Karlsruhe Institute of Technology

Socio- and Locality-Aware Overlays for User-Centric Networking

Diplom Thesis
by

Martin Florian

at the Institute of Telematics
Department of Informatics
KIT

First reviewer:	Prof. Dr. M. Zitterbart
Second reviewer:	Prof. Dr. H. Hartenstein
Supervisors:	Dr. I. Baumgart
	Dipl.-Inform. F. Hartmann

Day of Registration:	December 1, 2011
Day of Delivery:	May 31, 2012

I hereby declare that I have written this work independently and used no other than the specified sources and tools.

Karlsruhe, May 31, 2012

Zusammenfassung

Mobile Geräte erfreuen sich in den letzten Jahren einer immer höheren Beliebtheit und finden eine weite Verbreitung unter Endnutzern. Gleichzeitig lässt sich eine enorme Entwicklung in ihren Rechen- und Kommunikationsleistungen verzeichnen. Etablierte Kommunikationsparadigmen wissen diese neu verfügbaren Kapazitäten allerdings nicht zu nutzen und konzentrieren weiterhin, innerhalb des Cloud Computing sogar in noch höherem Maße, den gesamten Kommunikations- und Verwaltungsaufwand verteilter Anwendungen auf wenige zentral kontrollierte Server. Dies führt vor allem auch zu einer aktiven Bedrohung der Privatsphäre der Nutzer, was sich immer wieder in öffentlichen Diskussionen widerspiegelt.

Ausgehend von dieser Problematik wurde das *SODESSON*-Projekt ins Leben gerufen. SODESSON steht für *Service-oriented and decentralized social networks*, also die Erforschung von dienstorientierten dezentralen Computer-Netzwerken die auf sozialen Beziehungen zwischen Nutzern aufbauen. Dabei ist das Ziel nicht die Replikation von Funktionen wie sie aus populären Online-Kontaktnetzwerken (wie z.B. Facebook) bekannt sind, sondern die Entwicklung einer generischen *Middleware*, die die Realisierung von sicheren, nutzerorientierten Applikationen erlaubt.

Das Ziel dieser Arbeit war es, verschiedene Ansätze zur Verwirklichung einer solchen Middleware zu erforschen und dabei insbesondere auf die Lokalität der Kommunikation und die Beachtung und Ausnutzung sozialer Verbindungen zwischen Nutzern einzugehen. Konkret wurde zunächst eine Kategorisierung verschiedener Kommunikationsparadigmen unternommen. Darauf aufbauend wurden mögliche Lösungsansätze im Detail diskutiert und verglichen. Als Ergebnis dieser Analyse wurde ein Ansatz, der auf strukturierten Overlay-Netzen aufbaut, ausgewählt und einer weiteren, vertieften Analyse unterzogen. Schließlich diente er als Grundlage für den Entwurf eines Kommunikations- und Datenspeicherungssystems für SODESSON.

Zentrale Ergebnisse des Entwurfs sind unter anderem ein dezentrales System zur Vermittlung von Nachrichten (basierend auf dem *Publish/Subscribe*-Paradigma), das auch dann korrekt funktioniert, wenn der Empfänger zum Zeitpunkt der Sendung nicht verfügbar ist. In diesen Fällen wird die Nachricht ausgeliefert, sobald er wieder verfügbar ist, jedoch unabhängig von der Verfügbarkeit des ursprünglichen Absenders. Weiterhin wurden Konzepte für die Verbesserung der Lokalitätseigenschaften des Kommunikationssystems entwickelt. Außerdem wurde das Overlay-Protokoll *R/Kademlia* um eine Datenstruktur für befreundete Geräte erweitert und Ansätze zur Füllung dieser sozialen Nachbarschaftstabelle vorgeschlagen. Das gewonnene Wissen über die Identitäten befreundeter Geräte eröffnet zahlreiche Möglichkeiten zur Nutzung sozialer Verbindungen durch die SODESSON Middleware, z.B. zur Verbesserung der Datenspeicherung und des Routings.

Im Anschluss an den Entwurf wurde, basierend auf den entwickelten Konzepten, ein Prototyp für die Overlay-Simulationsumgebung *OverSim* implementiert. Dabei musste OverSim zunächst für die Simulation von nutzerorientierten Kommunikationsszenarien angepasst werden, in dem u.a. Nutzer-Entitäten und ein Generator für soziale Graphen hinzugefügt wurden. Außerdem wurde eine Testapplikation für die SODESSON-Middleware entwickelt, die darauf zielt die Anforderungen und das Nutzungsverhalten realer Instant-Messaging-Programme nachzubilden.

Mit Hilfe des implementierten Prototyps wurde schließlich eine umfassende Reihe von Simulationen durchgeführt um die Funktionalität und Leistung der Lösung zu prüfen. Dabei wurden zunächst geeignete Parameterkombinationen zur Maximierung der Systemleistung ermittelt. Simulationsergebnisse zeigten daraufhin, dass die entwickelten Konzepte in der Lage sind in realitätsnahen Netzen ein zuverlässiges Substrat für nutzerorientierte Kommunikation bereitzustellen. Sowohl die gemessenen Kommunikationslatenzen als auch der Bandbreitenbedarf der Lösung erwiesen sich als sehr zufriedenstellend.

Contents

1. Introduction

In recent years, mobile devices like smartphones and notebook computers have grown both more powerful and significantly more popular. The German Association for Information Technology *BITKOM*, for example, estimated that in the beginning of 2012, every third person in Germany owned a smartphone (34%) [31]. In the under 30 demographic, it was every second person (51%). In terms of processing power and communication capabilities, modern phones easily surpass state-of-the-art desktop computers that were common 15 years ago, while modern laptops are often in the same performance category as current desktop systems. Ubiquitous and affordable communication technologies enable mobile systems to exchange data independently of their location, at paces comparable to those of static systems. On top of all that, the performance and functionality increase of portable devices is still an ongoing trend, with more powerful devices entering the market every year.

Despite their elevated capabilities however, mobile devices play a rather passive role in most of today's communication applications. Even worse, with the popularization of cloud computing, they are more and more reduced to dumb terminals for services offered by third-parties. Instead of communicating directly with locally available devices, they form costly and unreliable long-distance connections with dedicated servers and relay their communication via them. Devices are usually not able to communicate with each other without global Internet connectivity, even if they are within wireless range of each another or connected to the same local network. The abundant resources that modern consumer devices have - be it idle processing power, an Internet connection or external pieces of hardware like printers - could also be shared locally to befriended users and their devices. This is also not reflected sufficiently in current communication paradigms.

In addition to the spreading of powerful mobile devices, a different but related trend surfaces today. Namely, the significant popularization of online social networking (OSN) platforms like Facebook and Google+. Like with many popular communication applications, those services are also based on a centralized design, i.e. the communication between users always passes through the respective platforms and the servers of their operators. In existing OSNs, the users' data is thus managed centrally by the organizations running the service. The strong privacy concerns this

raises are often the topic of public discussions. Legal regulations are implemented to address some of the concerns and companies make self commitments to the sensitive handling of personal data. However, the question arises whether a technical solution to those problems might not be more appropriate, by constructing social communication systems that are inherently privacy-friendly.

For researching the possibilities for creating such systems, while also making use of the resources available in modern mobile devices and enforcing a higher locality of traffic, the *SODESSON* project was started [6]. SODESSON stands for *Service-oriented and decentralized social networks*. Its core aim, however, is not the replication of standard OSN functionality as found in Facebook or Google+. Rather than that, a generic middleware is envisioned that leverages social information and provides a user based communication abstraction. Application developers are to be alleviated from worrying about low-level tasks like the discovery of devices, the delivery of messages or the identification and authentication of users. Such a networking approach, that is based around the addressing of users and the direct service provision by user devices, will be referred to as *user-centric networking* from now on. Based on the SODESSON middleware, the rapid development of secure, user-centric applications should be made possible.

1.1 Goals of the Thesis

This thesis deals with the exploration of approaches for an efficient storage and communication infrastructure for the SODESSON middleware, providing a functional substrate for user-centric networking. A thorough analysis of the solution space will be performed, taking the design goals of SODESSON into account. Characteristic properties of the user-centric networking scenario, like the existence of social connections between users and the frequent locality of inter-user communication, will be noted and discussed in the context of the considered approaches.

Based on the performed analysis, one general approach will be examined in greater detail. Specific concepts and mechanisms will be developed and the proposed designs will be implemented in a prototype. Additionally, a simulation environment for user-centric networking will be set up, as well as a test application for SODESSON. Using these tools, the feasibility and performance of the presented solutions will be evaluated.

1.2 Structure of the Thesis

Firstly, in Chapter 2, a few basic concepts and projects will be introduced, the knowledge of which is helpful for the understanding of the presented work. Following that, a first analysis of the problem and the possible solution space will be conducted in Chapter 3. Based on this preliminary analysis, a promising direction will be chosen for further study. The deeper discussion of the chosen approach will be conducted in Chapter 4, followed by the design of a specific proposal based on that approach, in Chapter 5. The implementation of a prototype of the design will be described in Chapter 6. In Chapter 7, simulation results obtained with this prototype will then be discussed and an evaluation of the proposed solutions will be conducted based on those results. Lastly, a summary of the work in this thesis and a collection of suggestions for future efforts will be presented in Chapter 8.

2. Background and Related Work

In this chapter, concepts and ideas will be discussed that form the basis for the presented work. Firstly, the SODESSON project will be introduced formally, followed by the introduction of the *publish/subscribe* pattern of communication and its proposed usage in the SODESSON middleware. Then, types of communication systems will be listed, culminating in a description of decentralized *overlay networks*. The *OverSim* simulation framework, used for implementing and evaluating the proposed designs, will be introduced after that. Lastly, similar and related work to this thesis will be listed and discussed shortly.

2.1 SODESSON

The *SODESSON* (*Service-oriented and decentralized social networks*) research project deals with the exploration of *user-centric networking* techniques. Its main goal is the development of a generic communication middleware for user-centric networking. *User-centric* in this context implies, that users and application developers are confronted only with the addressing of users and do not have to deal with issues related to the discovery of specific devices and the formation of communication links between them. Also, user devices are able to provide services to other devices directly and are expected to assume a central role for the realization of networking functionality. Some important assumptions made by the SODESSON project include:

- Users communicate predominantly with people they know, i.e. with people from their social network.
- Users have multiple devices at their disposal and user-centric applications could be running on all of them simultaneously.
- Devices are powerful and capable of providing services themselves instead of acting only as clients.

Based on those assumptions, the desire is to develop a middleware that is secure and privacy preserving, that is aware of the social context between communication

partners and that allows the provision of services by user devices. Locality and directness of communication are important as well and the middleware's independence from the availability of global Internet connectivity is desirable. These requirements will be elaborated and discussed further in subsequent chapters.

2.2 Publish/Subscribe

In the *publish/subscribe* communication pattern, one set of entities (the *subscribers*) first subscribes to a number of specific *topics*. *Publishers* send (publish) data to those topics. The publish/subscribe system then replicates and disseminates their messages to the respective subscribers. In a publish/subscribe based news service for example, users interested in politics might subscribe to the *politics* topic, while journalists writing about political matters would publish their articles to it. Data published to the *politics* topic gets disseminated to all users that subscribed to it, thus reaching the interested readers. As an important detail in the formal definition of the publish/subscribe pattern, publishers are not directly aware of the receivers of their messages and subscribers do not necessarily have to know the identity of the publishers. Publish/subscribe mechanisms thus create a layer of indirection between senders and receivers, making them oblivious of each other.

In the SODESSON middleware, the publish/subscribe pattern is envisioned as the base for a generic user-centric networking API. The goal is to provide a unified interface that abstracts from details related to the underlying network and enables the dynamic provision of services by individual user devices. Figure 2.1 demonstrates a proposed realization of this idea. In the presented approach, which is also suggested in current SODESSON drafts [6], services are represented as topics. Each device offering a particular service subscribes to the topic that corresponds to that service. Requests related to the service can then be published to this topic, reaching the service providers. Services can also be tied to a specific user, which is noted accordingly in topic titles. In the discussed context, typical user-centric applications like instant messaging are also understood as a form of service, e.g. an instant messaging service. To make a specific example, in an instant messaging application, one of Bob's devices could register to the topic *InstantMessaging@Bob*. When another user, e.g. Alice, wants to communicate with Bob via instant messenger, she can make a publish to that topic and thus reach all of his instant messaging-enabled devices without needing to discover or address them individually. Publish/subscribe topics act as rendezvous points for facilitating the mutual discovery of communication partners.

A deeper discussion of SODESSON's publish/subscribe based communication interface will also be conducted in later parts of this thesis.

2.3 Overlay Networks

Overlay networks have previously been suggested for the use with SODESSON (in [6]) and are also relevant in the context of this thesis. Thus, a short introduction to this concept will be performed here. First, different basic types of communication setups will be listed, giving an idea about overlay systems through their comparison with other approaches. Following that, some basic overlay-related termini will be introduced. Lastly, the two general categories of overlay networks will be discussed - structured and unstructured ones, focusing also on a popular storage system based on structured overlays.

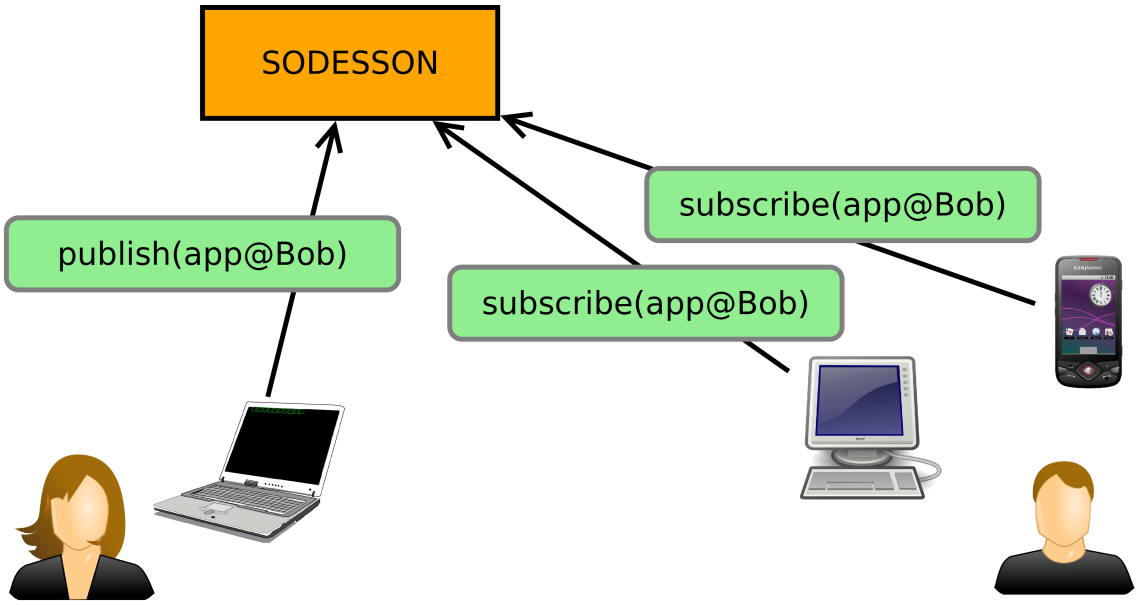


Figure 2.1: The publish/subscribe API envisioned for SODESSON.

2.3.1 Types of Communication Systems

In the following, several basic approaches for realizing distributed communication applications will be introduced.

2.3.1.1 Centralized Systems

In classical centralized systems, one dedicated node (the *server*) is at the center of all communication processes in a distributed application. Typically, the server also provides most of the resources in the system, e.g. computing power and storage space. Other participants (the *clients*) depend on it to provide them with data and relay their messages to other users. No direct client-client communication is possible. This is also referred to as the *client-server* model of communication.

It should be noted that most client-server systems used today follow a more relaxed responsibility distribution than the rigid one-to-many scheme presented above. Responsibilities are still concentrated on the serving side, but the number of servers is variable, allowing applications to achieve some degree of failure resilience and scalability, properties that are not inherent to centralized designs.

2.3.1.2 Hybrid Systems

Hybrid architectures share traits of both centralized and decentralized systems. This is realized by shifting more responsibilities on to clients, e.g. allowing them to form connections between each other, while still not eliminating the client/server separation entirely. The role of the server in such applications often becomes this of the coordinator, providing only little service himself but aiding clients with the discovery of relevant peers. A prominent example is the swarm-downloading system *Bittorrent*[1]. In its original version, dedicated servers (*trackers*) are used for connecting peers to the downloading swarm they are interested in. The actual data transfer is then performed in a decentralized manner between the clients in a swarm. However,

[1] http://www.bittorrent.org/

newer versions of the Bittorrent protocol however also support a trackerless mode of operation which is completely decentralized.

2.3.1.3 Decentralized Systems, Overlay Networks

In a completely decentralized system, no form of central coordination instance exists. All functionality is provided by nodes with equal responsibilities, hence the popular name *peer-to-peer network*. Nodes in a decentralized system can act both as clients and as servers, obtaining and providing services at the same time. Each node maintains connections to a few other peers, its *neighbors*. The resulting network is what keeps all nodes together and maintains the system's functionality.

Often, decentralized networks are formed by creating virtual connections between nodes in an existent network, e.g. the Internet. In these cases, the resulting virtual network is called an *overlay network*. The real network on top of which it is spanned is often referred to as the *underlay network* of that overlay. Overlay approaches can be divided into two main types - unstructured and structured ones - that will be discussed in the remainder of this section. Before that however, some basic overlay-related terminology will be elaborated first.

2.3.2 Overlay Network Terminology

The degree by which the topology of the underlying network influences the formation of links in the overlay, e.g. the mapping degree of the overlay to the underlying network, varies with different approaches. A common measure for this degree is the *stretch* between the length of an average overlay path between two nodes and the corresponding direct path between the same nodes in the underlay. Clearly, overlay paths are always at least as long as the best equivalent path in the underlying network.

Another important term in the context of overlay networks research is the concept of *churn*. Churn is said to exist in a scenario, when the node population forming the overlay is not static. Specifically, there is constant leaving and entering of nodes, either because of node failures and recoveries or due to normal user behavior (e.g. switching their devices on and off). The existence of churn poses a challenge to decentralized systems and measures for dealing with it are integrated into many popular overlay protocols and applications.

2.3.3 Unstructured Overlay Networks

Unstructured overlays are the earliest form of overlay networks. They are used, for example, in early file sharing protocols like Gnutella[2]. As their name suggests, they do not aim at realizing a particular network structure when forming overlay links and instead support a more flexible selection of overlay neighbors. Common techniques used for discovering nodes, datasets or services in a unstructured overlay include:

- *Flooding* - a node's query is duplicated and sent to all of the node's neighbors, who in turn forward it to all of their neighbors etc. Mechanisms exist for

[2] http://rfc-gnutella.sourceforge.net/

avoiding loops and multiple visits of the same node. Flooding queries are often also limited in scope, e.g. they are not forwarded anymore after having been forwarded a certain amount of times.

- *Gossiping* - knowledge gets disseminated in the network a-priori, so that no additional queries are necessary. Specifically, whenever a node generates a piece of information that other nodes might be interested in, it passes this information to a number of it's neighbors. They, in turn, pass it to some of their own neighbors and so on, until a significant portion of the node populations has learned about it.

- *Random walks* - similar to flooding, but queries get forwarded to only one random neighbor. The choice of neighbors can be biased by additional metrics and context-related information.

Often, combinations between those techniques are found. The main disadvantage of the presented mechanisms is their linear complexity. With n being the number of nodes in the overlay, $O(n)$ nodes need to be visited in all approaches in order to achieve correct results. For this reason, unstructured overlays in general do not scale well.

2.3.4 Structured Overlay Networks

Structured overlay approaches feature a dedicated logic that influences the formation of overlay links and the routing of queries. Specifically, the network is constructed reflecting a previously chosen topology structure, like a ring or a hypercube. Each overlay node is assigned an overlay addresses of a given length, also referred to as *key* or *node identifier* (node ID). The numerical space from which those addresses are taken is called the *key space* or *identifier space* (ID space).

The ID of a node determines its position in the overlay geometry. In a purely ring-based geometry for example, two neighbors are also the successors of one another in the identifier space, i.e. their overlay IDs are the numerical successors of each other in the set of all currently existing node IDs. The peers closest to a node in ID space are also referred to as that node's *siblings*. The set of neighbors of a node (the peers with which overlay connections exist) is called the *routing table* of that node.

Queries in the overlay are always routed towards some value in ID space, the destination key. They are forwarded from node to node based on this key, by selecting neighbors that are closest to it in ID space, using a predefined routing *metric*. The overlay structure is usually chosen in such a way, that the distance to the destination is not only guaranteed to decrease in each step, but is also likely do decrease by half or more. Thus, routing paths in structured overlays have an average length of $O(\log n)$ hops with n being the number of nodes in the overlay. This logarithmic routing complexity is a significant improvement to unstructured approaches, where $O(n)$ nodes have to be contacted for guaranteeing correct results. It is also one of the main advantages in using structured overlays in comparison to unstructured approaches.

The routing of overlay messages as described so far is also referred to as *key-based routing* (KBR). Hence, the term *KBR overlays* will be used interchangeably with structured overlay networks for the remainder of this thesis.

2.3.5 Distributed Hash Tables

By far the most popular application for structured overlays is their use for powering a *distributed hash table* (DHT). A DHT is a decentralized key-value storage system, realized by the nodes in the KBR overlay. DHT keys have the same length as the IDs in the underlying KBR overlay, i.e. both components share the same key space. Every KBR node becomes responsible for a portion of this key space, namely all values for which it is the closest node in the overlay in terms of its node ID. DHT requests for a specific key get routed through the overlay and arrive at this responsible node (through standard KBR logic). Data can then be stored on it using a *put* call, or retrieved with a *get* request. For dealing with churn, DHT-enabled nodes create replicas of all datasets they are responsible for and distribute them to other peers. Most commonly, *sibling replication* is employed, i.e. data is replicated and distributed to the overlay siblings of responsible nodes. By using replication, datasets are not lost whenever individual nodes leave the overlay. For all basic DHT mechanisms sketched so far, many suggestions for modifications and alternative approaches exist in the literature.

In different publications, the name DHT can be found do denote anything from a simple KBR overlay to the combination of a KBR overlay with a key-value storage mechanism as described above. In this thesis however, and in concordance to [16], the term DHT will be used only for denoting the storage and replication component described above. Mechanisms related to routing and the formation of a structured overlay are, per this definition, encapsulated in the KBR component, thus forming a clear separation between the two subsystems.

2.4 OverSim Simulation Framework

OverSim [7] is a framework for developing and testing overlay designs. It is based on *OMNeT++* [48], a popular discrete event network simulation environment. Like OMNeT++, OverSim enables the creation of complex simulation models by combining individual *modules*. Modules can be anything from a protocol implementation on an individual host to an abstraction of a whole network, with the composition of modules being very common. OMNeT++ and OverSim modules are defined in the *NED* language and their functionality is implemented in C++. The main method of intermodule communication is through the passing of message objects, although the remote calling of module functions is possible as well.

OverSim enhances OMNeT++ by a number of supportive modules for designing Overlay simulations. Additionally, it contains readily usable implementations of many popular overlay protocols and applications, including various KBR protocols and a DHT application. OverSim features a layered architecture together with common APIs on layer boundaries, to ensure the interchangeability and interoperability of individual components. The KBR component can be rotated freely without changing higher-layer modules, for example. This modularity of design is one of OverSim's greatest advantages. However, the framework's flexibility is not at the expense of its scalability and simulation performance. OverSim is capable of simulating networks with hundreds of thousands of nodes. A more thorough introduction to OverSim's architecture and its main components will be made in Chapter 6.

Figure 2.2 shows a screenshot of OverSim in action, demonstrating the framework's graphical user interface.

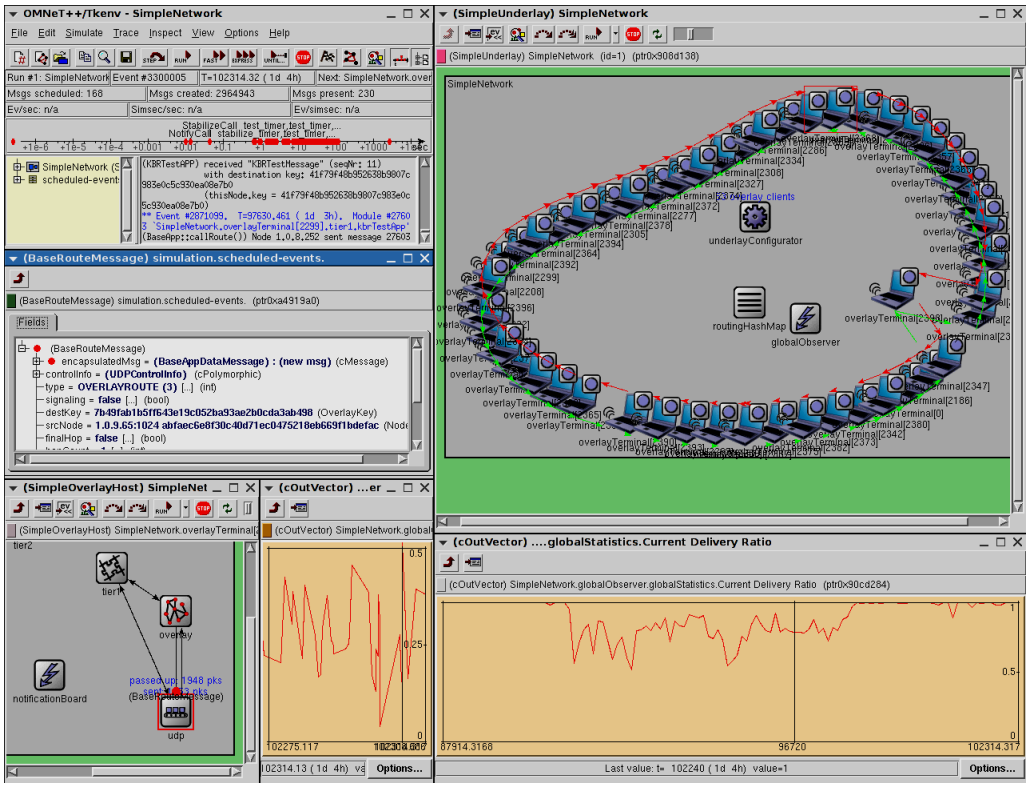


Figure 2.2: Screenshot of OverSim showing the visualization of an overlay topology under churn [7].

2.5 Related Work

In the widest sense, the idea of user-centric networking is as old as the Internet itself. Communication applications like email and instant messenger or even popular modern OSNs can all be described as user-centric. However, classical systems based on the client/server paradigm have a set of inherent problems like their bad scalability and the privacy concerns they rise. Furthermore, none of them provides a generic user-centric networking middleware that can be used for developing arbitrary new applications and enables the flexible service provision by user devices.

Approaches like the *Unified Internet Architecture* [18] (UIA) are closer to the latter goal, while also solving many of the problems found in centralized architectures. However, UIA is actually a device-centric system, with device instead of user addressing. Several decentralized OSN projects exist that are user-centric in this respect, for example [10, 14]. Amongst other things, those proposals also demonstrate the integration of social information into various distributed systems. However, they focus almost exclusively on privacy and the replication of classical OSN functionality. Specifically, they do not at aim at offering a generic communication middleware or enabling the flexible provision of services by powerful user devices.

The locality of communication was also not addressed yet in the context of user-centric networking. However, many contributions exist for improving the locality qualities of decentralized systems in general. Out of those, the concept of hierarchical structured overlays appears promising (e.g. as in [1, 19, 21]).

Unsurprisingly, the most strongly related work to this thesis is the previous work done in the SODESSON project, for example the ideas outlined in [6]. Currently

however, SODESSONs design is still very incomplete. Important details, like the choice of a storage system for application data or the mechanisms for realizing delay tolerant messaging, have not yet been specified sufficiently. Also, neither the locality of communication nor the prospects of leveraging social information for improving aspects of the system have yet been discussed in the context of SODESSON.

3. Analysis

The core aim of this thesis is to explore different approaches for an efficient storage and communication infrastructure for the SODESSON middleware. The resulting architectures should provide a basic set of functions to internal SODESSON components and applications build on top of SODESSON. They should provide persistent data storage mechanisms and a publish/subscribe interface as described in the original SODESSON paper [6]. In the end, they should be able to form the foundation for a wide range of different user-centric networking applications.

In developing a system like this, characteristic properties of the user-centric networking scenario must be taken into account. This includes the existence of social relationships between communication partners which can be leveraged to improve the performance, stability and security of an approach. This also includes the predominant locality of user-centric interactions, as socially connected people are often in physical proximity to each other as well and many communication processes between them are only of local interest. Solutions should thus be designed in a way that is locality-aware and avoids the triangular routing found in most centralized communication architectures today. Ideally, nodes in the same local network should be able to communicate with each other even when no global Internet connectivity is available.

