Doctoral Dissertation

Service Placement in Ad Hoc Networks

Dissertation zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) am Fachbereich Mathematik und Informatik der Freien Universität Berlin

vorgelegt von

Georg Wittenburg, M.Sc.

Datum der Disputation: 27. September 2010

Gutachter: Prof. Dr.-Ing. habil. Jochen H. Schiller
Prof. Dr.-Ing. habil. Andreas Mitschele-Thiel

Abstract

Service provisioning in ad hoc networks is challenging given the difficulties of communicating over a wireless channel and the potential heterogeneity and mobility of the devices that form the network. In order to optimize the performance of the network over which a service host provides a service to client nodes, it is necessary to continuously adapt the logical network topology to both external (e.g., wireless connectivity, mobility, churn) and internal (e.g., communication patterns, service demand) factors. Recent proposals advocate that nodes should dynamically choose which nodes in the network are to provide application-level services to other nodes. Services in this context range from infrastructural services such as the Domain Name System (DNS) to user-oriented services such as the World Wide Web (WWW).

Service placement is the process of selecting an optimal set of nodes to host the implementation of a service in light of a given service demand and network topology. The main questions addressed by service placement are: *How many* instances of the same service should be available in the network and cooperate to process clients' service requests; *where* these service instances should be placed, i.e., which nodes are best suited for hosting them; and *when* to adapt the current service configuration. The service instances of a distributively operating service are exact copies of the software component that provides the service, including both the executable binary and the application-level data. The set of nodes that host a service instance is referred to as the service configuration. A good service configuration increases the performance of a service according to application-specific quality metrics, while at the same time potentially reducing the overall network load. The key advantage of active service placement in ad hoc networks is that it allows for the service configuration to be adapted continuously at run time.

In this work, we propose the SP*i* service placement framework as a novel approach to service placement in ad hoc networks. The SP*i* framework takes advantage of the interdependencies between service placement, service discovery and the routing of service requests to minimize signaling overhead. We also propose the Graph Cost / Single Instance (GCSI) and the Graph Cost / Multiple Instances (GCMI) placement algorithms. The SP*i* framework employs these algorithms to optimize the number and the location of service instances based on usage statistics and a partial network topology derived from routing information. The GCSI and

GCMI placement algorithms only require minimal knowledge about the service they are tasked with placing in the network. They are novel in that they take the communication between service instances into account which is required to synchronize the global state of the service. Furthermore, when calculating the optimal timing of their placement decisions, the two algorithms explicitly consider the overhead of the actions required for implementing changes to the current service configuration.

Our implementation of the SP*i* framework on top of a special low-level API allows us to run the framework on a variety of evaluation platforms including major operating systems and network simulation tools. We examine the properties of our approach to service placement and compare it with other recent proposals in simulations, emulations, and real-world experiments on an IEEE 802.11 wireless testbed. The results of this evaluation show that the SP*i* service placement framework and our placement algorithms, in particular GCMI, are able to find service configurations that are superior across a variety of scenarios to those found by other approaches. As a consequence, service provisioning improves significantly with regard to its reliability and timeliness, while at the same time causing less network traffic. Furthermore, our results show that distributed service provisioning with active service placement – as implemented in SP*i* – generally outperforms services that are implemented in a traditional client/server architecture. From these results we conclude that our approach to service provisioning in ad hoc networks is a viable alternative to established architectures.

Zusammenfassung

Die Diensterbringung in Ad-hoc-Netzen stellt eine Herausforderung dar, weil in diesen Netzen die drahtlose Kommunikation per Funk sowie möglicherweise auch die Heterogenität und Mobilität der am Netz teilnehmenden Geräte neue Fragestellungen aufwerfen. Um die Leistungsfähigkeit des Netzes, über das die dienstanbietenden mit den dienstnehmenden Knoten kommunizieren, zu optimieren, gilt es, die logische Netztopologie kontinuierlich an sowohl externe als auch interne Faktoren anzupassen. Als Beispiele für diese Faktoren sind die Konnektivität, die Mobilität, und die Abwanderungsrate der Knoten einerseits, beziehungsweise das Kommunikationsverhalten und das Dienstanfragevolumen andererseits zu nennen. In letzter Zeit wurde vermehrt vorgeschlagen, jene Knoten dynamisch auszuwählen, die im Netz Dienste auf Anwendungsebene für die anderen Knoten erbringen sollen. Der Dienstbegriff umfasst in diesem Zusammenhang sowohl infrastrukturellen Dienste wie beispielsweise das Domain Name System (DNS) als auch benutzerorientierte Dienste wie das World Wide Web (WWW).

Die Auswahl einer optimalen Menge von dienstanbietenden Knoten angesichts der derzeitigen Nachfrage nach dem Dienst und der aktuellen Netztopologie wird *Dienstplatzierung* (engl. „service placement“) genannt. Dienstplatzierung in Ad-hoc-Netzen umfasst folgende Kernfragen: *Wie viele* Instanzen eines Dienstes sollten im Netz zur Verfügung stehen, um gemeinsam die Anfragen der dienstnehmenden Knoten abzuarbeiten; *wo* sollten diese Dienstinstanzen platziert werden, d.h. welche Knoten sind für die verteilte Diensterbringung am besten geeignet; und *wann* sollte eine Dienstkonfiguration angepasst werden. Hierbei sind die *Dienstinstanzen* eines verteilt arbeitenden Dienstes exakte Kopien der Softwarekomponente, die den Dienst erbringt — einschließlich des ausführbaren Programms und der Anwendungsdaten. Die Menge der Knoten, die jeweils eine Dienstinstanz beherbergen, wird *Dienstkonfiguration* genannt. Eine gute Dienstkonfiguration verbessert die Qualität der Diensterbringung gemäß anwendungsspezifischer Metriken bei gleichzeitiger Verminderung der Netzlast. Der wesentliche Vorteil einer aktiven Dienstplatzierung in Ad-hoc-Netzen ist die Möglichkeit zur kontinuierlichen Anpassung der Dienstkonfiguration zur Laufzeit.

In dieser Arbeit beschreiben wir das SP*i* Service Placement Framework, einen neuartigen Ansatz zur Dienstplatzierung in Ad-hoc-Netzen. Das SP*i* Framework nutzt die wechselseitigen Abhängigkeiten zwischen Dienstplatzierung, Dienstauffindung und dem Routing von

Dienstanfragen aus, um den erforderlichen Signalisierungsverkehr zwischen den Knoten zu minimieren. Wir beschreiben außerdem die Graph Cost / Single Instance (GCSI) und Graph Cost / Multiple Instances (GCMI) Dienstplatzierungsalgorithmen. Das SP*i* Framework verwendet diese Algorithmen zur Optimierung der Anzahl und der Position von Dienstinstanzen basierend auf Nutzungsstatistiken und einer partiellen, aus Routingdaten gewonnenen Netztopologie. Die GCSI und GCMI Dienstplatzierungsalgorithmen benötigen hierfür nur minimale Kenntnisse über den zu platzierenden Dienst. Sie gehen über den aktuellen Stand der Technik hinaus, indem sie explizit die Kommunikation zwischen den Dienstinstanzen berücksichtigen, die erforderlich ist, um den globalen Zustand des Dienstes über Instanzen hinweg synchron zu halten. Darüber hinaus beziehen die beiden Algorithmen bei der Berechnung des optimalen Zeitpunktes zur Anpassung einer Dienstkonfiguration die zu erwartende Netzlast der Maßnahmen ein, die für diese Anpassung erforderlich sind.

Unsere Implementierung des SP*i* Frameworks auf einer eigens für diesen Zweck entwickelten Softwareschnittstelle ermöglicht es uns, das System mit Hilfe einer Vielzahl von Evaluationsplattformen zu bewerten — einschließlich gängiger Betriebssysteme und Netzsimulatoren. Wir untersuchen die Eigenschaften unseres Dienstplatzierungssystems und vergleichen es mit anderen Ansätzen in Simulationen, Emulationen und Experimenten auf einem drahtlosen IEEE 802.11 Testbed. Die Auswertung dieser Experimente zeigt, dass das SP*i* Service Placement Framework und seine Dienstplatzierungsalgorithmen – insbesondere GCMI – in der Lage sind, in einer Vielzahl von Szenarien bessere Dienstkonfigurationen als andere Ansätze zu finden. Daraus resultiert eine Verbesserung der Verlässlichkeit und des Laufzeitverhaltens des platzierten Dienstes, während gleichzeitig weniger Datenverkehr im Netz für dessen Erbringung erforderlich ist. Darüber hinaus zeigen unsere Experimente, dass eine verteilte Diensterbringung mit aktiver Dienstplatzierung – wie sie in SP*i* implementiert ist – die Leistungsfähigkeit eines in einer klassischen Client/Server-Architektur erbrachten Dienstes übertrifft. Aus diesen Ergebnissen schließen wir, dass unser Ansatz zur Diensterbringung in Ad-hoc-Netzen eine interessante Alternative zu etablierten Architekturen darstellt.

Acknowledgments

The work presented in this document was carried out during my employment as a research assistant at the Institute of Computer Science, Freie Universität Berlin, and thereafter with grants from the Center for International Cooperation (CIC), Freie Universität Berlin, and the German Academic Exchange Service (Deutscher Akademischer Austausch Dienst, DAAD) at the Laboratoire d'Informatique de l'École Polytechnique (LIX), Paris. I am indebted to a number of amazing people who supported me during this venture — not only for their help and professional guidance, but also for making it a mostly cheerful and certainly unforgettable experience.

First and foremost, I would like to thank Prof. Dr.-Ing. Jochen Schiller and Prof. Dr.-Ing. Andreas Mitschele-Thiel for agreeing to advise and support me during this research project. I have greatly enjoyed working with Prof. Schiller and, for that matter, with his entire working group during the past years. In particular, I would like to thank him for his trust, dedication, and encouragement, but also for the great freedom with which I was able to pursue this and other projects. To Prof. Mitschele-Thiel I am especially grateful for the fruitful discussions and his encouragement during the final stages of this research effort.

