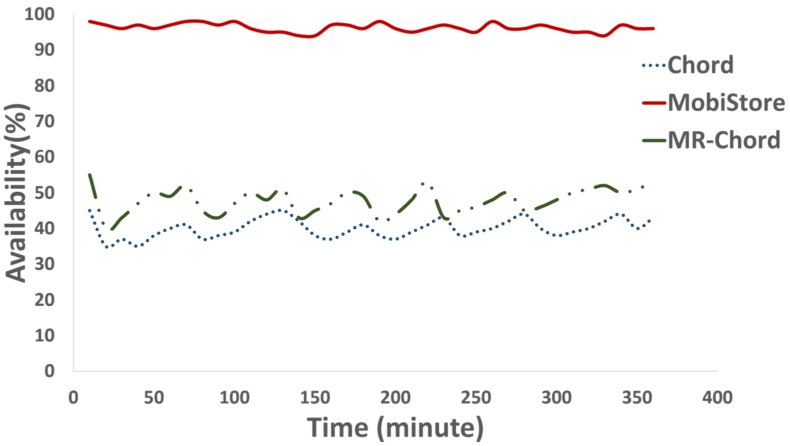


Figure 3.11 Effect of Load Balancing balance on availability over time (lookup success).

a Pareto distribution with the shape parameter set to 0.5 and the mean parameter set to 100 (the numbers correspond to file sizes in KB). We selected the file sizes in the range 10 KB-1MB as typical size of the data stored such that the simulation results are not dominated by the file size; instead, they show the effect of the number of stored files. After every update is posted, 40 randomly selected peers over the whole system attempt to retrieve the management data/content. These 40 peers do not experience churn. We recorded the time of posting the updates and the time of successful retrieval of the content by all 50 peers. The time difference is the update latency.

Figure 3.12 shows that the management update latency is less than 4 minutes for most session times. For very short sessions, it grows up to 7 minutes. In fact, the latency depends on the periodic global update interval, which was fixed at 2 minutes in this experiment. Therefore, even in the worst case scenario, the updates reach everyone in just four global update intervals. This result demonstrates the robustness of the update process. Aggregated updates along with MobiStore structure and interval selection based on local clock helped to achieve this result.

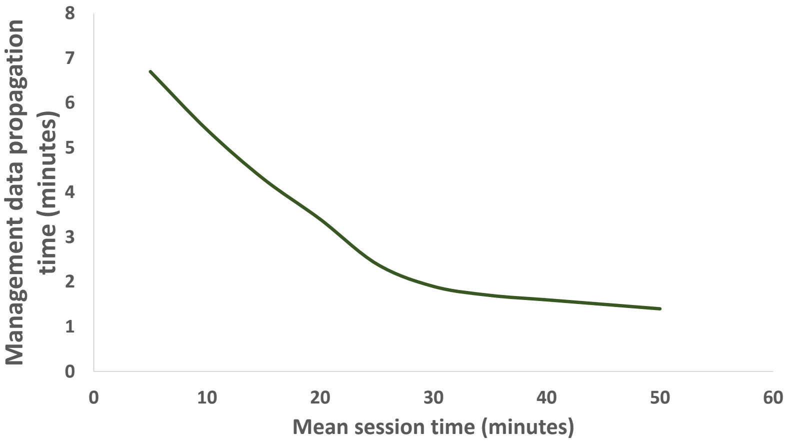


Figure 3.12 Management data update latency as a function of session time for periodic updates.

Figure 3.13 shows the content update latency as a function of the session time; the content updates are asynchronous. Once the content is posted, MobiStore starts the update process. Only the peers currently using WiFi take part in the content update process. The peers currently using cellular communication wait for the updates until they switch to WiFi. If a request for content comes to a cellular peer and the peer does not have the content, the peer forwards the request to a peer using WiFi (even without knowing anything about the content). VPs with 10-15 peers can find the content in 2-3 seconds in the worst case. Of course, larger file sizes would result in larger latency. Therefore, mobile applications are only limited by the wireless networking bandwidth. MobiStore does not increase the latency of the content update propagation; it rather minimizes the latency using silent forwarding among fellow peers of the VP.

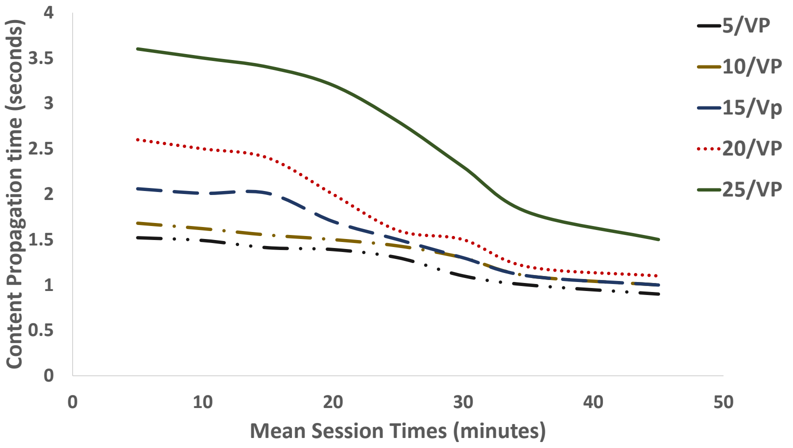


Figure 3.13 Content update latency as a function of session time.

3.4 Summary

This chapter presented MobiStore, a mobile collaborative/P2P data store for sharing user-generated mobile content. While P2P techniques are well known and understood, there is currently no good P2P solution in the mobile world. The main reasons for this situation are short wireless sessions, i.e., very high churn, and resource limitations in terms of battery and mobility related issues. The MobiStore design addresses these constraints with a new mobile P2P network structure and mechanisms able to adapt to failures and load variation. The results demonstrate that MobiStore achieves high availability, low latency, and good load balance, without incurring high overhead that could impact negatively the performance in relatively large scale networks. MobiStore is ideal for applications which can tolerate worst case key-value update delays of several seconds.

CHAPTER 4

PHILIA: COLLABORATIVE REPLICATION STORAGE FOR SOCIAL CONTENT

In this chapter, we shall discuss the design, implementation, and experimental evaluation of Philia, a collaborative replication storage system for online social networks. Section 4.1 describes the assumptions and overview of Philia's replication storage system. Section 4.2 discusses network centrality, proposes a new metric for network centrality, and shows how this metric can be used for efficient replica placement. Section 4.3 shows the replica placement algorithm. Section 4.4 shows experimental results for two real-life social graphs from Facebook and Google+.

4.1 Overview and Assumptions

Since replication requires peers to share storage with other users, we assume that users are willing to collaborate with their 1-hop social connections, but not with at any other users. There are two reasons behind this assumption which limits the number of peers available for replication. First, most sharing in social networks is done with 1-hop connections, and thus it makes sense to replicate data at these peers. This acts as an incentive for sharing because it improves application response time: the friends' content is already stored locally due to replication. Second, users might be unwilling to replicate some of their content on peers belonging to unknown people due to privacy. Replicating encrypted content could use all the peers in the network, but it slows down the applications and could be difficult to manage (e.g., key management).

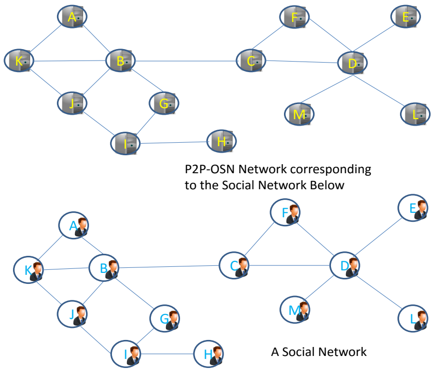


Figure 4.1 An example of social graph and corresponding P2P-OSN.

We assume that applications running over P2P-OSN decide their desired replication factor, and that the replication factor, r_f, satisfies the following condition,

$$r_f <= neighbors\_count + 1$$

A peer always stores one replica of its content locally. For example, applications may set the replication factor as follows,

$$replicas = 1 - \log(1 - desired_{availability}) / \log(1 - avg_{peer\_up})$$

Here, $desired_{availability}$ is the required availability of the content (decided by the application or the user), and $avg_{peer-up}$is the average uptime of the 1-hop peers. Peers can periodically query the neighbors and calculate $avg_{peer-up}$.

Figure 4.1 shows why replication is hard in P2P-OSN when only 1-hop peers are available to store replicas. Suppose C wants to store one replica for a piece of content. Intuitively, it should give priority to storing the replica to B or F instead of D. This is because D is the only storage option for three peers (M, L, E). However, if C completely ignores

D in all its replication decisions (when the replication factor is at most 2), this may have adverse effects on the neighbors of B and F because these two nodes receive all replicas from C. Furthermore, D may be a good candidate to replicate C's data if it receives very few replicas from its other neighbors (i.e., these users rarely generate new content).

From this example, we first observe that an effective replication scheme should consider the structural properties of the peers in the network. The replication is effective if it results in balanced storage usage across the peers over time. In the ideal case, the ratio of available local storage to total local storage is the same for every peer at any moment. That is, we always want to keep $max(\forall i \frac{AvailableStorage}{TotalStorage})$ to the minimum possible value. If we can maintain this, some peers will not finish all available storage space sooner than others. Therefore, all peers will be equally able/unable to store replicas, irrespective of their number of friends. The problem is, how to design such a replication scheme if the peers do not have a global view of the social graph? Let us recall that not disclosing the whole social graph to any peer is crucial from a privacy point of view.

To overcome this limitation, we propose to compute a localized network centrality metric, EasyRank, for each peer; the scores obtained from this metric are used to rank the peers with respect to their position in the network topology. As it will be explained in Section 4.3, the ranking effectively differentiate peers who have relatively limited storage space than others. A second source of storage imbalance among peers observed in our example is the different rate of content generation by users. Therefore, replication scheme also considers the currently available storage it can obtain from others.

Broadly, our distributed replica placement algorithm works as follows:

- Each peer updates its 1-hop neighbors with its EasyRank score and available storage, upon any local change.

- Upon receiving updates, each peer computes a compound score for each 1-hop neighbor using the reported values for EasyRank and available storage. Thus, the peer has a structural view of its 2-hop social network.

- Each peer ranks its neighbors according to the compound score.

- When replication is needed, the replicas are placed on the neighbor peers according to their rank. Higher-ranked peers are chosen sequentially (starting from the top-ranked), with a small probability of skipping peers. This random skipping of comparatively good peers is done to obtain a better global solution [68].

Existing social networking apps could be implemented over Philia. For example, SideStep lets people share travel experiences (with images and videos) and future travel plans with friends. The goal is to form a travel group and have fun with friends. This app can use Philia to replicate travel experience files (e.g., photos, videos) over the friends' storage. The replication factor will determine the efficiency of the app execution. The replication will maintain content availability for the owner and will reduce the content retrieval latency for the friends. As the content is replicated at different computers, image and video processing could be done using data parallel computation.

4.2 Balanced Replication with Network Centrality

Our idea is to use a network centrality metric to quantify the structural properties of the peers in P2P-OSN for efficient replication. This section reviews first a number of existing centrality metrics and explains why they are not expected to work well for our problem. Then, it introduces EasyRank, our new centrality metric and provides numerical examples to illustrate its benefits.

4.2.1 Existing Network Centrality Metrics

There are many centrality metrics for social networks, but our solution requires metrics that provide hints about neighbors' connections. Below, we investigate a few well-known metrics that satisfy this requirement.

Degree centrality is the node degree of a peer, i.e., the number of friends that the owner of the peer has. The number of friends captures to some extent the immediate neighbor connections, and this metric can be computed easily. However, it is inadequate for discovering social structure beyond the friend count, such as how the friends are connected with others in the network. For example, even though B and D have the same degree centrality in Figure 4.1, D's friends only option to get storage for replication is D, while B's friends have multiple options. This suggests that B and D should be treated differently.

Eigenvector centrality assigns higher scores to peers having connections with more central peers. The eigenvector centrality for peer i, c_i, is calculated in the following way:

$$c_i = \frac{1}{\lambda} \sum_j A_{ij} c_j$$

In the equation, A is the adjacency matrix, λ is the highest eigen value, and c_j is the eigenvector centrality of peer j. The eigenvector centrality of a peer is proportional to the eigenvector centrality of the neighbors. Therefore, the eigenvector centrality captures information of the network structure which spans beyond 1-hop. For example, in Figure 4.1, unlike degree centrality, eigenvector centrality will treat B and D differently (Table 4.1 shows the exact values).

A significant problem with eigenvector centrality is that its calculation needs multiple iterations to converge to reasonably accurate values. This calculation requires exchanging intermediate values around the network, which leads to high latency. In addition, churn in P2P networks may prevent reasonable convergence and stability of the calculated values. Furthermore, new peer joins or edge creations will force recalculation of the old values.

Pagerank is a variation of eigenvector centrality. While eigenvector centrality assigns higher scores to all the neighbors of a high centrality peer, pagerank adjusts the scores according to the number of neighbors. The pagerank for peer i, c_i is calculated using the following equation:

$$c_i = (1 - d) + d * \sum_j \frac{c_j}{deg_j}$$

Here, d is a damping factor, deg_j is the degree centrality of peer j, and c_j is the pagerank of peer j. For our problem, pagerank captures better the contribution of the higher centrality peers over lower centrality peers, as the contribution of each peer is scaled down with the number of neighbors. Similar to eigenvector centrality, pagerank captures information around a peer with respect to the whole network. It also requires multiple iteration to calculate a correct value and suffers from the same problems with eigenvector centrality.

Bonacich's power centrality assigns higher scores to the peers that are connected with powerless peers. According to this metric, if a peer has many friends which are not well connected, the peer is more central. This is one of the expected feature for replication in P2P-OSN because it captures the dependency among peers. It can be calculated using following formula:

$$c_i = \sum_j (\alpha + \beta c_j) * A_{ij}$$

Here, α is a normalizing constant, β indicates the importance of the neighbor's centrality, A is the adjacency matrix, c_j is the Bonacich centrality of peer j. The calculation of this metric also requires multiple iterations. Thus, it suffers from the same problems with eigenvector centrality and pagerank.

4.2.2 EasyRank

Our proposed metric, EasyRank, is designed to have similar benefits with Bonacich's power centrality (i.e., capture node dependency), and it is similar with degree centrality in terms of computation simplicity. Thus, it combines the benefits of existing metrics.

The EasyRank of peer i, r_i, is calculated using algorithm 2. Lines 2-4 calculate the centrality metric. Line 6 uses the sigmoid function to scale the centrality metric between 0 and 1. In this way, there is no need to use an expensive distributed algorithms to exchange the current values among nodes to find the normalized metrics which are necessary for comparison.

Algorithm 2 EasyRank Centrality Calculation

```
1: $r_i \leftarrow 0$
2: for all neighbors $j$ do
3:     $r_i \leftarrow r_i + \frac{1}{sqrt(deg_j)}$
4: end for
5: $r_i \leftarrow sqrt(deg_i) + r_i$
6: Normalize the rank using logistic function
7: $rank \leftarrow 1 - \frac{1}{sqrt(1+r_i^2)}$
```

EasyRank has the following benefits: (i) lightweight computation, as there is no need for distributed iterations in the algorithm; thus, it can be calculated in constant time and churn has no effect on the calculation; (ii) it captures the structural information needed for balanced replication. Peers whose neighbors do not have many neighbors, receive higher scores; (iii) newly joined peers can compute a value immediately; and (iv) new peers or connections (edges) only force the recalculation at the affected peers. Therefore, EasyRank is expected to work well within the constraints of P2P- OSN.