Proposed solutions must also be aware of other requirements to the SODESSON middleware that are not in the central focus of this thesis. This include properties of the usage scenario like the heterogeneity of participating devices and the desire to adapt to different network topologies, like ad hoc networks. This also includes security and privacy related constraints. Those requirements should be realizable on top of the resulting systems and thus be kept in mind. Some common practical challenges however, like bootstrapping and NAT traversal, will be ignored in most parts of this work for the sake of simplicity.

3.1 Requirements

This section serves the further specification of the requirements an efficient user-centric networking infrastructure needs to fulfill.

3.1.1 Design Constraints

The architecture that is being developed here will form the backbone of the SODESSON user-centric networking middleware and will be determent to many characteristics of the final system. Thus, it must be assured that SODESSON's basic design constrains can be met when building on top of the solutions developed in this thesis. This includes issues like reliability - SODESSON should be resilient to connectivity problems and node failures - and scalability - being able to handle worldwide networks with large numbers of users. In order to be deployed on a large scale, the solution should also aim at minimizing its impact on communication latency, bandwidth and the computational load in participating devices.

Amongst other things, the SODESSON middleware is also designed with a strong emphasis on security and privacy. While not a central focus of this thesis, the compatibility of the resulting solutions with a public key security infrastructure as mentioned in the original SODESSON paper [6] should thus be kept in mind. The same holds for privacy concerns, as a communication and storage solution that is inherently privacy-averse would not be a suitable basis for SODESSON.

3.1.2 Core Functionality

The core functionality required of the presented solutions reflects the basic design goals of the SODESSON middleware, namely the provision of a substrate for user-centric networking and communication. This implies the need for a per-user addressing scheme that alleviates application developers from issues like the selection of an appropriate device to communicate with. It is the responsibility of the solutions developed in this thesis to keep track of a user's devices and to make decisions about which device to contact. This is non-trivial as users might be logged on to several devices simultaneously.

In addition to user addressing and device selection, the SODESSON middleware aims at providing a set of communication and storage services for applications. The SODESSON paper [6] identifies three general categories of such services:

Direct Participating devices form a direct connection between each other through which data is being exchanged. This might be appropriate for applications like file sharing where large amounts of data are being transferred.

Persistent Data gets stored by the SODESSON middleware and is permanently available, even if the user's devices are offline. A persistent storage like this can be used for storing photo albums or profile pages like in more traditional online social networks.

Hybrid Direct communication with a possible fall back into persistent storage when target devices become unavailable. A prominent example here are instant messaging applications in which messages sent to an unavailable communication partner can be persisted for later delivery, even if the original sender becomes unavailable himself.

The concrete realization of those services is transparent to the actual applications. Decisions like routing or when to fall back into persistent storage are responsibilities

of the SODESSON middleware and, more specifically, of the communication and storage infrastructure that is being developed here. The presented approaches must also be aware of underlay-related issues like addressing, reachability and connection maintenance.

In SODESSON's current design, a great part of this transparency is achieved through the provision of a *publish/subscribe* interface. It is foreseen that the publish/subscribe paradigm will be a sufficient base for the development of sophisticated user-centric applications. In this respect, the realization of efficient publish/subscribe functionality in a user-centric context is one of the main concerns of this thesis and the solutions presented.

In addition to modeling user-centric communication, the publish/subscribe pattern can also be used for interacting with a persistent storage service. Interested peers could, for example, subscribe to a resource instead of just retrieving it once. Subscribed peers would then receive update notifications, enabling them to maintain a fresh local version of the resource without polling it constantly. Still, an interface that is tailored specifically to the data storage/data retrieval scenario might be more appropriate for internal use within SODESSON. Developed solutions should thus provide specialized data storage functions as well, e.g. a key-based *put/get* interface similar to the one used in hash tables.

To sum up, following key challenges are left to be addressed by the proposed communication and storage infrastructures:

- Defining a user-centric addressing scheme with several devices per user. Mechanisms for selecting the best device to communicate with should be supported.
- Providing a persistent data storage system independent of the availability of particular devices.
- Providing a communication abstraction based around the publish/subscribe paradigm that is able to persistently store data for later delivery if a communication partner is unavailable.

3.1.3 Social Context Leverage

The availability of social context information is a distinctive property of the SODESSON usage scenario. It is expected that most, if not all, of the interactions over the SODESSON middleware will be based on social relationships, i.e. will be between users that are connected socially. It is furthermore likely that users will be willing to assist their social contacts by helping them establish communication links or by storing data for them. In decentralized approaches, social information can be used for optimizing routing decisions. Trusted nodes can be preferred when forwarding messages, increasing the chance that a query will yield correct results, which improves the overall robustness of the system [42]. Social links might even be leveraged for alleviating the bootstrap problem in such approaches. The information from the social graph is thus a promising platform for optimizing communication and storage processes in SODESSON. The solutions developed in this thesis should be aware of this and aim at incorporating social information into their design.

3.1.4 Locality Awareness

Locality plays an important role in SODESSON as well, as socially related users are often also in proximity to each other[1]. This proximity can be leveraged for making communications processes between them more efficient. If two users reside in the same subnetwork for example, it is not very economic to route the traffic between them via a far away server, as is the case with many centralized applications today. Likewise, it is also more desirable to store data on nodes that are topologically close to both the data source and its expected receivers. In a similar way, locality information could also be leveraged when implementing caching schemes, so that interesting pieces of data can be copied and kept locally. Finally, focusing communication and storage to a local scope can also improve SODESSON's performance when there is no stable Internet connectivity. Ideally, SODESSON applications should be able to function without any global connectivity at all. With this possibilities in mind, it is desirable for the solutions developed in this thesis to be aware of locality and leverage it appropriately.

3.2 Possible Approaches

This section discusses possible approaches to a communication and storage infrastructure for user-centric networking. Only the basic characteristics of each approach will be evaluated. The most interesting approach will be analyzed and developed further in subsequent chapters.

The possible approaches can be divided into classes as depicted in Figure 3.1. The structure of this section is based around this classification.

3.2.1 Centralized Approaches

The client-server paradigm is at the heart of todays Internet and the foundation for many popular user-centric communication applications. It might thus be applicable to the scenario of this thesis as well - the development of a generic middleware for user-centric communication. A great part of SODESSONs core functionality has actually been implemented in a centralized manner already. Online social networks like Facebook and Google+, for example, need to provide very similar functionality to their internal components as the SODESSON middleware to SODESSON applications. Systems like this show that high performance can be achieved when following a client/server approach. They also achieve a high degree of fault tolerance and scalability, handling hundreds of millions of users per day[2]

3.2.1.1 Scalability and Fault Tolerance

However, this degree of scalability and fault tolerance comes at a cost. Since most resources are provided by a set of dedicated servers, the size of the user base is restricted by the amount of available servers and their combined bandwidth, storage

[1] The influence of geographic distance on friendship has been identified and discussed on several occasions, most recently in [47].

[2] In May 2011 for example, the popular OSN Facebook counted 721 Million active users, i.e. users that logged in at least once in a 28 day period. On average, those users logged into the site roughly once every 4 days [47].

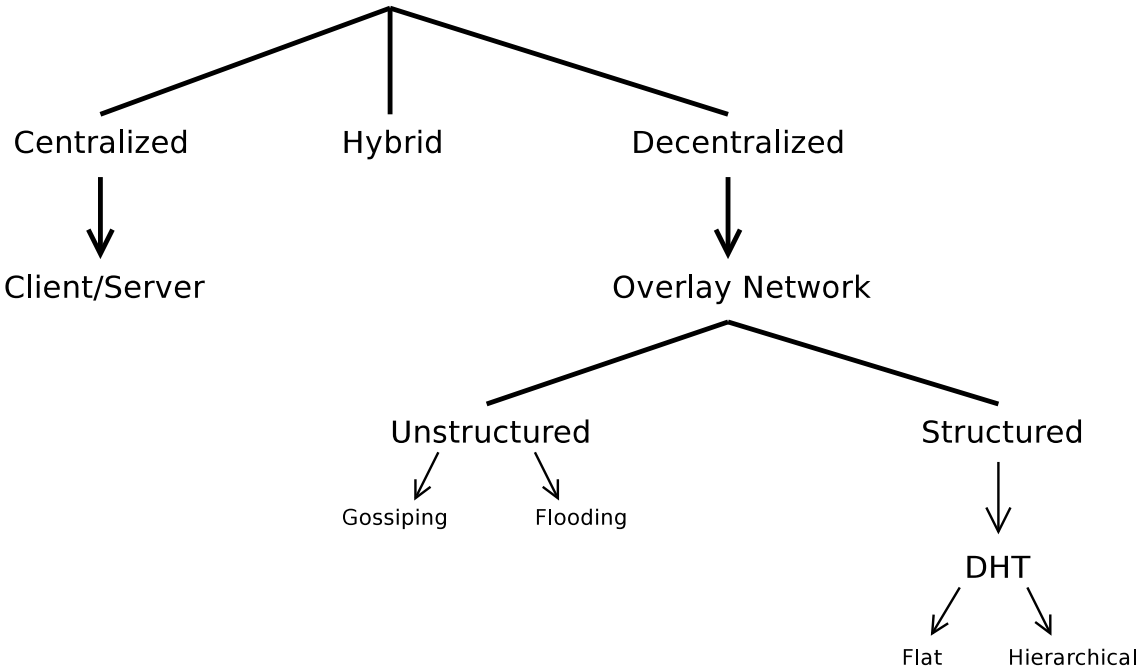


Figure 3.1: Possible approaches to a communication and storage infrastructure for user-centric networking

and processing capacities. This introduces the need for large server farms that are expensive and difficult to maintain. Data centers are also very energy consuming, leading to a negative ecological impact. The growth of the server set introduces challenges that are very familiar to the challenges of decentralized systems, like the dynamic adaptation to node failures and the maintenance of data consistency. However, client/server applications are still significantly more prone to node failures and denial-of-service (*DoS*) attacks than architectures with a more equal distribution of responsibilities. A problem in a few critical servers or infrastructure components can lead to a severe functionality decline for the whole system. For the same reasons, centralized systems are also prone to censorship, as connections to key servers can easily be blocked by ISPs.

3.2.1.2 Privacy

Another shortcoming of centralized architectures is the issue of ensuring user privacy. All communication paths in the client-server model need to pass through a centralized server which is, in most cases, operated by a third party. This potentially untrusted third party (the server operator) finds itself in a position where it can intercept and read sensitive user data. Clearly, this is highly undesirable. User privacy is usually protected through laws, usage agreements and privacy policies. However, legal rules can be broken on malicious intent and some of the agreements might even contain hidden loopholes inconceivable to the average user. Additionally, this form of security does not protect against unintentional information leaks, where client data may become compromised due to failures or security breaches on the serving side. Government agencies can also, under certain conditions, override legal obligations and request the release of sensitive information about particular users. While this might be desirable in some cases, it gives oppressive governments

a possibility for spying on its population and is thus, in general, an undesirable effect in the context of maintaining democratic societies.

A possible technical solution to the privacy problem is the protection of sensitive data through means of encryption. For example, messages and data can be secured through asymmetric cryptography, similar to the use of PGP in email. However, in this approach only the content of a message can be protected - meta information needed for the correct delivery (e.g. information about the sender and receiver) needs to remain visible to the server. The server operator can thus still make out who is participating in a specific communication processes, i.e. who communicates with whom. Additionally, the solution is only feasible when the receivers of a message are known in advance (as it needs to be encrypted with the correct public key), making it unusable for applications like photo sharing. Lastly, the incentive for service providers to implement privacy enforcing encryption systems on their servers is very low. Rather than that, many business models actually depend on the possibility to process an individual user's private data, e.g. for providing targeted advertisement.

3.2.1.3 Locality Awareness

Centralized approaches have inherently bad locality properties. Communication paths always lead through at least one distinct server, even when a more direct connection between client devices is possible. This problematic is also referred to as *triangular routing*. Triangular routing has an especially high impact in social user-centric applications, as socially connected users are often also physically close to each other as well, e.g. in the same building or city[3]. With centralized approaches however, nearby nodes still need to contact and route through a (probably far away) server in order to communicate. This leads to higher latencies and is wasteful to network resources. Another consequence of triangular routing is the dependence on global Internet connectivity. If no server can be reached the communication through a centralized application becomes impossible, even if the users wanting to communicate reside in the same subnetwork.

Several attempts exist to alleviate the problems of triangular routing, like the use of proxies and content delivery networks (*CDNs*) or the distribution of the server set into multiple server farms in different locations. However, those solutions induce additional costs and are thus limited in their scope and impact. They furthermore focus predominantly on improving the latency of applications and do not address problems like the dependence on global connectivity.

3.2.1.4 Conclusion

The client-server paradigm is a powerful foundation for sophisticated distributed applications. Centralized applications are comparatively easy to develop and capable of handling large loads with hundreds of millions of users. However, a chain of inherent problems, including the costs involved in maintaining such scalability, the issues concerning user privacy and the unavoidability of triangular routing, makes the suitability of this approach for the purposes of this thesis questionable. The exploration of more decentralized approaches seems appropriate.

[3]The authors in [47], for example, identify correlations between geographic proximity and friendship in the social graph of the OSN Facebook.

3.2.2 Hybrid Approaches

The introduction of more responsibilities for clients alleviates many of the problems of classical centralized systems, like the problem of scalability and, to some extent, the large detours in routing. Several central shortcomings remain untouched though, like the fact that the server still plays a critical part in the communication process. He is thus still a single point of failure and an entry point for disrupting and controlling the whole network. The global connectivity problematic remains unsolved as well, as clients need to connect to the server at least once in order to participate in the application. Often, a constant connection to the server is required for optimal functioning.

With those shortcomings in mind, the rest of this thesis will focus on the exploration of fully decentralized approaches.

3.2.3 Unstructured Overlays

Fully decentralized unstructured overlays are the simplest form of a pure peer-to-peer network. Their flexible structure makes them adaptable to different requirements and easy to design and implement. In comparison to DHTs, they allow for far more control over the storage locations of particular datasets. Any node can hold copies of any piece of data, while no node is obliged by a dedicated distribution logic to store specific items. This flexibility is desirable, as it enables users to hand-pick storage locations based on information from the social graph. They furthermore support fuzzy searches, where the exact key of the sought data does not have to be known in advance. This lookup technique could be expanded to support dynamic service discovery queries, e.g. "all befriended devices nearby that have spare CPU resources".

In general however, unstructured overlays offer worse performance than their structured counterparts [11, 37]. Most importantly, they scale very badly. Lookup operations, for example, have linear complexity in the number of participating nodes ($O(n)$). This implies that, in the worst case, a query needs to pass through all nodes in the network before a result can be determined. The lookup performance can be improved through aggressive caching and the replication of datasets across several peers. However, the impact of those techniques is limited as they do not protect against worst-case scenarios like the search for an object that doesn't exist in the network at all. Such cases might still require the querying of all nodes for the sake of correctness. Consistency and correctness requirements can be weakened, which allows for further optimizations like the limitation of flooding hops. However, consistency and correctness might be desirable properties in the context of a general-purpose user-centric networking middleware.

3.2.3.1 UIA Social Routing

An interesting solution to the scalability problem of unstructured overlays was proposed by Ford as part of his work on the *Unmanaged Internet Architecture* (UIA) [18]. UIA is a communication middleware implementing a *personal name system* for the easy addressing and discovery of befriended devices. It is similar in some ways to the SODESSON platform on which this thesis is based, with the important distinction that while SODESSON is user-centric, UIA is device-centric and deals only

with the identification and addressing of devices. UIA uses an overlay addressing scheme based on unique *endpoint identifiers* (EIDs) for each device. Several approaches have been proposed for routing with those identifiers. Here, the focus will be on *social routing*, an approach based on an unstructured peer-to-peer network between devices. The network is used mostly for providing EID lookups, resolving them into underlay addresses like IPs.

The difference between UIA's social routing overlay and classical unstructured overlays lies in the fact that the UIA overlay is based on social links. UIA assumes social connections between users and the devices in their personal name systems. It uses this information for choosing overlay neighbors and for restricting the flooding of queries in the network. Lookups are performed in rounds, flooding only directly befriended nodes at first and moving to larger friendship distances (friends of friends and beyond) if no results were found. In this way, queries can be restricted to nodes in the social neighborhood, which greatly improves the scalability of the overlay. This technique can be applied to other domains as well, not only to the lookup of device IDs. In fact, most applications realizable with unstructured overlays can be enhanced by restricting the flooding and gossiping traffic to socially close peers.

This technique has one major drawback however, namely that it is restricted to communication processes between socially connected nodes. If, for example, a required dataset is stored on a node outside a user's social circle (e.g. the node is looking up the EID of a random device), the user will not be able to efficiently locate that dataset. Evaluation results of UIA's social routing show that while nodes with friendship distances of one and two can be reached quickly and with high probability, the lookup performance degrades sharply at higher distances. At three hops (friends of friends of friends), lookups succeed only 50% of the time when 40% of the devices in the network are stable and up to 256 nodes are contacted during flooding [18].

It can be argued that this is not a problem in user-centric networking scenarios where most communication is based around social connections of one or two hops. The design of SODESSON reflects this standpoint as well - communication processes with no underlying social connectedness are not expected. Still, a restriction like this could lead to practical inconveniences that need to be discussed. For example, users might meet new contacts that are very far from their usual social neighborhood and that they do not want to add to their contact lists immediately. A chicken-egg problem surfaces - how can a user forge social bonds to nodes outside his social circle if he cannot communicate with them in the first place? Other challenges that need to be discussed include the implementation of publish/subscribe functionality and the possibility of integrating locality information into the solution. For the former, it is interesting to see whether the restriction of flooding improves the performance of publish/subscribe-style communication as well, as plain unstructured overlays have been found to perform poorly in this respect [4]. For the latter, proximity-restricted flooding could be discussed, e.g. for service discovery in local networks.

3.2.3.2 Storage

Finally, in order for this solution to satisfy the requirements outlined in section 3.1.2, a reliable storage system must be implementable on top of it as well. Unstructured,

fully decentralized storage networks have already been realized in filesharing applications like *Gnutella*[4] and *Turtle F2F* [36]. In those systems however, nodes store only data that is interesting for them themselves. It is unlikely that this behavioral model will provide a sufficient level of data availability in the scenario of this thesis, especially in cases where a user does not own many devices. A possible solution is the proactive replication of data to nodes that might not be interested in it directly, similar to the usage of friend peers for storing backups (as proposed in [46]). However, the coordination of those replicas, reacting to node failures and keeping replicas up to date, might be difficult on top of an unstructured network and will likely induce a high communication overhead.

3.2.4 Distributed Hash Tables

This section examines the feasibility of key-based routing (KBR) overlays and distributed hash tables (DHTs) as a basis for a user-centric networking middleware. KBR overlays appear promising, as they are decentralized (and thus not prone to many of the shortcomings of centralized systems) and yet offer high performance in comparison to unstructured approaches. They are especially interesting due to their scalability, with no operation involving more than $O(\log n)$ hops on the total number of overlay nodes. The potential of KBR/DHT systems has already been noted in the original SODESSON paper [6], where they are envisioned to power both the storage system and the public/subscribe mechanisms. Several related projects have chosen DHTs for enabling some of their core functionality, including *Safebook* [14], *PeerSoN* [10] and *the Social Distributed Hash Table* (SDHT) [33]. Routing mechanisms based on DHTs have been proposed for the Unmanaged Internet Architecture, as an alternative to unstructured social routing [18]. None of these projects focuses explicitly on providing general user-centric networking functionality. However, their scenarios are similar and many of the problems they face, like the lookup of users and devices, are relevant in the context of this thesis as well.

Structured overlays are an active research field, with many analytical publications and countless proposals for specific routing and storage algorithms. KBR overlays have been enhanced with locality awareness, making them adaptable to the underlay topology and improving the directness of their overlay routes (the *stretch* between the overlay path and the direct underlay route). Proposals that include this kind of optimizations include the protocols Bamboo [39] and R/Kademlia [22], which are also interesting for other reasons like their performance under churn. Routing improvements based on the integration of information from the social graph have also been discussed [42]. The findings of those works can be incorporated into the design of a general user-centric networking architecture.

On the downside, despite the extensiveness of structured overlays research, few production systems with a significant amount of users actually exist. KBR/DHT-based systems have been used mainly for filesharing, in applications like BitTorrent (trackerless mode). Their practical, real-life feasibility for powering large user-centric applications is thus still largely uncertain. Another general drawback of KBR/DHT systems is their lack of flexibility. In a classical DHT, data is, in effect, distributed randomly across peers, ignoring factors like locality or the existence of social ties. Its rigid data distribution scheme makes it difficult to influence the choice of storage

[4] http://rfc-gnutella.sourceforge.net/

locations. However, both the restriction of data to a certain topological scope, e.g. for data that is only interesting locally, and the explicit preference of befriended nodes as storage locations, e.g. for higher resilience against malicious nodes and for better privacy, might be desirable. Partial solutions to this problem have already been proposed. PeerSoN and Safebook use befriended node for storing sensitive data and put references to those nodes into the DHT, thus using it only as a lookup service. For optimizing storage locality, *hierarchical DHT* (HDHT) designs that organize peers into clusters based on their proximity in the underlying network might be considered as well [1, 2, 19, 21]. HDHTs might also be helpful for improving the overlay's performance on network partitioning and merging, as storage domains become more clearly separated. HDHTs are, however, still a very young area of research with many open questions, including the concrete mechanisms for cluster forming and the actual practical feasibility. An easier but less thorough solution for optimizing the local availability of stored data is to use replication and caching, distributing interesting data pieces evenly across the underlying network.

3.2.4.1 Realization of Core Functionality

This section explores the feasibility of KBR/DHT-based solutions to provide the basic functionality required by the SODESSON middleware, as outlined in section 3.1.2.

Addressing and Lookup For addressing and lookup, two challenges are faced. Firstly, finding an appropriate addressing scheme for devices and enabling the efficient lookup of a device's underlay address. Secondly, implementing a user-addressing scheme and a mapping from users to devices that supports both automatic device selection and the lookup of all devices associated to a user. For device addressing, a scheme might be chosen in which a device's unique address is the same as its identifier in the overlay. In this way, standard key-based routing can be used for looking up devices. Arbitrary addressing schemes without a connection to the overlay address can be used as well. In this cases, a DHT-based lookup system similar to UIA's *Identity Hash Routing* can be employed, which uses device addresses as keys to the device's current underlay address. A DHT lookup system like this can be used for solving the user addressing problem as well, as seen for example in PeerSoN.

Storage Users of the SODESSON middleware should be able to store data persistently, so that its availability does not depend on the availability of a particular user and its devices. A straightforward way for realizing this functionality is to put data directly into the DHT, leaving it up to the DHT implementation to handle availability and integrity related issues. Most DHT protocols are equipped with efficient replication strategies for such tasks, allowing them to achieve high data availability. However, in networks with a lot of churn those strategies might lead to a high overhead in bandwidth and storage space consumption, especially if large chunks of data are being stored. Alternative storage strategies might be used in this case, that are unconnected to the classic DHT logic and can eventually leverage node proximity and the existence of social trust. Safebook and PeerSoN propose storage approaches that go in this direction.

Publish/Subscribe Publish/subscribe mechanism are expected to form the foundation for most user-centric communication processes in SODESSON. A KBR/-DHT-based solution should thus provide an efficient interface for such functionality. Two principal approaches for implementing publish/subscribe-style communication on top of structured overlays can be found in the literature - one that is based around constructing a multicast delivery tree on top of the overlay topology and another that forms direct connections to each subscriber. *SCRIBE* [12] is an example for the former technique and a version of the latter is employed in the *Internet Indirection Infrastructure* (*i3*) [44].

As can be seen, the basic functionality requirements outlined in section 3.1.2 can easily be met when building upon KBR overlays and DHTs. Additionally, several different approaches to realizing each of the required mechanisms exist, which allows for greater flexibility when designing the system.

3.2.5 Discussion

Summarizing the analyses of all presented approaches, decentralized systems appear to be a better basis for a general user-centric networking middleware than client/server setups or hybrid approaches. They perform better in terms of resilience and costs and are more suitable for enforcing user privacy. As for the choice of an optimal overlay type, both unstructured approaches employing social information and KBR/DHT-based solutions appear promising to some extent.

However, unstructured overlays have a set of practical issues that make their feasibility for real-life applications questionable. They are also, despite the restriction of lookup tasks to social contacts, still likely to offer a lower overall performance than KBR-based systems for lookup and routing tasks[5]. Their suitability for powering a storage system with adequate availability is largely uncertain.

Apart from the rigidity of the data distribution and access schemes they impose, approaches based on KBR overlays and DHTs do not have any critical open issues or limitations. Most of the functionality that is relevant for supporting a general user-centric networking middleware has already been realized with the help of DHTs and with good performance. Additionally, the pursuit of a KBR/DHT-based approach offers a wide range of different design choices, leaving room for specific optimizations. Most prominently, the possibility for integrating mechanisms known from unstructured designs is given, like the free selection of storage locations or the support for fuzzy searches (thus combining the best from both worlds).

Thus, an approach based on structured overlays and distributed hash tables is found to be more suiting for the goals of this thesis than proposals based on unstructured overlays. Because of this, and since developing a working prototype for both approaches would exceed the scope of this thesis, the focus will be exclusively on developing a KBR/DHT-based system from now on.

[5]The amount of hops needed to route to a socially close node in a setup similar to UIA's social routing requires the querying of $O(m)$ nodes, m being the number of socially connected overlay neighbors per node. In many cases, this is still worse than the standard $O(n)$ complexity found in KBR overlays. Also, with KBR overlays, non-socially connected nodes can be efficiently found as well.

3.3 Summary

In this chapter, the requirements to a communication and storage infrastructure for user-centric networking were outlined. Based on those requirements, a number of general approaches for developing such a system were proposed. A short analysis of each of the approaches was performed, together with a discussion of their advantages and disadvantages. As a result of those discussions, decentralized approaches were found to be the most suitable for realizing the goals of this thesis. Specifically, an approach based on key-based routing overlays and distributed hash tables was found to be most promising, mainly due to its logarithmic scalability and the high degree of design freedom it allows.

The DHT-based approach to a communication and storage infrastructure for SODESSON will be put to a more thorough analysis in the subsequent Chapter 4. Following that, a specific solution will be designed and evaluated (in later chapters).

4. Structured Overlays for User-Centric Networking

This chapter features an additional, deeper analysis of the KBR/DHT-based approach outlined in Chapter 3. A more thorough review of the related work will be conducted first, discussing the use of key-based routing overlays and DHTs in projects similar to SODESSON. Following that, two publish/subscribe mechanisms based on structured overlays will be presented. Proposals for realizing the actual key-based routing and distributed storage functionality will then be regarded in greater detail. Storage mechanisms not based on standard DHT logic will be considered as well. Finally, ways for dealing with the partitioning and merging of the overlay will be discussed.

4.1 Related Work

Distributed hash tables running on top of structured overlays have already been proposed for the use with SODESSON. In the original design [6], they are envisioned as the foundation for a persistent storage system handling profile pages and similar small datasets. Other projects with similarities to SODESSON make use of DHTs as well, for example *PeerSoN* [10] and *Safebook* [14]. Those frameworks aim at mimicking the functionality of popular online social networks like Facebook and Google+, allowing the publishing of profile data, photo albums and status updates. In Safebook, the bulk of the storage (especially for privacy-sensitive data) is handled by an additional unstructured network of privacy-enforcing proxies called *matryoshkas*. A DHT is used for looking up storage locations in the matryoshka network. PeerSoN aims at reaching a similar configuration. It uses a DHT as a lookup tool while the actual data is either exchanged directly between source and destination or persisted on few individual nodes. However, the specific storage mechanics are still work in progress. Storage criteria like social connectedness or temporal and geographical diversity are being discussed, just as opportunistic delivery schemes similar to techniques used in delay-tolerant networks (DTNs). PeerSoN's current working prototype uses a DHT for persistently storing small pieces of data, e.g. for realizing asynchronous messaging.

The *Social Distributed Hash Table* (SDHT) [33], another related project, uses a DHT in combination with an additional overlay structure reflecting the social network between users. Again, the DHT is used mainly as a lookup service for nodes and resources. The social overlay is used for enabling flooding-based searches, similar to the lookup in UIA's social routing overlay (section 3.2.3.1). In this way, SDHT combines advantages of unstructured and structured overlays, allowing flexible, socially-aware searches and, at the same time, efficient resource retrieval through a DHT.

DHT-based lookup of device addresses is also being performed in *identity hash routing*, a routing mechanism proposed for UIA. In this scheme, all UIA nodes form a global DHT structure, using their endpoint identifiers (EIDs) as addresses in the DHT. EIDs can then be resolved to underlay addresses using standard DHT mechanisms. In addition to lookup however, the identity hash routing DHT can also be used for routing actual data traffic between nodes. This is seen as a solution for dealing with connectivity problems and NAT traversal [18].