I owe greatly to Dr. Kirsten Dolfus (née Terfloth) and Dr. Thomas Zahn who helped me to get started in this field, and both in their own way taught me how to get most out of one's research while having fun at it. I apologize to them for every sentence that turned out too long, and for every paper that I submitted to a second-tier conference.

I am also very grateful to my co-workers and friends. To Prof. Dr. Mesut Güneş, Matthias Wählisch, Dr. Katinka Wolter, and Philipp Reinecke for the insightful discussions and feedback on my work; to Bastian Blywis for his support and patience while I was preparing and conducting the experiments; and to Prof. Dr. Annika Hinze and Dr. Emmanuel Baccelli for hosting me during my stays at the University of Waikato in Hamilton, New Zealand, and at the École Polytechnique. The time at the Computer Systems & Telematics group at Freie Universität Berlin would not have been half as productive and half as enjoyable without the encouragement and cheerful company of Dr. Ilias Michalarias, Dr. Freddy López Villafuerte, Dr. Katharina Hahn, Norman Dziengel, Dr.-Ing. Achim Liers, and Stefanie Bahe (archaeologist). Same goes for Veronika Maria Bauer, Ulrich Herberg, and Juan Antonio Cordero Fuertes at the LIX.

Fabian Stehn, Jessica Schlegel, Maria Knobelsdorf, Katja Silligmann, and Agata Naumowicz made sure that I was mindful of the best practices of various research communities while writing this text. Thankfully, they also made sure that I did not forget that there are other things in life.

Last, but nowhere near least, I would like to extend my special thanks to my parents who lovingly and with great interest for my work supported me throughout this project. I am deeply grateful to my family – *und besonders meiner Großmutter in Berlin* – for being there for me in all circumstances, and for offering help and advice in all matters, from trivial to huge.

Contents

Chapter 1

Introduction

Over the past decade, there has been a continued research interest in the field of *ad hoc networking*. Generally speaking, ad hoc networks enable a potentially wide range of devices to communicate with each other without having to rely on any kind of support or services provided by an IT infrastructure [88, 96]. Given the recent emergence of wireless-enabled, portable devices in the IT end user market [74], one can anticipate a vast potential for wide-spread deployment of ad hoc networking technology.

1.1 Service Placement in Ad Hoc Networks

When designing and building ad hoc networks, fundamental design decisions from traditional, Internet-style networks need to be reevaluated. This affects all aspects of networking, starting with the choices for signal modulation and ranging up to the special requirements on privacy in these networks. In this general context, *service placement* deals with the question of which nodes in an ad hoc network are best suited for offering any given kind of service to other nodes [110]. Types of services may range from infrastructural services such as the Domain Name System (DNS) [78, 79] to user-oriented services such as the World Wide Web (WWW) [51].

1.1.1 Motivation

In ad hoc networks, the classical distinction between devices acting as either client or server is replaced by the paradigm of cooperation between a large number of potentially heterogeneous devices. These devices may differ in which hardware resources they incorporate, ranging from resource-constrained embedded devices employed in Wireless Sensor Networks (WSNs) to desktop-style hardware used in office mesh networks. Given the characteristics of communication over a wireless channel and the potential mobility of the devices, the physical topology of an ad hoc network is in a constant state of flux. In order to ensure that the network is operational at all times, it is thus necessary to continuously adapt the logical

configuration of the network, e.g., neighborhood lists and routing paths, to the conditions in the physical world. This adaptation is driven by both external (e.g., wireless connectivity, mobility, churn) and internal (e.g., communication patterns) factors. In the past decade, the focus of research has been on optimizing the routing of packets between the nodes of the ad hoc network, thus resulting in a great variety of reactive, proactive, and hybrid routing protocols. Some of these protocols already have or are currently undergoing the process of standardization [17, 19, 54, 87].

Taking a more application-centric view on ad hoc networks, the distinction between clients and servers still exists as part of the logical network structure whenever certain nodes request services provided by other nodes. Therefore, the question arises whether the performance of an ad hoc network can be optimized by carefully choosing exactly which of the nodes are to take the role of servers for a particular service. Key factors to take into account when answering this question are the connectivity between individual nodes, the service demand and request patterns of client nodes, and the suitability of services for being migrated between nodes.

The question of identifying the appropriate nodes in an ad hoc network to act as servers is referred to as the *service placement problem* whose goal it is to establish an optimal *service configuration*, i.e., a selection of nodes to host the instances of the service which is optimal in regard to some service-specific metric.

1.1.2 Problem Statement

The problem of service placement in ad hoc networks can be stated as follows: Given a service to be placed, a physical network topology, and the service demand of the client nodes, adapt the number and location of service instances in the network over time such that the service demand is met at a minimal cost.

The cost function may include metrics such as network traffic, energy expenditure, or service-dependent quality metrics. The choice of the cost function is mandated by the *service placement policy*. A placement policy states the goal that a *service placement system* aims to achieve. Goals, and as a consequence placement policies, vary depending on the exact type and area of application of the ad hoc network. A very fundamental placement policy – and in fact the one that we will focus on in this work – is the overall reduction of bandwidth required for service provisioning. Other goals may be pursued in more specialized application scenarios. For example, in an office mesh, the placement policy may aim at a reduction of service access time. Alternatively, in a WSN deployed in a hazardous environment, a regionally diverse placement of service instances may be preferable.

In order to adapt to regional-specific service demand, it is advantageous if the service can be provided by multiple *service instances* each of which is hosted on a different node. As the term suggests, a service instance is an exact copy of the software component that provides the service, including both the executable binary and the application-level data. Each service instance is capable of providing the complete service on its own. In fact, there is no distinction between service instances except for which node of the ad hoc network they are hosted on.

This concept of service instances gives rise to a distinction between *centralized* and *distributed* services: For centralized services, it is technically infeasible to create multiple instances of the service and hence there is only a single service instance. In fact, this single service instance of a centralized service is identical with the *server* in a traditional client/server architecture. For distributed services, however, it is technically possible to create multiple instances of the service. These service instances jointly process the clients' service requests and can be placed independently. However, they incur an additional overhead not present for centralized services, which consists of the acts of communication that are required to keep the global state of the service synchronized among all instances.

1.1.3 Clarification of Terms

Before proceeding, we give brief definitions for the terms that we have introduced informally in the previous section.

Service In this work, we use the term "service" in the same meaning as in the context of Service Oriented Architectures (SOAs) [73, Sec. 3.1]:

> A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. A service is provided by an entity – the service provider – for use by others, but the eventual consumers of the service may not be known to the service provider and may demonstrate uses of the service beyond the scope originally conceived by the provider. [...]

A service is thus a software component executed on one or several nodes of an ad hoc network. It consists of both service-specific logic and state. A service is accessed by local or remote clients by the means of issuing service requests that, in case of remote clients, are transmitted across the network. The content and structure of service requests and replies are specific to the service. Several different services may be active in a network simultaneously, and each node may host multiple services.

Several attributes of a service are of special interest in the context of service placement:

- **Node-specific vs. node-independent services:** A service may or may not be tied to a specific node of the network. For example, a service that provides a sensor reading of a particular node in a WSN is *node-specific* because the value depends on the physical location of the sensor node. In contrast, a directory service in an ad hoc network or a cluster head of a hierarchical routing protocol are typical examples for services that are *node-independent*.

  Obviously, service placement is only applicable to node-independent services.

- **Centralized vs. distributed services:** The semantics or the implementation of a service may require for it to run *centralized* on one node. This is particularly applicable to services for which processing a service request implies a complex manipulation of the internal state, or, more trivially, to services for which the implementation does not support distributed service provisioning. In contrast, a *distributed* service can be provided in the form of multiple service components spread out over several nodes. Distributed services may require some form of synchronization between service components when the global state of the service is manipulated by service requests.

  Service placement is applicable to both centralized and distributed services. For centralized services, it only controls on which node should host the service; for distributed services, it also manages the granularity with which the service is to be split up, i.e., the number of service components that are to be distributed in the network.

- **Monolithic vs. composite services:** A service may be structured in such a way that it can be decomposed into multiple interdependent subservices each of which contributes a different aspect of the overall service. The software component for each subservice of a distributed, *composite* service can be placed independently and communicates with the other subservices over the network. In contrast, a *monolithic* service cannot be split into subservices, either due to semantics or implementation concerns. Hence, if a monolithic service is to be provided in a distributed manner, then all software components across all nodes are identical.

  Service placement is equally applicable to monolithic and composite services. For monolithic services, it can freely create any number of identical instances of the service component; for composite services, it needs to take the service-specific interdependencies between the subservices into account.

We consider both centralized and distributed services in this work. As we will see, the former can be considered as a special case of the latter. Furthermore, while composite

services are certainly architecturally interesting, the main focus of this work is on monolithic services. The placement of the components of a composite service depends largely on the interactions and the semantics between the subservices. Hence, it is difficult to evaluate a placement system in a manner that is sufficiently independent of the service to claim generality of the results. We may address this type of service once the placement of monolithic services is well understood.

Service Instance If identical service components of a distributed, monolithic service are running on several nodes, we refer to these components as service instances. Service instances may have to exchange information in order to keep the global state of the service synchronized. The volume of this synchronization traffic depends on the clients' service requests as well as on the requirements of the service with regard to the data consistency across its instances.

In case of a centralized, monolithic service, i.e., a service with one single instance, the terms "server" and "service instance" may be used interchangeably.

Network Topology The *physical* network topology refers to the connectivity between the nodes that form the ad hoc network, i.e., to the quality of the wireless links that allow them to exchange messages. Due to node mobility, churn, and changing properties of the wireless channel it is subject to continuous change. The *logical* network topology consists of the links and multi-hop routing paths actually used for communication among the nodes.

Both physical and logical network topology can be modeled as a graph with the nodes as vertices and the communication links as edges. The graph of the logical network topology is implicitly created through the distributed operation of the neighborhood discovery and routing protocols. It is this graph that the placement of the service instances needs to be adapted to.