4.2.3 Examples of Replication Using Network Centrality

Considering the network example from Figure 4.1, we analyze the impact of different centrality metrics on storage allocation fairness, replica placement, and replication success rate.

Fairness in Storage Allocation One important requirement for storage allocation in P2P-OSN is fairness: peers should receive a similar amount of storage independent of how many neighbors/friends they have. This will lead to better global space utilization. Table 4.1 shows the storage allocation for the peers in Figure 4.1 using allocation policies based on the centrality metrics previously described. We assume that each peer initially donated 1GB of storage. Each peer divides the storage among its neighbors/friends proportionally with the centrality metric of the respective column. Each value in the table indicates how much storage each peer gets if all the peers apply the policy specified by the column. The bottom three rows of Table 4.1 show the 25-th and 75-th percentiles, and, standard deviation of the storage acquired by the peers.

We see that EasyRank is the most successful in fairly distributing the storage space irrespective of the number of neighbors the peers have. The highlighted lines in Table 4.1 show the situation of the peers having few neighbors (L and M) and many neighbors (B and D). The results demonstrate that EasyRank provides the highest amount of storage to the peers with few neighbors (M and L). It provides lower amount of storage to the peers having many friends, compared to PageRank or Degree centrality based policies. Finally the lowest

Table 4.1 Storage Space Allocated to Peers when Everyone Donates 1GB

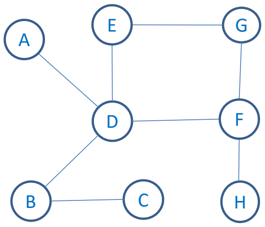
Peer	Degree	EasyRank	Bonacich	PageRank
A	0.35	0.47	0.02	0.37
B	2.61	2.11	2.49	2.56
C	0.99	0.94	1.35	0.91
D	4.04	3.91	3.70	4.10
E	0.13	0.17	0.10	0.14
F	0.41	0.50	0.72	0.43
G	0.48	0.53	0.78	0.49
H	0.17	0.25	0.30	0.22
I	1.65	1.82	2.19	1.68
J	1.03	0.94	0.95	0.97
K	0.88	0.99	0.16	0.85
L	0.13	0.17	0.10	0.14
M	0.13	0.17	0.10	0.14
25%	0.17	0.25	0.1	0.22
75%	1.03	0.99	1.35	0.97
SD	1.16	1.07	1.15	1.17

Table 4.2 Centrality Measurements for Example Network

Peer	Degree	Bonacich	EasyRank	EgVector	PgRank
A	0.40	0.15	0.63	0.57	0.06
B	1.00	1.00	0.85	1.00	0.14
D	1.00	0.80	0.89	0.43	0.17
F	0.40	0.40	0.63	0.33	0.07
C	0.60	0.02	0.75	0.58	0.09
G	0.40	0.00	0.63	0.47	0.06
H	0.20	0.23	0.40	0.14	0.04
I	0.60	0.80	0.78	0.43	0.09
J	0.60	0.00	0.75	0.71	0.08
M	0.20	0.23	0.36	0.14	0.04
K	0.60	0.44	0.76	0.74	0.08
E	0.20	0.23	0.36	0.14	0.04
L	0.20	0.23	0.36	0.14	0.04

standard deviation indicates that EasyRank-based storage distribution is the most balanced. Therefore, if the peers order their storage distribution based on EasyRank, the allocation is expected to be fair/balanced.

Replica Placement Table 4.2 shows the centrality values for the peers in Figure 4.1. Let us consider that C wants to store two replicas. It has three options, B, D and F (highlighted in the table). Peers for replica placement are chosen based on lowest centrality score. The first replica will be stored on F by all replication schemes because F has the lowest score.



Scheme 1		
From	To	Size
D	E,F	40
D	E,F	40
D	E,F	20
B	D	100

Scheme 2		
From	To	Size
D	A,E	40
D	A,B	40
D	A,F	20
B	C	100

Peer	A	B	C	D	E	F	G	H
EasyRank	0.38	0.71	0.45	0.84	0.66	0.79	0.67	0.40

Storage resource left after scheme 1 and Scheme 2									
Peer	A	B	C	D	E	F	G	H	
Storage Left (Sch 1)	100	100	100	0	0	0	100	100	38% peers (A,G,H) cannot use the system any more
Storage Left (Sch 2)	0	80	0	100	60	80	100	100	100% peers can still use the system

Figure 4.2 An Example Illustrating the Importance of Storage Balance for Replication Success Rate. Schemes 1 and 2 Show Two Policies for Selecting Peers to Store Replicas.

Intuitively, this is the correct choice because B and D have many neighbors, which may need to store replicas.

The problem becomes more complex when C needs to store the second replica. If we check the highlighted rows of Table 4.2, we see that B will be chosen by EasyRank and PageRank, while D will be chosen by Bonacich and EigenVector; Degree centrality will have a tie. Intuitively, it is clear that D must not be chosen because D has neighbors (M,L,E), which are solely dependent on it. Therefore, EasyRank and PageRank do a better job in this example.

Replication Success Rate In this example, we show how centrality-based schemes help improve the replication success rate in the network. They are able to prevent situations when certain peers cannot access any storage at their neighbors, while there is plenty of storage available in the network. Suppose each peer in Figure 4.2 donates 100MB of storage. Peers

B and D choose replication factors of 2 and 1, respectively. Suppose B stores one file of 100MB, while D stores three files of sizes 40MB, 40MB and 20MB. If we use scheme 1 to place the replicas for these files, peers A, G, and H will not be able to use the system in the future even though 50% of the global storage is still unused.

We could avoid this situation with scheme 2, which is based on EasyRank and provides more balanced storage. Scheme 2 combines the EasyRank scores with the available storage at peers in a new metric. For example, the metric could be defined as the sum of the EasyRank score of a peer and half of the unavailable storage percentage at this peer. In this case, the centrality has a higher weight than the unavailable storage. For the replicas generated at D, using this scheme, the scores of the neighboring peers will be at first [A=0.44, B=0.67, E=0.64, F=0.72] (no used space at any peer). D will choose A and E for the first file. The updated scores of the peers become [A=0.64, B=0.67, E=0.84, F=0.72]. Therefore, the second replica is placed at A and B (i.e., lowest scores). Next, the updated scores become [A=0.84, B=0.87, E=0.84, F=0.72]. Now, A and F will be chosen. For the replica generated at B, the scores of the neighboring peers will be [C=0.49, D=0.80]. Therefore, C will be chosen. We notice that this solution is more intuitive and balances better the storage across the nodes.

4.3 Replica Placement Algorithm

The previous section has illustrated how network centrality-based ranking of the peers can find an intuitive replica placement solution that leads to fair and balanced storage allocation. In addition, such a replication scheme avoids the problem of peers in P2P-OSN not being able to replicate their content at neighbors when there is plenty of storage available elsewhere in the network. We also saw that this placement scheme should incorporate a measure of the amount of empty storage at the peers, as the peers generate content at different rates.

Our algorithm for replica placement uses the following ranking function for the peers: $R_{ind} = W_{cent} * centrality + W_{st} * storage$

Here, $centrality$ is the centrality score of the peer, and its value is between 0 and 1. The $storage$ is the portion of unavailable (used) storage, and its value is between 0 and 1. Based on extensive experimentation, we chose $W_{cent} = 0.7$ and $W_{st} = 0.3$ for the implementation of our method (centrality has a higher weight than the unavailable storage). Each peer computes the ranking scores of its neighbors before placing replicas for new content. The peers are sorted in increasing order of the ranks. Therefore, peers with low centrality or high amounts of empty storage are at the top of the list. Peers are chosen from the top with a low probability of skipping the current peer and using the next peer in the list. This skipping of comparatively good peers is done to obtain a better global solution [68].

The algorithm is presented in Algorithm 3. This algorithm requires the EasyRank centrality scores. Lines 5-12 initialize the required data structures(e.g., pot_peers is the collection of potential peers for the current placement, and $skip\_thr$ is the threshold for skipping a more favorable peer). Lines 13-17 set the list for potential peers. If a peer is not available at the moment due to churn, it is not added to the list. Lines 19-20 asks the potential peers about their EasyRank scores and current available storage. Lines 21-23 calculate the

Algorithm 3 Replica Placement Algorithm

```
Requires:
$Centrality\ values\ calculated\ using\ algorithms$
$shown\ in\ section$ 4.2.2
Do the following for each replica:
Initialize:
$pot\_peers\_list \leftarrow empty$ {vector of (peerid,rank) tuples}
$cent\_list \leftarrow empty$ {vector of neighbors' centralities}
$storage\_list \leftarrow empty$ {vector of neighbors' available storage}
$skip\_thr \leftarrow THR$ {A pre-defined value for stable balancing}
$w\_c, w\_s \leftarrow WEIGHT$ {Pre-defined Weights for centrality and storage values}
$pindex \leftarrow 0$ {index to the $pot\_peers\_list$}
$repl\_placed \leftarrow 0$
$repl\_to\_place \leftarrow number\_of\_replicas()$ {Based on intended application specification}
for all neighbors n do
    $pot\_peer.peerid \leftarrow n.id$
    $pot\_peer.rank \leftarrow 0$
    $pot\_peers\_list.add(pot\_peer)$
end for
{Populate lists with neighbors' centrality and storage availability}
$cent\_list \leftarrow centrality(neighbors)$ {get centralities of all neighbors}
$storage\_list \leftarrow empty\_storage\_percent(neighbors)$ {get available storage of all neighbors}
for each peer p in $pot\_peers\_list$ do
    $p.rank \leftarrow w\_c * cent\_list.get(p.peerid) + w\_s * storage\_list.get(p.peerid)$
end for
$sort(pot\_peers\_list, decreasing)$ {sort the peers in decreasing order of the ranks($p.rank$)}
$skip\_probability \leftarrow random()$
$pcount \leftarrow pot\_peers\_list.length()$
$first\_repl \leftarrow TRUE$
while $repl\_placed < repl\_to\_place$ do
    if $((first\_repl == FALSE) \&\& ((pcount - pindex) > repl\_to\_place) \&\& (skip\_probability < skip\_thr))$
    then
        $pindex \leftarrow pindex + 1$
        $continue$ {while loop}
    end if
    $first\_repl \leftarrow FALSE$
    $is\_stored \leftarrow store\_data(pot\_peers\_list.get(pindex))$ {Store the replica in the corresponding peer}
    if $is\_stored$ == true then
        $pindex \leftarrow pindex + 1$
        $repl\_to\_place \leftarrow repl\_to\_place - 1$
        $repl\_placed \leftarrow repl\_placed + 1$
    else
        $pindex \leftarrow pindex + 1$
    end if
end while
```

Table 4.3 Properties of the Social Graphs

	GooglePlus	Facebook
Vertices	4903	1034
Edges	723321	26749
Density	0.06	0.05
Diameter	5	9
Clust. Coeff	0.24	0.5

peer ranks, and line 24 sorts the peers based on their ranks. The loop in lines 28-42 runs until all the required replicas are placed or there are no more peers left to consider.

The second condition in the if statement at line 28 makes sure that random skipping will not lead to storing fewer replicas; the third condition skips a favorable peer with probability $skip\_thr$ ($skip\_thr$ is constant and set to 0.1 in our implementation). The *complexity of the algorithm* is linear to the number of neighbors (lines 28-42).

4.4 Experimental Evaluation

We evaluated the solution through simulations using two real-life social graphs from [69]. One graph is from GooglePlus and has 4903 vertices; the other is from Facebook and has 1064 vertices. The properties of these two graphs are shown in Table 4.3. We built a new simulator for this work that fits our problem and allowed us to quickly evluate the replication methods. We could not evaluate larger graphs due to memory limitations in our simulator.

Nevertheless, we are confident that our solution will work for larger networks for two reasons. First, as we will see in this section, all results have similar patterns for both networks despite their different sizes. Second, the solution is P2P, and we expect that peers in larger networks will have similar responsibilities and behaviors.

We compared our scheme with a random scheme and other centrality-based schemes. All the evaluated schemes choose the peer/neighbor to store a given replica sequentially from an ordered list of neighbors. The difference comes from how they select the peers from the list. The random scheme selects a peer randomly (after making sure it has space to store the replica). The centrality-based schemes, including ours, use the algorithm described in this paper; however, each of them uses different values for W_{cent} in the ranking equation ($R_{ind} = W_{cent} * centrality + W_{st} * storage$).

4.4.1 Evaluation Scenarios and Metrics

We compared the replication schemes for three different scenarios: (1) stable social networks where the graphs do not change and the amount of storage remains the same; (2) stable social networks where the graphs do not change, but people are donating additional storage over time, (3) dynamic social networks, where people are joining the system and new connections (edges) are created over time.

For each scenario, we evaluated the mean and standard deviation for: (1) storage usage, which measures whether users are able to get storage when there is globally available storage in the network; (2) success rate in storing at least one replica; and (3) replication success rate, which measure whether the peers were able to store all the intended replicas. If these three metrics are clustered around median values, the replication scheme is balanced/fair (i.e., almost all the users are getting the same quality of service).

Table 4.4 Simulation Parameters

Parameter	Value
Storage donation	5-20 GB (from a uniform distribution)
Content generation 1	Pareto(α=0.3,β = 1.0) with prob=0.1
Content generation 2	LogNormal(mean=10.0,var=5.0) with prob=0.9
Maximum generated content size	100 MB
Churn	Exponential (mean=0.2)
Content generation rate	Top 20% every 6 hour, Rest every 12-36 hour
More storage donation	5-20GB
Forest fire burning probability	0.2
Peers removed from FB Graph	400
Peers removed from GP Graph	2000
Replica count	1-3
W_{cent}	1.00
W_{st}	0.50

4.4.2 Simulation Setup and Parameters

Table 4.4 shows the simulation setup and parameters. For each result, we executed 20 simulation runs. Peers initially donate storage ranging from 5 to 20 GB, selected from a uniform distribution. People generate content at different rates. Researchers analyzed content generation patterns in online social networks and found that 80% of the content is generated by 20% of the users [70]. To simulate this, we randomly select 20% peers, which generate more content than the others. These peers generate new content every 6 hours, and the rest generate content every 12 to 36 hours. Peers generate content at the same time with probability 0.1, which simulates multiple image/data posting at the same time.

The file sizes are selected from two distributions: with probability 0.9, the file size is log-normally distributed with mean 10MB and variance 5MB; with probability 0.1, the file size comes from a Pareto distribution with α set to 0.3 and β set to 1.0. The maximum size for the generated content is 100 MB. We generated this mix to represent mostly images, documents, and some other large files. The choice of distributions comes from [71].