The presented projects demonstrate how DHTs can be used in scenarios that are similar to that of this thesis. However, none of them shares the main goal of SODESSON, namely the creation of a generic middleware for user-centric networking. PeerSoN and Safebook focus more on the replication of classical OSN functionality and on guaranteeing strong privacy. SDHT concentrates on content sharing and search. UIA is a communication middleware, but addresses devices instead of users, providing a weaker abstraction than SODESSON. As a result, several important challenges are not addressed in those examples.

Firstly, none of the presented systems deals with the problematic of addressing users while handling multiple devices for each. Secondly, all of them lack a publish/subscribe-based communication abstraction as requested by the original SODESSON paper. Lastly, the examples are mostly not locality-aware and have not been assessed in scenarios involving network partitioning and losses of Internet connectivity.

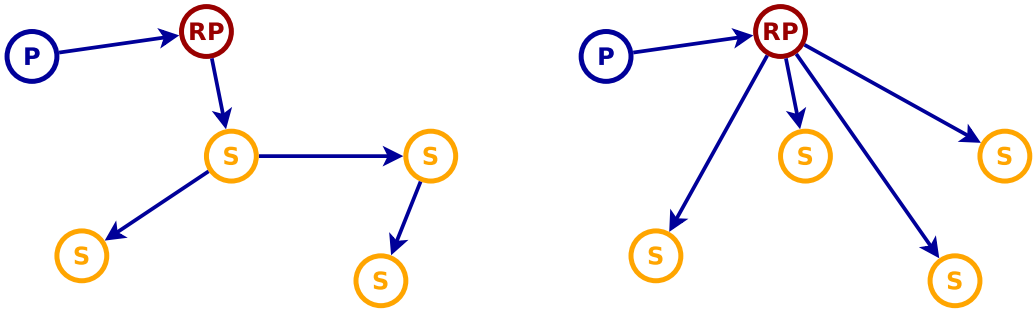
4.2 Publish/Subscribe

Communication in SODESSON is expected to rely exclusively on a publish/subscribe interface for message passing. However, publish/subscribe functionality was not yet discussed in the examples presented in Section 4.1. Still, several proposals for decentralized publish/subscribe systems exist in the literature, many of whom employ KBR overlays and DHTs.

The focus here will be on just two notable examples, *SCRIBE* [12] and *i3* [44]. Additionally, the possibility of implementing delay-tolerant messaging (i.e. hybrid services as described in Section 3.1.2) on top of those publish/subscribe mechanisms will be discussed.

4.2.1 SCRIBE

SCRIBE [12] is a multicast architecture based on the Pastry overlay [40]. It works by constructing a multicast tree from a *rendezvous point* (RP) to all group members (subscribers). Figure 4.1a shows an example of such a message dissemination tree. The rendezvous point is the overlay node responsible for the multicast group's *group*



(a) Publication dissemination in SCRIBE. (b) Publication dissemination in i3.

Figure 4.1: Comparison of publish/subscribe approaches.

ID. Group IDs can be generated from topic strings by hashing them into ID space. The tree is built using standard overlay links, following the lookup paths between RPs and subscribers.

The use of a multicast tree for message dissemination has the advantage of minimizing the bandwidth overhead when sending large pieces of data, as messages get duplicated less and closer to their destinations. This is efficient only in conjunction with a locality-aware overlay like Pastry, where proximity information is used for determining suitable overlay neighbors. In overlays with a high stretch, that reflect the underlay topology poorly, this technique can potentially lead to an increase in communication delays. Even with a low path stretch though, messages delivered over SCRIBE's multicast tree still arrive slower than messages sent directly. At the minimum, they need to pass through the RP, which is already a form of triangular routing. For small messages, that do not consume a lot of bandwidth, a protocol that delivers them outside of the overlay structure might thus be more appropriate. The same goes for multicast groups with only few subscribers.

Incidentally, most messages in SODESSON are expected to be rather small, e.g. like instant messages. Most topics, too, are unlikely to have significant amounts of subscribers at any given time. A SCRIBE-based publish/subscribe system might thus not be optimal for the use with SODESSON. Still, it might be considered for cases where large messages are being transmitted to many users, e.g. when a user pushes a photo album to all of his friends.

4.2.2 I3

The *Internet Indirection Infrastructure* (*i3*) [44] is another take on providing publish/subscribe functionality in a decentralized manner. Its main difference to the SCRIBE approach is that it doesn't construct a multicast tree, but instead disseminates messages in one hop from the rendezvous point to the subscribers[1]. Thus, all subscribers need to register directly at the RP. The direct dissemination mechanism is faster for smaller messages and fewer subscribers, but makes the RP a potential bottleneck. It is unlikely though that the RP will actually be overloaded in the average SODESSON use case. And even if, caching and the use of proxy nodes can be used to alleviate the problem, as mentioned in the original i3 paper. A graphical comparison of the message routing in i3 and SCRIBE can be seen in Figure 4.1.

[1]However, techniques for forming multicast trees on top of i3 have also been proposed [44].

However, the centralization of publish/subscribe traffic in the RP also raises security and privacy concerns. Furthermore, it still introduces an additional level of indirection and thus leads to triangular routing (like in the SCRIBE approach).

4.2.3 Hybrid Services

SODESSON's publish/subscribe system should also be able to support hybrid services as described in Section 3.1.2. However, none of the presented publish/subscribe systems is capable of persisting messages to offline recipients and delivering them at a later time. Solutions to that problem would need to face following challenges:

1. A reliable, decentralized storage system must be realized for the persistent storage of non-delivered messages.
2. The set of proper recipients for a message must be generated and stored, so that the message can be delivered to all of them but not to subscribers joining the topic after the initial publication of the message.
3. Information about who has already received a message must be gathered and stored, to avoid the duplicate delivery of messages and so that messages that have reached all of its recipients can be erased from the persistent storage system.
4. The delivery of a message must be triggered whenever one of its recipients, that did not receive it previously, becomes available.

While the first problem is trivial, no comparable work was found in the literature that addresses the latter three. Thus, the realization of delay-tolerant messaging services on top of a KBR/DHT-based publish/subscribe system remains an open question.

4.2.4 Conclusion

Based on the preceding analysis, a publish/subscribe architecture similar to i3 seems most appropriate for meeting the goals of this thesis. A system like this would perform well in situations where small pieces of data are being distributed to a small set of subscribers - a likely use-case in SODESSON. However, several open challenges remain with this approach, including the centralization of traffic and responsibilities in one single RP node and the lack of mechanisms for realizing delay-tolerant message delivery.

4.3 Overlay Structure and Routing

A multitude of different KBR protocols has been proposed in the literature. Out of those, *Bamboo* [39] and *R/Kademlia* [22] are especially interesting in the context of this thesis. They both show good locality properties and have performed well in previous evaluations. The two protocols will be introduced here in greater detail[2]. Additionally, hierarchical overlay structures will be presented and the possibilities for integrating social graph information into key-based structured overlays will be discussed as well.

[2]Adaptive algorithms like Accordion [28] and Chameleon [9] are interesting as well, but their consideration would exceed the scope of this thesis.

4.3.1 Bamboo

Bamboo [39] is a structured overlay with excellent locality properties and a low communication overhead. It is based on the Pastry protocol [40], optimizing its overlay maintenance mechanisms for a better handling of churn. Like Pastry, Bamboo forms a bidirectional ring topology in combination with a hypercube topology. Nodes are arranged on the ring based on their overlay IDs and keep track of several predecessors and successors, their *leaf set*. An additional *routing table* is used for establishing the hypercube. Nodes in the routing table are organized based on the length of their longest common prefix with the current node. When choosing entries for a given prefix length, nodes that are close in the underlying network are preferred, i.e. nodes with whom low communication latencies exist. This is referred to as *Proximity Neighbor Selection* (PNS) and is usually implemented by measuring round-trip times (RTTs) to potential neighbors or by using a network coordinate system like Vivaldi [15]. The routing table is filled by looking up random addresses with a given prefix and storing the result (which is the closest existing node ID to that address) or by asking neighbors for pieces of their routing table (e.g. all nodes they know with a given prefix). Depending on the implementation it is also possible to learn about new neighbors while forwarding their queries or while performing own lookups.

Having established a correct topology, messages are routed through the overlay in a recursive fashion, by forwarding them from node to node. At every hop, the currently visited node first checks if a member of its leaf set is responsible for the destination address. If so, this node becomes the next and final hop of the routing chain. Likewise, the current node also checks if it itself is responsible for the destination key and terminates the process if so. If neither the current node nor a member of its leaf set could be identified as the final destination, a suitable next hop is chosen from the current hop's routing table. The choice is based on a prefix-based metric, i.e. a metric where the distance between two keys id_A and id_B is equal to $lcp(id_A, id_B)$, the longest common prefix of the two. Thus, a neighbor N is chosen so that for all other neighbors M holds, that $lcp(id_N, id_{dest}) \geq lcp(id_M, id_{dest})$. In the rare event that no neighbor of the current node has a longer common prefix with the destination than the current node, a different metric is used instead. Namely, of all known peers, the one node is chosen as the next hop whose ID is numerically closest to the destination (i.e. not a prefix-based metric is used, but a metric based on numerical distance). Since most hops can be chosen using the prefix-based method however, diminishing the common prefix length with the target by at least one bit for each passed node, routing queries take at most $O(\log n)$ steps (n being the number of nodes in the overlay) until reaching their destinations. Still, in the worst case, where all routing tables in the overlay are empty and messages are routed only using the leaf set, requests are still guaranteed to succeed correctly.

An interesting effect of Bamboo's prefix-based routing in combination with PNS is that lookup paths tend to converge very quickly. Two lookup queries for the same address, performed by separate nodes that are topologically close in the underlying network, are likely to start following the same sequence of hops when getting closer to the destination. Specifically, this quality was attributed to Pastry in [12], where it is called *route convergence*. Since Bamboo and Pastry differ exclusively in their maintenance and node discovery mechanisms, it is safe to assume that the property

applies to Bamboo as well. Route convergence can be leveraged for implementing efficient caching schemes, like in PPR [41].

4.3.2 R/Kademlia

R/Kademlia [22] is a modified version of the Kademlia [34] protocol. In contrast to Pastry and Bamboo, Kademlia uses an XOR metric for determining the distance between two overlay keys (the distance between two keys id_A and id_B is calculated by solving $id_A \oplus id_B$). Kademlia does not construct an underlying ring topology, but relies only on the links in its routing table for discovering nodes. The routing table is organized in such a way that a hypercube routing geometry is spanned. It is based around a system of *buckets* for different XOR distances from a node's own key. Each bucket holds up to k nodes (with k usually being around 16), which means that, with high probability, multiple next hop candidates exist at each step. Kademlia leverages this by using *iterative* routing, where each hop reports its next hop candidates to the originator of the query who then chooses and contacts the next hops himself. By using iterative lookups, several paths can be expanded in parallel, which improves the overlay's resilience to churn and offers some protection against malicious nodes. However, iterative lookups are also slow and with bad locality properties, as every visited node has to report back to the originator of the query.

R/Kademlia aims at fixing those shortcomings by using a *recursive* type of routing that is similar to the one used in Pastry and Bamboo. Nodes forward received queries directly to the next hop instead of reporting back to the query originator. Also like in Pasty and Bamboo, it uses PNS for improving the locality of its queries. In addition to PNS, R/Kademlia implements *proximity routing* (PR) as well, a technique which incorporates latency information at the time of routing by preferring more local next hop candidates.

Explicit routing table maintenance in Kademlia and R/Kademlia is kept to a minimum, with the majority of nodes being discovered and refreshed while forwarding application-initiated messages and lookup queries. Different than in Pastry and Bamboo, this passive type of maintenance is typically sufficient, as Kademlia's routing table organization allows it to store redundant nodes which aid the recovery from node failures. The redundancy in Kademlia's and R/Kademlia's routing tables also enables instances of these protocols to know a larger part of the network, thus shortening lookup paths.

A deficit in Kademlia and R/Kademlia is their lack of an underlying ring topology that guarantees that nodes will know the nodes closest to them in ID space, i.e. their overlay siblings. Guaranteeing that nodes know their siblings greatly increases the chance that a query will succeed, as it ensures that there is always some, albeit suboptimal, path between two nodes in the overlay. In addition, the knowledge of a particular number of siblings is important for running a DHT on top of the KBR overlay. The DHT component needs to know all s closest nodes, so that it knows where replicas should be located (here s is identical to the number of replicas the DHT is required to create). Furthermore, when a node is storing a replica, it must be able to tell which of its siblings is responsible for the original data set. If using a ring topology, *2*s* siblings need to actually be known for this assessment to be made.

Kademlia's original protocol converges to a set of sibling nodes. In some cases however, correct siblings might not be added to any of the buckets, leading to an incomplete sibling list. The authors in [34] suggest modifying the standard routing table (bucket splitting) logic in such cases. However, such modifications introduce an additional level of complexity and can potentially worsen the algorithms performance. Another approach is to extend Kademlia by an explicit additional *sibling table* as suggested in [8]. The sibling table is exempted from Kademlia's standard bucket maintenance logic and always holds the $\eta * s$ closest other nodes (in ID space) that a node is aware of. If the parameter η is chosen accordingly, the sibling table can also be used for determining the responsible node for a dataset of which the current node is storing a replica, as mentioned earlier. A value of 5 was shown to be sufficient for this task [8].

4.3.3 Hierarchical Approaches

So far the focus has been on classical KBR protocols with a flat identifier and routing space containing all nodes in the overlay. An alternative to this approach is to organize nodes into clusters and hierarchies of clusters, achieving a better mapping to the underlying network. A variety of such *hierarchical KBR* (HKBR) overlays has been suggested in the literature, several of which will be presented in this section[3]. The goal of this survey is to find ways of improving the locality properties of the KBR layer, and the presented approaches are chosen accordingly.

HKBR overlays can be divided into two general categories - *homogeneous* overlays and *superpeer* designs[4]. Figure 4.2 depicts two examples for both types, assuming an overlay hierarchy of two levels. As the names of the categories suggest, all nodes in the homogeneous class share the same set of responsibilities, while some nodes in the latter category differ from the rest and need to contribute more resources. Those *superpeers* usually act as gateways between hierarchy levels, e.g. being part of both a local overlay and an overlay connecting superpeers of different local clusters. Heterogeneous designs using superpeers are used in other types of overlay networks as well, and for different purposes than improving the locality of overlay traffic (e.g. [29, 51]). In the particular case of improving the overlays locality properties however, superpeer designs seem to perform worse than homogeneous approaches, mostly due to their higher maintenance overhead and bad load balancing [2]. For this reason, and because the existence of heterogeneity inevitably leads to a higher degree of complexity, superpeer approaches will be discarded from consideration from now on.

In homogeneous approaches, every peer is part of several overlays simultaneously with each overlay being formed at a different level of the cluster hierarchy. Overlays at higher levels are typically constructed by merging the overlays from lower level clusters, as proposed for example by Ganesan *et al.* in [21]. The authors propose a generic construction (*Canon*) that can be used for transforming normal, flat KBR algorithms into hierarchical ones. Here, hierarchical overlays created in this way

[3]The wording used in the literature is *hierarchical DHTs*. However, since the interest here is mainly on the KBR part of those designs, the reference will be to hierarchical KBR overlays instead.

[4]In some places in the literature, the naming *horizontal/vertical* is used instead of homogenous/superpeer.

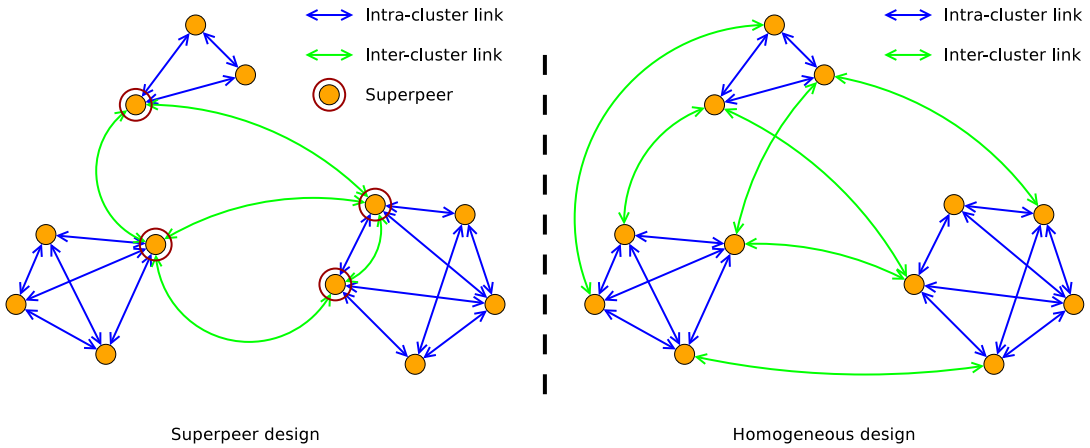


Figure 4.2: Types of hierarchical KBR overlays. Both depicted networks assume an overlay hierarchy of two levels. Note that, on the lower hierarchy level, each cluster of nodes forms a self-sufficient, isolated overlay. On the higher level, either an additional overlay is formed by distinct superpeers (in the superpeer approach), or a global overlay containing all nodes is constructed by mering the local cluster overlays (in homogeneous designs).

will also be referred to as *Canonical* HKBR overlays. In the Canon approach, the overlays at the lowest level of the hierarchy are formed using the standard mechanisms of the chosen KBR protocol, but using only nodes that should fall into the same cluster. When two clusters are merged, every node can create connections to nodes from the other cluster following the standard logic of the underlying KBR protocol, but obeying following additional rule: for every new neighbor M of a node N it must hold, that there are no neighbors in N's local overlay that are closer to N in ID space than M. In this way, Canonical HKBR protocols guarantee that (1) messages originating from the same cluster and targeting the same key always leave the cluster through the same exit node and (2) routes connecting nodes in the same cluster never exit that cluster. *HPastry* [19] is an application of the Canon paradigm to Pastry that combines the hierarchical approach with Pastry's locality optimizations. Additionally, HPastry also supports a more flexible interconnection of clusters that is not strictly hierarchical, in an attempt to better map to the underlying network. In these ways, HPastry represents a significant improvement to the protocols outlined in [21]. Eventually however, it could be enhanced even further by substituting its Pastry component with a more churn-resistant KBR protocol like Bamboo or R/Kademlia.

Cyclone [1] is another interesting approach to transforming regular KBR overlays into hierarchical ones. Like in Canon-derived approaches, a hierarchy of clusters is constructed, with higher level clusters being formed by merging lower level ones. The main difference in Cyclone is the use of unique cluster IDs stored as a suffix in all node IDs. The authors suggest, amongst other things, that this technique enables the use of simpler maintenance and routing mechanisms that the ones used in Canon. Cyclone furthermore supports the combination of different KBR protocols when merging clusters while Canon is confined to using the same KBR algorithm everywhere. Lastly, and perhaps most interestingly, the existence of cluster IDs enables the explicit addressing of a cluster, e.g. queries can be constructed that look for a given key only in a specific cluster. On the down side, the existence of unique

IDs for each cluster also raises scalability concerns and introduces the problem of agreeing on a cluster identifier.

An important open question in hierarchical KBR designs is the way in which clusters are formed. Clearly, proximity in the underlying network should be the main criteria for grouping nodes together. The authors in [50] propose a distributed binning scheme using well known landmark nodes for determining a node's cluster membership. However, the existence of fixed landmark nodes is problematic in the context of a fully decentralized system. In [20], nodes decide whether they are part of a cluster based on RTT measurements. With this method, network coordinate systems like Vivaldi [15] could also be used for speeding up the proximity assessment. The design of HPastry suggests that clusters should map to underlying Autonomous Systems, yielding one cluster per AS [19]. This approach provides a very clean mapping to the underlying network structure but introduces the practical problem of discovering a node's AS number. Closely related to cluster forming is the question of generating cluster identifiers in designs similar to Cyclone. The automatic selection of a cluster ID might involve voting procedures between nodes in each cluster, similar to the selection of connectivity domain identifiers in [35].

4.3.4 Social Routing

In the context of designing a user-centric networking overlay, the question to what extent social information can be integrated into KBR mechanisms becomes especially interesting. Several authors have suggested the use of social network information for protecting against *Sybil* attacks [17, 26]. In a Sybil attack, the attacker creates multiple fake overlay identities and uses them to disrupt or control routing procedures. However, since security is not a focus of this thesis, countermeasures against Sybil attacks will not be discussed further here. The authors of *SPROUT* [42] propose a related integration point for social network information, but one that has applications beyond security. Their suggestion is the preferred insertion of nodes, with whom social ties exist, into a node's routing table. When routing messages, friendly nodes should then be used for forwarding messages whenever possible, i.e. except in cases when no friendly nodes are known that are closer to the destination than the currently visited node. While the main focus of their work is again the improvement of the KBR's security (by routing over trusted nodes whenever possible), the social routing mechanisms they propose might be also interesting for other reasons. For one, routing over social links might be beneficial in applications where nodes in the same social neighborhood are likely to communicate with each other. Preferring friend nodes at every hop might furthermore lead to a fast convergence of routing paths targeting the same key, a beneficial property for caching and smart replica placement. However, strictly preferring socially connected nodes during routing might break locality-sensitive routing schemes like the ones used in Bamboo and R/Kademlia. Thus, a compromise between preferring local nodes and preferring socially connected nodes must be found in order to fully satisfy the requirements in this thesis. Even without SPROUT-like routing however, the prioritized inclusion of friend nodes into a node's routing table might still be beneficial. On one hand, it greatly increases the chance that befriended nodes will be overlay neighbors as well and that overlay messages between them will thus travel only one hop. One the other hand, with every node knowing about several other friend nodes, unstructured flooding techniques over friendship links could be realized, similar to the ones

used in UIA's Social Routing [18]. This could allow for more dynamic queries than are otherwise supported by a DHT, for example queries like "devices in my social neighborhood that can offer printing services right now".

4.3.5 Conclusion

Bamboo and R/Kademlia both show very good locality properties and behave well under churn. Both protocols are thus potential candidates for providing key-based routing functionality to the solutions in this thesis. Hierarchical KBR approaches can be used to further improve the overlay's mapping to the underlying network structure, but several unresolved questions and practical issues still exist with them. Lastly, social information can be integrated into existing overlay protocols to improve the performance of communication processes between socially connected nodes, to additionally enhance the security of the routing and to enable the construction of unstructured overlay structures formed by socially connected nodes.

4.4 Data Storage and Replication

This section focuses on exploring suitable storage and replication strategies for a KBR/DHT-based middleware for user-centric networking. Firstly, the nature of the data that solutions will likely have to work with will be discussed. Followed by that, actual strategies will be presented. Based on the stated requirements and expectations and on the discussed examples, an actual storage system concept will then be developed in Chapter 5.

4.4.1 Requirements and Expected Types of Data

In a DHT-based user-centric networking middleware like the one sketched in previous sections and chapters, specifically one that is based around an i3-style publish/subscribe architecture, data storage services will be needed for two general categories of data:

System Data Datasets required by SODESSON components, e.g. subscriber lists used by the publish/subscribe subsystem.

Application Data Data from SODESSON applications. The middleware is either explicitly instructed to store it or does so implicitly for realizing the delayed delivery of messages.

System data is expected to be small in size (e.g. less than 10 kilobyte per individual dataset). Frequent writes are likely, as well as frequent reads. Stored datasets should be available in the whole overlay and at all times, independent of churn and individual node failures. Application data will likely be more heterogeneous and vary across different applications. Table 4.1 shows several examples of application-generated datasets that might need to be stored, together with an estimation of their availability requirements and size. The size descriptors, "small" and "large", are not clearly defined but can be interpreted as below and above several megabytes per relevant collection of datasets, or below and above 10-20 kilobyte per individual dataset. "Target Group" denotes the set of entities that must be able to access

datasets, from the viewpoint of the user they originated from. The presented data types are chosen with several user-centric applications in mind, the realization of which might be interesting on top of the SODESSON middleware. Some of those applications however, like filesharing and the decentralized storage of personal files and backups, are not central to the goals of the SODESSON project and thus do not have to be considered very deeply.

Data Type	Availability Requirements	Size	Target Group
Persisted Instant Messages	high	small	individual users
Profile Pages	high	small	other users
Microblogging Posts	high	small	other users
Private Data (e.g. Contacts)	high	small	own devices
Persisted Multimedia Messages	high	large	individual users
Photo Albums	low	large	other users
Files for Filesharing	none	large	other users
Personal Files	high	large	own devices
Personal File Backups	low	large	own devices

Table 4.1: Possible application data types with requirements and properties.

A high degree of reliability is required for both system and application data, i.e. the storage system should not "forget" datasets or falsify their contents. Furthermore, it should be able to maintain these guarantees under conditions of churn and disturbances in the underlying network[5]. Lastly, while security is not in the focus of this work, solutions should not be inherently insecure and open to attackers, as the existence of malicious users can not be ruled out in realistic usage scenarios. Classical DHT mechanisms already fulfill those basic requirements. However, additions to the standard techniques, as well as alternative approaches, might offer a better performance in certain scenarios and will thus be considered as well in the following sections.

4.4.2 Example Strategies

Several example approaches for realizing and enhancing the data storage aspects of a DHT will be presented in the following sections. To ease the explanation of the different techniques, hash-table semantics will be assumed at all places. Furthermore, X_K will denote an arbitrary chunk of data that needs to be stored or retrieved and K its corresponding key, so that $get(K) \rightarrow X_K$. The node responsible for K according to standard DHT logic will be represented as N_K. For N_K holds, that there is no other node in the overlay whoose node ID is closer to K than that of N_K. N_K is also the node that is returned whenever a KBR lookup is made on K.

A number of additional criteria will now be defined for aiding the comparison of the presented approaches. These properties can be seen as an addition to the requirements outlined in Section 4.4.1. They are not directly related to the SODESSON

[5]Extreme disturbances leading to a partitioning of the overlay network will be discussed in greater detail in Section 4.5.

scenario, but aim at establishing a general frame for the discussion of storage and replication strategies. Following criteria can be defined[6]:

Accessibility Replicas resulting from the use of the technique should be useful and accessible to many potential requesters.

Put Locality It should be possible to store data without having to transmit it to far away-nodes. This is desirable for data that is mostly of local interest and changes frequently, as well as for larger pieces or data like photo albums and media files, which should be moved around the network as little as possible.

Get Locality Analogically to *Put Locality*, it is desirable that interesting datasets are available locally and can be obtained without needing to make long-distance connections.

Freshness Replicas should always hold the most up to date version of a resource. This is especially the case for important data that changes frequently, like subscriber lists in publish/subscribe applications.

Variability The choice of storage and replication locations should be variable to better adapt to different scenarios. Depending on the context, storage locations could be chosen based on social connections (and trust) or based on proximity characteristics, e.g. being well positioned in relation to all interested peers.

Independence Operations like the storage, editing and deletion of data should be independent from individual nodes. The existence of controller nodes can lead to asymmetric load distributions and a potential single point of failure. Additionally, the centralization of management tasks poses a security risk as it can place malicious nodes in positions where they can cause major disturbances.

Overhead The storage solution should not impose a significant additional communication overhead.

4.4.2.1 Sibling Replication

Sibling replication is the standard solution for realizing reliable storage in a DHT. During the initial put, X_K is sent to N_K who then stores it locally. N_K then creates s copies (replicas) of X_K and proactively pushes them towards its s siblings, i.e. the s nodes that are closest to N_K in ID space. In ring-based topologies like Pastry and Bamboo for example, those are the immediate ring neighbors of N_K. After the initial storage and replication of X_K, maintenance mechanisms ensure that the number of replicas is preserved independently of individual node failures and churn. Replicas are proactively pushed to new nodes entering the set of the $s + 1$ closest nodes to K. In that way, X_K is always replicated on that set of nodes and, for sufficiently high s, never gets "forgotten" by the system, as even if many nodes exit the overlay at the same time, some replicas are still likely to remain.

During data retrieval, more than one node can be queried, leading to a higher probability that X_K will be found and that a correct answer will be returned (the

[6]The list of criteria and the comparison table at the end of this section (Table 4.2) is inspired by the similar work in [5].

latter effect can be achieved by comparing all received answers). Additionally, the replica locations to be queried can be chosen based on proximity characteristics, so that get requests remain more local. This kind of server selection can happen implicitly when using locality optimized KBR-schemes, e.g. KBR algorithms using PR and PNS. An unavoidable drawback of standard sibling replication is that sibling nodes can be scattered around the whole network. Put requests and maintenance mechanisms thus both generate non-local traffic. Another drawback stems from the rigidity of standard DHT data distribution logic - assuming random node identifiers, there is no effective way of influencing the choice of storage locations for X_K[7].