Service Demand The service demand describes the scenario-dependent necessity of applications running on certain nodes to utilize a service. We refer to these nodes as "client nodes" or "clients". For each client, the service demand results in a stream of service requests over a period of time. Satisfying the service demand of all client nodes is the primary goal of the ad hoc network. Metrics that quantify the success of a service placement algorithm need to take into account in how far this goal has been reached.

Service Configuration The configuration of a service corresponds to the set of nodes that host the instances of this service. The service configuration is adapted continuously according to the placement policy as network topology and service demand change. These

adaptations reflect the fact that the *current* service configuration may differ from the *optimal* service configuration. In order to remedy this deficiency, new service instances may be created, and existing instances may be moved to other nodes or shut down.

Service Placement System A Service Placement System (SPS) is the set of software components that implements the functionality required to actively adapt the configuration of a service. This includes the tasks of measuring the quality with which the service is currently being provided to the clients, using this data to decide whether the service configuration needs to be adapted, and implementing any necessary changes. At the core of the SPS is the placement algorithm, which calculates the optimal service configuration, or an approximation thereof.

1.1.4 Assumptions

Employing service placement in an ad hoc network relies upon several assumptions about the context in which the ad hoc network is deployed and about the capabilities of the nodes. In detail, these assumptions are as follows:

- **Cooperation between nodes:** Service placement relies upon the assumption that the nodes are willing to cooperate with each other in order to achieve a common goal. The same assumption is also used in the core of MAC and routing protocols, and service placement applies it to the area of service provisioning.
- **Bounded heterogeneity of devices:** The heterogeneity of the devices with regard to their capabilities needs to be bounded, i.e., most, if not all, nodes in the network have to possess sufficient resources to host a service instance.

These assumptions are not uncommon in research into ad hoc networking. Therefore, we argue that they are reasonable and do not limit the validity of our results.

1.1.5 Potential Applications

Service placement as a fundamental technology is applicable to a wide variety of ad hoc networks. In its most basic form, which is also the focus of this work, it aims to reduce the bandwidth required for offering services to client nodes. This goal is achieved by balancing the bandwidth consumed by client requests against the bandwidth consumed for creating new service instances and keeping the global state of the service synchronized between them. This form of service placement is applicable across all types of ad hoc networks since communication bandwidth is a scarce resource for all of them.

More specialized forms of service placement with a different focus are more suitable in certain application scenarios. If a service is to be provided in an office mesh network, the quality of the service may be more important than the conservation of bandwidth. For instance, users may be more interested in high availability and low access times in this scenario. Therefore, a service placement system would create more service instances and place them in the close topological vicinity of the client nodes in this scenario.

Another area of application of service placement are wireless sensor networks that – depending on the deployment scenario – may be exposed to large numbers of node failures or intermittent communication problems. In order to increase the reliability of a service, a service placement system may aim to place the service instances in a regionally diverse manner. This would ensure that all nodes have a service instance in their topological neighborhood even if there is currently no demand for it. Furthermore, this approach would also increase the resilience of the service against large-scale regional node failures.

Service placement is equally applicable to ad hoc networks that emphasize node mobility, e.g., Mobile Ad hoc NETworks (MANETs) or Vehicular Ad hoc NETworks (VANETs). These networks are highly susceptible to the arrival and departure of nodes and to the changes to the physical topology of the network due to node movement. A service placement system may thus either aim to achieve an uniform distribution of service instances across a geographical area, or it may identify groups of nodes that exhibit similar mobility patterns and ensure that a service instance is present in each of these mobility groups.

The different placement policies for these areas of application are by no means mutually exclusive. In fact, scenarios that focus on multiple of the goals presented above may be the norm rather than the exception. For example, in an office mesh network, the placement policy may emphasize the quality of the service over the consumed bandwidth, but shift the priority to the latter in case the service has been met at a sufficient quality over a prolonged period of time. For these scenarios, it is also conceivable that a service placement system may interact with network components in charge of distributed resource allocation, e.g, the Resource ReSerVation Protocol (RSVP) [15].

1.2 Contributions

Our contribution towards enabling service placement in ad hoc networks is a service placement system for monolithic services, both in their centralized and distributed variants. The placement algorithms that we propose as part of this system implement a placement policy that aims to reduce the overall bandwidth usage of a service. We evaluate our approach by quantitative comparisons to other proposals from the literature in simulations, emulations, and real-world experiments.

1.2.1 Research Questions

As part of our contribution in the field of service placement in ad hoc networks, we focus on the following three key research questions:

1. ***Where* in the network are service instances to be placed?**
   Our service placement system aims to reduce the bandwidth required for service provisioning by placing the service instances in a way that shortens the routes between servers and clients as much as possible. This results in less contention on the wireless communication channel, less packet collisions, and thus ultimately in improved handling of service requests. Further, reducing the number of radio transmissions allows the nodes to save energy, thereby prolonging the lifetime of the network.

2. ***How many* service instances are required for optimal operation?**
   For distributed services, the optimal number of service instances strikes a balance between traffic between clients and service instances and synchronization traffic among service instances. Two extreme situations are conceivable: On one side, only one service instance may be present in the network, thus effectively making the service a centralized one. On the other side, each node in the network may host its own service instance. In the first case, a high volume of traffic is caused by the communication between clients and the service instance, but no synchronization of shared data between service instances is required. In the second case, no network traffic is required for communicating between clients and service instances, but network traffic may be caused by the service instances having to exchange data among them. For most services, there is an optimum between these two extremes. Hence, client-level traffic needs to be balanced against synchronization traffic in order to find an optimal number of service instances.

3. ***When* should the current configuration of a service be adapted?**
   As the network topology or the volume of service requests from different clients change over time, the current service configuration needs to be adapted to the new situation. However, migrating services or creating new service instances incurs additional network traffic. For this reason, the exact circumstances under which an adaptation actually makes sense are not trivial if the goal is the overall reduction of network traffic. For instance, a change in network topology may only be transient and not warrant starting a costly adaptation process. Hence, the timing of placement decisions must weight the expected reduction in bandwidth usage against the overhead required for implementing the adaptation.

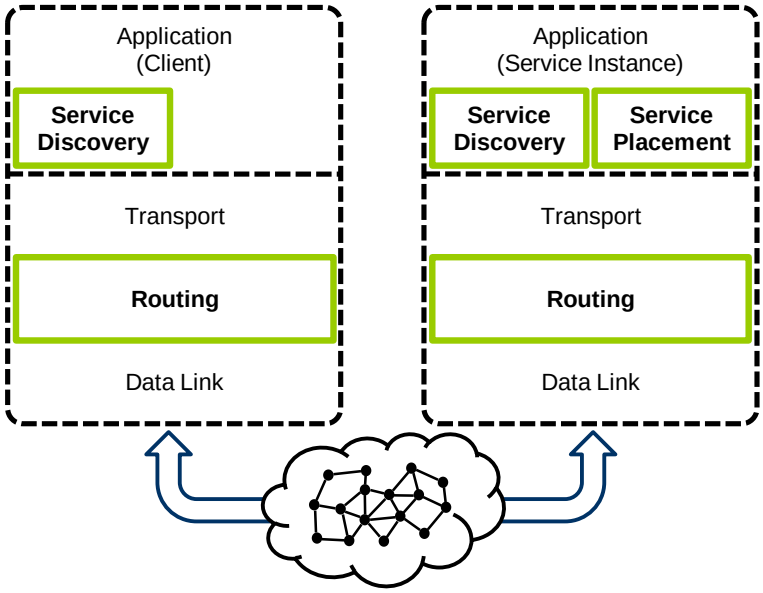


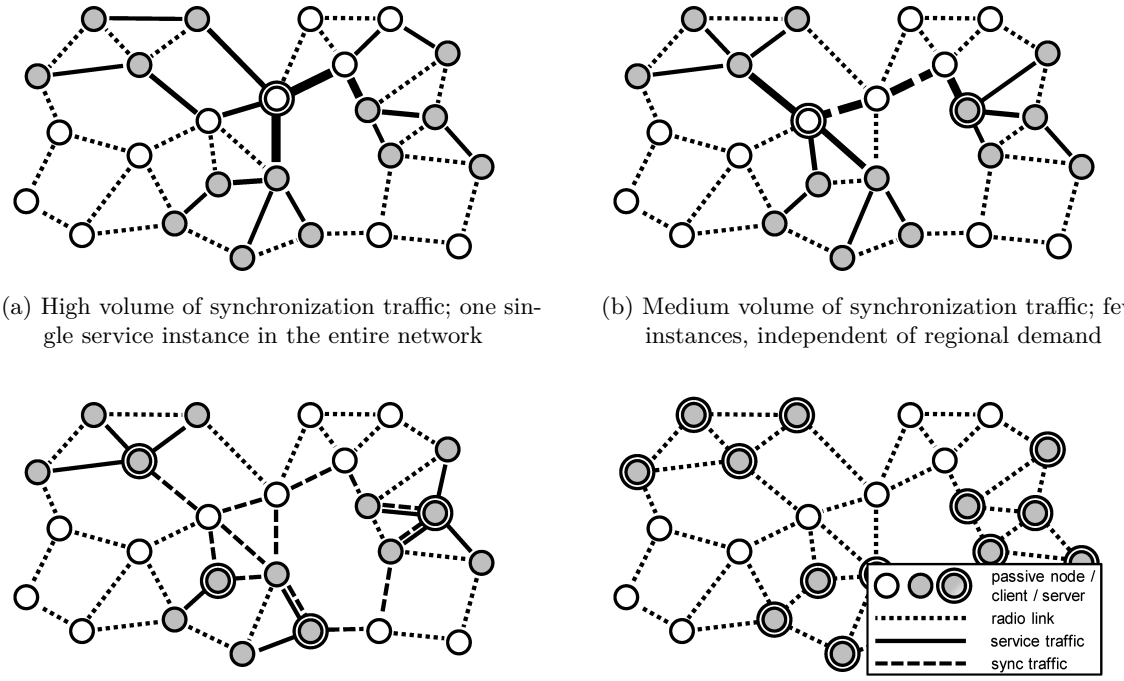
Figure 1.1: Components of the SP*i* service placement framework on client and server node

For any specific ad hoc network, the answers to these questions depend largely on the network topology and the service demands of the client nodes. Furthermore, the suitability of the service for migration of its instances as well as the requirements on the synchronization between the service instances need to be taken into account.