Every 24 hours, the peers decide whether to shut down with probability 0.1. Churn duration is generated from an exponential distribution with a mean of 4 hours. This is

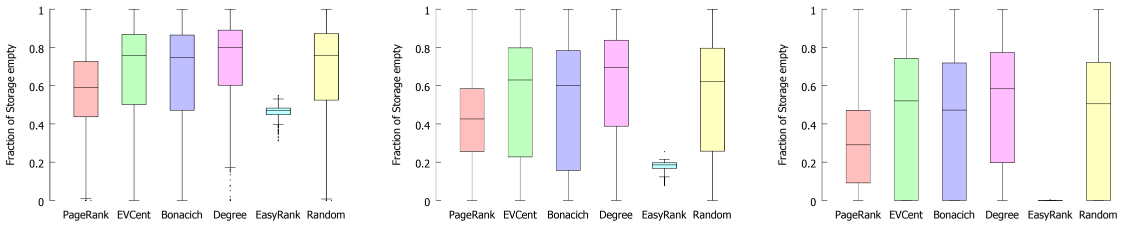


Figure 4.3 BoxPlot of the fraction of empty storage left at the peers for the stable Facebook graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

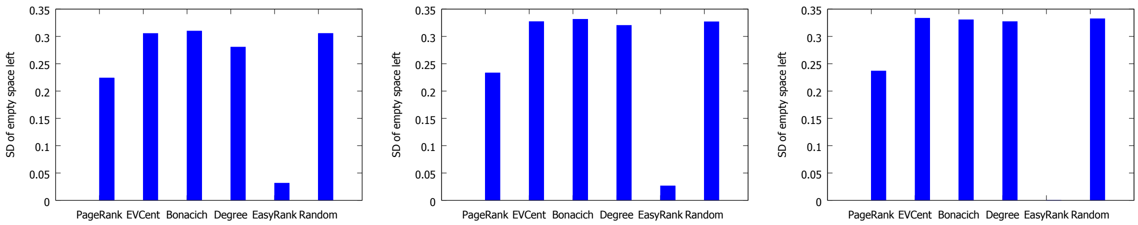


Figure 4.4 Standard deviation of the fraction of empty storage left at the peers for the stable Facebook graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

somewhat lower than usual P2P systems, but we expect the churn in P2P-OSN to be low as the network provides social incentives (e.g., the peers help their friends) [72]. The number of replicas a peer requests (in addition to its own copy) varies from 1 to 3 and comes from a uniform distribution.

We used pre-calculated stored values for PageRank, Eigenvector and Bonacich power centrality. This decision provides an advantage to these methods because churn would lead to inconsistent values in different parts of the network. Subsequently, the performance of these methods would decrease.

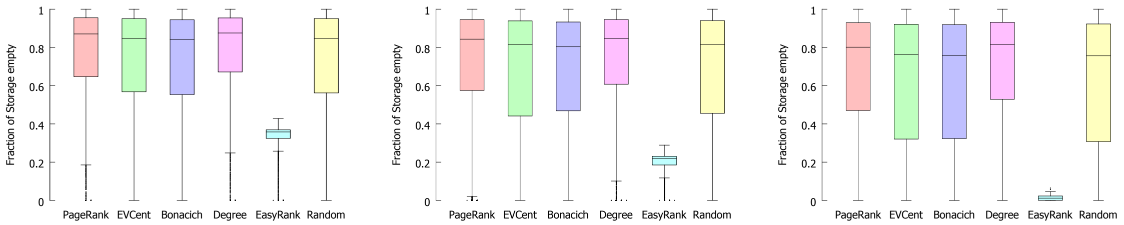


Figure 4.5 BoxPlot of the fraction of empty storage left at the peers for the stable GooglePlus graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

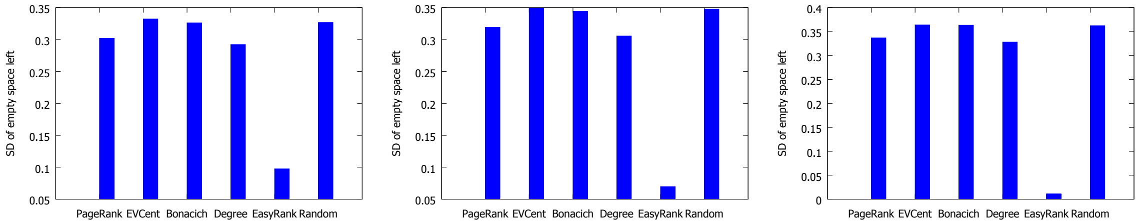


Figure 4.6 Standard deviation of the fraction of empty storage left at the peers for the stable GooglePlus graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

4.4.3 Results for Stable Social Networks

The first set of experiments analyze the steady state operations when the social graph and the amount of storage at peers do not change over time. We ran the simulations until the average available storage space at peers dropped to 40%, 20% and 1%.

Storage Usage. Figures 4.3 and 4.5 show the boxplot and Figures 4.6 and 4.4 show the standard deviation of the fraction of storage still unused at the peers for the Facebook and GooglePlus graphs, respectively. The boxplots show the median, 25-th and 75-th percentiles, and the overall clustering of storage usage at the peers. The histograms show the standard deviation of the storage usage at the peers.

The results demonstrate that EasyRank achieves the best performance in terms of storage usage for both networks; the storage usage is clustered around the median values. For the other centrality schemes and the random scheme, the storage usage is imbalanced

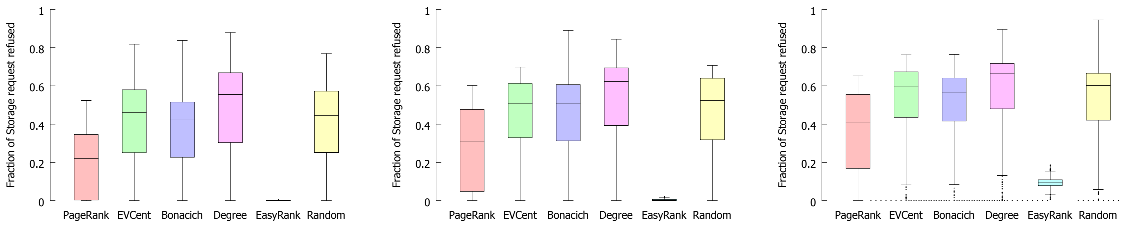


Figure 4.7 BoxPlot of the fraction of failed replication requests for the stable Facebook graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

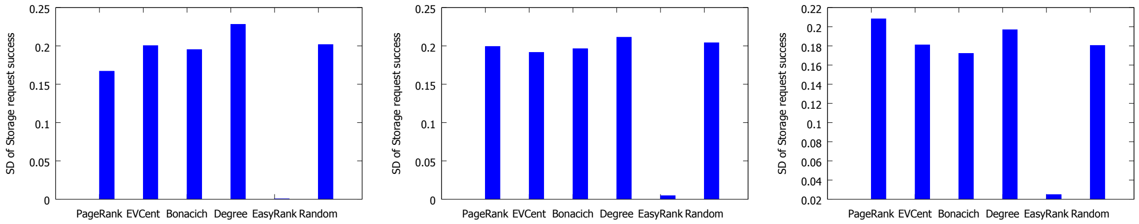


Figure 4.8 Standard deviation of the fraction of failed replication requests for the stable Facebook graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

and scattered over long ranges. Therefore, we conclude that the EasyRank-based scheme provides the fairest and balanced solution.

We also observe that the EasyRank-based scheme performs better than the other schemes when the percentage of available space decreases from 40% to 20% and then to 1%. The storage usage increases gradually for EasyRank and the scheme continues to provide balanced replication. The schemes show an increase in the level of imbalance with the decrease of the available/empty storage: there is little change in the median value, but they do not have clustering around median.

Success Rate in Storing at Least One Replica. The impact of poor/imbalanced storage usage is faced by the peers with many failed replication requests. We consider a replication request to be failed when no replica is stored. Figure 4.7, 4.8, 4.9, and 4.10

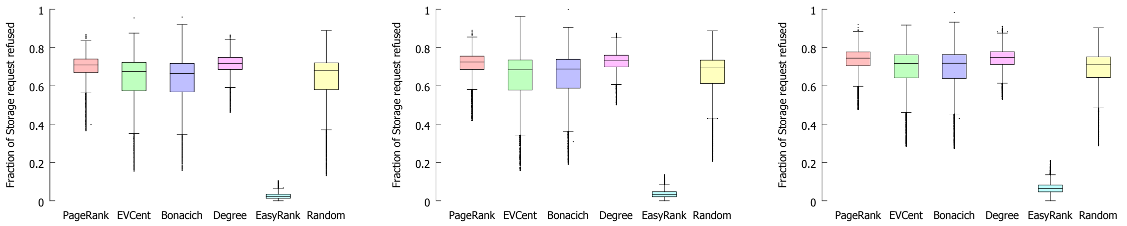


Figure 4.9 BoxPlot of the fraction of failed replication requests for the stable GooglePlus graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

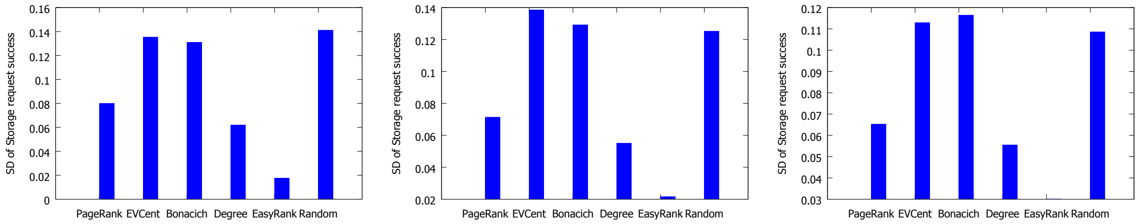


Figure 4.10 Standard deviation of the fraction of failed replication requests for the stable GooglePlus graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

show the fraction of failed replication requests for the stable social networks. We see that the fail rate for the EasyRank-based scheme is almost 0 and clustered around the median value for both networks. We also observe that the median fail rate increases steadily for both networks as the amount of available storage decreases. Overall, the EasyRank-based scheme outperforms the others.

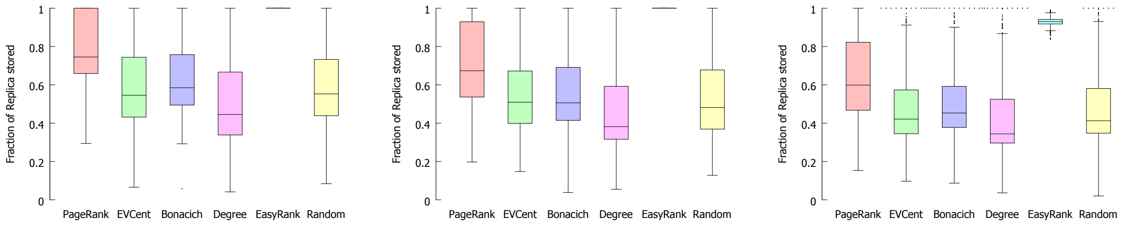


Figure 4.11 BoxPlot of the fraction of successful replication requests for the stable Facebook graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

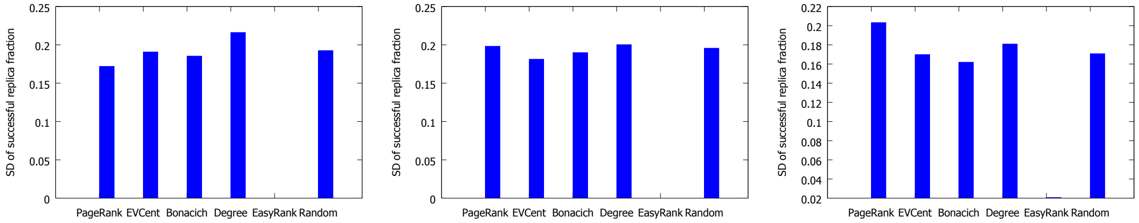


Figure 4.12 Standard deviation of the fraction of successful replication requests for the stable Facebook graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

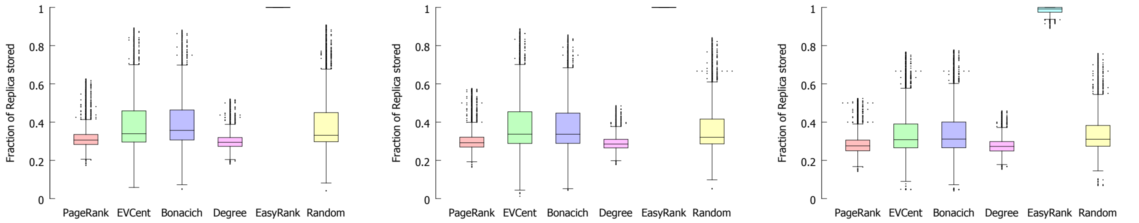


Figure 4.13 BoxPlot of the fraction of successful replication requests for the stable GooglePlus graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

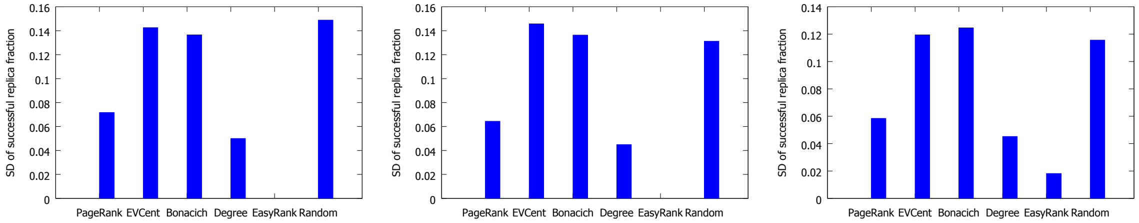


Figure 4.14 Standard deviation of the fraction of successful replication requests for the stable GooglePlus graph when the simulation was stopped at: average empty storage 40% (left), average empty storage 20% (middle), average empty storage 1% (right).

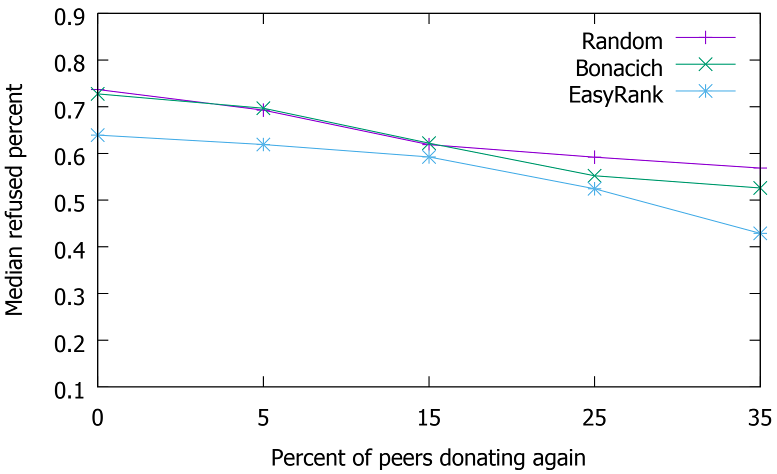


Figure 4.15 Change of median values with new storage addition for percent of failed replication requests.

Replication success rate. Figures 4.11, 4.12, 4.13, and 4.14 show the fraction successful replication requests (i.e., the desired replication factor is satisfied) for both stable networks. We again observe the effect of balanced and efficient storage usage for the EasyRank-based method: as long as storage is available, peers could store all the desired replicas. The success rate is close to 100% even when the available average storage at peers is 1%. On the other hand, the comparison schemes perform poorly even when the available average storage at peers is 40%, with the median success rate being less than 40%.