4.4.2.2 Passive Caching

Caching can improve the standard DHT storage performance by placing additional copies of X_K at key locations. If a cached copy of a dataset is placed along the path of a query for that dataset, the querying node's lookup path can be terminated at the cache location without needing to continue to N_k or one of its siblings. If copies exist that are close in terms of underlay network proximity, the retrieval times for X_K can shortened as well, as the data transfer remains more local.

Passive caching is the simplest form of caching, where copies of a given dataset are opportunistically created whenever it passes by. For example, if put requests or the answers to get requests get routed through the overlay, the nodes along the return path may all form local copies of the requested data. The next time they are requested to forward a get request for the same key, they can simply return their cached copies.

Passive caching is easy to implement and requires no additional traffic, thus having zero communication overhead. One of its biggest drawbacks however is its bad freshness, as cached version of X_K do not get updated proactively when new versions of the data are putted into the DHT. Additionally, it is often more efficient to transmit data payloads directly between nodes, without routing them through the overlay. With such an approach, passive caching techniques are not realizable at all.

4.4.2.3 Proactive Caching

Proactive caching denotes the explicit distribution of additional copies of X_K across a set of nodes. The pushing of X_K to those additional locations is handled by N_K. The authors in [38] and [41] suggest replicating datasets according to popularity, so that more copies are made for popular resources. This does not only increase the chance that local copies will exist for datasets that a node is interested in, but also leads to better load balancing as the task of delivering popular pieces of data gets distributed across a larger set of peers. In [41], it is additionally suggested to place replicas at locations where paths to a key are expected to converge and from where more requests are thus expected to pass, possibly leveraging the path convergence property of protocols like Pastry and Bamboo.

Proactive caching is a good way to shorten lookup times and to increase the probability that local copies of data will exist for interested nodes. The distribution of those copies is problematic though, because of the extra traffic it generates (which will likely be non-local) and because of the concentration of responsibilities in one responsible node, thus loosing independence.

[7] Modifying K can alter only the choice of the main responsible node for X.

4.4.2.4 Social Storage

Here, *social storage* denotes the general concept of storing data on nodes based on social connections, e.g. storing files on nodes belonging to befriended users. Several projects have previously proposed decentralized storage systems along those lines, including F2N [27] and Friendstore [46] and the decentralized OSN PeerSoN [10]. F2N and Friendstore employ unstructured overlays and distribute a user's data on a set of nodes belonging to social contacts of that user. Data availability is not recognized to be of critical importance and it is accepted that certain datasets might become temporarily unavailable if all the nodes that are storing it become inactive. Information about data locations gets stored by interested nodes and can be disseminated by flooding. PeerSoN, on the other hand, uses a DHT for storing and disseminating information about storage locations. The DHT is not used for storing the actual data, put holds only pointers to its real locations.

Storing data on socially connected nodes is especially helpful in cases, where (1) large amounts of data need to be stored, as users will likely be ready to spare more resources for helping friends, and (2) sensitive data needs to be stored, as nodes belonging to social contacts are usually considered more trustworthy. In addition, social storage offers a high degree of flexibility when choosing storage locations, allowing the choice of nodes based on complementary criteria as well, e.g. their proximity in the underlying network to potentially interested nodes. On a downside, data availability might not be guaranteed with the social storage approach. Data availability and a higher level of resilience might be achieved by implementing additional replication mechanisms. However, no specific proposals for such replication mechanisms were found in the literature and their complexity and communication overhead remains largely uncertain.

4.4.2.5 Hierarchical DHT Storage

The use of hierarchical KBR algorithms like the ones described in Section 4.3.3 enables the deployment of hierarchical storage and replication schemes, hence leading to actual *hierarchical DHTs*. Based on previous discussions, the focus here will be only on homogeneous hierarchical overlays, i.e. approaches where all nodes share the same set of responsibilities and rights. Hierarchical DHTs allow the restriction of the scope in which a certain piece of data gets distributed across the network. A common approach is to store data originating from a local cluster only within that cluster, i.e. in a local DHT formed by nodes only of that cluster. In this way both put and get operations become local for nodes within this local overlay. This a highly satisfying setup for datasets that are only of local interest anyway. For datasets that might be of interest for non-local nodes as well, a pointer to the location of the data can be put into a higher level of the DHT hierarchy [21]. A reference like this can either be a list of node IDs of the data's replica holders or, in overlays that support it, the ID of the cluster that the data is saved in. Using pointers, datasets in local clusters can be accessed globally using a two-step lookup procedure, resolving first the location of the pointer and second the address it is pointing to. In cases where the data is predominantly of global interest it can also be stored normally on a global scope using standard flat DHT logic.

Apart from enabling sophisticated data placement methods, the use of hierarchical overlays also improves the performance of caching techniques, due to the high degree

Criteria	Sibling Replication	Path Caching	Proactive Caching	Social Storage	HDHT Storage[1]
Accessibility	all	on path	all	friends	all or local
Put Locality	no	no	no	possible	yes
Get Locality	likely[2]	likely[2]	likely[2]	possible	yes
Freshness	yes	no	yes	yes	yes
Variability	no	constrained	no	yes	local or not
Independence	yes	yes	no	possible	yes
Overhead	low	none	low	uncertain	low

[1] In a hierarchical DHT without supernodes (homogenous design).
[2] When using proximity routing or proximity neighbor selection (PR and PNS).

Table 4.2: Comparison of different data storage and replication techniques

of path convergence they induce. Namely, lookups by nodes from a given cluster targeting the same key K always leave the cluster through the same node when moving to a higher scope. This exit node is thus an excellent location for caching datasets associated with K or for forming local replicas of them.

4.4.3 Discussion

Classical sibling replication is a stable base for realizing reliable storage in a DHT. However, its mediocre locality properties and its lack of variability in choosing storage locations make its unmodified use problematic in the context of this thesis. This is especially true in the case of large datasets, the storage and replication of which would induce a high amount of non-local traffic and burden many randomly chosen overlay nodes. Path caching offers a slight improvement in terms of data locality and incurs no extra bandwidth costs, but is not suitable for datasets that change frequently. Proactive caching is better in such cases. Still, it is unclear whether the overhead in maintaining its extra replicas will be worth the increase in locality and get performance. Additionally, the technique is dependent on a single controller node per dataset to distribute copies, thus placing a higher load on it and making it a potential point of failure. Social storage enables the hand-picking of storage locations for each dataset. This approach is suitable for larger pieces of data, as nodes can be chosen specifically based on their willingness to contribute more resources. However, as is, the concept is not suitable for datasets that need to be available 100% of the time. Lastly, hierarchical DHT approaches form an excellent base for achieving a high degree of locality in both the distribution and the retrieval of data. They are, however, coupled with the use of a hierarchical KBR overlay, with all challenges that stem from that.

A listing of the storage and replication techniques discussed so far together with a short evaluation of each one under the criteria from the beginning of Section 4.4.2 can be found in Table 4.2.

4.5 Consistency on Partitioning and Merging

One of the design goals of the SODESSON middleware is its ability to deal with temporary losses in global connectivity, i.e. its capacity to remain functional during both the partitioning of the network and its merging back together. For the

user-centric networking overlay discussed here, those challenges surface on two levels: on the KBR layer, which needs to knit around holes resulting from connectivity losses but also enable the reunification of separated overlay segments, and on the DHT/Storage layer, where data consistency needs to be maintained and the existence of local copies of important datasets needs to be ensured. Both KBR and storage-related challenges will be addressed in the remainder of this section.

4.5.1 KBR Component

KBR algorithms like Bamboo and Kademlia are already capable of dealing with network partitioning, due to their adaptable structure and their resilience to churn and other, more minor, network disturbances. In cases where there are frequent temporary breaks in internet connectivity however, standard overlay maintenance mechanisms might be too slow to efficiently rebuild the overlay structure. Also, when merging separated overlays, the additional challenge arises of actually bootstrapping the merging process. As a mitigation for some of these problems, the authors in [23] propose, amongst other things, that nodes to not remove neighbors from their routing tables when the network gets partitioned (possibly marking them as unavailable but not removing them entirely). In this way, the overlay can be reconstructed faster when the networks are rejoined. Additionally, and complementary, mechanisms for the explicit detection of partitioning and merging events in the underlying network can be used.

4.5.2 DHT Component

Provided that the underlying KBR topology remains stable on partitioning and merging events, two challenges remain to be faced by the DHT component. Firstly, ensuring that important data is available locally when a partitioning event occurs and secondly, that datasets remain consistent when two subnetworks are merged, e.g. that conflicts from a simultaneous modification of a dataset in both partitions get resolved appropriately.

The first task is not easily solvable due to the nature of the uniform data distribution scheme used in classical DHTs. The use of hierarchical DHT approaches might be helpful here, as relevant datasets could be replicated in the local cluster and, in this way, preserved locally if the cluster becomes cut off from the rest of the network. In flat designs, a similar effect might be achieved by using an appropriate caching technique. Lastly, it can also be left to applications and components initiating the data storage to refresh and reinsert their datasets themselves when it seizes to be available locally. In the case of the publish/subscribe subsystem for example, subscribers might resubscribe to a topic if the subscription information has become lost due to a partitioning event. The effectiveness of this approach rests on the assumption that only datasets originating from currently reachable nodes are important in the case of a partitioned network.

For the challenge of maintaining data consistency in the DHT, it is helpful to first consider the different types of put operations that might be employed. In the basic case, a DHT entry holds only one value per key that gets overwritten during every put operation on that key. For many applications however (including, for example, publish/subscribe systems similar to i3), this semantic is too weak, as several

datasets might need to be stored under the same key. Thus, DHT implementation often support multiple datasets per key, usually by supplying an additional identifier for each dataset. With this setup, conflicts resulting from the simultaneous modification of a resource can be avoided by separating the resource into distinct independent datasets with only one responsible author for each. Under the assumption that no entity (including users) can reside in two separated network partitions at the same time, sets of data can thus remain consistent during partitioning events.

For applications where the separation of data into independent parts with only one exclusive author for each part is not possible, for example a collaborative document editing application, the detection of consistency violations can be moved into the application layer. Mechanisms like the use of hashes, sequence numbers and timestamps can additionally be employed to support the detection and resolution of conflicts. The use of vector clocks is possible as well. However, vectors of dynamic size might be needed (if the set of data generating entities is non-static).

4.5.3 Conclusion

KBR overlays are inherently capable of dealing with partitioning and merging issues. Small modifications, e.g. in the way routing tables are maintained, can additionally enhance this quality. DHTs are also able to recover from a temporary partitioning of the network, but the consistency of datasets across partitions can not be guaranteed for resources which can be modified by multiple entities. Applications should thus be designed to avoid the existence of multiple contributors per dataset. In cases where this is not possible, conflict resolution might need to be performed on the application level.

4.6 Summary

The use of structured overlays and DHTs still appears promising for achieving the design goals of the SODESSON middleware. Existing projects like the PeerSon OSN [10] prove that basic user-centric functionality can readily be achieved with this approach. Effective publish/subscribe-style communication is realizable as well and many proposals exist for improving the locality aspects of both the overlay routing layer and the storage subsystem. However, several open questions remain. For one, a suitable addressing scheme has yet to be developed, that provides a mapping between users and devices and enables the implementation of smart device selection schemes when communicating with a user. Additionally, a mechanism for realizing delay-tolerant publish/subscribe communication is still lacking for KBR/DHT-based approaches. Lastly, existing publish/subscribe systems based on structured overlays all introduce an additional level of indirection and thus lead to triangular routing. Methods for avoiding this inherent lack of locality are required.

5. Design

This chapter introduces a specific proposal for a communication and storage infrastructure for the user-centric networking middleware SODESSON. Based on the analysis performed in previous chapters, the approach developed here will be based on a distributed hash table running on top of a key-based routing overlay. This chapter features the abstract design of the proposal, with implementation specific details following in Chapter 6.

Firstly, the core elements of the proposal will be introduced, namely the overlay routing component and the publish/subscribe subsystem. Following that, the focus will move to the specifics of the storage layer. After a discussion of addressing, device selection and related issues, locality optimizations to the approach will be suggested. Lastly, ways of leveraging social information will be proposed.

5.1 Basic Design

This section deals with the foundation of the presented proposal, namely the choice of an overlay routing algorithm and the design of a publish/subscribe subsystem running on top of it.

5.1.1 Overlay Structure

Structured, key-based routing overlays have previously been identified as a suitable base for powering a decentralized user-centric networking middleware. Both flat and hierarchical approaches have been proposed (Section 4.3). Here, hierarchal overlays will again be discussed shortly, before jumping to the comparison of specific protocols and formulating the final proposal.

5.1.1.1 Hierarchical KBR Overlays

Hierarchical KBR overlays offer several interesting advantages over their flat counterparts. For one, they offer a better adaptation to the underlying network topology by forming clusters on top of subnetworks and Autonomous Systems. They aim at minimizing inter-cluster and maximizing intra-cluster communication, thus keeping

overlay traffic as local as possible. Their strong path convergence properties make it easy to cache interesting resources locally, or even to form local replicas of them. When using them as part of a DHT, they enable the replication of data to specific clusters only, thus limiting its spread to regions where it is not likely to be needed. This locality of storage is also likely to be helpful in cases of network partitioning.

However, HKBRs also introduce a set of new challenges. For one, they are clearly more complex and difficult to implement that standard KBR protocols. Also, several practical issues have still not been addressed sufficiently in the literature, like the specifics of the local peer detection and cluster forming mechanisms or the compatibility of hierarchical approaches with NAT setups. Additionally, no actual prototype systems could be found at the time of writing of this thesis, so the real life practicability of hierarchical KBR approaches remains uncertain. Lastly, with approaches like Canon [21] and Cyclone [1], existing flat KBR overlays can easily be transformed into hierarchical ones. It thus seems appropriate to first construct a system based on flat overlays, leveraging the lesser complexity of this approach and its higher probability of yielding good results. Once such a system is in place, hierarchical approaches can be tested on it as an improvement to the original design.

5.1.1.2 Overlay Protocols

Two flat KBR protocols were previously identified as suitable for the use with the presented approach - Bamboo [39] and R/Kademlia [22]. Since both overlays offer very similar properties, a definite analytic decision for one or the other is difficult. Bamboo's main advantage over R/Kademlia is its additional ring topology that offers a clearer and more stable view on a node's intermediate neighbors in ID space (i.e. its siblings). In R/Kademlia however, a similar effect like that of an additional ring structure can be achieved by introducing a dedicated sibling bucket (like suggested in [8]). R/Kademlia's main advantage is the higher degree of redundancy in its routing table, which leads to a higher interconnection in the overlay and thus to a higher resilience to failures and lower lookup times. Still, Bamboo's routing table could easily be modified to hold more entries as well.

Ultimatively, the choice of a KBR routing protocol to power the proposal in this thesis falls to R/Kademlia. Since no clear advantages of one protocol over the other could be identified, the reasons for this decision are mostly practical in nature. Namely, a good R/Kademlia implementation for OverSim was available at the time of writing, upon which a prototype of the presented solution could be based.

5.1.1.3 Summary

The proposed system is based on the flat key-based routing protocol R/Kademlia [22], enhanced by an additional sibling bucket as proposed in [8]. Hierarchical approaches were discarded due to their higher complexity and unresolved questions regarding their practical feasibility. A later migration of the presented solution to a hierarchical design is likely to be possible. Concerning other flat KBR approaches, Bamboo is recognized as a valid alternative to R/Kademlia. In the end, R/Kademlia was preferred for practical reasons, i.e. the existence of a better implementation of the algorithm upon which a prototype could be based.

5.1.2 Publish/Subscribe Subsystem

The publish/subscribe architecture for SODESSON proposed here is based on the design of the Internet Indirection Infrastructure (i3) [44]. Like in i3, responsible nodes are chosen for each publish/subscribe topic. It is the task of those nodes, which are commonly referred to as *rendezvous points* (RPs), to manage subscribers and disseminate publications. The RP for a topic is the node who, upon hashing the topic title to an overlay identifier, is responsible for that identifier. Any common hashing algorithm can be used for mapping the topic title to an overlay identifier. However, hash functions that are difficult to reverse and for which collisions are unlikely and hard to generate are more appropriate (i.e. cryptographic hash functions), as they offer some additional degree of privacy and protection against disturbances by malicious nodes. In this thesis, the SHA-1 algorithm is used for hashing topics into ID space.

Whenever a node S wants to subscribe to a topic $\mathcal{T}$, it first produces a hash of $\mathcal{T}$'s title, yielding the topic's overlay identifier $id_{\mathcal{T}}$. S then sends a *subscribe request* via the overlay, targeted at $id_{\mathcal{T}}$. The request gets forwarded according to standard KBR logic[1] until it reaches $responsible(id_{\mathcal{T}})$, the node whose overlay identifier is closer to $id_{\mathcal{T}}$ than that of any other node in the overlay. This responsible node for $id_{\mathcal{T}}$ is also the proper RP for $\mathcal{T}$, i.e. $rp(\mathcal{T}) \equiv responsible(id_{\mathcal{T}})$. Upon receiving S's request, the RP creates a *subscription* object and stores it in the DHT. A subscription object holds information about one subscriber to a topic, most importantly its overlay ID (at this level, subscribers are always single nodes, e.g. devices). The underlay addresses of subscribers are not stored in the DHT as they might change over time. The tracking of subscribers is thus strictly left to the KBR layer. The subscriptions for a topic $\mathcal{T}$ are stored in the DHT under the key $id_{\mathcal{T}}$, i.e. the topic's mapping to ID space. An additional, random identifier is added to subscriptions when they are put in the DHT, to allow several subscriptions to be stored under one DHT key. Once the DHT put has been completed successfully, $rp(\mathcal{T})$ sends a *subscribe response* back to S, informing it about the success of the subscription attempt. By storing subscriptions in the DHT, no state needs to be kept in the RP and subscription information is automatically persisted across RP changes (an RP change can be caused by a failure of the old RP, for example).

Like subscribe requests, publications to $\mathcal{T}$ are also firstly routed blindly through the overlay towards $id_{\mathcal{T}}$, the topic's mapping to ID space. Upon arrival, the *publish request* gets evaluated by the RP. At this step, the RP may impose some form of access control and determine whether the publishing node P is eligible to publish on this topic. If the RP decides to deliver the publication, it first retrieves the set of subscribers from the DHT. Since $rp(\mathcal{T}) \equiv responsible(id_{\mathcal{T}})$, the RP is also the main replica location for datasets stored in the DHT under the key $id_{\mathcal{T}}$ (which includes the set of subscription objects for $\mathcal{T}$). Depending on the setup of the DHT, the retrieval of the set of subscriptions might thus involve no additional communication with other nodes and, consequently, be very efficient. Once the set of subscribers has been received by the SODESSON component of $rp(\mathcal{T})$, the RP can proceed to the delivery of the publication. To each registered subscriber, a separate *notification* message is routed through the overlay containing the data published by P. Figure 5.1 visualizes this mechanism. Alternatively, the actual publication can also be

[1] Since R/Kademlia is being employed, recursive routing is used.

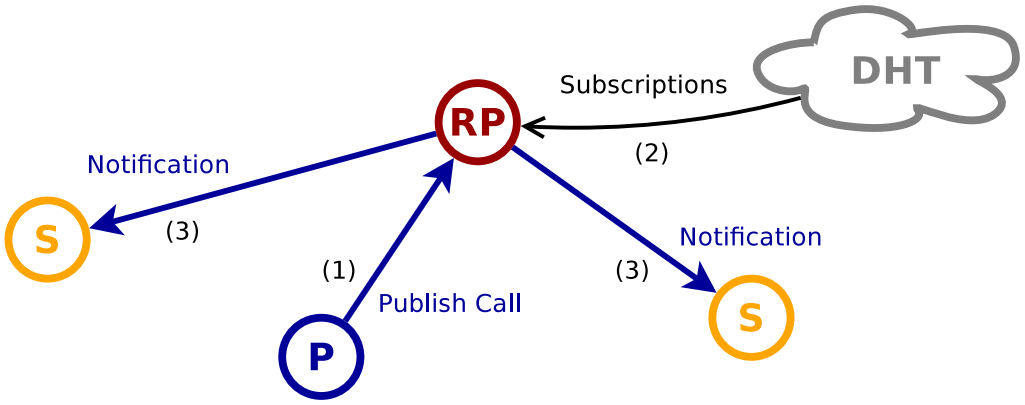


Figure 5.1: Basic Publish/Subscribe Mechanism.

omitted in those notifications and, instead, persisted. In such cases, subscribers that are interested in the actual data can make an additional *consume request* to the RP, upon which the data is transmitted. This is tightly related to the semantics of a publish/subscribe-based storage interface as proposed in Section 5.2.3.

5.2 Storage Subsystem

This section describes the decentralized storage component of the proposed solution, dealing first with its general design and, following that, discussing modifications for the handling of larger dataset. Additionally, the semantics of a publish/subscribe-based interface for the storage system will be elaborated.

5.2.1 General Approach

Based on the discussion in Section 4.4.1 it can be assumed, that the storage subsystem of the SODESSON middleware will mostly have to deal with datasets that are small in size, i.e. less than about 10-20 kilobytes per chunk. This includes datasets related to the correct functioning of the SODESSON middleware itself, for example data objects needed by the publish/subscribe infrastructure, as well as data that gets stored by SODESSON applications. Datasets might furthermore change very frequently, like, for example, the subscription information for a busy publish/subscribe topic.

For small chunks of data that get updated frequently, a standard DHT system with sibling replication is sufficient for providing reliable storage services. Caching can be used additionally, but its efficiency is questionable based on the preceding analysis. Unstructured storage techniques based on social connections (i.e. social storage), on the other hand, are not a good general solution for such types of data, as not all datasets that might need to be stored will stem from a social context (e.g. certain kinds of system data) and the low availability qualities of these storage approaches will likely be unacceptable in many cases. Also, the storage location selection procedures and the two-step lookup mechanism will likely induce a higher communication overhead than standard sibling replication and ultimately lead to a lower performance when dealing with smaller files. A hierarchical DHT approach might form a reasonable improvement to the classic sibling replication scheme when

dealing with small files, by allowing the local storage and replication of locally interesting datasets. However, hierarchical DHTs require the use of a hierarchical KBR overlay, which was ruled out previously.

Thus, classical DHT storage semantics and a standard sibling replication scheme will be used for dealing with common small files in the proposed system. This setup will form the base for the rest of the storage subsystem.

5.2.2 Storage of Large Datasets

While small data chunks that change frequently might form the majority of datasets that the developed storage system will have to deal with, larger pieces of data will likely also need to be stored. Examples for such datasets include photo albums and multimedia messages persisted for later delivery. Storing larger pieces of data (i.e. in the magnitude of several megabytes and more) might be problematic with standard DHT storage and replication schemes, as the distribution and maintenance of large chunks of data becomes difficult to manage and might cause a significant communication and storage overhead in concerned nodes. One solution is to separate larger datasets into several small chunks and place them into the DHT. However, this would still result in the same amount of data being stored in the overlay, so that the overall bandwidth and storage space consumption would not change. Most importantly, the burden for handling those elevated amounts of data would still fall to random nodes that have no significant motivation in providing an elevated amount of resources. Socially connected nodes, on the other hand, might be more willing to provide resources to each other, especially if they themselves have an interest in the data that is being stored (as, for example, might be the case with shared photo albums). It thus seems appropriate that an additional social storage system, inspired, for example, by the work in [10] and [46], be developed for the better handling of large datasets. However, the full specification of such a storage system would exceed the scope of this thesis and thus remains a subject for future work. The focus here will remain on the more common case of dealing with small datasets for which the standard DHT approach is sufficient.

5.2.3 Storage Interface

In the attempt to create a unified interface for the SODESSON middleware, the desire arises to use the publish/subscribe semantic for accessing persistent storage mechanisms as well. While this is typically handled by a put/get type of interface, the publish/subscribe pattern can also be used in this context. In the publish/subscribe system presented earlier, published data can be stored persistently by the RP, thus mimicking a put call. In addition to just putting the data though, it is also pushed to interested nodes (the subscribers) immediately, or notifications about the change are sent. Likewise, subscribe-requests can both return the current data set and register the originating node for updates. In the common case where a node is always interested in the most up-to-date version of a resource, updates by the RP reduce delays and the unnecessary overhead resulting from the continuous polling for changes. Control information about whether a publication should be persisted or not and about which parts of a publications should be delivered can be passed to RPs by adding appropriate flags in publish, subscribe and consume requests.

A publish/subscribe approach for handling persistent storage results in a cleaner and more unified programming interface. Additionally, the continuous polling for changes in a resource (busy wait) is being avoided. Thus, the described publish/subscribe interface is proposed as the standard interface for SODESSON applications accessing the storage system. Internal SODESSON components should be able to use it as well, if appropriate.

5.2.4 Summary

The presented storage solution is based on a classical DHT coupled with a standard sibling replication scheme. For small datasets that change frequently, which is expected to be the major type of data that the solution will have to work with, these techniques perform sufficiently well. For larger datasets, a complementary social storage approach is possible that distributes data on socially connected nodes. Lastly, a persistent storage interface based on the publish/subscribe communication pattern was introduced, as a way of creating a more unified programming interface for SODESSON and avoiding the continuous polling for resources in anticipation of changes.

5.3 User Addressing and Device Selection

In a user-centric networking middleware like the being developed here, two types of actors require a designated addressing scheme - the participating devices and the actual users. Additionally, a mapping between those groups must be implemented together with respective lookup mechanisms for resolving user IDs to the set of their devices and device IDs to addresses in the underlying network. An additional challenge in SODESSON is the problem of device selection - choosing with which one of a user's devices to communicate with in a given situation.

The presented challenges will be addressed in the remainder of this section.

5.3.1 Identifier Generation

Addressing schemes for both users and devices can be realized easily, as demonstrated for example in [18] (for devices) and [10, 14] (for users). Here, we will simply use the overlay ID of each device as its device identifier as well. This way, standard KBR mechanisms can be used for looking up a device's underlay address. Overlay IDs will be generated randomly following a uniform distribution. The uniform distribution of node IDs improves the load balancing in the overlay as every node becomes responsible for a roughly equal portion of the key space. Additionally, the randomness of node IDs implies an ID/locator split, i.e. the ID of a device becomes independent of its position in the underlying network and can be retained when it moves.

In the presented approach, user IDs will be generated randomly as well. For security purposes, they might additionally be tied to some form of cryptographic material, i.e. an asymmetric cryptographic key pair. This cryptographic material might even be used for generating user and device IDs, yielding *cryptographically generated addresses* (CGAs). However, since security and privacy are not in the central focus of this thesis, these approaches will not be elaborated further and the random generation of both sets of identifiers will be assumed.

5.3.2 Mapping Between Identifier Groups

At this level, the novel challenge surfaces of providing a mapping between user and device IDs in the specific case where several active devices may exist per user. The challenge can be divided into two subproblems - allowing devices to know which user they belong to and allowing entities that know a user's user ID to discover all devices that belong to that user. For the first half of the problem, a simple reference to the parent user can be stored on devices when setting them up as SODESSON nodes, together with some form of cryptographic material for authentication purposes (e.g. a certificate signed by the user). This would allow a device to both communicate its owner and, eventually, identify its siblings, i.e. devices by the same user. The second half of the problem can be solved by storing the list of each user's devices at a known point in the DHT and allowing devices to append their addresses to those lists. This is similar to creating a publish/subscribe topic for each user and having all of his devices subscribe to it when they become available. The resolution of user addresses to device IDs then condenses to making DHT get request using the user IDs as keys.

5.3.3 Device Selection

Having introduced identifiers for both users and devices and a mapping between those sets of addresses, the problem of device selection remains. Device selection can be seen as a form of anycast to a user address, where the best possible receiver (from the user's devices) is chosen based on the context of the communication. This choice is thus highly application and situation specific. In the instant messaging use case for example, the most appropriate device would be the one to which the user is paying attention to right now. For the photo album application, it would be the device where the sought photos are most likely to be stored. The devices themselves, together with the applications running on them, have the best view on the optimality of a candidate in a given situation. They should thus be able to communicate this information to the SODESSON middleware. Most communication processes in SODESSON will be based on a publish/subscribe interface. This introduces two options for restricting the circle of possible receivers and facilitating device selection. Firstly, topics could be organized as such that all subscribed devices are sensible targets for a request. For example, only devices with a running instant messaging program would subscribe to the instant messaging topic. Secondly, information about the status of a device can be communicated proactively to the RP. Devices can, for example, resubscribe to a topic periodically, communicating their activity level.

5.3.4 Summary

In the presented solution, both user IDs and device IDs are generated randomly following a uniform probability distribution. Additionally, a device's overlay address is set so that it is equivalent to the device's device ID. The mapping between user and device IDs is facilitated by storing user addresses on the devices they own and by storing device IDs in the DHT, using their owners' user IDs as keys. The DHT can then be used as a lookup tool for resolving user IDs to device addresses. Lastly, device selection was discussed, with the result that SODESSON applications must communicate their status and their preferences to RPs, in order to facilitate the correct delivery of messages.