1.2.2 Proposed Solution

The system we are proposing comprises the SP*i* service placement framework [111] as well as two placement algorithms: the Graph Cost / Single Instance algorithm for centralized services with a single instance, and the Graph Cost / Multiple Instances algorithm for distributed services with a variable number of instances. The SP*i* framework is a cross-platform tool that, for the first time, allows for light-weight implementation and deployment of placement algorithms and easy side-by-side comparisons. The two SP*i* placement algorithms, in particular the Graph Cost / Multiple Instances algorithm, improve upon the state of the art by taking a novel architectural approach to service placement that results in superior service configurations, more timely adaptation decisions, and ultimately in an increase in the overall performance of ad hoc networks.

The major components of the SP*i* framework are depicted in Figure 1.1. The *service placement middleware* collects usage statistics for each service instance by inspecting service requests and replies. The *routing component* extracts local network topology information from enhanced routing packets which piggy-back data concerning routing paths



(a) High volume of synchronization traffic; one single service instance in the entire network

(b) Medium volume of synchronization traffic; few instances, independent of regional demand

(c) Low volume of synchronization traffic; several instances, strongly related to regional demand

(d) No synchronization traffic; one service instance per client

Figure 1.2: Service configurations for different levels of synchronization traffic

and neighborhood connectivity. The *service discovery component* is used by client nodes to locate and select suitable service instances.

The placement algorithms, which are run as part of the placement middleware, calculate the optimal service configuration using a cost metric that is based on service demand in terms of combined data rates and the network topology in terms of hop count or link quality. When calculating the optimal service configuration, our placement algorithms explicitly take the synchronization traffic between service instances into account. The interdependency between synchronization traffic and optimal service configuration is illustrated in Figure 1.2: If a large volume of synchronization traffic is required, as it may be the case for distributed databases that provide transactional semantics, the optimal service configuration is to have only a single instance of the service (cf. Fig. 1.2a). On the other extreme, if no synchronization traffic is required, e.g., for a spell checking service, each client node hosts its own service instance (cf. Fig. 1.2d). For the more interesting cases of an intermediate or low volume of synchronization traffic (cf. Figs. 1.2b and 1.2c), the optimal service configuration consists of a variable number of service instances, distributed intelligently by the placement algorithm.

With the optimal service configuration as input, a set of actions is established that transforms the current service configuration into the optimal configuration. Possible actions comprise replicating a service instance (thus creating a new instance), migrating an instance, and shutting down an instance. The combined cost of these actions in terms of network traffic is then compared to the improvement of the optimal configuration over the current configuration. If the result is favorable for the newly calculated optimal configuration, the nodes that currently host service instances proceed to implement these actions, thereby adapting the service configuration.

1.2.3 Scope

The goal of this research project is to create and evaluate a practical system comprised of a set of protocols and algorithms that addresses the questions raised above. It is designed to interact closely with the domains of routing and service discovery. The implementation is complete and sufficiently robust to allow for the system to run successfully on a real-world ad hoc network.

There are several other, closely related areas of research, which we will not consider in depth in this work:

- **Mechanics of service migration and creation:** It is out of scope to consider how exactly services and their state are to be transferred from one node to another. For our purposes, it is sufficient to merely consider the network traffic incurred by these operations. We neither address the intricacies of lifecycle management of a service, the serialization of service state, nor the interactions with the operating system of the nodes of the ad hoc network.

- **Security:** It is beyond the scope of our current work to make the system robust against attacks from malicious nodes. However, we point out a several high-level strategies to enhance security as part of our discussion of future work in Section 7.2.

- **Incentives for cooperation:** It is out of scope to establish under which motivation nodes of an ad hoc network should decide to host service or forward packets for other nodes. The assumption that nodes are willing to cooperate to achieve a common goal is widely used in research into ad hoc networking. It is beyond the scope of this work to establish strong use cases for ad hoc networks that provide incentives for this kind of cooperation

All of these items are prerequisites for the successful deployment of an ad hoc network with service placement. It is merely due to need to focus the current research effort that we do not consider them in more detail.

1.3 Structure of this Work

This work is structured as follows: We begin with a review of the fundamentals of ad hoc networking and service placement in Chapter 2. In this discussion, we briefly cover the technical foundations of ad hoc networking and the underlying theory of service placement from the domain of operations research. We also present an in-depth review of the state of the art of service placement in ad hoc networks and classify current proposal by their architectural approach and applicability.

Chapter 3 covers the technical side of the methods that we used to conduct our research. Our goal is to make use of several tools during the design, implementation, and evaluation of our system, spanning from simulations over emulations to real-world experiments. We present our approach and programming abstraction for making a distributed system portable across evaluation environments, which is a prerequisite for changing back and forth between these tools as needed. We then compare this approach to other methods and demonstrate its effectiveness by presenting various use cases.

In Chapter 4, we present the first part of our main contribution – the SP*i* service placement framework. This framework expands upon the methods presented in the previous chapter by implementing all the components required for side-by-side evaluation of a variety of approaches to service placement in ad hoc networks. We discuss the requirements and the rationale of this framework and present the key components: the protocols for routing and service discovery as well as the middleware that hosts the placement algorithms. As part of this discussion, we also present the interface that the framework provides to placement algorithms, and our approach for replicating service instances including possible optimizations.

The second part of our contribution consists of the placement algorithms presented in Chapter 5. We propose the Graph Cost / Single Instance and Graph Cost / Multiple Instances algorithms for the respective placement of a centralized and of a distributed service. These algorithms are built around the concept of service provisioning cost which is used both for establishing the optimal service configuration as well as deciding on the appropriate point in time at which the current configuration should be adapted. At the end of this chapter, we give an in-depth example to illustrate the operation of the SP*i* service placement framework and its algorithms.

In Chapter 6, we present a quantitative evaluation of the SP*i* framework and its placement algorithms. Employing the methods presented in Chapter 3, we evaluate our system in a variety of scenarios covering different aspects of service placement. We present results from simulations that investigate the behavior of our and other approaches under different load scenarios and under a variety of patterns in the service demand. We use emulations to

verify the correct operation of the framework and the placement algorithms under real-time conditions and to evaluate their performance under varying quality of communication links. Finally, we present the results from real-world experiments on a IEEE 802.11-compliant wireless ad hoc network to verify our findings.

Finally, we sum up and discuss our results in Chapter 7, and point out directions for future research. Additional validation of our experimental setup as well as more detailed results are available in Appendices A to C.

Chapter 2

Background

This chapter gives brief introductions into the fields of ad hoc networking and facility location theory, both of which form the basis for service placement in ad hoc networks. We also provide an extensive review of the current state of the art in service placement and evaluate recent proposals. With this qualitative evaluation we lay the foundation for the quantitative evaluation of service placement algorithms presented in Chapter 6.

We begin this chapter with a brief overview in Section 2.1 and then introduce the reader to the fields of ad hoc networking and facility location theory in Sections 2.2 and 2.3. In the latter, we specifically cover the p-median and the uncapacitated facility location problems. We then continue to review current approaches to service placement in Section 2.4 and discuss their strengths and weaknesses. In Section 2.5, we summarize our findings.

2.1 Overview

Service placement can be seen as an application of facility location theory to ad hoc networking. In fact, when reviewing current approaches to service placement in [110], we found that there are two distinct ways in which this area of research is generally approached: It is either tackled as a byproduct of middleware research or as an application of facility location theory.

The middleware-related approaches commonly focus on centralized services. They employ heuristics based on information gathered from nodes in the neighborhood of the node that is currently hosting the service. These heuristics are usually tailored to specific applications and adapt the service configuration to provide good network-wide coverage of the service or to respond to group mobility. None of these approaches supports truly distributed services and they are thus not applicable to general-purpose service placement.

The approaches with a background in facility location theory solve the Uncapacitated Facility Location Problem (UFLP), either with centralized algorithms that aggregate the necessary information on a dedicated node or with distributed iterative approximations.

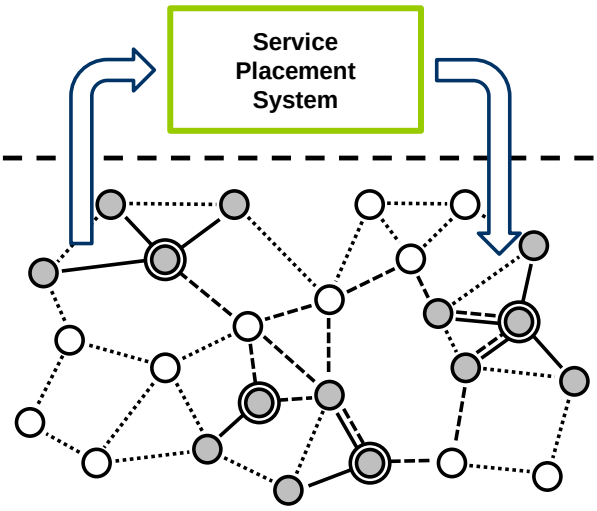


Figure 2.1: Service placement as a form of closed-loop control of an ad hoc network

Most of these approaches support adapting the number of service instances to the current service demand and the network topology. In contrast to the proposals with a background in middleware research, most UFLP-based approaches are not overly concerned with the overhead induced by gathering information or signaling between service instances.

Facility location theory is primarily concerned with static settings, i.e., problem specifications that are invariant over time. In ad hoc networks, however, the service demand and more importantly the network topology change continuously. Hence, a service placement system has to monitor the network and adapt the service configuration in response to relevant changes. This view on service placement corresponds to the model of *closed-loop control* from control theory [6].