The values for the GooglePlus graph are more clustered than those for the Facebook graph (for other centrality-based methods). This happened as the GooglePlus graph has more vertices; thus, peers have more neighbors to store for the same amount of load. But even in this case, EasyRank outperformed the other centrality-based methods.

4.4.4 Results for Stable Networks with Storage Addition

To assess the performance when storage is dynamically added during the simulation, we select a number of peers that add storage after their individual initial contributed storage (5-20 GB per) is 95% allocated. The rest of the peers do not add space even after their

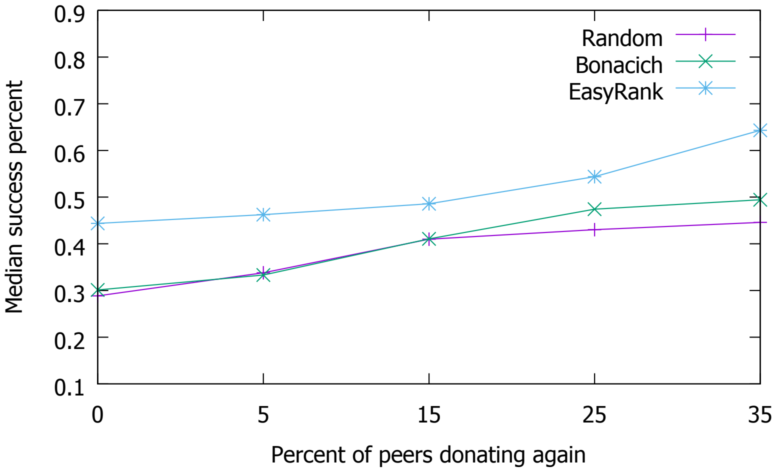


Figure 4.16 Change of median values with new storage addition for percent of replica successful replication requests.

storage is completely filled up. For this experiment, the percentage of peers contributing additional storage is varied from 5% to 35% of the total number of peers. These peers are selected randomly and donate extra 5-20GB of storage as described in Table 4.4.

The goal of this experiment is to find out how sensitive the replication schemes are to storage addition. A good scheme is expected to be able to readily use the newly donated storage and should exhibit comparatively lower replication fail rates and higher replication success rate. Let us recall that a replication is considered failed in no replica could be stored independent of the value of the replication factor. A replication is successful if the desired replication factor is satisfied.

Figures 4.15 and 4.16 show the median values of the fraction of failed replication requests and the fraction of successful replication requests. In this figure, we only plot the curves for EasyRank, Bonacich, and random schemes. Eigenvector and Pagerank schemes show similar results to Bonachich. The results demonstrate that EasyRank always performs better than the other schemes and exhibits steeper improvements if more than 30% peers contribute storage to the system. The results are worse than the ones for the stable networks

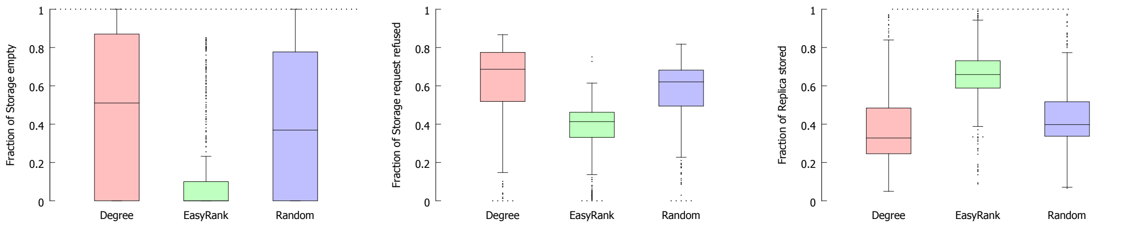


Figure 4.17 Boxplot of fraction of storage usage, failed replication requests (middle), and successful replication requests (right) for the dynamic Facebook graph. The simulation was run until the average available storage at peers was 1%, after all the edges and peers were added.

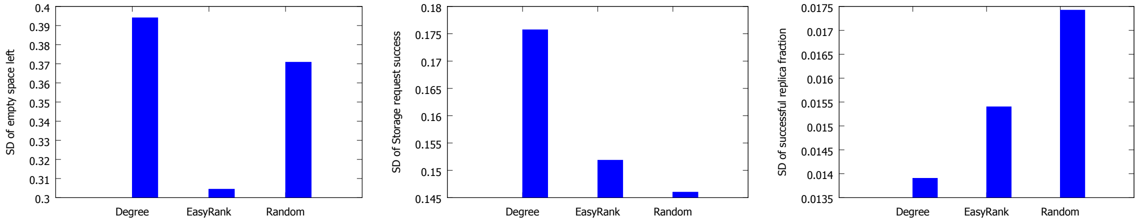


Figure 4.18 Standard deviation of fraction of storage usage, failed replication requests (middle), and successful replication requests (right) for the dynamic Facebook graph. The simulation was run until the average available storage at peers was 1%, after all the edges and peers were added.

because there is a large storage imbalance between the peers adding storage (5%-35%) and those not adding (65%-95%). The result of this imbalance is that many peers fill up before the average available storage across all the peers become 1%. This phenomenon does not happen in the previous experiments. Thus, more replication requests will completely fail or will not be able to store the desired number of replicas.

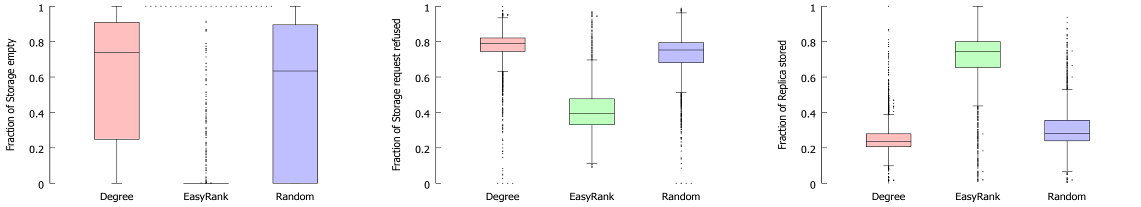


Figure 4.19 Boxplot of fraction of storage usage, failed replication requests (middle), and successful replication requests (right) for the dynamic GooglePlus graph. The simulation was run until the average available storage at peers was 5%, after all the edges and peers were added.

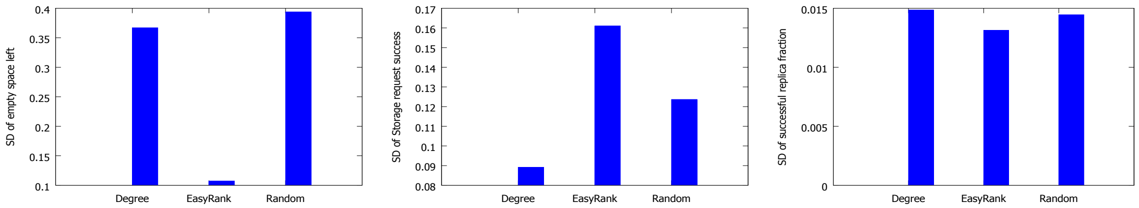


Figure 4.20 Standard deviation of fraction of storage usage, failed replication requests (middle), and successful replication requests (right) for the dynamic GooglePlus graph. The simulation was run until the average available storage at peers was 5%, after all the edges and peers were added.

4.4.5 Results for Dynamic Social Networks

This experiment evaluates the performance while peers and edges are being added to the social networks. For that, we need a social graph which is forming over time. Therefore, we generated back-in-time graphs by removing 400 peers from the Facebook graph and 2000 peers from the GooglePlus graph. We used the forest fire method discussed in [73] to generate the most likely back-in-time social graphs. We also removed additional random edges from the existing social graphs. We started the simulation with the back-in-time social graph. Then, we added back the peers and edges in the same order they were removed, one per hour, after the average available space dropped to 60% for the Facebook graph and 75% for the GooglePlus graph (the difference in percentages is due to the different graph sizes).

Figures 4.17,4.18, 4.19, and 4.20 show the results for the dynamic social networks. We only considered the Degree centrality, EasyRank, and Random schemes. Since the graph is changing, there is no simple way to finds the correct centrality values for the other metrics/schemes. For both networks, we ran the simulations until the average available storage at peers was 5%, after all the edges and peers were added. The results demonstrate that EasyRank achieves the most balanced storage usage, the lowest replication failure rate, and the highest replication success rate.

4.5 Summary

This chapter presented Philia, an efficient replication method for storing replicas of user generated content in P2P-OSN, which uses EasyRank, a new network centrality metric. We designed our solution to be fair and balanced: peers can use the system irrespective of the number of friends they have. We discussed why network centrality can be used for compensating for the peers' limited view of the network. Then, we showed how EasyRank-based replica placement can achieve balanced storage usage. We evaluated the EasyRank-based replication method with real social graphs and compared it against other centrality-based replication methods and a random replication method. The results demonstrate that Philia allocates the storage in a fair and balanced way, which allows for an increased lifetime of the system. The peers can find replication storage at neighbors as long as there is global storage available.

CHAPTER 5

MOITREE: PROGRAMMING MODEL FOR MOBILE COLLABORATIVE APPS

In this chapter, we shall describe the Moitree programming model. We shall start with the Avatar architecture and assumptions in Section 5.1. In Section 5.2 we shall describe the programming API. In Section 5.3, we shall present the middleware which maintains the pairing between the avatar and the mobile device and provides a seamless programming environment. We shall show experimental results in Section 5.4.

5.1 Avatar Overview

The Avatar system [19] is motivated by the strong demand for fast, scalable, reliable, and energy efficient distributed computing over mobile devices. A few research efforts have investigated cloud support for mobile distributed computing [15,74,75]. These projects focus mostly on enabling communication among mobile users. Other efforts have investigated offloading mobile code execution to the cloud in the context of stand-alone apps [13, 14, 76, 77]. While Avatar leverages high-level ideas from these efforts, its goal is fundamentally different - integrating the benefits of modern mobile devices and cloud computing, to provide a novel platform for distributed collaborative apps (which are inherently peer-to-peer).

In Avatar, a mobile user is represented by one mobile device and its associated "avatar" hosted in the cloud. An avatar is a per-user software entity which acts as a surrogate for the user's mobile device, to the extent possible, thus reducing the workload and the demand for storage and bandwidth on the mobiles. Each avatar is instantiated as a VM (Android x86) in the cloud in order to provide resource isolation and to simplify per-user resource management. Avatars run the same operating system as the mobiles and can thus run unmodified apps or app components (e.g., functions, threads, etc.). Implicitly, they save energy on the mobiles and improve the response time for many apps by executing certain tasks on behalf of the mobiles. The avatars are always available, even when their associated mobile devices are offline because of poor network connectivity or simply turned off. Each avatar coordinates with its mobile device to synchronize data and schedule the computation of avatar apps on the avatar and/or mobile device. A mobile device does not interact directly with the avatars of other mobile users. User-to-user communication always goes through the avatars.

While the avatars could be hosted by any cloud provider, mobile network operators may be particularly willing to offer the service. They have already started to offer cloud services to back up data from mobile devices. At the same time, having information about their customers and their mobile devices, mobile network operators may be able to offer better services with higher efficiency than other cloud providers.

The Avatar platform provides the architecture for cloud-assisted mobile distributed computing, and Moitree offers the required runtime system (i.e., middleware) as well as the API to build mobile distributed apps over Avatar.

5.2 Moitree Programming Model

This section first describes the key ideas in Moitree and the distributed app execution model. Then, we present an example app, *LostChild*, to illustrate Moitree's advantages in developing distributed apps. Finally, we describe the complete API provided by Moitree.

5.2.1 Key Ideas

Groups and Group Hierarchies A *group* is a fundamental unit over which mobile distributed computation is done. A group is a set of users selected and organized based on properties, such as user locations, time, and social connections. Each group is app-specific, and each app can create as many groups as it needs. All members of a specific group, by default, run the same app. A user can be part of multiple groups at the same time.

Groups in the same app can form hierarchies with the groups at lower levels being subgroups of the ones at upper levels, and are maintained in a tree structure. This helps programmers to structure the distributed computation for certain apps. For example, subgroups can be recursively created to solve location-based computations using the divide-and-conquer strategy (e.g., finding a free parking spot in a city).

Communication Channels A communication channel provides high level messaging support and makes it easy to offload communication to the cloud. There are four types of channels: (i) *broadcast* for sending messages to all members of a group; (ii) *anycast:* for sending messages to a random member of a group; (iii) *point2point:* for sending messages to a specific member; and (iv) *scatter-gather:* for sending messages to all members of a group and then receiving answers from some group members as a function of their computation results. The *broadcast* channel is unidirectional, while the other three are bidirectional.

When a function invokes a communication channel on a mobile, the call is intercepted in the middleware and always forwarded to the cloud (avatars) to carry on. User-to-user communication always occurs via the avatars.

While some messages (e.g., point2point and anycast communication) are not persistent, the messages exchanged within the whole group (e.g., broadcast and scatter-gather communication) can be designated as *persistent* by the programmers. These messages are useful for forwarding data to new comers to the group (e.g., a person who entered the region that defines the group after the group was formed). Persistent messages are stored in the group buffer in the cloud and distributed to new members when they join a group.

Dynamic Group Membership Since people move and their context changes over time, group membership also changes. The dynamic group membership in Moitree shields the programmers from handling group dynamics. The middleware selects and maintains group members automatically based on properties specified by the programmers. For example, a group can be formed for a given geographic region during a specific time interval. The middleware will add/remove group members based on who enters/leaves the region during that interval.

5.2.2 App Execution

A distributed app is instantiated by all group members who collaborate within the app. Thus, the distributed app consists of app instances executed in parallel on the set of mobile/avatar pairs belonging to the group members. The distributed app runs in a single program, multiple data style. The app group is formed when the first user (i.e., the initiator) starts its app instance, which in turn invokes a group creation function. This user becomes the owner of the group. Other group members are selected and added to the group automatically by the middleware based on their properties and permissions. The permissions are defined in the users' collaboration policies and specify the conditions under which a user is willing to participate in a group. The app instances at the other group members are created in response to the request for group creation. This is done through events delivered to the Moitree components on the avatars/mobiles.

The app code is the same for each group member. However, due to factors such as resource availability on mobiles and privacy policies, the way in which the computation tasks are partitioned between avatars and mobile devices may differ for different users. Programmers do not have to worry about app partitioning because Moitree hides it from them. Tasks in an app may also differ with respect to different users based on their roles in the distributed computation.

During the execution, instances can send messages using the four communication channels already described. The communication is event-based. Messages are received asynchronously (i.e., similar to the "select" system call in Unix), using callback functions registered by the programmers with the Moitree middleware.

The app instance at the initiator is responsible for deleting its groups before terminating. This operation triggers events in Moitree that will terminate app instances at all the other group members.

Let us finally note that Moitree apps invoke the Moitree API for group operations and communication. For everything else, it uses the local platform API (e.g., Android). Finally, it is important to point out that Moitree apps co-exist with native apps on mobile devices.