5.4 Delay-Tolerant Publish/Subscribe

A publish/subscribe-based storage interface as suggested in Section 5.2.3 could also be used for implementing hybrid, i.e. delay-tolerant communication services as requested in Section 3.1.2. However, several additional challenges arise when persisting messages only temporarily for later delivery. For one, the RP must be able to find out which nodes are supposed to receive a given message it has persisted, as nodes subscribing after the initial publish might not be meant to receive it. Then, it must be ensured that all proper recipients receive the message and its transmission is triggered accordingly when a node becomes available. When the message has been delivered to everybody or a timeout is reached, it should be deleted, i.e. removed from persistent storage. Lastly, the delivery of delayed messages should be resilient to churn and independent of the availability of one particular node, e.g. one particular RP.

The proposed solution to these challenges involves the concept of *pending acknowledgements* (pending ACKs or *PACKs*). In many communication systems, ACK messages are sent by the receivers of a message to communicate to the sender that it was delivered successfully. Since in an i3-based publish/subscribe system like the one that was previously suggested, the RP is responsible for disseminating messages to the final subscribers, the RP is also proposed here as the collector of acknowledgements. With this setup, PACK objects are employed as follows: for every publication p_i to a topic $\mathcal{T}$ that an RP receives, and every subscriber S to $\mathcal{T}$, the RP generates one unique PACK object $pack(T, S, p_i)$ and stores it in the DHT. Every PACK object represents the expectation that one specific ACK message will be received. Thus, for each ACK an RP receives, it deletes the corresponding PACK from the DHT. A schematic describing this mechanism can be found in Figure 5.2.

Every RP keeps track of its subscribers, marking their state in the DHT. If a node does not acknowledge publications that were sent to it, or if other nodes report it to be down, the RP marks it as inactive. Amongst other things, publications are not forwarded to nodes that have been marked as inactive, until they resubscribe or otherwise signal their availability. When a node is active however, and pending ACKs for it exist in the DHT, the RP periodically attempts to retransmit the corresponding publications. Likewise, whenever a node resubscribes to a topic after a period of inactivity (e.g. after it has been offline), the responsible RP uses the information from the stored PACKs to immediately trigger the retransmission of all publications that the node has missed. It is in this specific way that the delayed transmission of messages is effectively realized.

Since PACKs are generated at the moment of publication, they also implicitly encapsulate information about the proper receivers of a publication. If the set of subscribers changes for example, the existing PACKs are still retained, so that (1) all the proper receivers get the publication and (2) it doesn't get distributed to any nodes that were not subscribed to the topic at the time of publication. PACKs are deleted whenever their corresponding ACK arrives. In this way, no duplicate retransmissions can be triggered based on this PACK and the message will be delivered only once. Additionally, the existence of PACKs for a publication is an indicator that the publication must still be delivered to some nodes and its storage is thus still necessary. Analogically, once the last PACK for a topic was deleted and the persistence of the publications was not explicitly requested (e.g. by setting an

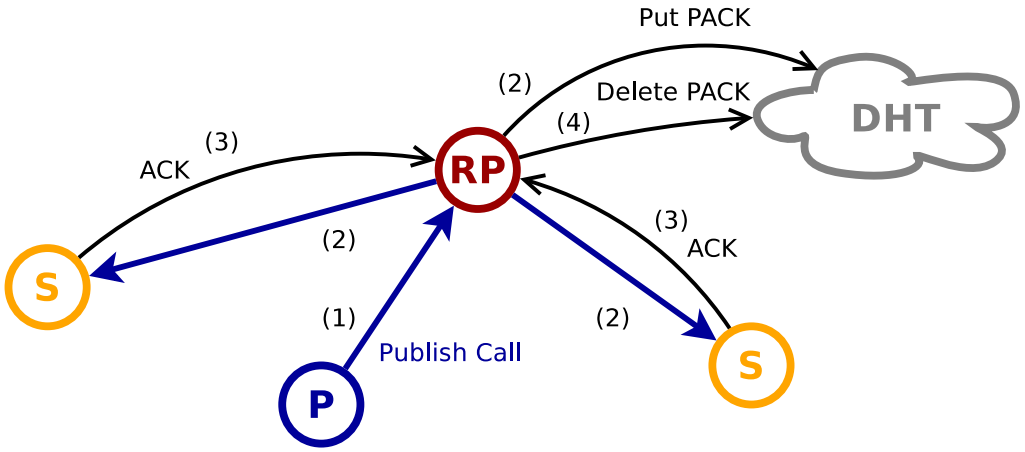


Figure 5.2: Pending ACKs Mechanism.

appropriate flag in the publish request), it can be safely deleted. Lastly, since PACK objects are stored in the DHT, the described mechanism does not depend on the availability of one particular RP node. Once the responsibility for a topic changes (e.g. when the old RP leaves the network), the new RP simply requests the relevant set of PACKs from the DHT again, so that no information is lost.

5.5 Direct Publish

This section describes an optimization of the publish/subscribe infrastructure introduced in Section 5.1.2. The aim of the presented technique is to improve the locality of the communication system, by eliminating the additional routing hop over the RP.

Firstly, the reasons for developing this enhancement will be elaborated. Following that, the actual proposed mechanisms will be described. Lastly, the effects of the presented modifications will be discussed.

5.5.1 Motivation

The publish/subscribe system outlined in Section 5.1.2 provides a reliable, decentralized infrastructure for both direct and, with the introduction of PACKs, delay-tolerant communication. In the context of this thesis however, it has one major drawback: the existence of triangular routing in the way publications are distributed. Everytime a node wants to publish a set of data to a topic $\mathcal{T}$, it has to pass it to $rp(\mathcal{T})$ first. Only then does it get forwarded to the subscribers. Thus, publications always make a detour over the RP, introducing an additional routing hop and higher communication latencies. Additionally, since the distribution of node identifiers is uniform and unconnected to the topology of the underlying network, the additional routing hop $rp(\mathcal{T})$ will be some random device in the network, most likely far away in terms of underlying network proximity from both the publishers and the subscribers to $\mathcal{T}$[2]. Even worse, it is expected that in the SODESSON usage scenario publishers and subscribers will be relatively close to each other in the social graph, which was shown to correlate with geographical proximity [47]. Geographical proximity,

[2]This might be avoidable if building upon a hierarchical KBR overlay.

in turn, is likely to imply some degree of proximity in the Internet graph as well. Thus, in the context of this thesis, publishers and subscribers are expected to often be close to each other in the underlying network. In this way, the communication detour over a far away rendezvous point introduces an even higher increase in latency in comparison to direct communication as well as an non-neglectable amount of avoidable non-local traffic (especially when large amounts of data are transferred, e.g. in file transfer applications). For these reasons, the need is seen for a more direct method of publishing data. Consequently, a *direct publish* mechanism will be proposed here.

5.5.2 Mechanism

The main design goal of the presented approach is the elimination of the additional routing step over the RP, thus enabling the direct data exchange between publishers and subscribers. In order for publishers to be able to transmit their notifications directly to subscribers, the set of subscribers for the target topic $\mathcal{T}$ must be known to them. Thus, as the first step of the presented algorithm, the RP of $\mathcal{T}$ informs potential publishers about the subscribers to $\mathcal{T}$. For simplicity, it will be assumed from now on that all potential publishers to a topic are also subscribers to that same topic. If needed, all presented mechanisms can be adjusted for relaxing this constraint, although at the cost of a slightly more complex design. All subscribers to $\mathcal{T}$ being potential publishers as well, $rp(\mathcal{T})$ informs them about the other subscribers to $\mathcal{T}$ using dedicated *subscription notifications*. These notification messages are sent whenever a node subscribes to a topic. They are sent to the newly subscribing node, carrying information about all nodes that are currently subscribed, and they are sent to all of the currently subscribed nodes, notifying them about the new subscriber. Likewise, subscribers get informed whenever a node unsubscribes from a topic or becomes unavailable. In this way, the subscriber lists of all subscribers to $\mathcal{T}$ remain up to date.

Whenever a node P wants to publish to the topic $\mathcal{T}$ it does the following: (1) it sends notification messages carrying its publication to all subscribers it knows and (2) it sends a standard publish call to the RP that also includes information about the subscribers that the message was already sent to. The additional message to the RP is necessary for a variety of reasons. Firstly, it can never be guaranteed that P's information on the subscriber set is up to date. P might, for example, have missed a subscription notification because of routing failures, or an additional subscriber might have joined the overlay shortly before the publication was sent. Secondly, in the case of unavailable subscribers, the publications must be persisted for later recovery. Additionally, PACKs must be generated. In the current design, all those tasks are carried out by $rp(\mathcal{T})$, so it is unavoidable that it gets informed about the publication as well.

Whenever $rp(\mathcal{T})$ receives P's publish call, it first checks whether P was aware of all currently active subscribers to $\mathcal{T}$. The RP delivers the publication to each subscriber that was missed by P. Additionally, $rp(\mathcal{T})$ also sends a subscription notification to P, informing the publisher about its error and fixing its local view on the subscribers to $\mathcal{T}$. The information about which nodes a publication was delivered to can be provided as a list of node IDs or as a hash of all of them. Hashing the node IDs together makes the publish calls smaller and thus saves bandwidth. However, it also

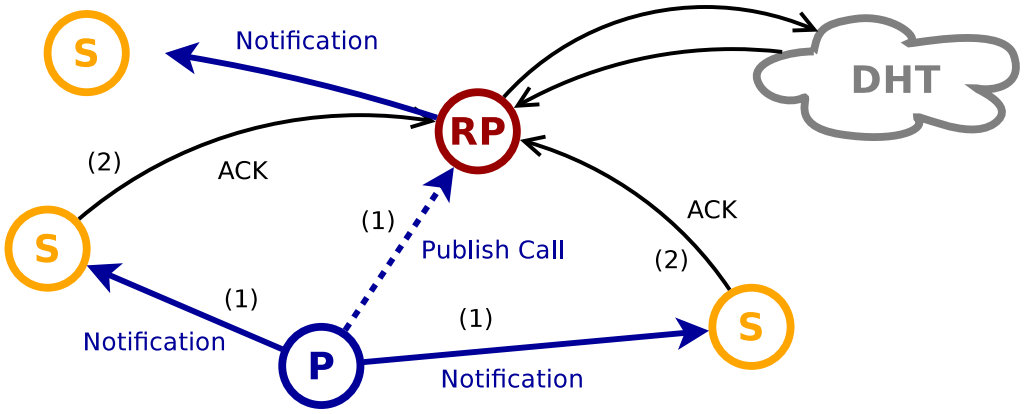


Figure 5.3: Direct Publish Mechanism.

makes the design more complex. A simple hashing method is to sequentially apply XOR on all subscriber IDs, distilling them all to one value. In order to verify P's view of $\mathcal{T}$'s subscriber set, $rp(\mathcal{T})$ needs to calculate the same hash from its own knowledge about the subscribers and compare the result. As an additional feature of the XOR method, if only one subscriber is missing or wrong in P's view (which is a likely case), the ID of that subscriber can be discovered by applying XOR on both versions of the subscriber hash. This mechanism can be made arbitrary more complex to allow the discovery of two, three and more missed nodes, using the two subscriber hashes and the RP's knowledge about recent changes to the subscriber set. If no fit can be found however, the publication must be retransmitted to all subscribers.

In addition to checking whether P was sent to all of $\mathcal{T}$'s subscribers, the RP also generates PACKs for the publication in the same way as it would in the case of a non-direct publish. Thus, as an important detail, the subscribers that receive P's publication still send their ACK messages to $rp(\mathcal{T})$, despite the fact that they might have received it directly from P. Figure 5.3 visualizes this detail as well as the rest of the direct publish mechanism.

5.5.3 Discussion

The presented improvement to classical i3-style publish/subscribe communication greatly improves the latency between the publication and the arrival of messages, by omitting the detour over the RP in most cases, transmitting publication directly from source to destination. Direct publish can also be used as a foundation for improving the privacy characteristics of the system, as the amount of information that is sent to the RP can be restricted at will (at the expense of reliability and functionality). Lastly, direct publish is also expected to increase the overall resilience of the system, by shifting responsibility away from centralized RP nodes.

In future designs, additional mechanisms could be developed that shift even more responsibilities away from individual, randomly chosen RPs. Instead of persisting a given message or dataset himself, for example, the RP could keep a list of subscribers that have received it and are storing copies. Interested nodes can then be redirected to one of those subscribers and request the actual data from them. This setup is especially interesting for large datasets (like photo albums) and messages with

private contents. Higher levels of indirection are also possible. For example, the responsibilities of the official RP for a topic (that is chosen based on its node ID) can be reduced further, so that it only holds a reference to a node performing the actual RP functionality. In this way, RPs can be chosen freely, based on criteria like their location in the underlying network or the amount of social trust between them and the nodes interested in the topic. This technique would harmonize well with the use of hierarchical KBR overlays - the RP for a topic of mainly local interest could be chosen from the node population of the local cluster (keeping the data traffic local), while still making it possible for external nodes to subscribe and publish. In combination with the direct publish improvement, approaches like this could also be developed to realize the social storage idea outlined in Section 5.2.2. However, for choosing RP and storage locations based on social trust, mechanisms for electing an RP from a set of publishers and subscribers might additionally need to be devised.

5.6 Social Routing

So far, methods for achieving the core functionality of the SODESSON middleware have been discussed, together with improvements for a better locality of communication. The presented solution, however, is still lacking a social dimension as no specific integration points with the social graph have yet been proposed. This section rectifies this oversight. A *social routing* mechanism will be introduced that leverages social information for the improvement of overlay routing.

Firstly, two assumptions will be stated that will form the frame for integrating social information into the KBR component. Following that, an additional data structure for R/Kademlia will be introduced, that holds information about socially connected nodes. Ways of filling that data structure will then be elaborated. Lastly, uses for the gathered social routing data will be discussed, including actual social routing and the usage of social links for distributing queries independently of standard KBR logic.

5.6.1 Assumptions

For simplicity, following assumptions will be made for the social network between users and its basic connection points with the SODESSON middleware:

1. There is only one class of social connections, namely ”friendship”. All friendship links are bidirectional.

2. Each user is aware of all of his friends and their user identifiers. This information is present in the SODESSON middleware (e.g. in the storage system) and all of the user's devices can access it freely.

The constraint expressed in the first assumption can be relaxed at will without rendering the presented techniques unfeasible, but at the cost of simplicity. The realization of the second assumption is trivial and implementation specific. It will not be elaborated further here.

5.6.2 Social Table

Having established the abovementioned preconditions, an additional data structure is introduced here - a dedicated *social table* (or also: *social bucket*). The social table of a node holds the overlay IDs of all devices that belong to friends of that node's owner. For example, if the users Bob and Alice are friends and both own one SODESSON-enabled device, then Alice's device would be included in the social table on Bob's device and vica versa. In R/Kademlia's bucket maintenance logic, the social table has a lower priority than the sibling table but a higher priority than ordinary buckets. Thus, in the previous example, if Alice's and Bob's devices were siblings in the overlay, they would be added in each others sibling tables instead of their social tables. If else, or if they stop being candidates for each others' sibling tables, they are always added to their respective social buckets as long as Alice and Bob remain friends. Only if no other condition applies, i.e. if the two devices are neither siblings nor belonging to befriended users, are their node IDs added to ordinary buckets. When routing however, the entries of all three data structures - sibling table, social table and ordinary buckets - are all treated equally in the current design.

5.6.3 Friend Node Discovery

There are two basic approaches for filling the social table:

Passive Whenever a node receives a publication directly from the publisher (when using direct publish), it checks whether the publishing user is a friend of the receiving node's owner. If so, the publishing node is added to the social table. Additionally, a *friend device notification* is sent to the publishing node. Since friendship links are assumed to be bidirectional, the publishing node can then add the subscriber to its own social table as well.

Active In this approach, the identifiers of all devices belonging to a user are stored in one location in the DHT, namely under the user's user ID (see Section 5.3.2). Thus, whenever a device joins the overlay it first adds its own device ID to this collection. Additionally, it retrieves the node IDs of the devices of both his owner and that of his owner's friends. Upon receiving the identifiers, it sends friend device notification messages to all those devices, informing them about its own presence. Upon receiving a response from a contacted friend device, it can be added to the social table. Device entries are persisted only for a limited amount of time in order to better adapt to churn. Thus, available devices refresh their own entries periodically by reinserting their own device IDs into the DHT.

The first method is only applicable in combination with the direct publish mechanism described earlier. Additionally, it does not guarantee that all befriended nodes will be discovered. More specifically, only befriended nodes with whom communication occurs will be added to the social table. In the publish/subscribe scenario however, this would already introduce an improvement, as it increases the probability that publishers and subscribers to a topic will be present in each others routing table (under the assumption that publishers and subscribers to a topic are often socially connected). This would lead to one-hop distances between publishers and

subscribes when routing over the overlay which, depending on the implementation, might greatly improve latencies when using the direct publish mechanism. Finally, this approach distinguishes itself with its next to nonexistent communication overhead. The active method of discovering friend nodes, on the other hand, is expected to induce a relatively high bandwidth consumption. Specifically, all node IDs need to be stored in the DHT (with all maintenance efforts involved) and accessed and updated frequently. As a result however, it is expected that the greater majority of node's friend nodes will become known shortly after joining the overlay. In addition to the already mentioned improvements to publish/subscribe performance, this also enables nodes to form more links in the overlay quicker, leading to a better overall interconnection and routing performance. Lastly, both presented methods can be combined, possibly leading to an even higher internal coverage of a node's friend nodes set.

5.6.4 Uses for the Social Table

So far, only trivial uses for the additional social table were stated, namely for improving the overall interconnection in the overlay and increasing the chance that message exchanges with friend nodes (which are likely to occur) will be performed in one hop. However, those applications of the social bucket only scratch the surface of what can be done with the additional social intelligence. Following additional uses for the social table appear promising:

Actual Social Routing Until now, the entries in the social table were not treated differently than the entries of regular buckets when making routing decisions. However, friend nodes have a few characteristic differences to nodes with whom no social connection exist. For one, they are likely to be more trustworthy than averages nodes, thus making them a better medium for forwarding sensitive data. In addition, in networks where malicious nodes might exist, the preferred routing over friendship links might increase the overall stability of the routing. A routing approach along those lines was already proposed in SPROUT [42]. In the scenario of this thesis however, caution must be taken, as the preference of social nodes when routing could break the routing layer's locality characteristics and conflict with strategies like PR, that prefer routing hops based on their proximity in the underlying network.

Unstructured Flooding over Social Links As demonstrated in [18], unstructured overlay techniques like flooding loose some of their disadvantages when restricting the set of queried nodes to only such nodes with whom social connections exist. Following this line of thought, the proposal here is to use the entries from the social table for the socially-restricted flooding of dynamic service discovery queries. For example, if a nodes is interested in a given service and is looking for a node from its social neighborhood that is offering it, it can simply flood its request over the entries from its social table and wait for replies. Nodes receiving the query can pass it on to their friend nodes, using standard flooding techniques to restrict the spread of the request and to avoid loops. The socially restricted flooding technique could be additionally enhanced for considering locality characteristics as well, i.e. flooding only over socially connected nodes that are also nearby in terms of underlying network proximity.

Unfortunately, the more in-depth discussion of those approaches would exceed the scope of this thesis. Their further development is left as a subject for future work.

5.7 Summary

The presented design is based on the KBR overlay R/Kademlia in combination with a publish/subscribe system strongly resembling i3. Standard DHT mechanics are used for realizing the persistent storage of most types of data. Additionally, a storage interface based on the publish/subscribe pattern is proposed. For the addressing of users and devices, the generation of random IDs is suggested and ways for mapping between the identifier groups have been elaborated. Concerning the delay-tolerant delivery of messages, a system based on pending acknowledgement (PACK) objects is proposed. For each publication and each peer that needs to receive that publication, one PACK is generated, that can be used for triggering delivery attempts and gets persisted in the DHT until a corresponding ACK is received. In addition to this core functionality, an approach for improving the locality of the publish/subscribe subsystem is suggested. Namely, the direct passing of publications between publishers and subscribers without taking a detour over the RP. Lastly, an additional social table is introduced for R/Kademlia that holds devices belonging to befriended users and thus forms an integration point for information from the social graph.

Apart from those specific propositions, several other suggestions were made that could not be developed in greater detail. These include the usage of an additional, unstructured storage system for dealing with larger datasets and sensitive data, the employment of cryptographic methods for generating user and device addresses and the usage of information from the social table for influencing routing decisions and enabling the unstructured flooding of queries in the social neighborhood.

6. Implementation

In addition to the analytical work in this thesis, a prototype of the proposed system was implemented. The goal was to enable the evaluation of the suggested designs in a simulated environment and to verify the general feasibility of the approach. Additionally, the development of reusable components for the use in future research efforts was desirable.

The solution was implemented using the overlay simulation framework OverSim [7] introduced in Section 2.4. The architecture of the framework will be described firstly, focusing on the integration points of the developed modules and giving an overview of the relevant folders in OverSim's directory structure. Following that, new components will be introduced that facilitate the evaluation of user-centric applications. Additionally, the design and implementation of a test application for the SODESSON middleware will be outlined. Then, the actual SODESSON prototype will be introduced, focusing on its module structure and elaborating a set of implementation-specific particularities. Details about the new social table in Kademlia will be discussed next. Lastly, an additional number of smaller changes and additions to the OverSim codebase will be listed.

During the implementation of the prototype, several small bugs and inconsistencies were discovered in existing OverSim components. They will not be discussed further here, but patches containing fixes have been made available on the CD accompanying this thesis.

6.1 OverSim Simulation Framework

As mentioned earlier, one of the main advantages of the overlay simulation framework OverSim [7] lies in its modular, layered architecture. Existing components can be reused and combined to a great extent, a fact that was leveraged extensively while implementing the discussed prototype. For example, both an R/Kademlia and a DHT component were available for the use in this work and were leveraged to provide core functionality to the developed system. Figure 6.1 depicts OverSim's layered architecture and a selection of key modules. SODESSON-related components are inserted in this diagram as well, to give a sense of perspective about how

they integrate with the rest of the framework. The global observer module dealing with users is new as well and will be described in subsequent sections.

The architecture of OverSim can be divided into three main layers that reside on top of one another. Starting from the bottom, the *underlay layer* simulates real IP networks upon which overlay designs can be spanned. Several underlay network models exist for OverSim that offer a variable degree of realism. For this thesis, the *SimpleUnderlay* module is used. It offers a rudimentary approximation of a IP routing topology that is sufficient for the goals of this thesis and allows the simulation of large networks, due to its low impact on simulation performance. OverSim's underlay layer is also responsible for spawning and removing network nodes according to predefined patters. This work is done by dedicated *churn generators*. For this thesis, *LifetimeChurn* generators are used in all simulations, that spawn and remove nodes based on the time they spent in and out of the network.

Moving higher, the *overlay layer* sits on top of the underlay and implements the actual decentralized networking behavior. KBR protocols like Kademlia and Bamboo reside in this layer. For the presented implementation, the Kademlia module is used, with parameters set so that it behaves like R/Kademlia with an additional sibling table (as described in Section 5.1.1). The main task of the overlay layer is to provide a unified routing abstraction to higher components. Members of the *application layer* can then be developed independently from individual overlay approaches. The unified overlay API is defined in the *BaseOverlay* class (*src/common/BaseOverlay.h*). All overlay implementations inherit from this class.

Arriving at the application layer, an additional division into tiers can be made, with higher tiers relying on the functionality of lower ones. In the presented implementation, the actual SODESSON middleware is at the second tier. It uses a DHT component at tier 1, but can also communicate directly with the KBR overlay. Actual SODESSON applications reside at tier 3.

Several supportive components exist that can not be clearly associated with any of the presented layers. Out of those, the *GlobalObserver* category of modules is especially relevant for this work. Global observers are unique entities that form a supportive frame for the simulation environment, by collecting and disseminating global knowledge. Important global observers are for example the *GlobalStatistics* observer, that collects and processes statistical information from all modules, and the *GlobalNodeList*, that tracks all nodes currently residing in the overlay. In this work, an additional global observer module was implemented that generates user entities and a social network between them, thus delivering a frame for user-centric networking simulations. The module will be introduced in greater detail in Section 6.2.1.

On a more practical level, Figure 6.2 shows the portion of OverSim's directory structure that is relevant for understanding the presented implementation effort. Those are also the paths under which modifications were made in the course of this work.

6.2 User Model and User Behavior

Before presenting the actual SODESSON prototype, components required for establishing a suitable simulation scenario will be described first. Specifically, new

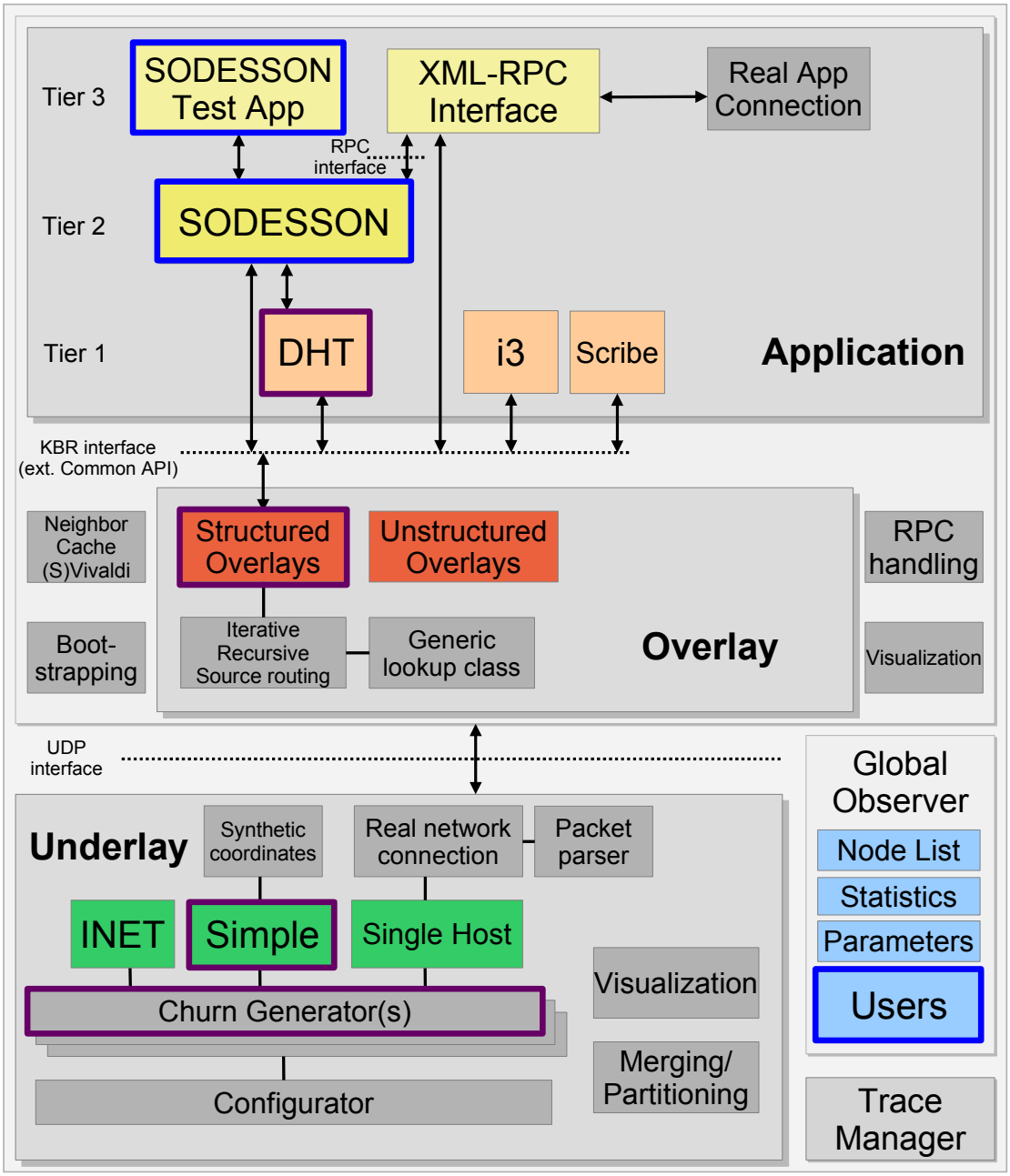


Figure 6.1: OverSim architecture with SODESSON-related modules (original diagram from [7]). New components are encircled in blue, components where modifications occurred are encircled in violet.