As illustrated in Figure 2.1, a service placement system can be seen as an external entity that controls (part of) the operation of an ad hoc network. A key difference to classical control theory is that measurement, signaling, and the implementation of control decisions all occur in-band, i.e., they utilize the same resources as the system under control. For example, if a service instance needs to be migrated from its current node to a new host, then the resulting network traffic will temporarily occupy bandwidth that would otherwise be available for service provisioning. Depending on the implementation of the service placement system, the network traffic required for controlling the network operation may be non-negligible and adversely impact the quality with which the service can be provided to client nodes. Therefore, a service placement system must consider carefully *how much* measurement and control is warranted. We explore this question and the relevant tradeoffs in greater detail in Section 4.2.

2.2 Ad Hoc Networks

A general definition of ad hoc networks, as stated by Perkins and Royer in [90], is as follows:

> An ad hoc network is the cooperative engagement of a collection of mobile nodes without the required intervention of any centralized access point or existing infrastructure.

Given the need to be self-reliant in all matters regarding the organization of the network, the two key principles in ad hoc networking are cooperation among nodes and autonomous management of the network. The latter of these two motivates a set of self-x properties, e.g., self-configuration, self-healing, self-management, and self-optimization, that are desirable for ad hoc networks [27, 76].

This section only provides a brief overview over ad hoc networking. A more in-depth treatment of the subject can be found in [88, 96].

2.2.1 Types of Ad Hoc Networks

Ad hoc networking is a fundamental technology that can be employed in various areas of application. This gives rise to various specialized types of ad hoc networks which – while all relying on the same basic principles – emphasize different aspects of ad hoc networking or make different design tradeoffs in light of the varying application scenarios.

In the following, we give a brief overview over the most prominent types of ad hoc networks. For illustration, exemplary devices that form ad hoc networks are shown in Figure 2.2.

Wireless Mesh Networks

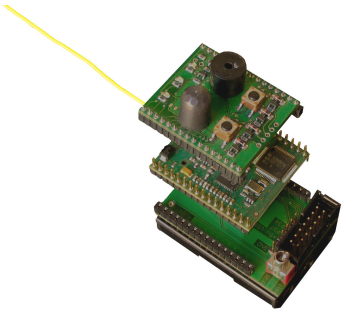
Wireless Mesh Networks (WMNs) [3] are a very general form of ad hoc networks. They consists of mesh routers, mesh clients, and optionally gateways. The mesh routers (cf. Fig. 2.2a) communicate with each other via radio transmissions and employ special-purpose routing protocols such as Optimized Link State Routing (OLSR) [19] or Ad hoc On-Demand Distance Vector (AODV) routing [87] to form a wireless multi-hop backbone. Mesh clients (c.f. Figs. 2.2b and 2.2c) access the network via one of the routers, while gateways provide connectivity to other networks, e.g., the Internet. Radio communication is handled using one of several standards, e.g., IEEE 802.11 (WLAN, cf. Sec. 2.2.2) or IEEE 802.16 (WiMAX, cf. Sec. 2.2.2).

Research in the area of WMNs is mostly concerned with questions regarding routing of packets and self-configuration of the nodes that form the network. As of this writing, there are several large-scale WMNs in operation such as the wireless networks in Athens [7] and

(a) Linksys WRT54G wireless broadband router [18]

(b) OLPC XO-1 laptop [33] (Picture by Mike McGregor)

(c) Google Nexus One smartphone [23]



(d) ScatterWeb MSB sensor node [97] (Picture from [30])

Figure 2.2: Exemplary devices that support ad hoc networking

the Freifunk network in Berlin [35]. Both networks employ the OLSR routing protocol. These networks have been setup to provide connectivity to the Internet in areas in which it is impossible to achieve the same connectivity in terms of quality or price by classical means alone. Another prospective application of WMNs is office mesh networking, i.e., to provide IP-connectivity in an office building without having to install a wired infrastructure as it would be required for an IEEE 802.3 network.

Mobile Ad Hoc Networks

Like WMNs, Mobile Ad Hoc Networks (MANETs) are a very general form of ad hoc networks. They are built on top of similar communication technology as WMNs, but emphasize the mobility of nodes as the key property of the wireless network. Given this mobility, it is impossible to built a persistent backbone as in WMNs. Instead, all nodes that form a MANET need to contribute to maintain network-wide connectivity by routing packets for other nodes. From this follows that the heterogeneity of nodes in MANETs as compared to WMNs must be rather limited.

Research into MANETs is also concerned with routing protocols, but does so with a greater focus on mobility patterns and the churn rate, i.e., the rate at which new nodes join the network or current nodes leave. Several specialized flavors of general-purpose MANETs exist, the two most prominent ones being Vehicular Ad Hoc Networks (VANETs) and certain types of wireless sensor networks [20].

Personal Area Networks

Another form of wireless ad hoc networks are Personal Area Networks (PANs). As the term suggests, these networks generally cover a very limited area, i.e., they emphasize direct communication between nodes. At the same time, the form factor of the nodes plays a major role. Since the size of the battery contributes significantly to the form factor, the energy available to each node is limited as a consequence. Depending on the exact area of application, PANs may be built around communication technologies standardized as part of IEEE 802.15 (cf. Sec. 2.2.2), e.g., IEEE 802.15.1 (Bluetooth) or IEEE 802.15.4 (ZigBee).

PANs are used to connect peripherals to PC-style hardware and to establish transient connections between portable devices such as smart phones. Research is mostly focused on providing direct connectivity between nodes for specific applications and on energy preservation. A special form of PANs are Body Area Networks (BANs) in which the nodes are attached to or even implanted into the human body for medical applications.

Wireless Sensor Networks

Wireless Sensor Networks (WSNs) [2] are commonly made up of a potentially large number of highly embedded and resource-constrained devices (cf. Fig. 2.2d). These sensor nodes collaboratively employ their built-in sensors to gather data from the environment and report it back to a base station. Like in PANs, energy conservation is a major design parameter for WSNs. Since WSNs are targeted to cover large areas for prolonged deployment times, self-organization and multi-hop communication also play major roles. The latter is, however, frequently restricted to specific communication patterns such many-to-one communications which are predominant when nodes report measurements to the base station. Among many alternatives, WSNs may employ IEEE 802.15.4 (ZigBee, cf. Sec. 2.2.2) for communication between nodes.

The core application of WSNs is precision sensing in application scenarios that range from inaccessible areas to the monitoring of industrial machinery. One variant is smart metering, i.e., the wireless metering of utilities that enables near real-time monitoring and billing. The research focus in WSNs is similar to that in PANs. However, the setting is more demanding given the even smaller form factor of the devices, the need for multi-hop

communication, and strong reliance on the self-organizing capabilities of the network. In many scenarios, the data gathered by sensor networks is only valuable if mapped to objects or locations of the physical world. Therefore, an additional challenge is to establish the exact position of randomly deployed sensor nodes.

As can be seen from this list, ad hoc networks cover a wide range of application scenarios. Hence, the form factor of the nodes ranges from miniscule embedded devices used in WSNs over embedded and highly integrated devices employed in VANETs to desktop-style PC hardware found in office mesh networks. We note that service placement as a fundamental technology is equally applicable to all these types of ad hoc networks.

2.2.2 Standards and Current Approaches

Standardization of the technologies required for ad hoc networking is conducted by the Institute of Electrical and Electronics Engineers (IEEE) and the Internet Engineering Task Force (IETF). Specifically, computer networking employing variable-sized packets is covered by the IEEE 802 family of standards. Of particular interest for ad hoc networking is the standardization effort of the working groups that deal with wireless communications, i.e., IEEE 802.11 – wireless local area networks, IEEE 802.15 – wireless personal area networks, and IEEE 802.16 – wireless broadband. At the IETF, there are three working groups that address topics that are of interest to ad hoc networking. Their two main lines of standardization focus on questions regarding the routing of packets and self-configuration.

IEEE 802.11 (Wireless Local Area Networks)

IEEE 802.11 [102], better known as wireless local area networking (WLAN) or under the term *Wi-Fi*, is the most widely used of the three sets of standards. The first IEEE 802.11 standard was published in 1997, but it was quickly amended to support higher data rates and enhanced security (IEEE 802.11a/b in 1999, IEEE 802.11g in 2003, IEEE 802.11i in 2004, and IEEE 802.11n in 2009). The draft of IEEE 802.11s specifically addresses the requirements of mesh networking.

IEEE 802.11 devices communicate over a set of channels in the 2.4 GHz and 5 GHz frequency bands. Signals are modulated using either Direct-Sequence Spread Spectrum (DSSS) or Orthogonal Frequency-Division Multiplexing (OFDM). Nodes communicate with each other using variable-length frames addressed to the MAC address of their network interface cards. Different types of frames exist for network management, control, and data transmissions. Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) with an optional RTS/CTS mechanism is employed for channel access. Priorities of different types of frames as well as fair access to the wireless channel are implemented via intelligently

varied back-off timers before sending a packet. The maximum raw data rate of the current IEEE 802.11n standard is 150 Mbits/s. The first security mechanism of IEEE 802.11 networks, Wired Equivalent Privacy (WEP), was quickly replaced with Wi-Fi Protected Access (WPA and WPA2) after it was found to be easily vulnerable to attack.

Devices that operate according to this standard fall predominantly into the category of portable computers, but support for IEEE 802.11 has recently emerged in printers, digital cameras, and smartphones.

IEEE 802.15 (Wireless Personal Area Networks)

The IEEE 802.15 working group standardizes protocols for wireless personal area networks (WPANs), but the same technology is also partially applicable to WSNs. The first standard, IEEE 802.15.1, was published in 2002 in cooperation with the Bluetooth Special Interest Group (SIG) and later updated in 2005 [100]. These two standards correspond to version 1.1 and 1.2 of the Bluetooth specification respectively. They are mostly concerned with connecting peripheral devices and thus include provisions for a variety of profiles for specific use cases. As a second line of standardization, the IEEE 802.15.4 set of standards – first published in 2003 – is concerned with low data rate networks that operate under severe energy constraints, e.g., WSNs.