5.2.3 Code Example *LostChild* App

To illustrate the development of a Moitree app as well as the main features of the middleware, let us consider a *LostChild* app. A parent could use it to locate a child lost in a crowded area using the child's photo(s) to search for the child among recent photos taken by nearby mobile users. Based on the places where the photos that contain the child have been taken,

Code 5.1 Code running at the parent's mobile/avatar

```
searchForLostChild(){
 MembershipProperties prop = new MembershipProperties();
 prop.setLocationBound(LOCATION, RADIUS);
 prop.setTimeBound(TIME_FROM,TIME_TO);
 Group group = Avatar.createGroup(null,prop,false,LIFETIME);
 ReadCallback callback = new ReadCallback(){
   public void scatterGather(ChannelID cid,Message msg){
     //msg contains results sent by participants
     //add result to potential trajectories
     //update user with latest trajectories
     if (ChildFound){
       DONE=true;
     }
   }
  ... ... ... //other channel callbacks
 }
 group.setReadCallback(callback);
 ChannelID cid = generateChannelID();
 group.scatterGather(cid, childImage);
 while (!DONE){ //block }
 group.deleteGroup(group.credential());
}
```

the app builds a real-time trajectory of the child's movement to aid the parent to find her quickly.

Code 5.1 shows the code executed by the app instance running at the parent's mobile/avatar. Lines 2-4 specify the context properties that must be satisfied by group members (i.e., region and time interval), and the group is created in line 5. The middleware is responsible for group formation and dynamic maintenance. Alternatively, a group may also be created from a specific list of users. Before adding a user to the group, the middleware verifies that its app/group collaboration policies allow it to become a group member (e.g., the user may refuse to collaborate due to location privacy reasons).

Lines 6-16 show the callback function implementation for message communication. In this example, we have only one channel type, *scatter-gather*. The callback function is registered with the middleware in line 17. Then, in line 19, the app sends a photo of the child to all the group members and then waits for responses until the child is found. When

Code 5.2 Code running at the participants' mobiles/avatars

```
onCreateGroup(Group group){
 ReadCallback callback = new ReadCallback(){
   public void scatterGather(ChanelID cid, Message msg){
     Image image = msg.getData();
     //run face recognition with image
     //if child image is recognized, set the location and time associated with the
         matching photo in result
     group.scatterGather(cid, result);
   }
   ... ... ... //other channel callbacks
 };
 group.setReadCallback(callback);
}
```

responses are received from individual users, the scatter-gather callback is invoked and the potential child trajectory is updated and presented to the parent. When the parent confirms that the child was found, the app deletes the group and terminates (lines 11-12 and 20-21).

Code 5.2 shows the app processing done at the participating users. The app is activated by the middleware at the participating users when they are added to the group (at group formation time or later). The app starts with the *onCreateGroup* method in line 1. A reference to the group is passed as a parameter in this method. A *scatter-gather* callback is implemented in lines 2-10 and registered with the middleware in line 11. The callback is invoked when the photo of the child is received from the parent. Face recognition is performed for all the photos of this user that satisfy the group location and time criteria to see if they match the child image (line 5). If any of them does (line 6), its associated location and time are sent to the parent over the *scatter-gather* channel (line 7).

This example shows that Moitree makes mobile distributed computing concise and easy to program and understand.

Table 5.1 Moitree API

Group-related API - Avatar Class

Method	**Description**
Group *createGroup*(Group parent, MembershipProperties prop, Boolean enableLeader, Double groupLifetime)	Creates a group with members selected based on membership properties *prop*; if *enableLeader* is true, the group has a special member with leader role. *groupLifetime* specifies how long the group should exist without receiving any messages from the members.
Boolean *changeParentGroup*(Group newParent)	This method is used to re-organize the group tree.
Boolean *onCreateGroup*(Group group)	Callback method executed by Moitree to start the app for a user when the user is made member of the group. Parameter *group* is supplied by the middleware.
Boolean *joinGroup*(User user, Group group, Credential c)	Called to join a group after the group was created. The credential ensures that the user has appropriate permissions to join the group. Credentials are generated when a group is created and distributed to the members as part of group creation.

Group-related API - Group Class

Method	**Description**
void *removeFromGroup*(User usr)	Called to remove *usr* from a group.
void *deleteGroup*(Credential c)	Deletes an existing group. Credentials are used to ensure that the callee has permission to delete the group.
List⟨User⟩ *getMembers*()	Returns the list of group members.
User *getLeader*()	Returns the group leader.
Group *getRoot*()	Returns a reference to the root of the group.
Group *getParent*()	Returns a reference to the parent of the group.
List⟨Group⟩ *getChildGroups*()	Returns the list of children groups of the group.

5.2.4 API Description

The Moitree API, presented in Tables 5.1 and 5.2, is divided into three classes. The *Avatar* class provides methods for group creation and joining. The *Group* class offers methods for group management (e.g., leave/delete the group, create subgroups) and group communication. The *MembershipProperties* class is a utility class that has methods for specifying the group properties. The same Moitree API is used independent of the execution place (i.e., mobile or avatar).

Table 5.2 Moitree API

Group Membership API - MembershipProperties Class

Method	**Description**
void *setTimeBound*(Time from, Time to)	Used to set the time property for identifying users active in the given time interval (typically used in conjunction with the location property).
void *setLocationBound*(Location center, Double radius)	Used to set the location where a user is/has been/will be (typically used in conjunction with the time property).
void *setSocialNetwork*(SocialNetwork network, Activity a)	Used to identify group members who are part of the user's social network based on activities such as friendship, work, sports, etc.
void *setList*(List⟨Users⟩ users)	Used to add specific users to a group.

Group Communication API - Group Class

Method	**Description**
void *setReadCallBack*(ReadCallBack callBack)	Registers the read callback methods for incoming messages. *ReadCallBack* is an interface with four callback methods corresponding to broadcast, anycast, scatter-gather, and point2point.
void *broadcast*(Message msg)	This method is used to broadcast messages to a group.
void *anycast*(Message msg)	Used to send a message to a random member of the group.
void *scatterGather*(ChannelID cid, Message msg)	Used to broadcast messages to a group and get responses from group members back to the broadcaster. An app can use as many scatterGather channels as required by using different *ChannelID* to different channel
void *point2point*(Message msg, User to)	Used for user-to-user communication.
void *sendToLeader*(Message msg)	Used for sending a message to the group leader.

Group Creation, Membership, and Deletion In addition to the *prop* parameter already discussed, *createGroup* takes three other parameters: *parent*, *enableLeader*, and *lifetime*. The *parent* parameter is used for group hierarchies. If it set to *null*, it means the group is the potential root of a group hierarchy; however, groups are not required to be part of hierarchies. More details on group hierarchies are presented in Sub-section 5.2.4.

Some groups may need leaders to implement functions such as consensus or scheduling among their members. If the *enableLeader* is set to *true*, then the user who creates the group

Code 5.3 Creating hierarchical groups

```
//Top–Down Group Creation
Group newark = Avatar.createGroup(null,membershipProperties,true,lifetime);
Group zip07102 = Avatar.createGroup(newark,membershipProperties2,true,lifetime);
... ... ...
//Bottom–up Group Creation
Group zip07102 = Avatar.createGroup(null,membershipProperties,true,LIFE_TIME);
Group zip07103 = Avatar.createGroup(zip07102.getParent(),membershipProperties,true,
    LIFE_TIME);
Group newark = Avatar.createGroup(zip07102.getRoot(),membershipProperties,true,
    LIFE_TIME);
zip07102.changeParentGroup(newark);
... ... ...
```

becomes the leader. If the leader leaves the group, the middleware selects a new leader. Currently, Moitree selects a random user as a new leader, but other leader selection policies could be implemented. The method *sendToLeader* allows any user to send messages to the leader, without the need to use the ID of the leader, which improves fault-tolerance. The method *getLeader* is normally used to determine if the local user is the leader of the group; if yes, the app needs to run leader-specific functionality.

The *lifetime* parameter specifies that a group has to be deleted by the middleware in the absence of any group communication for the *lifetime* duration. In this way, Moitree deallocates the resources associated with a group when the group is not active. The apps receive an exception and terminate.

Groups are dynamic in that members can come and go. Users can join a group using the *joinGroup* method. The user invoking this method must know the group ID and have the right credentials. For example, a new user is invited to a multi-player mobile game and is provided the group ID and the credentials. A user can leave a group by calling *removeFromGroup*. Currently, this method is used only to remove the user invoking it. However, we plan to explore if this method should be allowed to remove other users; such functionality could be useful for group creators and/or leaders.

Group Hierarchies By default, if a user belongs to a group, it is also a member of the parent group. For simplicity, we do not allow overlapping sibling groups or overlapping groups on different branches of the hierarchy tree.

A user in a subgroup can get a reference to the parent group using the *getParent* method or to the root group using *getRoot*. Similarly, a user in a group can get a reference to the child subgroups (i.e., one level down in the hierarchy) using the *getChildGroups* method. Group hierarchies can be created top-down or bottom-up as shown in Code 5.3. The top-down approach is used when the hierarchy can be defined before the app starts. The bottom-up approach allows for dynamic creation of hierarchies at runtime. The example in Code 5.3 illustrates the group hierarchy for an app that finds free parking spots around a given destination. The region around the destination can be divided into several levels of sub-regions, with each sub-region associated with a group in the hierarchy. The processing can be done in parallel for each subgroup, and the results can be gathered at the initiator.

Group Communication Apps may use any combination of communication channels as needed. Each channel is instantiated by the middleware upon its first invocation in the app. Each app instance can use its own scatter-gather channel. To distinguish these channels, they have unique ChannelIDs.

The communication on all channels is asynchronous. Anyone can send a message anytime and receive messages through callback methods. Each sending communication channel is paired with a receiving callback method.

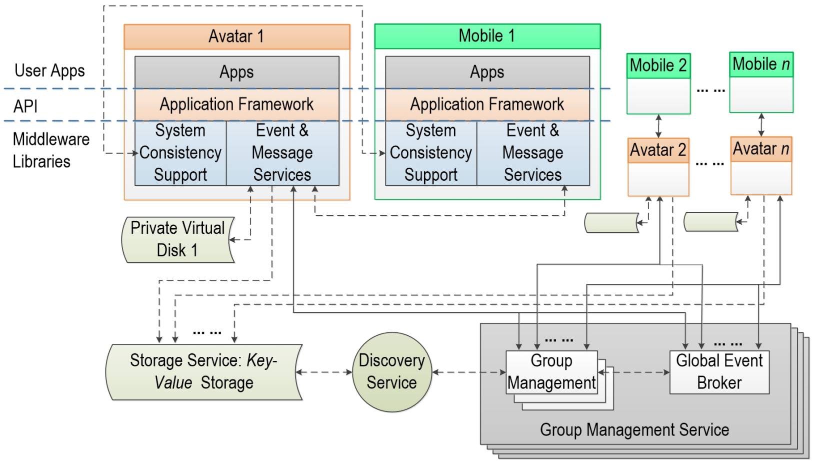


Figure 5.1 Moitree middleware architecture.

5.3 Middleware

The Moitree middleware is responsible for providing the runtime support for the API discussed in Section 5.2. The overall structure of the middleware is shown in Figure 5.1. The middleware is implemented in libraries, which process app requests with the support of standard Android system services and a few Avatar-specific services.

The middleware has the following components. *System Consistency Support (SCS)* synchronizes data and system states between a mobile device and its associated avatar to create a consistent execution environment for apps. *Event and Message Services (EMS)* manage and dispatch events and messages to drive the app execution. *Group Management Service (GMS)* manages groups and implements all group related functionality. It also supports communication among group members by managing communication channels. *Directory Service (DS)* responds to queries requesting user information. *Storage Service (SS)* facilitates shared storage space for the middleware. SCS and EMS reside on both mobile devices and avatars. The other components are provided as part of the cloud infrastructure and run on dedicated servers. The number of servers is dynamically adjusted based on the workload.

5.3.1 System Consistency Support (SCS)

The avatar and the mobile device of a user adopt the same OS (Android in our current implementation). This allows the same app code to run unmodified on both platforms. SCS ensures a consistent system environment for the app instances on the platforms. SCS synchronizes the necessary data (e.g., photos, contacts, app-specific data) and system/app state (e.g., current location of the mobile device). To achieve low synchronization overhead, Moitree runs a daemon on each mobile and its avatar. The daemons rely on each app to define the data to be synchronized and specify when and how frequently the data should be synchronized. The cost of synchronization depends on user policies and the amount of data to synchronize. A policy that requires to keep everything always in sync will be costly in terms of energy and bandwidth usage. The other extreme policy could be to synchronize only using WiFi and while the phone is charging. Currently, we are working on fine grained policies which will provide optimized solutions in terms of usability and energy/bandwidth cost.

5.3.2 Event and Message Services (EMS)

Moitree API calls from an app are translated by the middleware as a set of events or messages. EMS forwards the events and messages to the appropriate app instances in the group with help from GMS. On each system (mobile device or avatar), the EMS component is implemented as a set of Android services, which are shared by all the apps on that system. To handle events and messages, two queues are established: an event queue (EQ) and a

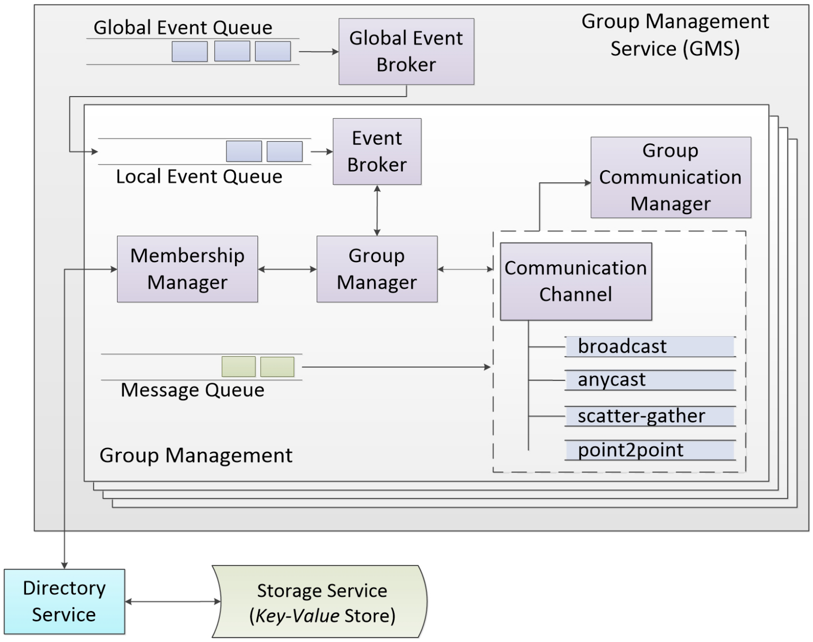


Figure 5.2 Group Management Service architecture.

message queue (MQ). Events generated from apps (e.g., group creation event in LostChild) are posted to EQ. MQ is used to distribute data and receive results (e.g., photos of the lost child, photos matching the lost child and the related location/time information).