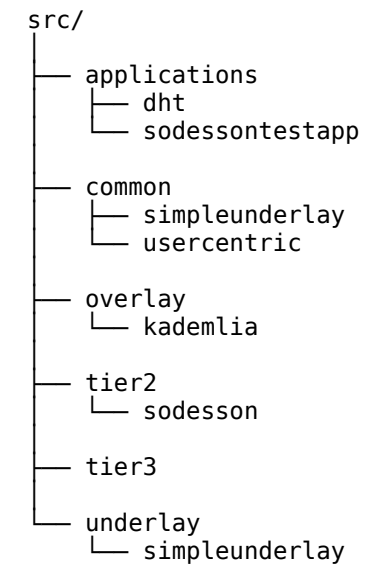


Figure 6.2: Relevant paths in OverSim's source directory.

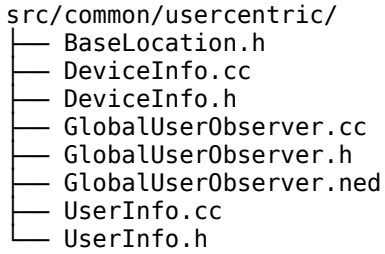


Figure 6.3: Source directory of the new user-centric networking components.

modules enabling the simulation of user-centric networking scenarios will be introduced, together with a test application for SODESSON that models the behavior and usage patterns of a generic instant messaging application. Additionally, modifications enabling the deliberate clustering of nodes in the underlying network will be presented.

6.2.1 User-centric Networking Support

Several additions to the OverSim framework were required for the establishment of a user-centric simulation environment. For one, the notion of users had to be introduced, together with a way of expressing social connections between them. Based on this, ways of actually generating users and a social graph had to be devised and the user-related information had to be made available to interested components. In the context of the SODESSON middleware, an internal mapping between users and devices was required as well.

In the presented implementation, these tasks are handed by a novel *GlobalUserObserver* and a few related modifications. Figure 6.3 shows the directory containing the discussed additions. The *BaseLocation* class (defined in *BaseLocation.h*) is currently not required for providing user-centric networking functionality, but may be used in the future for tying individual users to locations in the underlying network (see also Section 6.2.3). *UserInfo* and *DeviceInfo* objects (defined in *UserInfo.h* and *DeviceInfo.h*, respectively) are used internally for representing users and devices. Amongst

other things, UserInfo objects store the list of friends for each user and a list of DeviceInfo objects for the devices it owns. DeviceInfo objects hold only a device's overlay ID and a flag denoting its availability status. Upon startup, the actual GlobalUserObserver generates a number of UserInfo objects equal to the number n of nodes in the system, as set in the OverSim configuration. LifetimeChurn generators are used in all SODESSON simulations, with the average lifetime of nodes equalling the time they spent offline. Additionally, the reentry of nodes is configured, so that in effect only $2 * n$ distinct nodes are generated during the course of one simulation. Conclusively, by setting the number of users equal to n, every user will be mapped to, on average, two devices, which is deemed a realistic constellation. The actual connection of devices to users is done whenever a node (re-)enters the overlay. Upon receiving its overlay ID, the node calls the observer's *getUser()* function, passing it its newly learned identifier. The observer module then checks if a previous association with a user exists and creates one if not (choosing the new owner randomly). Listing 6.1 contains the definition of the GlobalUserObserver class, from which more elements of its public API can be seen.

```
class GlobalUserObserver: public cSimpleModule
{
public:
    GlobalUserObserver();
    ~GlobalUserObserver();

    UserInfo* getUser(const OverlayKey& userId);
    UserInfo* getRandomUser();
    UserInfo* getRandomFriend(const OverlayKey& userId);

    UserInfo* registerDevice(DeviceInfo& device);
    DeviceVector getSiblingDevices(const OverlayKey& deviceId);
    DeviceVector getFriendDevices(const OverlayKey& deviceId);

    bool isFriendDevice(const OverlayKey& deviceIdA,
                        const OverlayKey& deviceIdB);

protected:
    virtual void initialize();

private:

    void doGenerateFriendships();

    UserVector users; // all UserInfo objects
    UserMap userMap; // userId -> UserInfo

    KeyMap deviceOwnershipMap; // deviceId -> userId
};
```

Listing 6.1: GlobalUserObserver class (*src/common/usercentric/GlobalUserObserver.h*).

```
src/applications/sodessontestapp/
├── GlobalSodessonTestObserver.cc
├── GlobalSodessonTestObserver.h
├── GlobalSodessonTestObserver.ned
├── SodessonTestApp.cc
├── SodessonTestApp.h
└── SodessonTestApp.ned
```

Figure 6.4: Source directory of the SODESSON test application.

An additional key responsibility of the GlobalUserObserver class is the generation of a social network between users. This, too, happens during initialization, shortly after generating the actual user set. For the generation of the social network, a small-world graph generation algorithm is being used, namely the one proposed by Watts and Strogatz in [49]. As proposed in previous chapters, only one class of social connections is used at this level - *friendship*. For the parameter K of the Watts' and Strogatz' algorithm (denoting the average amount of connections per user), the value 100 is chosen. The number is inspired by the median friend count measured in the Facebook online social network, which is 99 [47]. The parameter β (that specifies the randomness of the graph) was set to 0.05. This yields a clustering coefficient of about 0.8 - a value likely to be found in real social networks[1]. In this way, the technique produces fairly realistic interconnection patterns between users.

6.2.2 SODESSON Test Application

For verifying the functionality of the developed prototype and evaluating its performance, a test application was implemented as one of the first steps in the development process. Figure 6.4 gives an overview of the component's source directory. The test functionality is realized by two core modules - the actual *SodessonTestApp* and an additional *GlobalSodessonTestObserver*.

The SodessonTestApp was designed to simulate the behavior and usage patterns of a generic instant messaging application. Instances subscribe to a set of topics and publish short messages to them at random intervals. The GlobalSodessonTestObserver acts as a point of reference, by noting all published messages and helping the SodessonTestApp instances to register missed publications. The information from the global observer is used for statistical purposes, for calculating the ratio of correctly received publications. Additionally, for direct messages, i.e. messages where both sender and receiver were available at the time of sending, the latency between the initial publish and the receive by the subscriber is measured and recorded.

For the simulations in this thesis, the parameters for the SODESSON test application were chosen with the goal of providing a close approximation of real instant messaging traffic. A snipped from OverSim's configuration file that shows the parameters of the component can be seen in Listing 6.2. The average send period of 200 seconds was inspired by a paper by Avrahami and Hudson [3], where an average of 17.6 messages per hour was measured for participants in a study on instant messaging behaviour. The maximum number of subscriptions (*maxNumSubscriptions*), i.e. the maximum number of topics that a node will be subscribed to simultaneously, is also derived from the results of that study. From the authors' data, an average value

[1] A clustering coefficient of 0.8 is found for example in collaboration graphs of film actors, which are common surrogates for social graphs [49].

of seven instant messaging sessions per day can be derived[2]. Thus, subscriptions to seven topics were also assumed for SodessonTestApp instances (the implementation is such, that test application instances always subscribe to the maximum number of topics that they are allowed to). The length of the instant messages sent by the test application is constant and hardcoded to a value of 68 characters. This number reflects the average of 13.5 words per instant message measured in [25] and the assumption of an average of 5 characters per word in an ordinary English text. The value for the *receiveTimeout* parameter is arbitrary, additional evaluations will be made to determine the messaging delays of the solution. Lastly, the parameter *chooseTopicsSocially* enforces that a node subscribes to only such topics where all other subscribers are friends to the node's owner. This reflects the assumption that only befriended users will engage in instant messaging conversations.

```
**.tier3*.sodessontestapp.averageSendPeriod = 200s
**.tier3*.sodessontestapp.maxNumSubscriptions = 7
**.tier3*.sodessontestapp.receiveTimeout = 120s
**.tier3*.sodessontestapp.chooseTopicsSocially = true
```

Listing 6.2: SodessonTestApp configuration (*simulations/omnetpp.ini*).

6.2.3 Clustering of Nodes in the Underlying Network

To model the assumption that the devices of socially connected users are often also close to each other in the underlying network (a claim backed, to some extent, by the results in [47]), modifications to the LifetimeChurn churn generator were made so that the underlay positioning of created nodes could be influenced. Specifically, the coordinates of a point in the SimpleUnderlay's synthetic coordinate system (used, for example, for calculating delays) can now be passed to the component, together with a maximum round-trip time (RTT). Node locations in the underlay network are then generated randomly in the proximity of that point, so that the RTT between each two nodes placed in this way does not exceed the preconfigured value. Thus, the described modification enables the clustering of nodes in the proximity of a specific location. Listing 6.3 shows a code snipped demonstrating the generation of node coordinates in this approach (for *useLocations* = true). The variables *locationsCenterX* and *locationsCenterY* reflect the coordinates set in the configuration file, while the *locationsScatterRadius* is calculated from the desired maximum RTT. The *createNode* functions of the *UnderlayConfigurator* (*src/common/UnderlayConfigurator.h*) and *SimpleUnderlayConfigurator* (*src/common/simpleunderlay/SimpleUnderlayConfigurator.h*) had to be modified as well, to support the passing of specific coordinates at which the created node should be placed.

```
TransportAddress* ta;
if(useLocations) {
    NodeRecord* location = new NodeRecord();
```

[2]By calculating the number of sessions recorded per hour (Total Sessions / Total Hours Recorded) and multiplying it with the average number of hours recorded per participant per day.

```
    location->coords[0] =
        uniform(locationsCenterX-locationsScatterRadius,
                locationsCenterX+locationsScatterRadius);

    location->coords[1] =
        uniform(locationsCenterY-locationsScatterRadius,
                locationsCenterY+locationsScatterRadius);

    ta = underlayConfigurator->
            createNode(createType, initialize, location);
} else {
    ta = underlayConfigurator->
            createNode(createType, initialize);
}
```

Listing 6.3: Clustering of nodes around a previously specified set of coordinates (*src/common/LifetimeChurn.cc*).

6.3 SODESSON Component

In the presented implementation, the actual SODESSON middleware resides at tier 2 in OverSim's layering logic. It employs a standard DHT implementation with sibling replication for all of its storage needs. The used DHT component was already part of the OverSim framework. In the given test scenario, all stored applications messages have sizes below the kilobyte range. Thus, the usage of a DHT as the exclusive storage system is acceptable, as discussed earlier. Mechanisms for dealing with larger files, e.g. through means of distributing data on socially connected and local nodes, were not implemented. On the overlay layer, the developed prototype uses OverSim's Kademlia module, with parameters set so that it behaves like an R/Kademlia implementation with an additional sibling table.

6.3.1 Module Structure

Figure 6.5 depicts the directory containing the actual developed SODESSON implementation. The architecture of the prototype is such that a clear separation between client and RP-related functionality exists. Consequently, all client-related functionality is encapsulated in the *SodessonClientModule* class[3]. It is also in this class that the communication with SODESSON applications is handled. RP-related functionality is encapsulated in *SodessonRPModule*. This separation of functionality aims at a higher clarity of design and is purely organisational. From the standpoint of OverSim, there is only one SODESSON module, namely the one defined by the *Sodesson* class. Thus, all responsibilities concerning the communication with other OverSim components is concentrated in this main module. It provides send functionality to its submodules and forwards them messages from other OverSim components.

In addition to the discussed classes, a number of *SodessonEntry* classes can be found in the presented source tree, e.g. *SodessonSubscriptionEntry*. Those classes serve the

[3]Unless otherwise noted, classes in this section are always implemented in the source files with the same name.

```
src/tier2/sodesson/
├── Sodesson.cc
├── Sodesson.h
├── Sodesson.ned
├── SodessonAckEntry.cc
├── SodessonAckEntry.h
├── SodessonClientModule.cc
├── SodessonClientModule.h
├── SodessonContactManager.cc
├── SodessonContactManager.h
├── SodessonMessage.msg
├── SodessonPublicationEntry.cc
├── SodessonPublicationEntry.h
├── SodessonRPModule.cc
├── SodessonRPModule.h
├── SodessonSubscriptionEntry.cc
├── SodessonSubscriptionEntry.h
├── SodessonTopicEntry.cc
└── SodessonTopicEntry.h
```

Figure 6.5: Source directory of the SODESSON prototype.

internal representation of data and, in several cases, also encapsulate data objects that need to be stored in the DHT. The file *SodessonMessages.msg* holds the different message types employed by SODESSON. The *SodessonContactManager* module is currently only a placeholder and is not being used. Information about a node's social context, i.e. its owner and the friend users of that owner, is retrieved directly from the GlobalUserObserver by components that require it.

6.3.2 Particularities of the Implementation

In the remainder of this section, non-trivial implementation details of some SODESSON components will be discussed. This includes the initialization of new RPs, the detection of unavailable subscribers, the PACK mechanism and the storage of arbitrary data objects in the DHT.

6.3.2.1 RP Initialisation

Whenever a node joins the overlay, it can happen that it immediately becomes responsible for a given topic $\mathcal{T}$ (due to its overlay ID). In such cases, publications and other requests to the RP might arrive at the new node before it has had a chance to fully complete the join process. Specifically, the node might still be unaware of some of its overlay siblings and it might not yet have received the DHT entries corresponding to $\mathcal{T}$ (being the RP for $\mathcal{T}$, it is also the responsible DHT node for datasets associated with that topic). When handling publish requests, such cases manifest itself in the fact that, upon requesting all subscribers from the DHT (in order to properly disseminate the message), the RP component does not receive any datasets in return. When this happens, the current implementation does not respond to the publish request. The publish is then retried by the SODESSON client component at the publishing node, with exponentially increasing intervals between publish attempts. Eventually, the RP becomes fully initialised and the publish succeeds.

6.3.2.2 Detection of Unavailable Subscribers

The detection of failed or otherwise departed nodes is handled by registering failed message deliveries. In OverSim, the latter is usually handled by the RPC subsystem

with it's built-in acknowledgement and timeout mechanisms. In the present scenario however, the available RPC mechanisms do not work, for following reasons:

- SODESSON messages are always sent to overlay keys instead of individual underlay addresses.
- When routing a message to a specific node ID, and the node belonging to that ID is currently not available, the message is instead delivered to the overlay node whose address is closest to the destination key.
- With the current implementation, this accidental receiver is not aware of the original destination ID of the message and thus processes it normally. Specifically, it also sends a normal reply to the sending node.
- Since the sending node has received a reply (acknowledgement) for its message, it cannot tell that the real destination node is actually not available. Specifically, the processing of replies is handled by the RPC subsystem and, again, does not compare the real target key with the ID of the receiver.

So, the detection of unavailable SODESSON nodes by SODESSON components would require a partial rewrite of OverSim's RPC subsystem. Namely, an additional sending mode must be implemented that enforces the routing of a message to a specific overlay ID, returning an error if no node with this specific key is found. Here however, a different workaround to this problem was devised that was easier to implement. The following was done:

- SODESSON messages typically targeted at clients, for example notification messages, were equipped with an additional *destID* field holding the target node's overlay ID.
- With this additional field, the SODESSON client can verify whether it is the real destination of a message.
- If not, it still sends a normal reply to the sender (to obey OverSim's RPC logic), but also sends a *NodeFailNotification* message to the RP corresponding to the topic that the received message is associated with.

All NodeFailNotification messages for subscribers to a specific topic are gathered at that topic's RP. The RP stores a counter for each subscriber it has received NodeFail messages for. The counter is incremented for each received fail message and gets reset whenever a message by the respective subscriber arrives. Once the fail counter reaches a certain value (here, 8 was chosen), the subscriber is marked as inactive in the DHT and no more notifications are sent to it until it resubscribes or otherwise demonstrates that it is active.

6.3.2.3 PACK Mechanism

As an important element of the PACK mechanism, the PACKs for all currently active subscribers must be known by the RP, and the transmission and retransmission of publications must be triggered based on those PACKs. In the present implementation, this is handled by maintaining a dedicated *PACK cache* for each topic that

the RP is responsible for. The cache is filled with *AckCacheEntries* representing pending acknowledgements for currently active subscribers. Every cache entry is associated with a timer, for timing periodic delivery attempts. The PACK for a topic gets filled immediately after a node learns it is responsible for that topic. After that, new PACKs are retrieved from the DHT whenever a node subscribes or becomes active after a period of inactivity, as in both cases publications could be pending for it. Lastly, all PACK caches maintained by a node are updated periodically as well, by first getting all active subscribers for each topic from the DHT and, based on that, all currently relevant PACKs. Depending on the configuration of the DHT component, this periodic updating might generate a lot of extra traffic. A better way for the RP to stay updated about changes in the DHT is for the DHT to notify the RP component whenever changes to some of its datasets occur. However, this was not easily realizable at the time of implementation and thus not done.

6.3.2.4 Storage of Arbitrary Data Objects in the DHT

On many occasions, the presented SODESSON prototype needs to store complex data objects in the DHT, e.g. PACK entries or subscription entries with extra information like the availability status of the subscriber. With the currently available DHT implementation however, only the storage of *BinaryValue* objects is possible, which are, in essence, strings. Thus, an rudimentary serializer was integrated into all classes whose objects needed to be storable in the DHT. Listing 6.4 demonstrates the used technique (dubbed the *poor man's serializer*) as implemented in the *SodessonSubscriptionEntry* class. The *pack()* function basically outputs the object's data to a string using standard C++ streams. Its counterpart, *unpack()*, can then reconstruct the original object from that string. A more complex example of this technique can be found in the implementation of the *SodessonAckEntry* class.

```
// poor man's serializer
BinaryValue SodessonSubscriptionEntry::pack() {
    std::stringstream ss;
    ss << subscriber.toString(16) << endl;
    ss << isActive;

    return BinaryValue(ss.str());
}
void SodessonSubscriptionEntry::unpack(const BinaryValue& v) {
    std::stringstream ss;
    ss << v;

    std::string subscriber_string;
    ss >> subscriber_string;
    subscriber = OverlayKey(subscriber_string);

    ss >> isActive;
}
```

Listing 6.4: Basic serializer (*src/tier2/sodesson/SodessonSubscriptionEntry.h*).

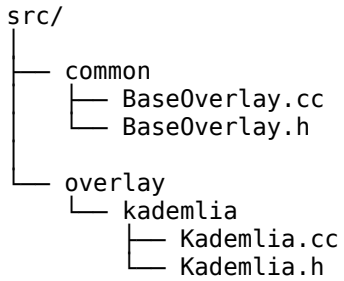


Figure 6.6: Files modified in the course of implementing social routing functionality.

6.4 Social Routing Additions

This section describes the implementation effort of integrating social information into OverSim's overlay routing layer. Figure 6.6 shows the locations of the performed modifications. The made changes are limited almost exclusively to the Kademlia module, where a social table was added in accordance to Section 5.6. The modifications in *BaseOverlay.h* and *BaseOverlay.cc* reflect changes to OverSim's generic overlay API that were required by the new social components and were deemed useful for further efforts into implementing social routing mechanisms. Most importantly, those changes include the introduction of the *addFriendNode()* function, a function that can be used to inform the overlay about devices belonging to befriended users.

In the previously introduced SODESSON component, the addFriendNode() function is called in following cases:

- A notification message about a publication is received directly from the publishing device and the publishing user is a social contact.
- In configurations where all devices store their IDs in the DHT, under their owner's user ID as was suggested previously, every node that joins the overlay immediately requests the addresses of all devices belonging to its owner and to friends of its owner. Node addresses received from the DHT as a result of this query are all passed to addFriendNode().
- In the former scenario, where the IDs of befriended devices can be retrieved from the DHT, *FriendDeviceNotification* messages are sent to all devices discovered in this way. In this way, nodes are notified whenever a new befriended device becomes available and can pass this information on to the overlay.

The modified R/Kademlia implementation reacts to *addFriendNode()* calls in two ways. Firstly, it inserts the node ID in question into the *friendDevices* set, a data structure holding the overlay identifiers of all known befriended nodes. Secondly, it checks whether the new friend device is already present in some ordinary Kademlia bucket. If it is, it is moved to the social table. The friendDevices set is extensively used in Kademlia's *routingAdd()* function, where the decision is made in which bucket a given overlay neighbor must be put. After gathering all socially connected nodes in the social table however, they are still treated in the same ways as normal nodes, e.g. when routing messages.

6.5 Other Modifications and Additions

In addition to the already mentioned changes, a number of smaller modifications and additions to the OverSim codebase were made:

- In the DHT module, additional statistical functions were implemented that categorize messages based on the SODESSON data types they are associated with (a put call on a subscription entry is associated with a subscription entry, for example). The messages are then logged in such a way that, in the end, the amount of DHT traffic that each individual SODESSON data type induces can be seen in the simulation output.
- The GlobalStatistics module was enhanced by the possibility of generating quantile histograms, through the statistical class *cPSquare* provided by OMNet++. The new GlobalStatistics function is called *recordPSquare()* (defined in *src/common/GlobalStatistics.h*).
- Kademlia and the GlobalNodeList component were modified to support the evaluation of the status of Kademlia's sibling table. It is now possible to log the sibling table's completeness as a function of time, i.e. the percent of a node's real siblings that are registered in the node's sibling table. This functionality is optional and can be activated by setting Kademlia's new *siblingReportInterval* parameter.
- A new plotting tool was implemented based on the *plot.py* script (*simulations/tools/plot.py*) available in OverSim. The new program, *barplot.py* (*simulations/tools/plot.py*), is invoked similarly to plot.py but produces bar diagrams instead of normal plots.

6.6 Summary

In this chapter, an implementation of the previously developed concepts and mechanisms was presented, using the OverSim simulation framework. In addition to the implementation of an actual SODESSON component providing user-centric networking functionality as proposed in Chapter 5, several modifications to OverSim were made to create a suitable simulation environment for user-centric networking. Specifically, user entities were introduced and a mechanism was implemented for generating a social network between them. The clustering of nodes around previously specified locations in the underlying network was made possible and a test application for SODESSON was developed, that models user behavior in the context of an instant messaging application. Previously discussed improvements to SODESSON's basic design, namely the direct publish modification and the introduction of a social table to R/Kademlia, were integrated into the implementation as well. Lastly, several smaller changes and additions were made to existing OverSim components, mostly related to the collection of statistical data.

7. Evaluation

For the evaluation of the designs presented in Chapter 5 and their implementation in Chapter 6, a set of simulation studies was performed in the overlay simulation framework OverSim. The studies were performed with following goals in mind:

- To discover good parameter combinations for the different system components.
- To evaluate the basic feasibility of the solution to act as a foundation for the SODESSON middleware.
- To further evaluate the performance of the approach and the impact of the different modifications proposed for it.

Based on these goals, four groups of simulations were performed: a pure DHT study to determine good parameters for the DHT component, a small exemplary study with the SODESSON prototype to assess its core properties and, lastly, two larger studies with the SODESSON implementation to aid the comparison of the proposed modifications and demonstrate the behavior of the solution in larger node populations. The four simulation families will be discussed extensively in the remainder of this chapter. First, however, invariant aspects of the simulation setup will be elaborated and some common terminology and abbreviations used for presenting the simulation results will be introduced.

7.1 Common Simulation Parameters and Terminology

The simulations in this thesis were run using the *SimpleUnderlay* underlay abstraction, as noted previously. The SimpleUnderlay network model is based around a synthetic coordinate system for calculating communication delays. In this work, node coordinates were generated based on real Internet latency measurements, using the *Skitter* [24] datasets provided by the *CAIDA* [13] project. In all simulation runs, 20000 seconds of activity were simulated and observed, with additional initialization and transition phases in the beginning of each run during which no measurements

occurred. For each evaluated parameter combination, 5 simulation runs with different seeds (for OverSim's random number generator) were performed. The sizes of the simulated networks were varied across simulation studies and individual configurations. An average number of simultaneously active nodes was configured for each run, with the actual node population fluctuating slightly due to churn.

Unless otherwise noted, all simulations in this chapter were run in conditions of *moderate churn*. Namely, OverSim's *LifeTimeChurn* churn generator was used with an average life and dead time of 10000 seconds. Based on the measurements taken for the *KAD* network in [45] and [43], this value is estimated to reflect realistic average user behavior. Whenever nodes leave the system, they do so in a non-*graceful* way, i.e. without notifying their peers. The reentry of nodes is configured, so that whenever a dead node gets recreated, it retains its old node ID. At the beginning of each simulation, a user population is generated as described in Section 6.2.1. Like stated there, the size of the user population equals the configured number of simultaneously active nodes, resulting in an average of 2 devices per user.

Unless otherwise noted, presented statistical variables are always related to single nodes. So, if a value like *Bytes Sent/s* is given, for example, it denotes the amount of bytes that each individual node in the simulation sends on average. Result values were produced by averaging between all simulation runs performed for a given parameter combination. Also, confidence intervals were calculated based on the values from those runs. All confidence intervals specified in this thesis represent a 95% confidence level.

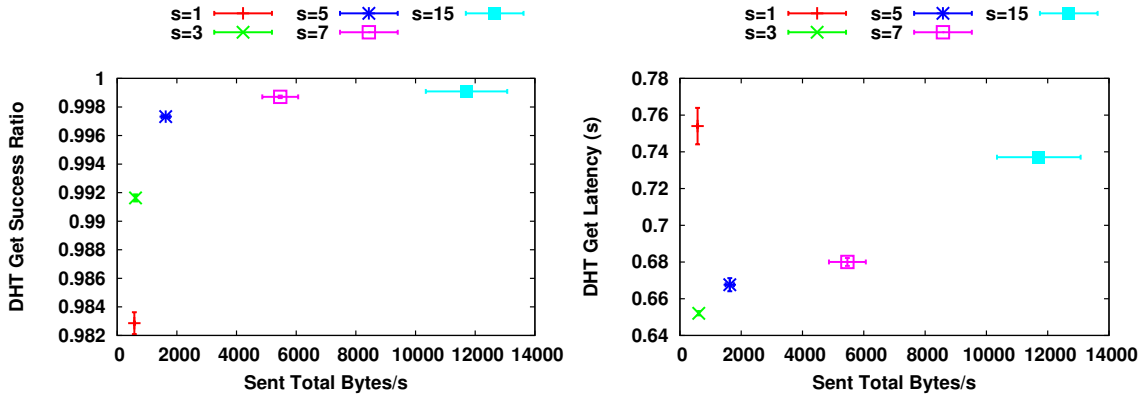
When visualizing the results, a few abbreviations are additionally used for denoting different parameter combinations. *DP* signifies the activation of the direct publish mechanism introduced earlier. *ST* denotes the existence of a social table in R/Kademlia, with *active* and *passive* in this context referring to the two approaches for discovering befriended nodes. Also, the terms *undelayed* and *delayed* are used, for describing different types of publications. With undelayed publications, both the subscriber and the publisher of a message are available at the time of publication so the message is transmitted directly, i.e. without persisting it in the DHT. Delayed publications, on the other hand, are persisted by the RP because of inactive subscribers and get delivered once their destination nodes become available.

7.2 Selection of DHT Parameters

Before the actual evaluation of SODESSON, a preliminary study of the DHT component was performed to determine good configuration parameters for the DHT and KBR modules.

7.2.1 Simulation Setup

Above all, the goal of this study was do determine a suitable value for the number of siblings s. This number determines the size of the KBR component's sibling table and also tells the DHT how many replicas of its datasets it should create. Two studies were performed in this context, using R/Kademlia, OverSim's standard DHT implementation and the *DHTTestApp* included in OverSim. The parameters of the test application were set to resemble the expected DHT usage patterns of the



(a) Success ratio and bandwidth consumption.

(b) Latency and bandwidth consumption.

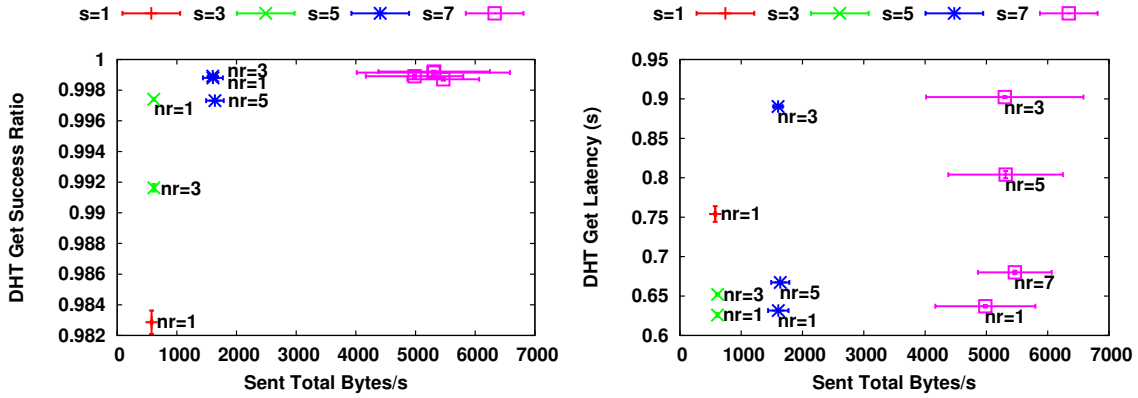
Figure 7.1: DHT parameter selection: variation of the number of siblings s. The number of siblings contacted for each request is also s. $0.5 \cdot s$ equal replies are required for a get request to succeed.