Like IEEE 802.11, Bluetooth operates in the 2.4 GHz frequency band. It employs Frequency-Hopping Spread Spectrum (FHSS) across 79 channels to establish communication links. Depending on the modulation scheme, Bluetooth can achieve raw data rates of up to 3 Mbit/s. Medium access is centrally controlled by the master node of a Bluetooth piconet. A piconet can consists of up to eight nodes. Bridging between piconets is possible by a master node joining a second piconet as a slave. The Bluetooth protocol stack comprises a variety of protocols for functions such as link management, service discovery, and serial communication. Requirements on the protocol stack for specific tasks are encoded in profiles. Bluetooth specifications after version 1.2 have been published by the Bluetooth SIG without cooperation with the IEEE, the latest one being the Bluetooth Core Specification version 4.0 in 2010.

IEEE 802.15.4 (ZigBee) [101] devices communicate on frequency bands at 868 MHz, 915 MHz, and 2.4 GHz. They employ a DSSS modulation scheme and achieve raw data rates between 20 kbit/s and 250 kbit/s depending on the frequency band. Like in IEEE 802.11, CSMA/CA is used for channel access. IEEE 802.15.4 nodes may form multi-hop networks and nodes in such networks take on similar roles to those found in WMNs: end devices (clients), routers, and gateways. As low energy consumptions is one of the main goals of IEEE 802.15.4, these roles translate into different hardware capabilities of the nodes.

IEEE 802.16 (Wireless Broadband)

IEEE 802.16 [103], also known as *WiMAX*, is a set of standards that is targeted at high data rate metropolitan area networks, i.e., the goal is to wirelessly bridge the "last mile". The first IEEE 802.16 standard covering physical and MAC layer issues was published in 2001. In subsequent standards, the focus was expanded to include topics such as mobility and quality of service.

The frequency bands used by IEEE 802.16 for communication range from 2 GHz to 66 GHz. Signals are modulated with Orthogonal Frequency-Division Multiple Access (OFDMA). The currently supported raw data rate is 40 Mbit/s, but work is underway to push the raw data rate to 100 Mbit/s for mobile and 1 Gbit/s for fixed nodes. A key difference to other ad hoc networking technologies is that IEEE 802.16 employs a connection-oriented MAC layer and is thus able to offer guarantees with regard to the quality of service.

According to [32], there are 592 WiMAX networks in operation as of this writing.

IETF Working Groups

The three working groups within the IETF that are of particular interest to ad hoc networking are: Ad-Hoc Network Autoconfiguration (autoconf) which deals with self-configuration issues such as address assignment, Mobile Ad-hoc Networks (manet) which deals with general-purpose routing protocols, and Routing Over Low power and Lossy networks (roll) which deals with routing in more specialized scenarios. These working groups published their first standard in the form of a Request for Comments (RFC) in 2003.

The standardization effort encompasses both proactive routing protocols such as Optimized Link State Routing (OLSR) [19], and reactive protocols such as the Ad hoc On Demand Distance Vector (AODV) [87] and Dynamic MANET On-demand (DYMO) [17] protocols. The key difference between these two approaches lies in the question whether routing information should be exchanged between nodes continuously or only in response to pending packets that need to be forwarded. Generally speaking, the former is more suitable to static scenarios such as WMNs and certain deployments of WSNs, while the latter is targeted at the dynamic conditions found in MANETs and VANETs.

Recent work also addresses the special requirements of Low power and Lossy Networks (LLNs) such as WSNs. The IPv6 Routing Protocol for LLNs (RPL) [107] leverages the special communication patterns in these scenarios. Since most data is exchanged in the form of many-to-one communications, i.e., between nodes that report measurements to a base station, packets can be routed along energy efficient, tree-like topologies.

2.3 Facility Location Theory

Facility location theory [77] is a branch of operations research. It employs mathematical models to solve problems dealing with the optimal placement of facilities such as factories, distribution centers, or, in the domain of networking, switching centers. Of the various problems studied in facility location theory, two problems are applicable to service placement in ad hoc networks: the p-median problem and the uncapacitated facility location problem. In this section, we present both problem statements and discuss their applicability to service placement.

2.3.1 The p-median Problem

Summarizing the introduction provided by Reese [91], the p-median problem can be stated as follows:

> Given a graph or a network $G = (V, E)$, find $V_p \subseteq V$ such that $|V_p| = p$, where p may either be variable or fixed, and that the sum of the shortest distances from the vertices in $V \setminus V_p$ to their nearest vertex in V_p is minimized.

In the context of ad hoc networks, the nodes of the network correspond to the vertices V and the wireless communication channels between the nodes correspond to the edges E.

More formally, Hakimi [45] defines the *p-median* of G as a set of vertices $\hat{V}_p$ such that

$$\bigvee_{V_p \subseteq V} \sum_{i=1}^{n} w_i \cdot d(v_i, \hat{V}_p) \leq \sum_{i=1}^{n} w_i \cdot d(v_i, V_p)$$

where $v_i \in V$ is a vertex of the graph, w_i is the weight of vertex v_i, $d(v_i, v_j)$ is the weight of the edge connecting v_i and v_j, and $d(v_i, V_p)$ is the shortest distance from vertex v_i to its nearest element in V_p.

The p-median problem belongs to the class of *minisum location-allocation problems*. It operates on a discretized solution space, e.g., on a set of vertices V, while the similar Fermat-Weber problem deals with a continuous solution space, e.g., the Euclidean plane. A special case of the p-median problem is the 1-median problem, for which the single vertex in $\hat{V}_1$ is referred to as the *absolute median* of the graph G [44].

The p-median problem can be formulated as an integer program [92]. Let ξ_{ij} be an allocation variable such that

$$\xi_{ij} = \begin{cases} 1 & \text{if vertex } v_j \text{ is allocated to vertex } v_i \\ 0 & \text{otherwise} \end{cases}$$

The integer program is to

$$\begin{aligned}
\text{minimize} \quad & Z = \textstyle\sum_{ij} W_{ij} \cdot \xi_{ij} \\
\text{subject to} \quad & \forall_{j=1\ldots|V|} \textstyle\sum_{i=1\ldots|V|} \xi_{ij} = 1, && (2.1)\\
& \textstyle\sum_{i=1\ldots|V|} \xi_{ii} = p, && (2.2)\\
& \forall_{i,j=1\ldots n} \xi_{ij} \leq \xi_{ii}, && (2.3)\\
& \xi_{ij} \in \{0,1\}. && (2.4)
\end{aligned}$$

where W is the weighted distance matrix $W_{ij} = w_i \cdot D_{ij}$, and D is the shortest distance matrix $D_{ij} = [d(v_i, v_j)]$.

Constraint 2.1 ensures that each vertex is allocated to one and only one element in the p-element subset. Constraint 2.2 guarantees that there are p vertices allocated to themselves, which forces the cardinality of the p-median subset to be p. Constraint 2.3 states that vertices cannot be allocated to non p-median vertices. Constraint 2.4 specifies the possible values for ξ_{ij}. The p-median is $\{v_i \in V \mid \xi_{ii} = 1\}$.

The p-median problem on general graphs for an arbitrary p is NP-hard [38, 56]. For a fixed p, the problem can be solved in polynomial time [38]. A recent discussion of solution methods for the p-median problem can be found in [91].

2.3.2 The Uncapacitated Facility Location Problem

Cornuejols, Nemhauser, and Wolsey discuss the Uncapacitated Facility Location Problem (UFLP) in detail in [22]. The problem statement according to them is as follows:

> An economic problem of great practical importance is to choose the location of facilities, such as industrial plants or warehouses, in order to minimize the cost (or maximize the profit) of satisfying the demand for some commodity. In general, there are fixed costs for locating the facilities and transportation costs for distributing the commodities between the facilities and the clients. This problem [...] is commonly referred to as the *plant location problem*, or *facility location problem*.

More formally, consider a set of clients I with a given demand for a single commodity and a set of sites for facilities J. Opening a facility on site $j \in J$ incurs a fixed cost f_j. Serving a client $i \in I$ from a facility $j \in J$ incurs a fixed cost of d_{ij}. The solution to the UFLP is to open facilities on the subset of the available sites $\hat{J} \subseteq J$ that minimizes the total cost while satisfying the demand of all clients. Each client i satisfies its demand from the facility j for which d_{ij} is minimal. Thus, the total cost of an optimal selection of facilities $\hat{J}$ is $\sum_{i \in I} \min_{j \in \hat{J}} d_{ij} + \sum_{j \in \hat{J}} f_j$.

While similar to the p-median problem in goals and solution methods, there are three important differences between the two problems [91]:

- The UFLP introduces a fixed cost f_j for establishing a facility at a given vertex $j \in J$.
- It has no constraint on the maximum number of facilities (except for the obvious upper bound $|J|$).
- Formulations of the UFLP usually separate the set of possible facilities from the set of clients, i.e., there are separate sets I and J for clients and potential facility locations as opposed to the uniform set of vertices V of the p-median problem.

Like the p-median problem, the UFLP can be formulated as an integer program [22]. Let ψ_j and ϕ_{ij} be allocation variables such that

$$\psi_j = \begin{cases} 1 & \text{if facility } j \in J \text{ is open} \\ 0 & \text{otherwise} \end{cases}$$

$$\phi_{ij} = \begin{cases} 1 & \text{if the demand of client } i \in I \text{ is satified from facility } j \in J \\ 0 & \text{otherwise} \end{cases}$$

The integer program is to

$$\begin{aligned} \text{minimize} \quad & Z = \textstyle\sum_{i \in I} \sum_{j \in J} d_{ij} \cdot \phi_{ij} + \sum_{j \in J} f_j \cdot \psi_j \\ \text{subject to} \quad & \forall_{i \in I} \ \textstyle\sum_{j \in J} \phi_{ij} = 1, && (2.5) \\ & \forall_{i \in I, j \in J} \ \phi_{ij} \leq \psi_j, && (2.6) \\ & \forall_{i \in I, j \in J} \ \psi_j, \phi_{ij} \in \{1, 0\}. && (2.7) \end{aligned}$$

Constraint 2.5 guarantees that the demand of every client is satisfied. Constraint 2.6 ensures that clients are only supplied from open facilities. Constraint 2.7 specifies the possible values for ψ_j and ϕ_{ij}. The solution to the UFLP is $\{j \in J \mid \psi_i = 1\}$. If we extend this integer program with an additional constraint $\sum_{j \in J} \psi_j = p$ for a new parameter p and set $f_j = 0$ for all $j \in J$, then the problem statement corresponds to the p-median problem.