Moitree events and messages do not depend on network addresses/names to reach their destinations. Instead, they rely on the groups and channels established in each group. Specifically, events are defined as (*app*, *group*, *type*, and *data*) tuples; messages are defined as (*app*, *channel*, *group*, *type*, and *body*) tuples. The recipients of an event are the members in the group of the app specified by the *app* and *group* fields. Within a group, the channel types (i.e., *type* field) help demultiplexing messages. GMS assists event/message forwarding with group and channel information.

When an event or a message is generated on a mobile device, it is first forwarded to its corresponding avatar before it is delivered to each recipient of a group. Compared to directly sending the event/message to every recipient, sending it only once to the avatar can significantly reduce the mobile data traffic and its potential monetary cost. When the event/message arrives at the avatar, EMS forwards it to a GMS server, where the recipient list can be determined. Then, the event/message is dispatched by the GMS server to the recipient avatars, and these avatars finally forward it to the corresponding apps.

5.3.3 Group Management Service (GMS)

GMS is the core of the middleware. It is designed to handle group operations, events, and communication. GMS runs on a group of dedicated servers, and its internal modules are shown in Figure 5.2.

For each group hierarchy, a *group manager* is used to maintain its tree structure, and a *membership manager* is used to maintain a list of the current members of each group or subgroup. These two modules are also responsible for handling requests, such as creating and managing groups, changing membership, etc. To keep the data structures up-to-date, they use keep-alive messages to find out failures of the members; members are removed from groups when failures are detected.

For each group, there is an *event broker* in charge of delivering events within the group, and a *group communication manager* to maintain communication channels and to forward messages to recipients. Thus, the handling of events and messages is separated to prevent a large number of messages to delay a few important events. The middleware gives higher priority to event handling compared to message handling because events are associated with important group/system state changes that must be reflected in real-time in apps.

The forwarding workload of the GMS servers can be offloaded to group leader avatars. However, this requires that group and channel information is duplicated to these avatars.

This may lead to privacy concerns and additional overhead to maintain the information consistent. Another optimization aiming to reduce the load on GMS servers is to save large messages into a shared key-value store. Instead of forwarding complete messages, the GMS servers just forward the keys of the messages. When an avatar receives a message key, it reads out the message from the shared storage. However, this method increases the workload of the storage service. Thus, we have not included these optimizations in our current implementation because they need additional experimental evaluation.

5.3.4 Storage Service (SS)

SS provides a shared and permanent storage space for the middleware and is implemented as a key-value store. Specifically, this service uses the Redis [78] key-value database because it is fast, reliable, and can work on both a single node and a multi-node configuration.

SS maintains an app registry, which serves the purpose of finding which app is installed on which user's device and avatar. An entry in the app registry is created when an Avatar app is installed on a user's mobile device and avatar. The registry entry contains the avatar ID and the app's name. Information about the users' dynamic location and time is also stored by this service to assist the DS component. Finally, this service could be used for sharing large messages as described in section 5.3.3.

Each avatar has a virtual disk directly attached to it; these disks are not part of the storage service. Each disk serves as the private and primary storage for the avatar and the apps running on it.

Table 5.3 Number of Lines of Code for Our Apps Using Moitree and JXTA

Application	Moitree	JXTA
Lost Child	85	178
Video Conferencing	100	219

5.3.5 Directory Service (DS)

DS provides answers for queries such as “which users have the LostChild app installed and were present in Times Square between 5PM and 6PM today?”. DS uses SS as its data repository. Normally, we expect the mobile carrier to provide this service.

5.4 Evaluation

We implemented a prototype of Moitree and two apps: the LostChild app and a video-conferencing app that allows users to create group hierarchies. The Moitree prototype consists of 2722 lines of Java and Android code. The experimental evaluation has three goals: (1) verify the *effectiveness* of the Moitree programming model to reduce programming complexity by implementing two apps; (2) assess the *performance* of an app developed over Moitree, and (3) test the *efficiency* of the middleware through micro-benchmarks.

Our testbed uses Android-based mobiles and Android x86 VMs running in an OpenStack-based cloud. The mobiles are mix of Nexus 6, Nexus 5, Moto X Pure, Kindle Fire, and HTC One M7. Unless otherwise specified, each avatar VM runs Android 4.4 and is configured with 4 virtual CPUs and 3GB memory. The mobiles communicate with the cloud using our institution’s secure WiFi network.

5.4.1 A Case Study on Programming Effort

To quantify the benefits of the Moitree programming model, we used the LostChild app, a video conferencing app, and an app for finding interesting places. In the video conferencing app, users share real-time video streams with friends, family or acquaintance. These three types of groups have different levels of permissions. Friends and family have permissions to see all the streams, while acquaintance can view only selected streams. The InterestingPlace app helps tourists to find interesting places in unknown or relatively less explored areas (for which there are not many Yelp or GoogleMap entries). The tourists request help from the people around them about places such as restaurants, historic landmarks, etc. The people in the area receive the request along with the location of the tourist. Then, they draw their recommended path on the map from the tourist to the interesting place. In addition, they may annotate the path with photos. The tourist can follow the map and visit the interesting place. All three apps are typical mobile distributed apps that may involve a large number of mobile users. Their implementations must deal with the common issues that are usually faced in mobile distributed apps (e.g., identifying and coordinating groups of participants). Therefore, they can act as a good test for Moitree.

We first compared two implementations of the lost child and video conferencing app: one done using Java and JXTA [79], and one done in Java and Moitree. JXTA is selected to compare to Moitree for two reasons: (1) JXTA is designed for peer-to-peer systems, in which peers are conceptually similar to sets of autonomous avatar/mobile pairs, and (2) it also has group concepts, although different from those used in Moitree.

For the two implementations of these two apps, we compared the sizes of their source code. In this comparison, we only counted the lines written by our programmers.

The code in other libraries (e.g., OpenCV [80] for face recognition and Kryonet [81] for network communication) is not counted toward the effort to develop the app. The app implementations include mostly group management and group communication features; the rest is done through library function invocations.

We show the numbers of lines of code (LOC) in Table 5.3. We found that Moitree decreases LOC by a factor higher than 2. This is a promising result that illustrates how Moitree can simplify the programming of mobile distributed apps. Currently, we are implementing several additional apps for a more thorough evaluation.

5.4.2 Performance of the LostChild App

To understand the real-time performance of Moitree apps and the effect of avatars on latency, we measured the response time for the LostChild app in two scenarios: (1) the major workload in the app, including face detection and recognition, is handled on the mobiles, and (2) the major workload is executed at the avatars. Let us note that mobile to mobile communication in the first scenario is mediated by Moitree services in the cloud. Figure 5.3 shows the end-to-end response time from the time of submitting the initial request until the time of receiving the final results. The figure also shows the breakdown of the latency between the face detection/recognition operations and the networking/middleware operations. In these experiments, we used one mobile device as initiator and three other mobile devices as participants. All the avatars were instantiated on the same server. In this experiment, each avatar VM runs Android 6.0 and is configured with 6 virtual CPUs. Each participant has a database of 47 images containing 60 faces stored in her avatar. In addition, each participant returns a result because all participants have photos of the lost child. The training process for face recognition is done before the app starts.

The results demonstrate that avatars help reduce the end-to-end response time to half when compared to the scenario where the mobiles handle the major workload. A substantial part of this improvement is due to offloading the computation for face detection and recognition to the avatars. We also observe that the latency incurred by Moitree and networking is reduced by 14.4%; this is due to offloading the communication part in these workloads to the cloud.

Figure 5.4 shows that the response time when we varied the number of participants from 2 to 7. In this experiment, the face detection and recognition are executed at the avatars. The experiment was conducted for two scenarios: (1) all the participant avatars run on a single server; and (2) each participant avatar runs on a different server. All the other parameters are the same with those of the previous experiment. The figure plots the median response latency and the latency of the last received response as experienced by the initiator. Let us recall that in this experiment each participant sends a response. In a real-life situation, most participants are not expected to send responses. Therefore, the curves for the latency of the last received response represent the worst case scenario.

The results show that the absolute values are reasonable (generally, between 500ms and 700ms). In addition, the application scales well with the number of participants for this experiment. The number of participants has almost no effect on the median latency values. However, the latency of the last received response is affected by the number of participants. We found two reasons for this problem. First, our current Moitree implementation sequentializes the communication among the members and adds a few of milliseconds to every message transmission. Second, avatars do not send responses at the same time; in most experiments, we noticed one or two stragglers. We are working on improving the message delivery system of the middleware and planning more experiments to

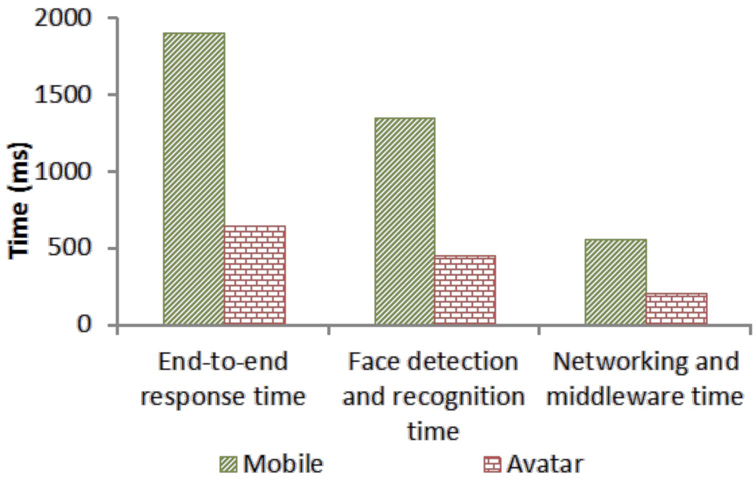


Figure 5.3 End-to-end response time for *LostChild* app when the workload is at the mobile and avatar, respectively.

better understand the second problem. Finally, let us note that running one avatar per server improves the response time and, as expected, is relatively constant. Running all avatars on one server, on the other hand, leads to higher latency and this latency increases with the number of participants. This is caused by the resource contention incurred by the increasing number of avatars on the same machine.

Figure 5.5 shows the power savings achieved on a LostChild participant phone when face recognition is done on the avatar vs. on the phone. The power measurements in our experiments were taken with Power Tutor [82]. When face recognition runs on the phone, the energy consumption is approximately 225J. When the face recognition is run at the avatar, the energy consumption on the phone is about 9J. Thus, we conclude that Moitree and Avatar lead to significant improvements in response latency and energy savings on the mobiles.

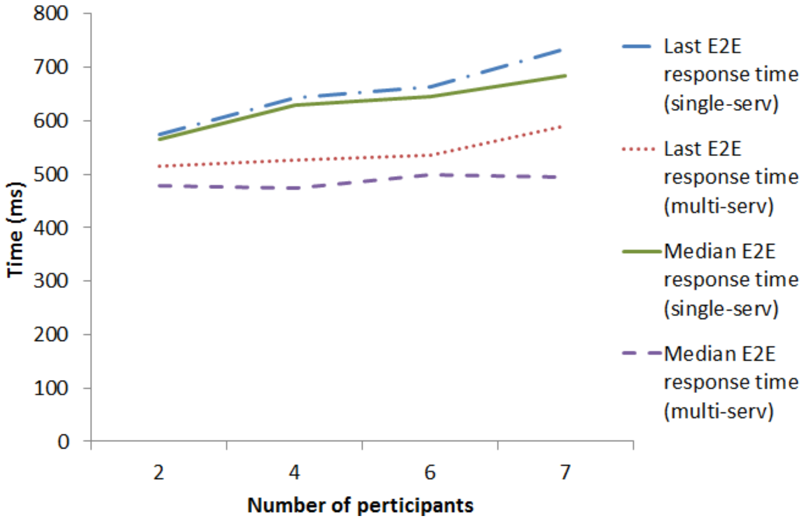


Figure 5.4 End-to-end response time for *LostChild* app vs. number of participants: (i) single-serv: all participant avatars run on one server; (ii) multi-serv: each participant avatar runs on a different server.

Table 5.4 Moitree's Energy Consumption on Phones

Component	**Energy/Power Consumed**	**Comment**
Moitree in idle state	5.5 mJ/sec	Middleware could run for two and half month before draining the battery
Moitree API calls	2.3 mJ/call	One and half million API calls with a full battery
Data transfer by middleware & Plain TCP	0.5 mJ/KB & 0.15 mJ/KB	Energy consumed in addition to WiFi being ON for the transfer

5.4.3 Experience with InterestingPlace App

This app, shown in Figure 5.6, requires a group with all the people in the area at a specific time. It is challenging to gather all these people and communicate with them efficiently. Yet, with the Moitree library, we had to call just two functions to set up the group. The communication required invoking one function call (for the scatter-gather channel). The easiest part was that we could use the same function to send and receive messages. Of

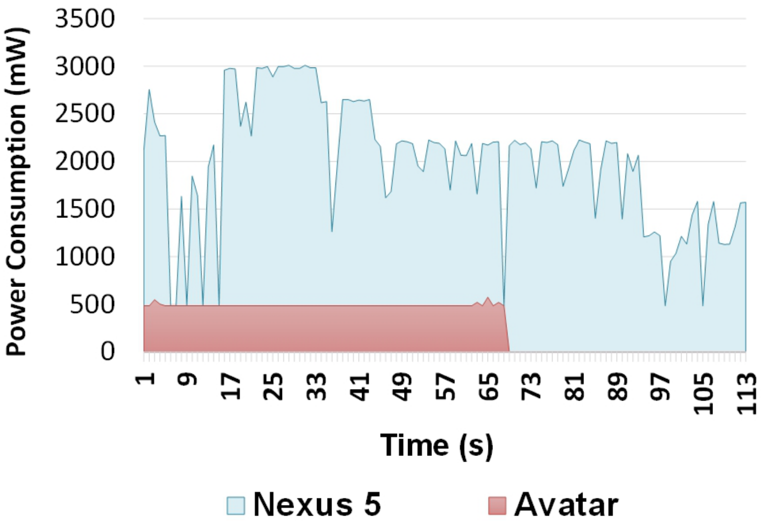


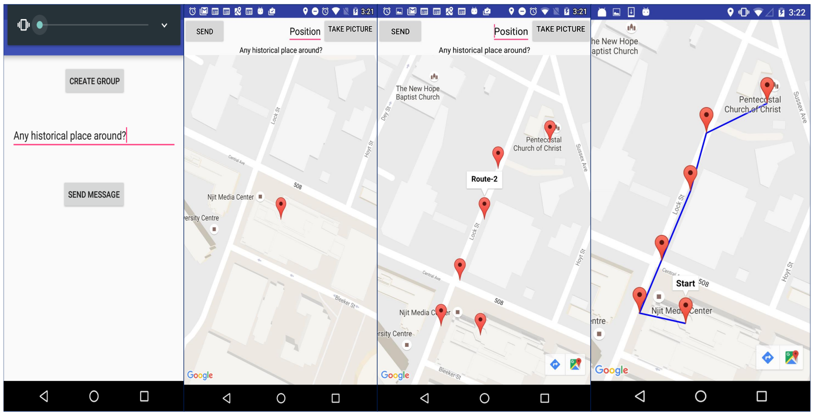
Figure 5.5 Power consumption comparison for participant *LostChild* app phone with and without avatar help.

course we had to write the callback functions, which represent 12 LOC. Therefore, the entire dynamic group creation and communication required mainly three functions calls.