SODESSON prototype and the SodessonTestApp. All simulations were performed for a population of 10000 nodes. The results in this section were plotted using *scatter diagrams*. Each point in these plots represents one parameter combination. The x and y coordinates of the point correspond to the values of two statistical variables measured for its respective configuration. In this way, scatter plots can give a sense for the performance versus cost ratios of the presented parameter combinations, by choosing the plotted variables accordingly. Often, they are thus also referred to as *performance versus cost* diagrams.

7.2.2 Results

Figure 7.1 shows the results of the first simulation study. Here, only the parameter s was varied as described earlier. For each DHT get request, s siblings had to be contacted. From these s contacted siblings, $0.5 \cdot s$ had to reply with the same value for the get request to succeed. The result diagram clearly shows that 3 and 5 are good candidate values for s, while $s = 15$ induces an extra amount of bandwidth overhead that is disproportionate to the success ratio increase it induces. Based on these insights and the fact that simulations with $s = 15$ run significantly slower than the other configurations, the parameter combination $s = 15$ was omitted in the second study.

The second group of simulations evaluates more fine-grained configurations of the DHT component. Namely, the *numGetRequests* parameter of the DHT component was altered as well. In the following, it will be denoted only as nr. The nr configuration variable denotes the number of individual requests a get call induces. For example, if $nr = 3$, the responsible node and 2 of its siblings are contacted when performing the operation. Clearly, it always holds that $nr \leq s$. In the performed simulations, from the nr nodes being contacted, $0.5 \cdot s$ need to reply equally for a request to succeed. If $nr < 0.5 \cdot s$, then only nr contacted nodes need to reply equally instead. So, for example, for $(s = 7, nr = 5)$, 4 equal replies need to be received, while for $(s = 7, nr = 3)$, 3 equal replies are sufficient. For $nr = 1$, the



(a) Success ratio and bandwidth consumption.

(b) Latency and bandwidth consumption.

Figure 7.2: DHT parameter selection: variation of the number of siblings s and the number of requests per get operation nr (the number of contacted nodes). $0.5 \cdot s$ or nr equal replies are required for a get request to succeed, whatever is lower.

answer of the queried node is always assumed to be correct by the querying node. Figure 7.2 shows the results of the study.

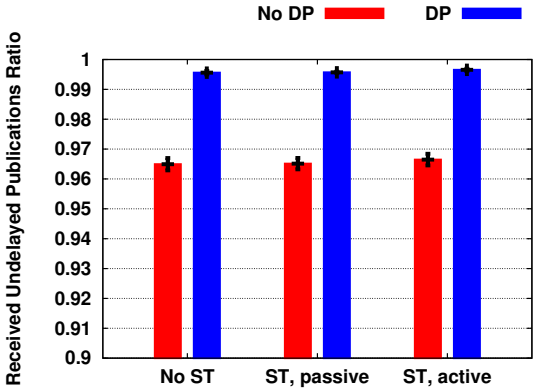
Looking at the resulting plots, the combinations $(s = 3, nr = 1)$ and $(s = 5, nr = 1)$ appear most promising, i.e. having the best cost versus performance ratios. However, the former configuration has a significantly lower bandwidth overhead than the latter, with a just slight decrease in reliability. Resting on the assumption that a success rate increase of less than 0.2% does not justify a doubling of bandwidth consumption (a total increase of about 1000 bytes per second), $(s = 3, nr = 1)$ is thus the better candidate configuration for the DHT component. On a side node, the contacting of only one node per get request raises serious security concerns, as this node could be malicious. In the current SODESSON design however, the main querier of the DHT are nodes acting as RPs and, again, only one RP is contacted in all SODESSON mechanisms. Thus, this problem should first be addressed in the SODESSON layer. Additionally, when $nr = 1$, an RP does not have to make any requests to other nodes at all, as he is always also the main DHT replica location for the datasets it requires.

7.2.3 Conclusion

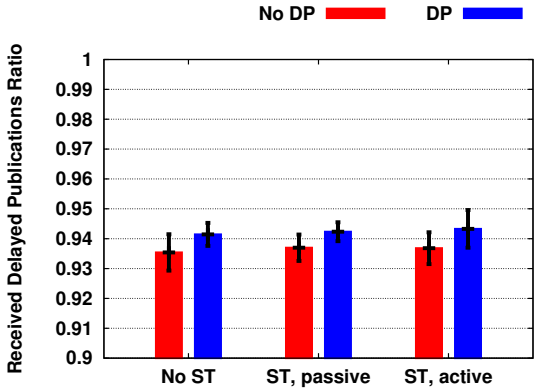
Based on two simulative studies of the DHT component, the parameter combination $(s = 3, nr = 1)$, i.e. 3 overlay siblings per node of which one is contacted during each get operation, was determined as best for the current prototype of the SODESSON middleware. All subsequent simulations in this chapter were performed with this DHT configuration.

7.3 Basic Feasibility

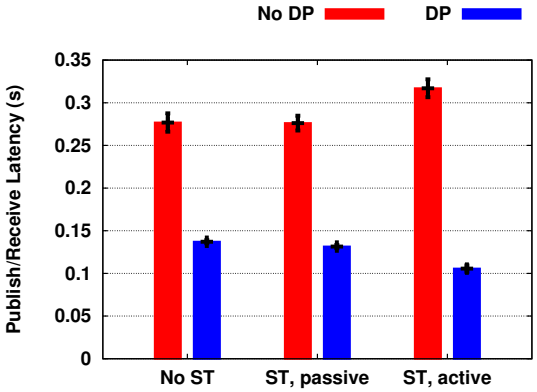
In this section, the question will be answered whether the developed prototype represents a valid solution to the problems stated previously. The reliability, latency and bandwidth consumption of the developed solutions will be assessed for an exemplary network size of 1000 nodes. Basic characteristics of the prototype and its different configurations will be noted based on the results.



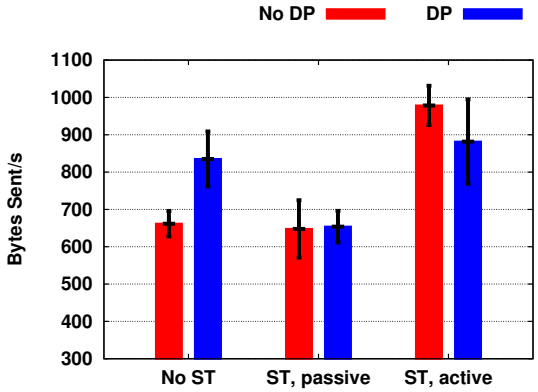
(a) Success ratio for undelayed publications.



(b) Success ratio for delayed publications.



(c) Latency for undelayed publications.



(d) Bandwidth consumption.

Figure 7.3: Exemplary simulation results for a network with 1000 active SODESSON nodes (on average) and moderate churn. *DP* denotes whether the direct publish modifications are active. *ST* signifies the usage of a social table in R/Kademlia, with *active* and *passive* representing the two approaches for discovering befriended nodes.

7.3.1 Simulation Setup

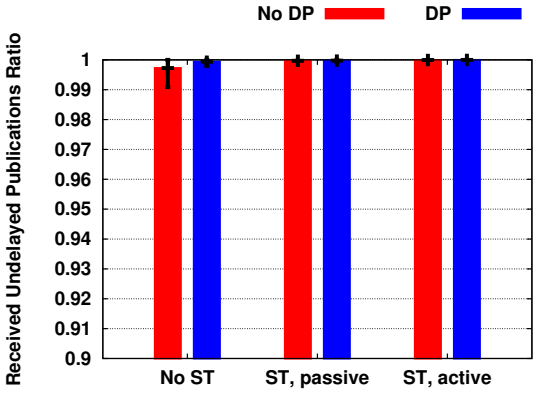
Various possible configurations of the developed prototype were simulated in a exemplary scenario with a node population of 1000 nodes (on average). SODESSON and SodessonTestApp instances were running on all nodes. The parameters for the DHT component at each node were set as described in the last section. The SODESSON parameters being evaluated are marked in the result diagrams using the abbreviations introduced in Section 7.1.

7.3.2 Results

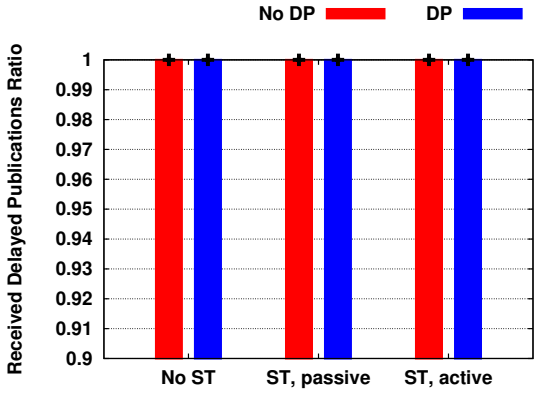
Two simulation studies were performed - one with moderate churn and one without any churn at all (using OverSim's *NoChurn* churn generator). The results of the two studies can be seen in Figures 7.3 and 7.4, respectively.

7.3.2.1 Delivery Ratios

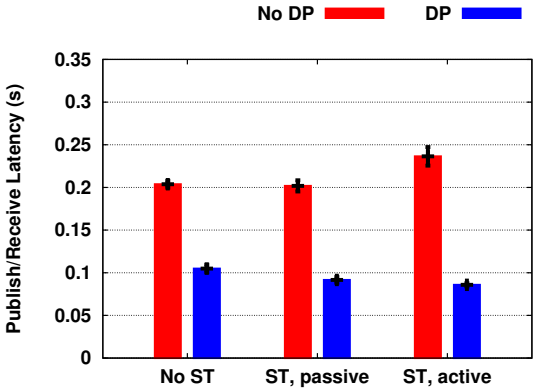
From the results of the simulations with churn, it can be seen that the average success ratio of publications is below 100%. For every 20 messages a node receives,



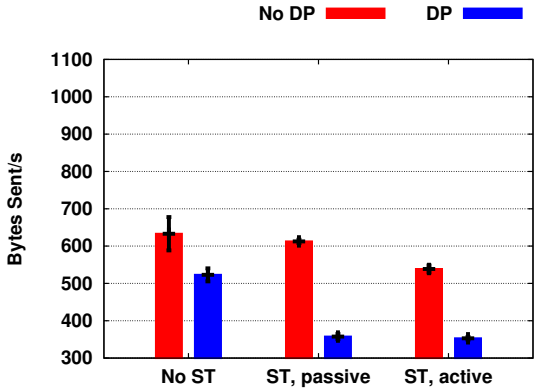
(a) Success ratio for undelayed publications.



(b) Success ratio for delayed publications.



(c) Latency for undelayed publications.



(d) Bandwidth consumption.

Figure 7.4: Exemplary simulation results for a network with 1000 active SODESSON nodes and no churn. Parameter names as in Figure 7.3.

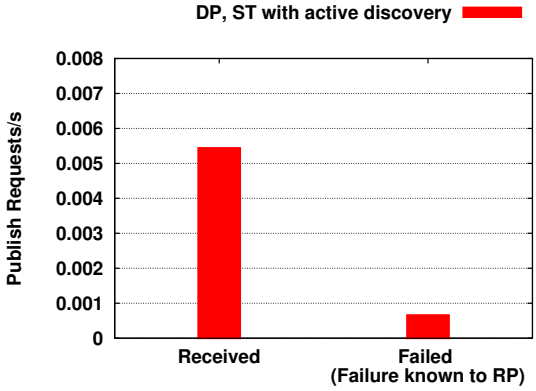


Figure 7.5: Publish requests received by an RP and the number of them that could not be processed because the RP was not ready.

it misses one. As can be seen from the comparison with the results from Figure 7.4, these losses are mainly due to churn. Mostly, they are caused by the following situation that was not accounted for sufficiently in the implementation of the prototype: Whenever a new node joins the overlay and immediately becomes the RP for one or more existing topics (due to its node ID), requests for these topics might reach the new RP before it is fully prepared for its new role. Most importantly, the datasets corresponding to these topics might not yet be fully available to the new node, as it takes some time for it to discover its siblings and assume its role as a DHT storage location. Whenever an RP senses that it might not yet be ready for servicing a publish request (i.e. when it notices that it does not have any data about the topic in question), it does not respond to the call. In this way, the publisher can learn that its publication attempt was unsuccessful and retry it. After 3 retries, SODESSON clients in the performed simulations stop trying. In such cases, the publishing node knows that its publication was unsuccessful. However, the test applications still counts the publication when calculating delivery rates. So, in many of the cases where messages are counted as lost, the publishers are actually aware that something is wrong and might act differently. Figure 7.5 shows the frequency of publish requests that could not be processed by an RP in comparison to the total number of publish requests received by an RP per second. It can be seen that the percentage of publications rejected at an RP is about 9%. Keeping in mind that publishers retry their publications up to 3 times if they don't receive a reply from the RP, many of these rejected publications are actually retransmission attempts of publications that were rejected previously. As an estimation (based on the number of retries), the number of unique publications dropped by the RP because he was not fully initialized for serving them is thus somewhere between 2% and 4.5%. Consequently, cases where the RP drops messages because he is not ready for serving them really are responsible for many of the failed message deliveries registered by the test application.

On a side note, the error rates for delayed messages (Figure 7.3b) are usually significantly higher than those for undelayed ones. This is simply due to the fact that for delayed messages, the services of the RP are required two times - once for registering the original publication and generating PACKs, and then for detecting that a subscriber has become available again and retransmitting the message. Failures due to an insufficient initialization of the RP may surface during both of these steps.

The improvement of delivery ratios in i3-style publish/subscribe communication was not the main focus of this thesis and several possible approaches for avoiding message losses were not considered for implementation. For one, the migration to new RPs could be made slower, i.e. new RPs could relay messages to the respective old ones until they become fully initialized. Also, publishers could exert more control over RPs, checking more thoroughly whether their publications were processed correctly.

7.3.2.2 Effects of the Direct Publish Mechanism

Such a form of control is partially realized through the use of the direct publish modification, where publishers keep track of subscribers and thus manage to compensate, to some extent, the RPs lack of knowledge. The resulting reliability improvement is especially visible for undelayed publications (Figure 7.3a), as there messages can be delivered without any help from the RP. According to expectations, the direct

publish improvement also introduces a significant lowering of communication latencies, by more than half in the simulated scenario. In addition to the measurement of the communication delay over the SODESSON prototype, the latency of a hypothetical direct transmission through the underlay network (between publisher and subscriber) was calculated as well for each publication and subscriber, yielding an average value of 79 milliseconds. Since SODESSON is based on an overlay network spanned on top of the considered underlay, this value is the lowest achievable latency average between communication partners in the performed simulations. The latency measured for the best parameter combination of the SODESSON prototype ($DP = true, ST = true$)) is close to this value, with 106 milliseconds. This is a highly satisfying result. It should be noted, that the achievement of close to optimal latencies also implies the avoidance of large communication detours and thus indicates a high degree of communication locality.

7.3.2.3 Effects of the Social Table

The introduction of a social table also leads to improvements in the communication latencies and message delivery ratios, albeit not consistently and on a smaller scale than the direct publish modifications. As expected, the active friend device discovery technique has a higher impact than the passive one. When direct publish is activated, it leads to a visible latency improvement in comparison to the passive approach (Figure 7.3c). This can be explained by the fact that with the active method, nodes learn about their future communication partners (which are all befriended devices) immediately after joining the network. Thus, one hop overlay routes between publishers and subscribers are possible from the beginning on, leading to next to ideal latencies. If direct publish is deactivated however, the latencies actually get worse through the use of active node discovery. The reason for this most likely lies in the fact that when direct publish is deactivated, direct messages are exchanged only between publishers and RPs and between subscribers and RPs. Since, usually, no social connections exist between those entities, the knowledge about befriended devices does not bring any benefits in this setup. However, during the initial node discovery step, a great amount of overlay traffic is produced that also leads to the discovery of many non-befriended nodes. Those nodes might not be relevant for the communication processes in later stages, but are still included in R/Kademlia's routing table. This potentially leads to the premature filling of some of R/Kademlia's buckets, so that nodes with whom frequent communication occurs, e.g. the RP of a topic a node is publishing to, can not be added later on. The impact of the social table and direct publish modifications on latencies will be discussed more thoroughly in Section 7.4, based on simulations with larger node populations.

7.3.2.4 Bandwidth Consumption

Turning to the evaluation of the bandwidth consumption of the prototype, Figure 7.3d demonstrates that for the worst configuration in this respect, the amount of sent data per second is still bellow 1 kilobyte. This is an acceptable amount of traffic, even for mobile devices. Furthermore, it can be seen that the active node discovery mechanism leads to a bandwidth consumption increase of up to 450 bytes per second. A more precise evaluation of the reasons of this increase in communication overhead will be conducted in Section 7.5, where the development of the bandwidth consumption with growing node populations will be addressed as well.

7.3.2.5 Effects of Churn

Lastly, Figure 7.4 demonstrates the effect that the exclusion of churn has on all measured variables. The effects of churn on the reliability of the solution were already discussed in the beginning of this section - most configurations yield delivery rates of 100% in setups without churn. On an important side node, the plot in Figure 7.4b is not really relevant here, as no delayed message deliveries occur if all participating devices remain online during the whole simulation. Comparing Figures 7.3c and 7.3d with 7.4c and 7.4d, latencies and bandwidth consumption are also better with static node populations. In the simulations performed in this chapter, the effects of churn are even stronger than in realistic networks, as all node leaves from the network are non-graceful, i.e. nodes leave the overlay suddenly and without notifying any of their peers. For this reason, the loss of a node does not get detected immediately, leading, for example, to misroutings and lost messages.

7.3.3 Conclusion

The developed prototype was shown to perform well in a network with 1000 active nodes. In terms of delivery ratios, there is still room for improvement. Optimizations in this aspect are possible but were not attempted in the scope of this thesis. Advantages of the direct publish mechanism have become visible, as well as the big impact of churn on the overall performance of the prototype.

7.4 Effects of Modifications

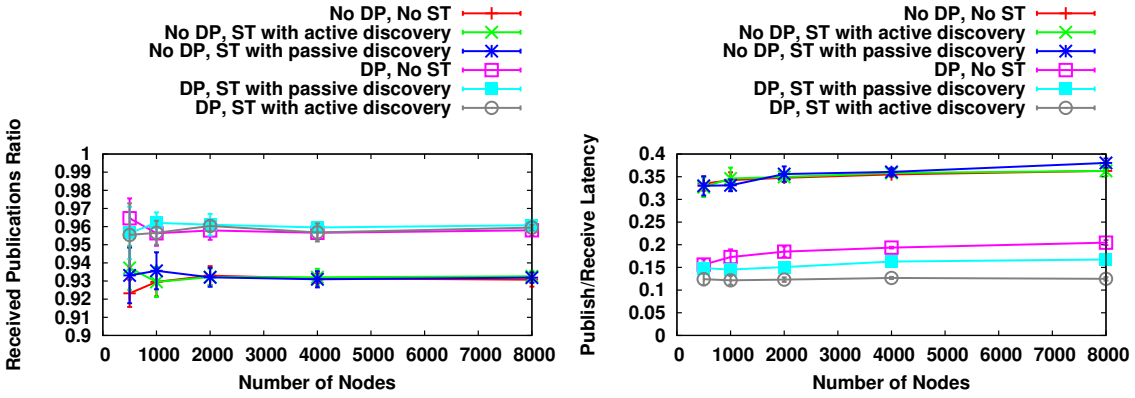
The simulation studies discussed in this section aim at further assessing the effects of the improvements proposed to SODESSON's base design, most importantly the direct publish mechanism and the introduction of a social table.

7.4.1 Simulation Setup

The same parameter combinations for SODESSON were used as in Section 7.3, but larger node populations were employed. Also, for shortening simulation times, *mixed* networks were constructed where only 10% of nodes run an instance of the SodessonTestApp (and thus initiate communication processes) while the other 90% perform only RP-related functions. The former category of nodes will be called *active* from now on. In the presented scenario, active nodes are also the only ones associated with a user entity. Social connections thus exist only between active nodes. Splitting the simulated networks in this way allows the simulation of larger node populations for less time while still receiving realistic results for delivery ratios and communication latencies. However, mixed simulations do not offer a correct value for the amount of bandwidth consumed by nodes in realistic setups. Thus, no bandwidth-related plots will be shown in this section.

7.4.2 Results

Two studies were performed, both with mixed networks as described above. The results are shown in Figures 7.6 and 7.7. The depicted received publications ratios refer to the total of all messages that should have been delivered at some point during a simulation, i.e. both direct and delayed message deliveries. In the second



(a) Average success ratio for all publications. (b) Latency for undelayed publications.

Figure 7.6: Simulation results for a mixed network setup with 10% active SODESSON nodes (running the SodessonTestApp) and 90% RP-only nodes. Friendships exist only between the owners of active devices. Moderate churn. *DP* denotes whether the direct publish modifications are active. *ST* signifies the usage of a social table in R/Kademlia.

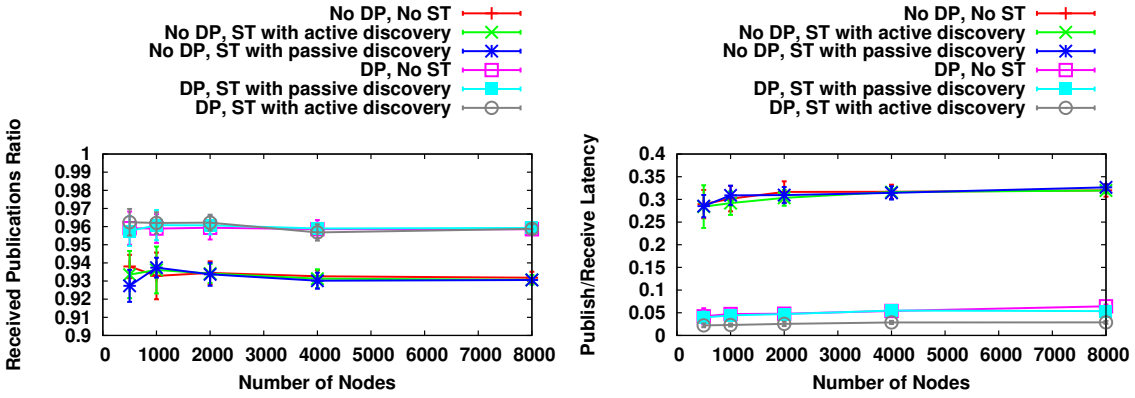
study (Figure 7.7), the underlay locations of active nodes were modified so that all of them were spawned in the proximity of one another. Specifically, they were all placed in such a way, that no pair of active nodes had an RTT between them of more than 20 milliseconds. The idea behind this setup is to reflect the fact that, in some scenarios, socially connected users are likely to be geographically close to each other as well. The nodes were clustered around a point in the underlay model's synthetic coordinate system that, given the use of node coordinates based on real Internet measurements (datasets from the CAIDA/Skitter project), roughly corresponds to the city of Karlsruhe.

7.4.2.1 Overall Performance

Several interesting properties of the prototype and its different parameter combinations can be seen from the results. For one, for all simulated configurations, both the delivery rate and the communication latency of the system remain relatively stable with growing node populations. This hints at a good scalability of the developed designs. However, an evaluation of the bandwidth consumption of the solution in larger networks is needed as well. Such an evaluation will be performed in Section 7.5. For smaller network sizes, the measured values show a large variation, especially concerning message delivery ratios. This can be explained through the fact that smaller node populations also produce fewer measurements, so that outliers have a stronger effect on the final result. Furthermore, outliers are more frequent in small node populations, as smaller overlays are more easy to destabilize and the existence of accidental one hop paths between nodes is more likely in them.

7.4.2.2 Direct Publish Mechanism

As was also noticed in Section 7.3, the direct publish mechanism introduces significant improvements to all considered performance variables. Also, the absolute difference in the results of runs with direct publish and runs without direct publish appears to be unaffected by the size of the simulated node population. For success



(a) Average success ratio for all publications. (b) Latency for undelayed publications.

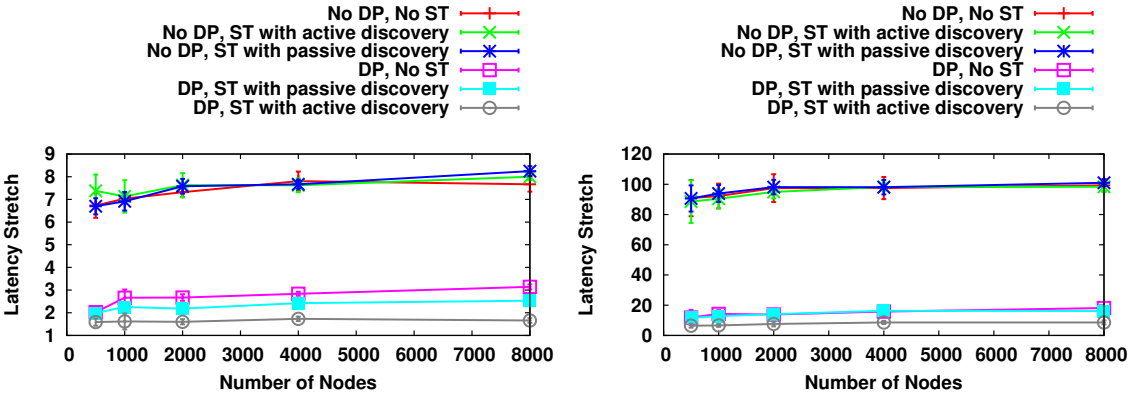
Figure 7.7: Simulation results for a mixed network setup with 10% active SODESSON nodes (running the SodessonTestApp) and 90% RP-only nodes. All active nodes are close to each other in the underlying network, with a maximum RTT of 20 milliseconds between any pair of them. Friendships exist only between the owners of active devices. Moderate churn. Parameter names as in Figure 7.6.

ratios, this absolute improvement is nearly 3%. As was already discussed, the delivery ratios can easily be enhanced further by shifting even more responsibilities to publishers and subscribers and introducing more control mechanisms for the RP. Concerning the communication latency of the system, direct publish alone results in an improvement of more than 40% (Figure 7.6b). Whenever latencies are being discussed, it is important to note that beyond showing the solution's applicability for powering time-critical applications, they are also a strong indicator for the locality of the produced traffic. The avoidance of long distant connection results in communication processes that are faster, more cost efficient and more reliable. Moreover, systems with strong locality properties are more unaffected by breaks in global Internet connectivity.

The locality improvement introduced by the direct publish modification is even stronger when modeling the geographical proximity of befriended devices (Figure 7.7b). Here however, the measured communication latencies over SODESSON need to be considered in relation to the theoretical ideal latencies between communicating nodes (i.e. when they exchange messages directly over the underlay), which are considerably lower in this scenario than in setups with randomly distributed nodes. The ratio between those two latencies was thus calculated for each received message, yielding the *latency stretch* of the solution. Figure 7.8 depicts the latency stretches measured in the mixed simulation scenarios described above. Again, some variance exists for smaller network sizes. However, assuming that latency stretch correlates with locality of communication, it can be clearly seen that direct publish modification improves the locality characteristics of the solution by nearly an order of magnitude.

7.4.2.3 Social Table

Turning to the addition of a social table to R/Kademlia, it does not introduce any visible improvements to the delivery ratios of the solution (except for small node populations, but there the variance of the results is too high to make definitive claims). In combination with the direct publish modification however, improvements to the



(a) Nodes have random underlay coordinates.

(b) Befriended nodes are close to each other in the underlay.

Figure 7.8: Stretch between the latency of sending a publication via SODESSON and the delay when transmitting it directly through the underlay (between sender and receiver). The measurements were taken in mixed network setups as described in Figure 7.6 and Figure 7.7.

communication latency of the solution are induced when the active discovery of befriended devices is used. When modeling the geographical proximity of socially connected nodes, this configuration results in a close to 50% lower latency stretch in comparison to the direct publish approach without social additions (Figure 7.8b). This is likely due to the fact that nodes learn about their future communication partners sooner during their lifetime, allowing them to form one-hop overlay connections with them from early on. Also, more befriended nodes are discovered by active probing than through passive discovery. This leads to more redundancy when choosing next hops during overlay routing procedures and thus, since all befriended nodes are also close to each other in the underlying network, increases the locality of the overlay routing as well.

All other configurations including the use of a social table do not show any changes in the measured latencies. However, the knowledge about socially connected nodes has benefits beyond the improvement of publish/subscribe performance that are not reflected in the presented diagrams. Potential uses for the additional social information have been discussed extensively in this thesis, most prominently in Section 5.6.4. In the scope of this chapter, it is thus more important to assess the cost of realizing such an integration of social information into the KBR layer. Since no negative impact of the social mechanisms on the locality or the delivery ratios of the solution could be noted, they remain to be evaluated only in terms of induced bandwidth. Such an evaluation will be conducted in the following Section 7.5.