Just like the p-median problem, the UFLP is NP-hard. Solution methods to the UFLP are presented in [22].

2.3.3 Applicability to the Service Placement Problem

It is apparent from their definitions that both the p-median and the uncapacitated facility location problems strongly resemble the service placement problem. In fact, service placement can be seen as an application of facility location theory to service provisioning in ad

hoc networks. Of the two problems, the UFLP is more similar to the service placement problem since the optimal number of the facilities depends on the cost of establishing them as well as on the demand.

There are, however, several important differences between the UFLP and the service placement problem:

- The UFLP does not capture the fact that service instances need to exchange data in order to keep the global state of the service synchronized. The concept of fixed cost per facility cannot adequately describe the synchronization cost, since the synchronization overhead depends on the overall configuration of the service, i.e., number of current service instances and their locations.
- The network topology in an ad hoc network and the service demand of the client nodes is variable over time. Hence, the service configuration needs to be adapted as soon as the current configuration becomes inadequate. The UFLP, however, has no notion of variable inputs over time and hence cannot answer the question when to adapt a service configuration.
- The UFLP assumes that the information about possible locations for facilities J as well as about the cost for satisfying the demand of a client d_{ij} is readily available. In the context of service placement, this information depends on the changing network topology and needs to be collected at run time. Gathering this information for central processing causes a non negligible communication overhead. This overhead needs to be taken into account when solving the service placement problem, while it is out of scope for the UFLP.

Given these differences, it is not possible to directly reuse solutions to the UFLP for placing services in ad hoc networks. However, since the problems are similar, we can base our own algorithms on solution methods from this field. In fact, the Graph Cost / Multiple Instances placement algorithm as proposed in Section 5.5 is in part inspired by previous work on the p-median problem [75, 94].

2.4 Current Approaches to Service Placement

Service placement is part of the self-organization paradigm of ad-hoc networks. A survey on the topic focussed on MAC and routing has been recently published by Dressler [27]. Further, service placement is also a topic in the context of application distribution as surveyed by Kuorilehto et al. [60], who conclude that current approaches are too complex, and motivate the necessity for simpler algorithms with less control traffic, even at the expense of placement quality.

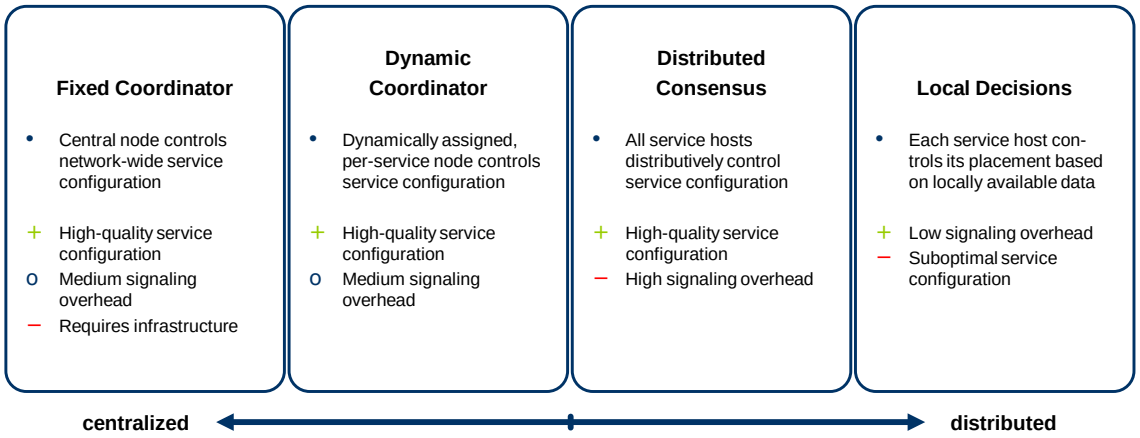


Figure 2.3: Design space of service placement systems

2.4.1 Design Space

The fundamental tradeoff in the design of a service placement system lies in the structure of the placement algorithm. This algorithm can either run centrally on a node selected for this very purpose, or it can be distributed over several nodes who jointly take a placement decision. Given the distributed nature of an ad hoc network, one may be tempted to instinctively prefer a distributed solution. However, as illustrated in Figure 2.3, the architectural tradeoffs are more intricate.

Distributed approaches generally have the drawback of either incurring a high signaling overhead or having problems in achieving high-quality service configurations. Centralized approaches do not suffer from these problems, but have the disadvantage of introducing a single point of failure into the optimization process. We discuss these tradeoffs in greater detail in Section 4.2. For the understanding of the review of current approaches presented in the following sections, it is helpful to keep this very basic classification and the resulting system properties in mind.

2.4.2 General-purpose Service Placement

This section summarizes the approaches that deal with general-purpose service placement, both in theory and practice. These publications are the ones most closely related to our own work.

Coverage-based Placement of Distributed Services in MANETs

In [93], Sailhan and Issarny propose an architecture for scalable service discovery in MANETs built around a dynamic and homogeneous deployment of service directories within the

network. The homogeneous placement of the directories is achieved by an initiating node broadcasting a query for available resources and for parameters regarding the surroundings of a node, e.g., currently available directories and connectivity to its neighborhood. Queried nodes either reply with the corresponding information or refuse to participate in the process of selecting new service hosts. Based on this data, the initiating node chooses a node for hosting a new directory. The main selection criteria are the expected coverage of the new directory in terms of number of direct neighbor nodes and the number of other directories in the vicinity. The node that best matches these criteria is then notified by the initiating node of this decision and begins hosting the service directory. In case of simultaneous election processes, only one election process is allowed to proceed based on the unique network address of the initiating node. For load balancing, the node currently hosting the directory periodically initiates a new election process in order to move existing directories from node to node.

Coverage-based Service Placement in Sensor Networks

In [67], Lipphardt et al. propose the Dynamic Self-organizing Service Coverage (DySSCo) protocol. This protocol places the instances of a distributed service with the goal of achieving a given network-wide coverage, i.e., hosting a service instance on a preconfigured portion of the nodes. The resulting service configuration thus solely depends on the network topology, but not on the actual service demand of client nodes.

DySSCo operates by periodically exchanging beacon messages between all neighboring nodes of the network. These beacon messages encode information about how many service instances are currently available in the local region of the network, i.e., the 1-hop neighborhood of each node. This metric is used as an indication on whether the portion of globally available service instances matches the preconfigured value. If the regional number of service instances is too low, then a new service instance is created on the local node. If the regional number of service instances is too high and would remain sufficiently high without the local instance, then any local instance is shut down. Additional beacons are exchanged whenever such a change in the service configuration takes place. The evaluation of the DySSCo protocol using a real-world WSN deployment shows that the service configuration converges and reaches the requested coverage.

Iterative, Topology-based Migration of a Centralized Service in MANETs

As part of the REDMAN middleware, Bellavista et al. [8–11] propose a method for electing a node in a dense MANET that is responsible for coordinating the replication of data items in a given area. The election process is started once a node detects that it has become

part of a dense network region without a replication manager. The goal is to place the replication manager at the topological center of the dense network region. In order to achieve this, the service is iteratively migrated one hop towards the node which is furthest away from the current node until no better candidates other than the current node are available. Heuristics are used to establish migration targets and the distance of the most distant node.

Iterative, Request Volume-based Migration of a Centralized Service in MANETs

Oikonomou and Stavrakakis propose in [85] a method for the optimal placement of a centralized service in a MANET. Their approach is to iteratively migrate the service to the neighboring node from which the highest volume of service request has been received as long as this request volume makes up for more than half of the total request volume. One can think of this procedure as similar to the hill climbing approach in optimization, which has the problem that it may terminate in a local optimum as opposed to the global optimum. However, the authors prove that under the assumption of a tree-like network topology their algorithm eventually reaches the globally optimal position. The approach has the additional advantage that the service migration process is naturally restarted if the network topology or the service demands change. In the following, we refer to this placement algorithm as *tree-topology migration*.

Iterative Migration of Multiple Software Components in MANETs

The MagnetOS project as described in [68] by Lie et al. aims at building a distributed operating system for MANETs that has the ability to migrate software components, e.g., Java objects, across the network. By placing the components close to those nodes that generate the most request for them, communication overhead can be reduced and network lifetime be prolonged. The algorithmic approach is to discretize time into epochs and at the end of each epoch relocate a service component according to one of five strategies:

- **LinkPull** moves the component one hop in direction of the most requests it receives.
- **PeerPull** moves the component to the host that caused most requests.
- **NetCluster** moves the component to a random node of the one hop cluster whose nodes collectively caused most requests.
- **TopoCenter(1)** moves the component to a node that – based on a partial view of the network – minimizes the sum of migration cost and the future service communication cost.

- **TopoCenter(Multi)** works similarly as TopoCenter(1) but with additional information on network topology.

The evaluation of these algorithms is conducted through simulations and real-world experiments. It shows that depending on the scenario the network lifetime can be increased significantly.

Rule-based Replica Placement in Ambient Intelligence Infrastructures

Herrmann proposes the Ad hoc Service Grid (ASG) for placing replicas, i.e., instances, of a distributed service [49]. The motivation of this work is to enable the self-organization of Ambient Intelligence systems which have to operate efficiently without any human intervention. To this end, ASG implements a set of three services that implement self-configuration for ad hoc networks: self-organized placement of distributed services, service discovery, and synchronization of service instances.