This app demonstrates how programmers can use efficiently the network bandwidth and CPU power. When the app sends images, we used a flag indicating that the image should be processed at the avatar, not at the mobile. In this way, we saved wireless bandwidth and CPU cycles on the mobile. Implicitly, this saved battery power.

5.4.4 Experiments with Micro-benchmarks

Moitree itself consumes very limited resources. The mobile part of Moitree takes 35 MB of memory and registers almost no CPU usage in the idle state. However, to maintain the avatar pairing, it periodically checks for data synchronization and keeps waiting for communication requests. That is why we see the low energy usage in Table 5.4. This result shows that energy consumption on mobiles in the idle state is negligible for all practical purposes. Similarly, Table 5.4 shows that the energy consumed per average API call is very



Any historical place around?

Figure 5.6 InterestingPlace app.

low (a full battery charge allows one and a half million calls). In terms of data transmission, Table 5.4 shows that Moitree introduces a relatively high overhead when compared with plain TCP. This is due to the Kryonet [81] library used for communication, which simplifies programming at the cost of overhead. Nevertheless, the absolute values are still low.

Table 5.5 GMS Initial Memory Consumption

Component	Resource Consumption
GMS	70 MB
Primitive Data Structures	45%
Concurrency, Hash, and Sets	22%
Others	33%
Threads	23

The GMS runs on servers over Linux. Table 5.5 shows the initial resource consumption of the GMS server. In the beginning, it only consumes about 70 MB of memory and spawns 23 threads. To optimize the GMS server, we used primary data structures whenever possible. That is why we see that around 45% of the initial memory is used by primary data structures.

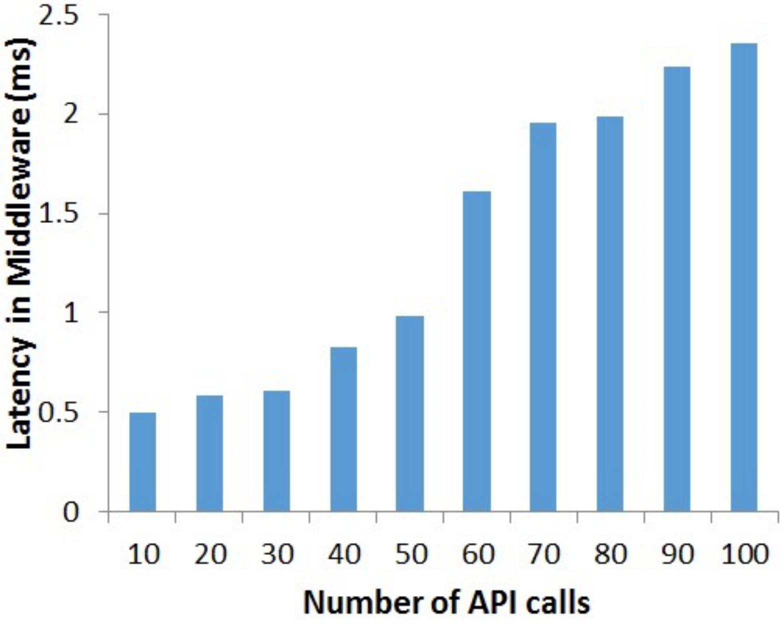


Figure 5.7 Average end-to-end latency for concurrent API calls in Moitree (including network communication).

An additional 22% of the memory is consumed by the concurrency management objects, hash maps, and sets. Java streams and the daemons use rest of the memory.

The most important work of the GMS server is to maintain the groups. Figure 5.9 shows the amount of memory used by the newly created groups. The increase in group count increases the memory usage almost linearly. The reason is that groups are separated entity. Unless groups are members of the same hierarchy, they do not share memory. We found that each group consumes about 7MB of memory. This value is almost constant over the lifetime of the experiments and in our daily usage.

Another important task of GMS is to manage the participants' location in real time. Users periodically update their locations to the GMS. It uses these location values to manage the group membership dynamically. Figure 5.10 shows the processing time for the location updates. We see that GMS can process the location updates in micro-seconds. On average,

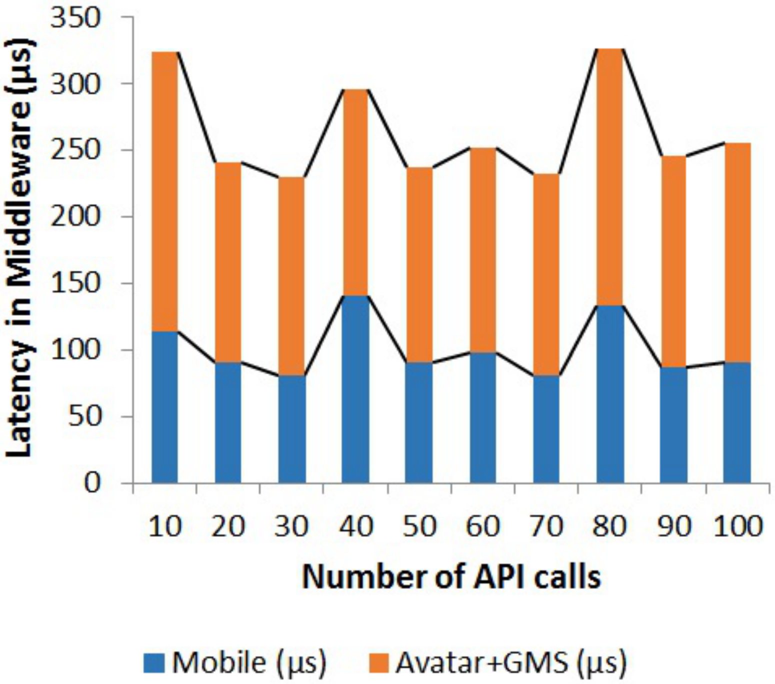


Figure 5.8 Average processing time for API calls in Moitree on mobile and avatar (no network communication).

each location update, including matching with current live groups, is accomplished in 3 microseconds. Therefore, location handling in GMS is fast and unlikely to create any processing bottleneck for the GMS server.

Location-based groups use persistent messages for maintaining state about location, time, etc. Figure 5.11 shows the latencies to store and retrieve persistent messages. We see that messages having size up to 1MB can be stored in milliseconds. Larger messages take exponentially longer time to store. We found that the Kryonet library takes more time to fragment/defragment the larger messages. The retrieval time is also affected as we have to send the messages to participants using the Kryonet library. The slight reduction in timing is due to the use of concurrent data structure where reading is faster than writing (no need for locking). The results indicate that to store larger message, it is more efficient to break the message into 1MB chunks before storing them.

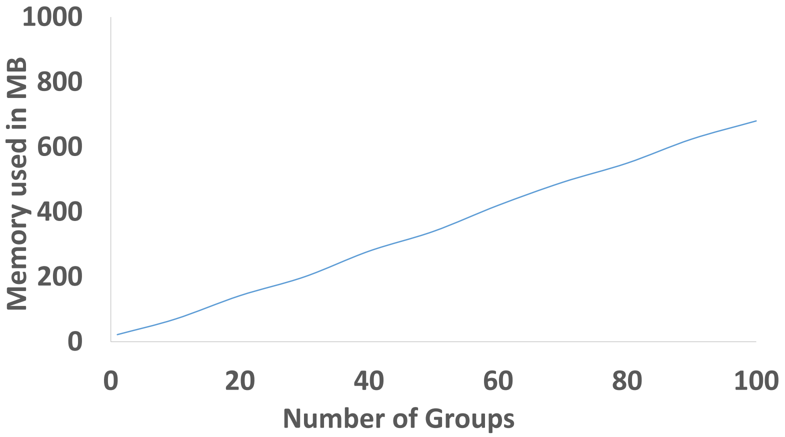


Figure 5.9 Memory required to create new groups.

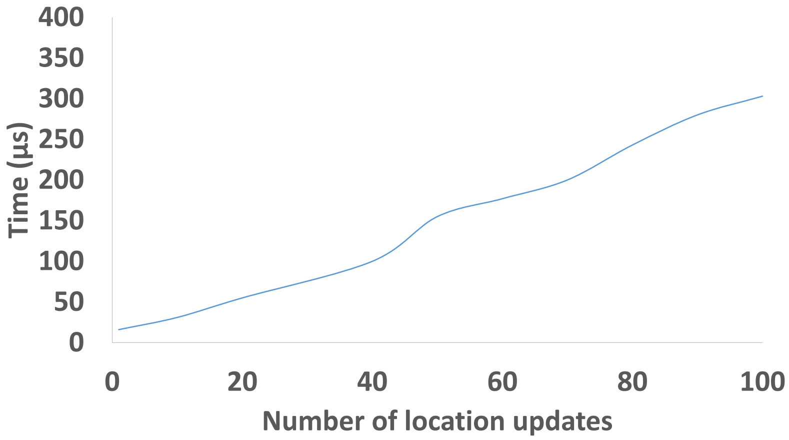


Figure 5.10 Processing time for simultaneous location updates.

GMS helps to route messages over the logical communication channels. We checked the communication latencies for the GMS message handlers with concurrent participants. We see from figure 5.12 that with the increase in participants count, latency increases but the rate of increase is sub-linear. GMS uses application level routing and sends messages sequentially to the participants. Therefore, there is some sequential delay in processing the messages. It should be noted that the latency shown in this graph only considers the latency incurred by the GMS server processing, not the end-to-end delay.

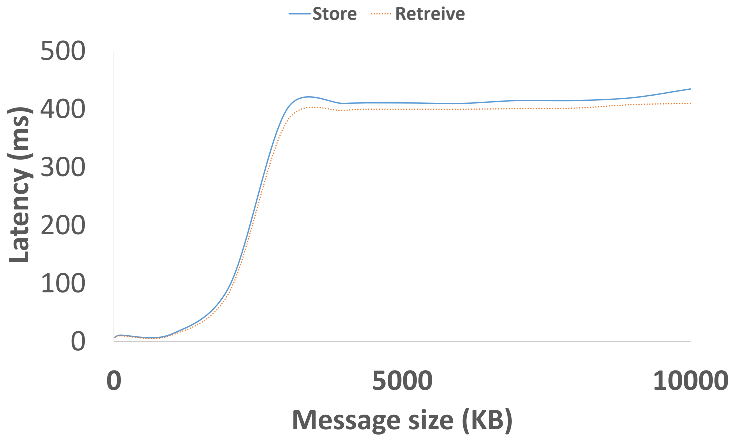


Figure 5.11 Latency for persistent message storage and retrieval.

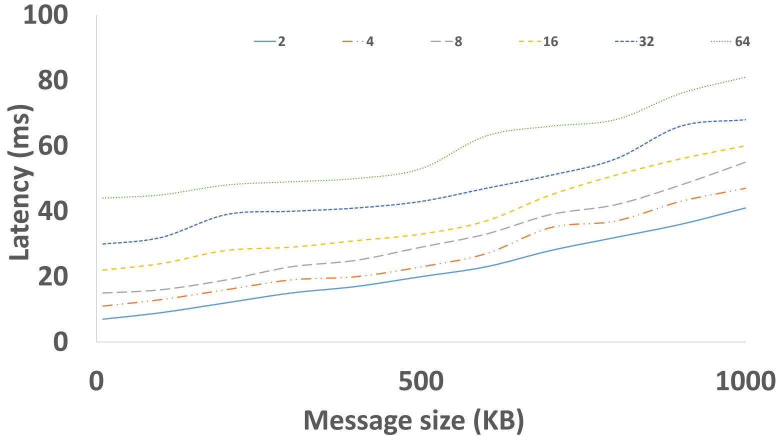


Figure 5.12 Latency for sending different size of messages to different number of participants.

Our next set of experiments show the scalability and the pairing capability of the mobile/avatar pairs. For the scalability tests, we measured end-to-end latency for concurrent API calls. Test programs concurrently called random APIs from mobiles and avatars, and then measured the latency to complete these calls. The API calls start at the *App* layer in Figure 5.1, pass through the *API and middleware* layers, reach the GMS service layer, and then the results are returned via the reverse path.

Table 5.6 Sensor Data Reading Latency in Moitree

Sensor	Reading latency (ms)
Accelerator	3.26
Gyroscope	1.96
Magnetometer	3.60

Figure 5.7 shows the latency of the concurrent API calls. We used up to 100 API calls at the same time, which is a significant load. The aim of the experiment is to assess the delay imposed by the middleware to process the calls. The results show that even 100 concurrent API calls can be resolved in 2.4ms. Next, we instrumented the middleware and discarded the communication time (used by the kryonet [81] library). The results are shown in Figure 5.8. We see that the execution of the API calls remains approximately constant at about 320μs. This demonstrates that Moitree scales well at a load of up to 100 concurrent API calls.

Our final set of experiments show the pairing capability of the middleware. The most important metric for pairing is the latency to resolve API calls spanning across the mobile and the avatar. For this experiment, our test program runs on an avatar and reads sensor data on the corresponding mobile. The middleware on the avatar intercepts the calls, requests data from the mobile, and transparently provides the data to the running programs. Table 5.6 shows the average latency incurred for three of the sensors present in the Nexus 5 phones. We deducted the network delay from the readings as it depends on external factors not related to the Moitree middleware. All the latencies are under 4ms, which is fine for most real-time apps. We are working on decreasing the networking latency which may hamper the performance of real-time apps.

5.5 Summary

In this chapter, we described the Moitree collaborative programming model along with its associated middleware for the Avatar [19] system. To the best of our knowledge, Moitree is the first middleware for mobile distributed apps assisted by the cloud. We described the API with an example app and discussed elaborately the components of the middleware which provide the pairing of the mobile-avatar. The results of our evaluation are promising. Moitree is able to reduce the number of lines of code to less than half when compared to an existing solution. In addition, Moitree scales well when multiple APIs are invoked concurrently and helps users save energy on mobile devices at the cost of a reasonable latency overhead.

CHAPTER 6

CONCLUSION

Smartphones and tablets have become part of our daily life. The amount of mobile user-generated content and mobile sensing data will become very large in the near future. These data will enable novel mobile applications that provide new and rich user experiences. However, people will be reluctant to share these data if they cannot own or control the data. We believe that mobile collaborative computing will empower users with control of their own data and will foster the creation of many novel apps.

The experimental results show that we can build collaborative storage networks with mobile devices, which are able to maintain high content availability and quickly locate the content in the network. Furthermore, the proposed collaborative social replication storage system is capable of achieving the fairest storage allocation and, thus, providing the highest rate of replication success.

The Moitree programming framework is easy to use for collaborative programming. Several programmers with different levels of expertises used the API with ease and built complex apps. The middleware scales well and introduces low delays for API calls and message routing. In addition, it saves a substantial amount of mobile bandwidth and power.

We believe privacy will play a greater role in the upcoming years for online services. People will be reluctant to hand over ownership and control of private data for getting free services. On the other hand, consumer-grade mobile devices and cloud resources are becoming more reliable and cheaper day by day. Therefore, collaborative mobile storage and computing, with potential help from the cloud, will play a greater role in supporting novel apps in the future.

REFERENCES

[1] H. Wang, D. Lymberopoulos, and J. Liu, “Local business ambience characterization through mobile audio sensing,” in *The 23rd International Conference on World Wide Web*. ACM, 2014, pp. 293–304.