7.4.3 Conclusions

Based on the studies conducted in this section, the performance of the system appears to remain stable with growing node populations. Furthermore, the direct publish mechanism was shown once more to induce a significant improvement in terms of latency and publication delivery rates in comparison to the standard i3 method of message dissemination. This is especially true in cases where befriended nodes (and thus - communication partners) are also close to each other in the underlying

network. The social table additions introduce slight improvements to communication latencies as well. The main advantage of the social table, however, lies not in its effect on publish/subscribe performance but in its potential for powering novel, socially-aware mechanisms for the SODESSON middleware.

7.5 Scalability and Bandwidth Consumption

The last series of simulations performed for this chapter aim at assessing the scalability of the solution with its most promising configurations, focusing mainly on the bandwidth consumption of the design.

7.5.1 Simulation Setup

Fully active node populations were used, i.e. all nodes were associated with a user entity and ran an instance of the SODESSON test application. Based on the results from Section 7.3 and Section 7.4, only configurations with direct publish were considered here. Parameter combinations without an activated social table were omitted from the simulations as well, as the existence of a social table was considered highly desirable in the context of the SODESSON middleware (amongst other things, the existence of a social table could enable the development of novel mechanisms based on social trust). Both the active and passive approaches for discovering befriended nodes were evaluated in the simulations.

As an experiment, additional runs were also made where *proximity neighbor selection* (PNS) was turned off in the overlay routing component. PNS is a mechanism used in some KBR protocols to prefer the addition of proximate nodes when filling the routing table, thus promoting a greater locality of communication[1]. The extra runs were motivated by the suspicion that a large part of the current prototype's bandwidth consumption is due to ping traffic induced by the PNS implementation in R/Kademlia.

7.5.2 Results

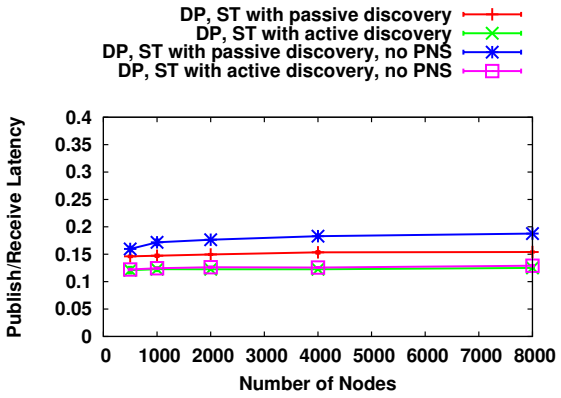
Results from the discussed simulations are shown in Figure 7.9. Concerning latency and delivery ratios (Figures 7.9a and 7.9b), the results are, as expected, mostly identical to the results from Section 7.4 (Figures 7.6a and 7.6b). Again, and despite the fact that all nodes are actively producing overlay traffic, the system's performance in these aspects remains nearly constant with growing node populations.

7.5.2.1 Effects of PNS

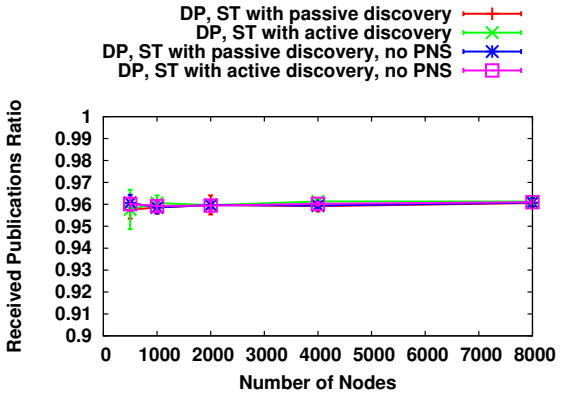
Concerning the latency of the solution, a slight worsening can be noticed when PNS is deactivated while the passive approach for discovering befriended nodes is used. No such change can be seen when the active discovery of befriended nodes is configured. PNS is an important topology adaptation mechanism in R/Kademlia. Without PNS, only *proximity routing* (PR) is left as a means for improving the locality characteristics of the overlay routing[2]. PR, however, can only optimize overlay routing decisions based on the nodes that are already included in the routing

[1] A more thorough explanation of PNS can be found in Section 4.3.

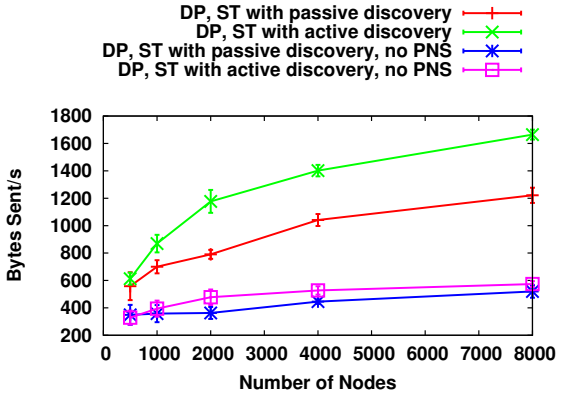
[2] See Section 4.3 for detailed description of the PR mechanism.



(a) Latency for undelayed publications.

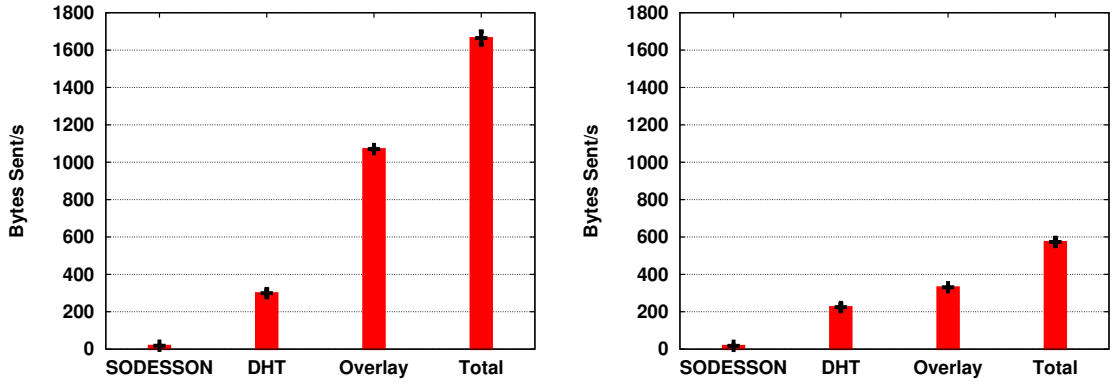


(b) Average success ratio for all publications.



(c) Bandwidth consumption.

Figure 7.9: Simulation results for network setups consisting only of active nodes. Moderate churn. *DP* denotes whether the direct publish modifications are active. *ST* signifies the usage of a social table in R/Kademlia, with *active* and *passive* representing the two approaches for discovering befriended nodes. *PNS* referrers to the activation of R/Kademlia's PNS mechanism.



(a) With PNS in R/Kademlia.

(b) Without PNS in R/Kademlia.

Figure 7.10: Bandwidth distribution in a simulation with 8000 active nodes. Direct publish is turned on and a social table with active friend device discovery is being used. The *Overlay* category includes ping and overlay maintenance traffic as well as the overhead induced by the sending of application messages over the overlay (e.g. due to extra packet headers).

table, thus being unable to fully compensate for the lack of PNS (hence the higher latencies when using passive friend device discovery). When using a social table with active friend device discovery however, all potential communication partners of a node become known to it soon after it joins the overlay. After they are placed in the social table of the node, one hop overlay routes can be used for pushing publications to them (when using the direct publish modifications). In this way, no entries from regular buckets are used for routing notification messages between publishers and subscribers and PNS seizes to have a significant effect.

Turning to the evaluation of the solution's bandwidth consumption, the measured values are relatively high for configurations with a standard R/Kademlia setup using PNS (Figure 7.9c). The additional runs with deactivated PNS demonstrate that a non-negligible amount of this overhead is due to signaling and ping traffic induced by the PNS mechanism. A comparison between the additional Figures 7.10a and 7.10b, that show the distribution of communication traffic between the different system components, confirms this observation. There, the bandwidth consumption generated by the overlay component sinks by more than 60% when PNS is turned off.

To summarize, the deactivation of PNS in configurations with active friend device discovery leads to a significantly lower bandwidth overhead while causing no significant change in communication latencies for configurations with active friend device discovery. Thus, its deactivation seems appropriate in the context of the SODESSON prototype and the presented simulation scenario. However, turning PNS off entirely is not the only method for avoiding the large signaling overhead it induces. Most interestingly, a network coordinate system like Vivaldi [15] could be employed for avoiding the constant pinging of nodes. Due to time constraints however, this was not considered in this thesis. In the following, only configurations where PNS is deactivated will thus be discussed.

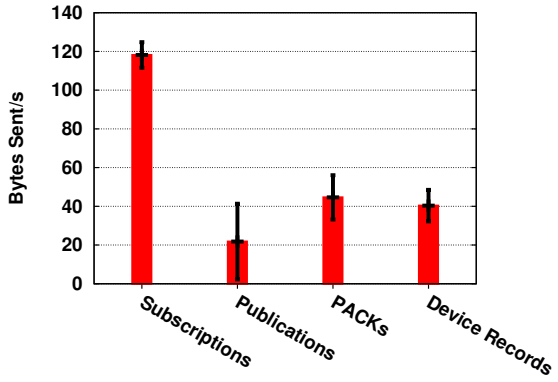


Figure 7.11: Amount of DHT traffic generated for different SODESSON data types being stored, in a simulation with 8000 active nodes. Direct publish is turned on and a social table with active friend device discovery is being used. PNS in R/Kademlia is turned off. For comparison, an average of 8.59 subscribers was measured per topic, for a total of 11963.4 existing topics on average.

7.5.2.2 Impact of Discovery Mechanisms for Befriended Nodes

For node populations of 8000 nodes and configurations without PNS, the bandwidth consumption of an individual node stays below 600 bytes per second (Figure 7.9c). The measured values are nearly identical for both discovery mechanisms for befriended nodes. Since the passive discovery mechanism leads to worse latencies (Figure 7.9a) and since the knowledge about a greater portion of a node's friend population might be desirable for future applications of the social table, configurations using passive friend device discovery will be omitted as well from the remaining discussions in this section. Thus, only one configuration will be evaluated from now on, namely the combination of the active node discovery mechanism with the deactivation of PNS. According to the simulation results, this parameter combination maximizes the performance of the SODESSON prototype in the chosen scenario.

7.5.2.3 Bandwidth Consumption for Large Node Populations

Focusing on the bandwidth consumption for the proposed configuration, the measured values appear to increase logarithmically with the growth of the node population, especially when considering network sizes of 2000 nodes and more (Figure 7.9c). Extrapolating based on this assumption, a traffic consumption in the range of about 1.1 kilobytes per second can be estimated for a network with 10^6 nodes. For a node population of 10^9 nodes, the value is in the range of 1.9 kilobytes per second. These are highly satisfying results. A bandwidth consumption in these ranges implies that an average of up to 3.6-7 megabytes of data will be sent per hour by nodes in a very large network. Additionally, approaches are possible for specifically alleviating devices for which bandwidth might be a limiting factor. For example, an alternative KBR approach could be used that adapts to the available amount of bandwidth (e.g. [9, 28]), or a client-only mode could be enabled for bandwidth-restricted nodes, similar to the proposal in [51].

7.5.2.4 Analysis of Generated Traffic

Moving to an analysis of the sources of the generated traffic, the aforementioned Figure 7.10 shows the distribution of bandwidth consumption between different sys-

tem components. It becomes clear that the SODESSON modules itself produces a negligible amount of traffic in comparison to the DHT and overlay routing components. The overhead produced at the overlay layer, stemming e.g. from ping and maintenance traffic and from additional message headers added when routing data through the overlay, forms more than 50% of the bandwidth consumption of the whole system (Figure 7.10b). Figure 7.11 presents an additional analysis of the traffic generated by the DHT component. It shows the bandwidth consumption induced at the DHT module for the storage and retrieval of individual SODESSON data types. Interestingly, the most significant part (about 50%) of the generated DHT traffic is due to the maintenance and retrieval of SODESSON subscription records. From the presented results, it can also be seen that the PACK mechanism introduced in this thesis does not induce a particularly high bandwidth consumption. Neither does the technique of storing the overlay IDs of devices in the DHT (for enabling the active discovery of friend devices).

7.5.3 Conclusion

The solution was, again, shown to scale well in terms of delivery ratios and communication latencies. In terms of bandwidth, a large overhead was induced by signaling and ping traffic associated with R/Kademlia's PNS mechanism. Also, the active mechanism for discovering befriended nodes was determined to be superior, as the latency improvements it induces did not appear to come at the cost of a significantly higher traffic overhead. By turning off PNS and using the active method for discovering befriended devices, bandwidth consumption drops to a level of about 600 bytes per second per node for a node population of 8000 nodes and raises only slowly with increasing network sizes, following a logarithmic curve. Lastly, the contributions of different system components to the generated traffic was evaluated as well, with the result that actual SODESSON messages are responsible for a negligible part of the overall bandwidth consumption in comparison to the DHT component and the overhead produced at the overlay routing layer.

7.6 Summary

The results of four sets of simulation studies were discussed in this chapter. In the first set, suitable parameter combinations for the DHT component were determined using OverSim's DHT test application. As a result of this study, a configuration with 3 replica locations per dataset, of which only one is queried during get requests, was determined to be best for the use with the SODESSON prototype. Using this DHT configuration, the basic feasibility of the proposed concepts was then demonstrated in an exemplary test scenario with 1000 nodes. Two subsequent studies were performed, assessing the scalability of the approach and the effects of the various modifications proposed for it.

As one of the results of this evaluation, the direct publish mechanism was shown to introduce significant improvements to both the locality properties and the reliability of the system. A configuration, where the direct publish mechanism is employed in combination with a social table with active friend device discovery and where R/Kademlia's PNS mechanism for topology adaptation is deactivated, was shown to produce the best results in the chosen simulation scenarios. For this parameter

combination, highly satisfying latencies and a moderate bandwidth consumption were measured that were noted to increase only slowly with growing node populations, hinting at the good scalability of the solution.

8. Summary and Further Directions

In recent years, mobile devices like smartphones and notebook computers have become increasingly more popular and widespread. At the same time, their computational and communicational capabilities have developed significantly. However, established communication paradigms do not leverage these newly available capacities and continue to organize distributed applications in ways that concentrate responsibilities in a handful of centrally controlled servers. Amongst other things, this raises strong privacy concerns that are often in the center of public attention.

With this problematic in mind, the *SODESSON* project was brought to life. SODESSON stands for *Service-oriented and decentralized social networks*. The main aim of the project is the development of a generic communication middleware that allows the deployment of secure user-centric applications. The term *user-centric networking* is used in this context, describing communication approaches based around the addressing of users and the direct service provision by user devices.

This thesis deals with the development of concepts for a user-centric networking middleware as envisioned by the SODESSON project. A special focus was placed on the locality of inter-user communication and the integration of information from the social graph.

8.1 Results of the Thesis

Firstly, a categorization of possible approaches for realizing a user-centric communication middleware was performed. Based on the discussion of the presented classes and the analytical comparison between them, a design based on structured overlay networks and distributed hash tables (DHTs) was found to be most suitable. A second, more deeper analysis of this approach was then performed, discussing specific mechanisms for realizing the required communication and data storage functionality. Following that, an actual proposal was designed based on those discussions. Specifically, the developed solution was based on the following components: the *R/Kademlia* key-based routing (KBR) protocol, a standard DHT with sibling replication and a

publish/subscribe-based communication system inspired by the *Internet indirection infrastructure* (*i3*). For the publish/subscribe system, a mechanism was developed to allow the delay-tolerant delivery of messages without requiring the sender and receiver to be available at the same time. Modifications to the base design were additionally devised for improving the locality of publish/subscribe communication. Lastly, a novel social table was also introduced to R/Kademlia, that stores socially connected nodes as overlay neighbors and thus serves as an integration point for social information. Mechanisms for discovering befriended devices and filling the social table were proposed as well.

The developed concepts were implemented in the overlay simulation framework *OverSim*. In addition to the introduction of an actual SODESSON prototype, the framework was also enhanced by components enabling the simulation of user-centric networking scenarios. User entities were introduced, as well as a mechanism for generating a social network between them. A user-centric test application was developed as well, offering a realistic emulation of instant messaging behavior and traffic.

Various simulation studies were finally performed using the implemented components. The aim was to evaluate the functionality and performance of the proposed designs. Based on the results from those studies, the solution was found to meet expectations and perform well in networks with thousands of nodes. Using the developed locality optimizations to the publish/subscribe system, the communication latency of the solution decreased by more than 40%, reaching an average of 125 milliseconds for the best parameter combination. For this parameter combination, a stretch of 1.5 was measured between the communication latency of the developed system and the hypothetical latency for sending messages directly over the underlying network. This result is taken as an indicator for the good locality properties of the developed system. The inclusion of a social table was found to introduce only small performance improvements, but was noted to be an important prerequisite for the development of mechanisms based on social information.

The delivery ratios and communication latencies of the solution were found to remain stable with growing node populations, hinting at the excellent scalability properties of the approach. Concerning the amount of required bandwidth, a logarithmic increase was noted in dependence of the simulated number of nodes. Through extrapolation based on this assumption, a traffic consumption of around 1.9 kilobytes per second was estimated for the evaluated scenario in a network with 10^9 nodes. In addition to the evaluation of the system's performance, opportunities for additions and improvements to the presented designs were noted and discussed.

8.2 Ideas for Future Work

This thesis can be seen as a first step towards the goal of developing an efficient and secure middleware for user-centric networking (as envisioned by the SODESSON project). Thus, a myriad of possibilities for future research efforts exist. For one, aspects of the implementation could be improved and tweaked. For example, the use of a network coordinate system for eliminating R/Kademlia's large PNS overhead could be attempted. Also, a better interwork between the DHT component and the developed prototype could be realized, to avoid the continuous polling for datasets

and offer nodes a better view on the topics they are responsible for. As the security aspects of the solution were explicitly excluded from consideration, they remain an important open question for future work. Most importantly, alternative approaches to the one-responsible-node-per-topic setup might need to be devised. It is especially interesting to asses the effects of such security-enhancing modifications on the system's overall performance.

Finally, the introduction of a social table and the discovery of befriended nodes enable the realization of various socially-powered mechanisms that need to be researched further. Socially connected devices could be used for storing sensitive data, for example, or datasets that are too large to be efficiently stored in the DHT. Social information can be used to improve the resilience of overlay routing. Lastly, social flooding techniques can also be attempted using nodes from the social table, potentially enabling the realization of powerful dynamic service discovery queries without causing the bad scalability effects usually associated with unstructured overlay techniques.

Bibliography

[1] Artigas, Marc Sánchez, Pedro García López, Jordi Pujol Ahulló and Antonio F. Gómez Skarmeta: *Cyclone: a novel design schema for hierarchical DHTs.* In *Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on*, pages 49 – 56, aug.-2 sept. 2005.

[2] Artigas, Marc Sánchez, Pedro García López and Antonio F. Gómez Skarmeta: *A Comparative Study of Hierarchical DHT Systems.* In *Local Computer Networks, 2007. LCN 2007. 32nd IEEE Conference on*, pages 325 –333, oct. 2007.

[3] Avrahami, Daniel and Scott E. Hudson: *Communication characteristics of instant messaging: effects and predictions of interpersonal relationships.* In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 505–514, New York, NY, USA, 2006. ACM.

[4] Baldoni, Roberto, Leonardo Querzoni and Antonino Virgillito: *Distributed Event Routing in Publish/Subscribe Communication Systems: a Survey.* Technical Report, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienzia", 2005.

[5] Bauer, Daniel, Paul Hurley and Marcel Waldvogel: *Replica Placement and Location using Distributed Hash Tables.* In *Local Computer Networks, 2007. LCN 2007. 32nd IEEE Conference on*, pages 315 –324, oct. 2007.

[6] Baumgart, Ingmar and Fabian Hartmann: *Towards secure user-centric networking: Service-oriented and decentralized social networks.* In *Proc. First International Workshop on Socio-Aware Networked Computing Systems at 5th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SaSo)*, October 2011.

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

[8] Baumgart, Ingmar and Sebastian Mies: *S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing.* In *Proceedings of the 13th International Conference on Parallel and Distributed Systems (ICPADS '07)*, December 2007.

[9] Brown, Alan, Mario Kolberg and John Buford: *Chameleon: An Adaptable 2-Tier Variable Hop Overlay.* In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1 –6, jan. 2009.

[10] Buchegger, Sonja, Doris Schiöberg, Le-Hung Vu and Anwitaman Datta: *PeerSoN: P2P social networking: early experiences and insights.* In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, SNS '09, pages 46–52, New York, NY, USA, 2009. ACM.

[11] Castro, Miguel, Manuel Costa and Antony Rowstron: *Debunking some myths about structured and unstructured overlays.* In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, May 2005.

[12] Castro, Miguel, Peter Druschel, Anne-Marie Kermarrec and Antony Rowstron: *Scribe: a large-scale and decentralized application-level multicast infrastructure.* Selected Areas in Communications, IEEE Journal on, 20(8):1489 – 1499, oct 2002.

[13] claffy, k.: *CAIDA: Visualizing the Internet.* Internet Computing Online, page 88, Jan 2001.

[14] Cutillo, Leucio Antonio, Refik Molva and Thorsten Strufe: *Safebook: A privacy-preserving online social network leveraging on real-life trust.* Communications Magazine, IEEE, 47(12):94 –101, dec. 2009.

[15] Dabek, Frank, Russ Cox, Frans Kaashoek and Robert Morris: *Vivaldi: a decentralized network coordinate system.* In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '04, pages 15–26, New York, NY, USA, 2004. ACM.

[16] Dabek, Frank, Ben Zhao, Peter Druschel, John Kubiatowicz and Ion Stoica: *Towards a Common API for Structured Peer-to-Peer Overlays.* In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, February 2003.

[17] Danezis, DGeorge and Prateek Mittal: *Sybilinfer: Detecting sybil nodes using social networks*, volume 9. NDSS, 2009.

[18] Ford, Bryan Alexander: *UIA: A Global Connectivity Architecture for Mobile Personal Devices.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2008. Adviser-Kaashoek, M. Frans.

[19] Fotiou, Nikos, Konstantinos Katsaros, Xenofon Vasilakos, Christos Tsilopoulos, Christopher N. Ververidis, George Xylomenos and George C. Polyzos: *H-Pastry: An Adaptive Multi-level Overlay Inter-Network.* Technical Report, Mobile Multimedia Laboratory / Department of Informatics / Athens University of Economics and Busines, October 2011.

[20] Freedman, Michael and David Maziéres: *Sloppy Hashing and Self-Organizing Clusters.* In Kaashoek, M. and Ion Stoica (editors): *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 45–55. Springer Berlin / Heidelberg, 2003.

[21] Ganesan, Prasanna, Krishna Gummadi and H. Garcia-Molina: *Canon in G major: designing DHTs with hierarchical structure.* In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 263 – 272, 2004.

[22] Heep, Bernhard: *R/Kademlia: Recursive and topology-aware overlay routing.* In *Telecommunication Networks and Applications Conference (ATNAC), 2010 Australasian*, pages 102 –107, 31 2010-nov. 3 2010.

[23] Heer, Tobias, Stefan Götz, Simon Rieche and Klaus Wehrle: *Adapting distributed hash tables for mobile ad hoc networks.* In *Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on*, pages 6 pp. –178, march 2006.

[24] Huffaker, B., D. Plummer, D. Moore and k. claffy: *Topology discovery by active probing.* In *Symposium on Applications and the Internet (SAINT)*, pages 90–96, Nara, Japan, Jan 2002. SAINT.

[25] Isaacs, Ellen, Alan Walendowski, Steve Whittaker, Diane J. Schiano and Candace Kamm: *The character, functions, and styles of instant messaging in the workplace.* In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, CSCW '02, pages 11–20, New York, NY, USA, 2002. ACM.

[26] Lesniewski-Laas, Chris and M. Frans Kaashoek: *Whanau: a sybil-proof distributed hash table.* In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[27] Li, Jinyang and Frank Dabek: *F2F: reliable storage in open networks.* In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, February 2006.

[28] Li, Jinyang, Jeremy Stribling, Robert Morris and M. Frans Kaashoek: *Bandwidth-efficient management of DHT routing tables.* In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 99–114, Berkeley, CA, USA, 2005. USENIX Association.

[29] Lv, Qin, Sylvia Ratnasamy and Scott Shenker: *Can Heterogeneity Make Gnutella Scalable?* In Druschel, Peter, Frans Kaashoek and Antony Rowstron (editors): *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 94–103. Springer Berlin / Heidelberg, 2002.

[30] Lynch, Nancy, Dahlia Malkhi and David Ratajczak: *Atomic data access in distributed hash tables.* In *In Proceedings of the International Peer-to-Peer Symposium*, pages 295–305. Springer-Verlag, 2002.

[31] Marc Thylmann, Bernd Klusmann: *Jeder Dritte hat ein Smartphone.* BITKOM, April 2012.

[32] Martinez-Yelmo, Isaias, Ruben Cuevas, Carmen Guerrero and Andreas Mauthe: *Routing Performance in a Hierarchical DHT-based Overlay Network.* In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 508 –515, feb. 2008.

[33] Matuszewski, Marcin and Miguel A. Garcia-Martin: *Social Distributed Hash Table.* In *Wireless Communications and Networking Conference, 2007.WCNC 2007. IEEE*, pages 2812 –2818, march 2007.

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

[35] Mies, Sebastian and Oliver P. Waldhorst: *Autonomous detection of connectivity.* In *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, pages 44 –53, 31 2011-sept. 2 2011.

[36] Popescu, Bogdan C., Bruno Crispo and Andrew S. Tanenbaum: *Safe and Private Data Sharing with Turtle: Friends Team-Up and Beat the System.* In *Security Protocols Workshop*, pages 213–220, 2004.

[37] Qiao, Yi and Fabián E. Bustamante: *Structured and unstructured overlays under the microscope: a measurement-based view of two P2P systems that people use.* In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 31–31, Berkeley, CA, USA, 2006. USENIX Association.

[38] Ramasubramanian, Venugopalan and Emin Gün Sirer: *Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays.* In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[39] Rhea, Sean, Dennis Geels, Timothy Roscoe and John Kubiatowicz: *Handling churn in a DHT.* In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[40] Rowstron, Antony and Peter Druschel: *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems.* In Guerraoui, Rachid (editor): *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer Berlin / Heidelberg, 2001.

[41] Sanchez-Artigas, Marc, Pedro Garcia-Lopez and Antonio G. Skarmeta: *On the Relationship between Caching and Routing in DHTs.* In *Proceedings of the 2007 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Workshops*, WI-IATW '07, pages 415–418, Washington, DC, USA, 2007. IEEE Computer Society.

[42] Sergio Marti, Prasanna Ganesan and Hector Garcia-Molina: *SPROUT: P2P Routing with Social Networks.* Technical Report, Stanford University, 2004.

[43] Steiner, Moritz, Taoufik En Najjary and Ernst W Biersack: *Long term study of peer behavior in the KAD DHT.* IEEE/ACM Transactions on Networking, 17(5), 12 2009.

[44] Stoica, Ion, Daniel Adkins, Shelley Zhuang, Scott Shenker and Sonesh Surana: *Internet Indirection Infrastructure.* In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '02, pages 73–86, New York, NY, USA, 2002. ACM.

[45] Stutzbach, Daniel and Reza Rejaie: *Understanding churn in peer-to-peer networks.* In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 189–202, New York, NY, USA, 2006. ACM.

[46] Tran, Dinh Nguyen, Frank Chiang and Jinyang Li: *Friendstore: cooperative online backup using trusted nodes.* In *Proceedings of the 1st Workshop on Social Network Systems*, SocialNets '08, pages 37–42, New York, NY, USA, 2008. ACM.

[47] Ugander, Johan, Brian Karrer, Lars Backstrom and Cameron Marlow: *The Anatomy of the Facebook Social Graph.* CoRR, abs/1111.4503, 2011.

[48] Varga, András and Rudolf Hornig: *An overview of the OMNeT++ simulation environment.* In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[49] Watts, Duncan J. and Steven H. Strogatz: *Collective dynamics of 'small-world' networks.* Nature, 393:440–442, June 1998.

[50] Xu, Zhiyong, Rui Min and Yiming Hu: *HIERAS: a DHT based hierarchical P2P routing algorithm.* In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 187 –194, oct. 2003.

[51] Zoels, Stefan, Simon Schubert, Wolfgang Kellerer and Zoran Despotovic: *Hybrid DHT Design for Mobile Environments.* In Joseph, Sam, Zoran Despotovic, Gianluca Moro and Sonia Bergamaschi (editors): *Agents and Peer-to-Peer Computing*, volume 4461 of *Lecture Notes in Computer Science*, pages 19–30. Springer Berlin / Heidelberg, 2008.