[2] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, “Handling churn in a dht,” in *The USENIX Annual Technical Conference*. Boston, MA, USA, 2004, pp. 127–140.

[3] A. Shaker and D. S. Reeves, “Self-stabilizing structured ring topology p2p systems,” in *Fifth IEEE International Conference on Peer-to-Peer Computing*. IEEE, 2005, pp. 39–46.

[4] A. Binzenhofer, D. Staehle, and R. Henjes, “On the stability of chord-based p2p systems,” in *The IEEE Global Telecommunications Conference*, vol. 2. IEEE, 2005, pp. 5–10.

[5] N. Kourtellis and A. Iamnitchi, “Leveraging peer centrality in the designof socially-informed peer-to-peer systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 9, pp. 2364–2374, 2014.

[6] R. Narendula, T. G. Papaioannou, and K. Aberer, “A decentralized online social network with efficient user-driven replication,” in *International Confernece on Social Computing (SocialCom)*. IEEE, 2012, pp. 166–175.

[7] ——, “Privacy-aware and highly-available osn profiles,” in *The 19th IEEE International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE)*. IEEE, 2010, pp. 211–216.

[8] L. Han, M. Punceva, B. Nath, S. Muthukrishnan, and L. Iftode, “Socialcdn: Caching techniques for distributed social networks,” in *The IEEE 12th International Conference on Peer-to-Peer Computing*. IEEE, 2012, pp. 191–202.

[9] N. Kourtellis, J. Finnis, P. Anderson, J. Blackburn, C. Borcea, and A. Iamnitchi, “Prometheus: User-controlled p2p social data management for socially-aware applications,” in *The ACM/IFIP/USENIX 11th International Conference on Middleware*. Springer, 2010, pp. 212–231.

[10] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, “The little engine (s) that could: scaling online social networks,” in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4. ACM, 2010, pp. 375–386.

[11] D. A. Tran, K. Nguyen, and C. Pham, “S-clone: Socially-aware data replication for social networks,” *Computer Networks*, vol. 56, no. 7, pp. 2001–2013, 2012.

[12] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, “The case for vm-based cloudlets in mobile computing,” *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[13] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “Clonecloud: Elastic execution between mobile device and cloud,” *The 6th EuroSys Conference (EuroSys)*, pp. 301–314, 2011.

[14] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “Maui: making smartphones last longer with code offload,” *The 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pp. 49–62, 2010.

[15] S. Kosta, V. C. Perta, J. Stefa, P. Hui, and A. Mei, “Clone2clone (c2c): Peer-to-peer networking of smartphones on the cloud,” in *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2013.

[16] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy, “Customizable and extensible deployment for mobile/cloud applications,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2014, pp. 97–112.

[17] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, “Innovative technology for cpu based attestation and sealing,” in *The 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, vol. 13, 2013.

[18] A. Baumann, M. Peinado, and G. Hunt, “Shielding applications from an untrusted cloud with haven,” in *The 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 267–283.

[19] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath, “Avatar: Mobile distributed computing in the cloud,” in *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2015 3rd IEEE International Conference on*. IEEE, 2015, pp. 151–156.

[20] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: a scalable peer-to-peer lookup protocol for internet applications,” *The IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.

[21] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *The IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2001, pp. 329–350.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content-addressable network,” in *The 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM ’01. New York, NY, USA: ACM, 2001, pp. 161–172.

[23] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, “Tapestry: A resilient global-scale overlay for service deployment,” *The IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, 2004.

[24] P. Druschel and A. Rowstron, “Past: A large-scale, persistent peer-to-peer storage utility,” in *The Eighth Workshop on Hot Topics in Operating Systems*. IEEE, 2001, pp. 75–80.

[25] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer *et al.*, “Oceanstore: An architecture for global-scale persistent storage,” *ACM Sigplan Notices*, vol. 35, no. 11, pp. 190–201, 2000.

[26] J. W. Lee, H. Schulzrinne, W. Kellerer, and Z. Despotovic, “mdht: multicast-augmented dht architecture for high availability and immunity to churn,” in *The 6th IEEE Consumer Communications and Networking Conference*. IEEE, 2009, pp. 1–5.

[27] I. Gupta, K. Birman, P. Linga, A. Demers, and R. Van Renesse, “Kelips: Building an efficient and stable p2p dht through increased memory and background overhead,” in *Peer-to-Peer Systems II*. Springer, 2003, pp. 160–169.

[28] I. Woungang, F.-H. Tseng, Y.-H. Lin, L.-D. Chou, H.-C. Chao, and M. S. Obaidat, “Mr-chord: Improved chord lookup performance in structured mobile p2p networks,” *IEEE Systems Journal*, vol. 9, no. 3, pp. 743–751, 2015.

[29] C.-L. Liu, C.-Y. Wang, and H.-Y. Wei, “Cross-layer mobile chord p2p protocol design for vanet,” *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 6, no. 3, pp. 150–163, 2010.

[30] W. Jiang, C. Xu, M. Huang, J. Lai, and S. Xu, “Improved chord algorithm in mobile peer-to-peer network,” in *The International Conference on Advanced Intelligence and Awareness Internet*. IET, 2011, pp. 239–242.

[31] M. Zulhasnine, C. Huang, and A. Srinivasan, “Towards an effective integration of cellular users to the structured peer-to-peer network,” *Peer-to-Peer Networking and Applications*, vol. 5, no. 2, pp. 178–192, 2012.

[32] Q. Hofstatter, S. Zols, M. Michel, Z. Despotovic, and W. Kellerer, “Chordella-a hierarchical peer-to-peer overlay implementation for heterogeneous, mobile environments,” in *The 8th International Conference on Peer-to-Peer Computing*. IEEE, 2008, pp. 75–76.

[33] T. Horozov, A. Grama, V. Vasudevan, and S. Landis, “Moby-a mobile peer-to-peer service and data network,” in *International Conference on Parallel Processing*. IEEE, 2002, pp. 437–444.

[34] J. Rybicki, B. Scheuermann, M. Koegel, and M. Mauve, “Peertis: a peer-to-peer traffic information system,” in *Proceedings of the sixth ACM international workshop on VehiculAr InterNETworking*. ACM, 2009, pp. 23–32.

[35] Q. Cao and S. Fujita, “Load balancing schemes for a hierarchical peer-to-peer file search system,” in *The International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. IEEE, 2010, pp. 63–70.

[36] M. Mordacchini, L. Ricci, L. Ferrucci, M. Albano, and R. Baraglia, "Hivory: Range queries on hierarchical voronoi overlays," in *The IEEE Tenth International Conference on Peer-to-Peer Computing*. IEEE, 2010, pp. 1–10.

[37] L. Liquori, C. Tedeschi, L. Vanni, F. Bongiovanni, V. Ciancaglini, and B. Marinković, "Synapse: A scalable protocol for interconnecting heterogeneous overlay networks," in *International Conference on Research in Networking*. Springer, 2010, pp. 67–82.

[38] B. G. Karthik, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in dynamic structured p2p systems," in *The IEEE INFOCOM*. Citeseer, 2004.

[39] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in structured p2p systems," in *Peer-to-Peer Systems II*. Springer, 2003, pp. 68–79.

[40] A.-K. Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, and C. Diot, "Mobiclique: middleware for mobile social networking," in *The 2nd ACM Workshop on Online Social Networks*. ACM, 2009, pp. 49–54.

[41] P. Stuedi, I. Mohomed, M. Balakrishnan, Z. M. Mao, V. Ramasubramanian, D. Terry, and T. Wobber, "Contrail: Enabling decentralized social networks on smartphones," in *Middleware 2011*. Springer, 2011, pp. 41–60.

[42] S.-W. Seong, J. Seo, M. Nasielski, D. Sengupta, S. Hangal, S. K. Teh, R. Chu, B. Dodson, and M. S. Lam, "Prpl: a decentralized social networking infrastructure," in *The 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*. ACM, 2010, p. 8.

[43] (2013, Nov.) Join diaspora. [accessed 10-May-2016]. [Online]. Available: https://joindiaspora.com/

[44] D. Liu, A. Shakimov, R. Cáceres, A. Varshavsky, and L. P. Cox, "Confidant: Protecting osn data without locking it up," in *Middleware 2011*. Springer, 2011, pp. 61–80.

[45] A. Shakimov, H. Lim, R. Cáceres, L. P. Cox, K. Li, D. Liu, and A. Varshavsky, "Vis-a-vis: Privacy-preserving online social networking via virtual individual servers," in *The 3rd International Conference on Communication Systems and Networks*. IEEE, 2011, pp. 1–10.

[46] S. Sodsee, "Placing files on the nodes of peer-to-peer systems," Ph.D. dissertation, FernUniversität in Hagen, Nonthaburi, Thailand, 2012.

[47] D. N. Tran, F. Chiang, and J. Li, "Friendstore: cooperative online backup using trusted nodes," in *The 1st Workshop on Social Network Systems*. ACM, 2008, pp. 37–42.

[48] K. Chard, S. Caton, O. Rana, and K. Bubendorfer, "Social cloud: Cloud computing in social networks," in *The 3rd International Conference on Cloud Computing*. IEEE, 2010, pp. 99–106.

[49] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, “Farsite: Federated, available, and reliable storage for an incompletely trusted environment,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 1–14, 2002.

[50] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker, “Ght: a geographic hash table for data-centric storage,” in *1st ACM International Workshop on Wireless sensor networks and Applications*. ACM, 2002, pp. 78–87.

[51] F. Araujo, L. Rodrigues, J. Kaiser, C. Liu, and C. Mitidieri, “Chr: a distributed hash table for wireless ad hoc networks,” in *The 25th IEEE International Conference on Distributed Computing Systems Workshops*. IEEE, 2005, pp. 407–413.

[52] M. Caesar, M. Castro, E. B. Nightingale, G. O’Shea, and A. Rowstron, “Virtual ring routing: network routing inspired by dhts,” in *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4. ACM, 2006, pp. 351–362.

[53] O. Landsiedel, K. A. Lehmann, and K. Wehrle, “T-dht: topology-based distributed hash tables,” in *The 5th International Conference on Peer-to-Peer Computing*. IEEE, 2005, pp. 143–144.

[54] R. Chakravorty, S. Agarwal, S. Banerjee, and I. Pratt, “Mob: A mobile bazaar for wide-area wireless services,” in *The 11th Annual International Conference on Mobile Computing and Networking*. ACM, 2005, pp. 228–242.

[55] U. Lee, J. Lee, J.-S. Park, and M. Gerla, “Fleanet: A virtual market place on vehicular networks,” *IEEE Transactions on Vehicular Technology*, vol. 59, no. 1, pp. 344–355, 2010.

[56] H. Cao, O. Wolfson, B. Xu, and H. Yin, “Mobi-dic: Mobile discovery of local resources in peer-to-peer wireless network.” *IEEE Data Engineering*, vol. 28, no. 3, pp. 11–18, 2005.

[57] B. Pásztor, M. Musolesi, and C. Mascolo, “Opportunistic mobile sensor data collection with scar,” in *The Internatonal Conference on Mobile Adhoc and Sensor Systems*. IEEE, 2007, pp. 1–12.

[58] J. Lin, E. Shing, W.-K. Chanand, and R. Bagrodia, “TMACS: Type-based distributed middleware for mobile ad-hoc networks,” in *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, July 2008.

[59] J. Liu, D. Sacchetti, F. Sailhan, and V. Issarny, “Group management for mobile ad hoc networks: design, implementation and experiment,” in *The 6th international conference on Mobile data management*, May 2005, p. 192199.

[60] M. Mamei and F. Zambonelli, “Programming pervasive and mobile computing applications with the tota middleware,” in *The second IEEE Annual Conference on Pervasive Computing and Communications*, March 2004, pp. 263–273.

[61] A. L. Murphy, G. P. Picco, and G.-C. Roman, “Lime: A coordination model and middleware supporting mobility of hosts and agents,” *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 3, pp. 279–328, July 2006.

[62] S. Guinea and P. Saeedi, “Mobile application development with MELON,” in *The 13th International Conference ADHOC-NOW*, June 2014, pp. 265–278.

[63] N. Brouwers and K. Langendoen, “Pogo, a middleware for mobile phone sensing,” in *The 13th International Middleware Conference*, December 2012, pp. 21–40.

[64] A. Gupta, A. Kalra, D. Boston, and C. Borcea, “Mobisoc: A middleware for mobile social computing applications,” *Springer MONET*, vol. 14, no. 1, pp. 35–52, Jan. 2009.

[65] P. Erdos and A. Rényi, “On the evolution of random graphs,” *Mathematics Institute of Hungary, Academy of Science*, vol. 5, pp. 17–61, 1960.

[66] A. Montresor and M. Jelasity, “PeerSim: A scalable P2P simulator,” in *The 9th International Conference on Peer-to-Peer Computing*, Seattle, WA, Sep. 2009, pp. 99–100.

[67] “A mobility scenario generation and analysis tool,” 2016, [accessed 23-March-2016]. [Online]. Available: http://sys.cs.uos.de/bonnmotion/

[68] S. Kirkpatrick, D. G. Jr., and M. P. Vecchi, “Optimization by simmulated annealing,” *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[69] L. Jure. (2015, Nov.) Stanford large network dataset collection. [accessed 1-Feb-2016]. [Online]. Available: http://snap.stanford.edu/data/index.html

[70] L. Guo, E. Tan, S. Chen, X. Zhang, and Y. E. Zhao, “Analyzing patterns of user content generation in online social networks,” in *The 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 369–378.

[71] M. Mitzenmacher and B. Tworetzky, “New models and methods for file size distributions,” in *The Annual Allerton Conference on Communication Control and Computing*, vol. 41, no. 1. The University; 1998, 2003, pp. 603–612.

[72] J. Li and F. Dabek, “F2f: Reliable storage in open networks.” in *The 5th International Workshop on Peer-to-Peer Systems*, 2006, pp. 1–6.

[73] J. Leskovec and C. Faloutsos, “Sampling from large graphs,” in *The 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 631–636.

[74] K. Kim, S. Lee, and P. Congdon, “On cloud-centric network architecture for multidimensional mobility,” *Computer Communication Review*, vol. 42, no. 4, pp. 509–514, 2012.

[75] Y. Qin, D. Huang, and X. Zhang, “Vehicloud: Cloud computing facilitating routing in vehicular networks,” *The 11th IEEE TrustCom*, pp. 1438–1445, 2012.

[76] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, “Odessa: Enabling interactive perception applications on mobile devices,” in *The 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’11. ACM, 2011, pp. 43–56.

[77] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, “Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” *The IEEE Infocom 2012*, pp. 945–953, 2012.

[78] “Redis,” http://redis.io/, [Online; accessed 10-May-2015].

[79] L. Gong, “Jxta: A network programming environment,” *The IEEE Internet Computing*, vol. 5, no. 3, pp. 88–95, 2001.

[80] “OpenCV,” http://opencv.org/, [Online; accessed 10-May-2015].

[81] “Kryonet,” https://github.com/EsotericSoftware/kryonet, [Online; accessed 10-May-2015].

[82] Z. Yang, “Powertutor-a power monitor for android-based mobile platforms,” *University of Michigan, EECS*, vol. 2, p. 19, 2012.