As part of the placement subservice, the author presents two distributed placement algorithms: ASG / simple and ASG / Event Flow Tree (EFT), the latter one being a refinement of the former. Both placement algorithms are a rule-based algorithms that are executed on each node that currently hosts a service instance. Neither of the two algorithms requires nodes to explicitly exchange information in order to take placement decisions. For ASG / simple, migrations are triggered if more service requests are received from one neighbor than from all other neighbors and the current service host together. Replications are triggered if the service requests that have been forwarded by a migration target have traveled more than a preconfigured number of hops. Service instances are shut down if the service demand they serve falls below a threshold. ASG / EFT builds upon the same basic principle, however it also considers more distant nodes as potential targets for migrations and replications. As a result, ASG / EFT requires less operations before converging to an optimal service configuration. The rules of both algorithms rely upon preconfigured parameters that encode knowledge about the optimal distribution of the service instances in relation to the network topology. Suitable values for some of these parameters can be derived analytically from properties of the network topology, but for other parameters, scenario-specific expert knowledge is required. The ASG system is evaluated using simulations. In their discussion, the authors also emphasize its properties with regard to self-organization in general.

Service Placement and Replication in Fixed, Internet-style Networks

As a follow-up to [85], Laoutaris et al. [62] map the two problems of finding the optimal locations for a given number of services and of additionally calculating the optimal number

of services to the uncapacitated p-median and the uncapacitated facility location problems respectively (cf. Section 2.3). Most approaches to solving these problems are centralized algorithms that in a network-related scenario required global knowledge about the network topology, the service demands of the nodes, and (optionally for the UFLP) the cost of hosting a service.

In order to avoid the centralized collection of this information, the authors propose distributed versions of both problems. The idea behind these new algorithms is to only solve the problems for the n-hop neighborhood of nodes hosting a service instance. The service demand of distant nodes is mapped to the nodes on the border of the neighborhood that forward the respective service request packets. A crucial part in this procedure is to appropriately merge overlapping neighborhoods of different nodes that host a service. For evaluation, the network traffic incurred by the respective solutions of the centralized and the distributed algorithm are compared and put into the context of required iterations of the distributed algorithm.

Local Facility Location in Distributed Systems

Krivitski et al. propose a local algorithm for the facility location problem for large-scale distributed system in general [58] and for WSNs in particular [57]. Given a set of clients, a set of possible locations for facilities, a cost function, and the desired number of facilities, the proposed algorithm establishes which locations are to be used for the desired number of facilities in order to minimize cost. This is achieved without a coordinating central entity based on information available in the neighborhood. The algorithm employs speculative hill climbing and local majority votes: Each node speculatively calculates the next step in the hill climbing process based on the locally available information and then – rather than calculating the cost directly – the optimal step is chosen via a distributed majority vote in the network. Network overhead is kept low by avoiding votes on alternatives that do not reduce the cost as compared to a currently known best next step.

Distributed Facility Location in WSNs

In [34], Frank and Römer describe how the facility location problem can be solved in a distributed manner and apply their solution to the area of WSNs. The algorithm operates by locally running algorithms to establish locations for facilities and mapping clients to them over several rounds. In each round, a candidate facility location is chosen based on expected clients, and clients decide whether using the services offered by this facility is optimal for them with regard to other facility candidates. If the actual decision of the clients matches the expectation of the facility candidate, the new facility is opened. Network traffic is

reduced by iteratively expanding the search radius. The reduction in service communication cost achieved by the algorithm is evaluated using both simulation and traces from a real deployment.

Partition Prediction and Service Replication

Wang and Li [66,105,106] address the problem of service availability with regard to network partitions due to node mobility. Based on the group mobility model proposed in [64] (later republished in [65]), their approach is to group nodes by their velocity vectors and predict the event of such a group moving out of the radio range of another group. Since the resulting two groups have no means of communicating with each other, this effectively partitions the MANET. In case of a single node providing a service to both mobility groups, a new service instance is created on one node in the mobility group that would be left without access to the service otherwise.

The algorithm to select a specific node in the departing group to host the new service instance is run centrally on the node that is currently hosting the service. The main goal of the node selection process is unclear: According to [105], the goal is to delay the time of replication as much as possible, and therefore select the node *most distant* from the current server along the velocity vector of the leaving mobility group. In contrast, according to [66], the goal is to start the replication process as soon as possible, and therefore select the node *closest* to the current server. In both cases, the exact motivation for selecting this particular node is not given. One can only assume that the intended strategy is to delay the replication as much as possible and therefore select the node *closest* to the current server. In any case, neither of these two strategies considers the placement of the service within the leaving mobility group with regard to other service placement parameters.

2.4.3 Service Placement of Composite Services

Not as closely related to this proposal as general-purpose service placement, but worth taking into consideration is the placement of composite services. Subservices are commonly organized in a tree-like structure which is then mapped to the actual network topology.

Service Chain Placement in MANETs

As part of their Mobile Service Overlays project [98,99], Seshasayee et al. propose a method for distributing the components of a service across the nodes of a MANET. The aim of this project is to use dynamic load balancing as a mean to save energy and ultimately prolong the lifetime of the network. Algorithmically, they decompose the services into several components that depend on each other, thereby establishing the flow graph of the

service. In order to map the vertices of the graph to the nodes of the network, the graph is first partitioned into chains which are then mapped to the nodes while taking the energy resources of each node into account. The mapping of a single component along a chain, an entire chain, or the entire flow graph is evaluated over time and may be readjusted if necessary. Small scale experiments with five PDA-style devices show that this approach prolongs the network lifetime by up to 10%.

Operator Tree Placement in WSNs

Bonfils and Bonnet [14] deal with with the distribution of long-term query operators over the nodes of a WSN. Operators form a logical tree and filter, aggregate, and correlate data in the network. If placed correctly, this reduces the overall network traffic. The algorithm used for operator placement relies on exploring the nodes in the neighborhood of the current location of the operators and evaluating whether a migration would reduce the network traffic. Due to the tree-like structure of the operators, migrating one operator to another node may cause the migration of the subtrees that depend on an operator. Hence, the cost of migrations is calculated recursively over the subtrees. While the proposed approach inherently incurs the risk of getting stuck in local optima, simulative results show that near-optimal placement can be achieved for a variety of network topologies.

Service Composition Graph Placement in WSNs

In [1], Abrams and Liu describe the service placement strategy of their Service-Oriented Network proGramming of Sensors (SONGS) platform [69]. They introduce a set of hierarchical structured services that make up a user task, the so-called *service composition graph*, and their work deals with mapping the services of the service composition graph onto the network. Coming from a background in WSNs, they further restrict the service composition graph to have a fixed root, i.e., the base station of the WSN, and fixed leaves, i.e., the nodes equipped with the physical sensors. The contribution of their work is a set of solutions to the problem of finding the most appropriate nodes in the network for hosting the remaining mobile services of the service composition graph. They propose an algorithm requiring global knowledge for undirected network graphs, two theoretical heuristics with similar requirements but a reduced complexity, and a distributed implementation of a heuristic that uses limited network flooding. The distributed implementation is evaluated through simulations with respect quality of the solution and communication overhead.

2.4.4 Clustering

Clustering is an application of service placement predominantly found in the domain of WSNs. As node mobility is generally considered of lesser importance in WSNs (especially when compared to MANETs), clustering based on the physical location of the nodes can be expected to have a significant payoff in this domain. Nevertheless, some of these approaches may be suitable as inspiration for general-purpose service placement and are thus worth considering.

Decentralized Probabilistic Clustering

In [48], Heinzelman et al. describe a clustering method for WSNs that is part of their Low-Energy Adaptive Clustering Hierarchy (LEACH) protocol. The clustering algorithm is based on local probabilistic choices of individual nodes. Global parameters such as number of nodes and the desired number of cluster heads need to be configured on all nodes before a deployment can take place. Optionally, the algorithm can also take the remaining energy on each node into account. This ensures that the additional energy expenditure that results from serving as a cluster head is shared equally between all nodes. In this optional case, the algorithm need to exchange information with other nodes in the network to estimate the global energy level. The base case does not require any communication at all, each node decides autonomously whether to become cluster head or not.

The major drawback of the decentralized approach to clustering is that the quality of the placement of the cluster heads is not taken into account. To address this issue, Heinzelman et al. propose LEACH-C as a centralized variant of their clustering algorithm. In LEACH-C, each node transmits its location to a base station which then assigns the cluster heads after calculating the most suitable nodes with a heuristic based on simulated annealing.

LEACH and LEACH-C are evaluated using the network simulator `ns-2`. The results show that both protocols can extend the lifetime of the network when compared to other routing schemes.

Exact and Heuristic Clustering with Centralized Facility Location Algorithms

Furuta et al. [37] improve upon the general architecture of LEACH-C [48], a centralized clustering algorithm for WSNs summarized above. Their formulation of the clustering problem as an uncapacitated facility location problem allows them to derive the optimal solution at the base station of the network. This information is then used to adapt the clusters to current energy levels and prolong the lifetime of the network.

Using the Xpress-MP optimization software to simulate a 100-node network, the authors evaluate their clustering algorithm and compare it with LEACH-C. The metric used is the

survival rate, which is defined as the percentage of alive sensor nodes over all sensor nodes. The results show that the UFLP-based algorithm outperforms LEACH-C and the network remains operational for a longer period of time.

The major drawback of the UFLP-based clustering is the computational overhead of solving the NP-hard UFLP for large networks. In their follow-up work [36], Furuta et al. propose an alternative clustering algorithm method based on Voronoi heuristics. While no detailed complexity analysis is given, experiments show that according to wall clock time the heuristic runs several orders of magnitude faster than the exact solution while retaining comparable improvements in network life time.

2.4.5 Related Work in Other Fields

While not directly related to service placement in ad hoc networks, some recent work in adjacent areas of research also provides some interesting insights. In this section, we briefly summarize selected approaches from the fields of pure facility location theory and distributed databases.

Refinements of Facility Location Theory

Ghosh [39] discusses algorithms to heuristically solve the UFLP. Starting with an initial solution, alternative solutions are evaluated in the neighboring solution space, for which different structures are discussed. As for all approaches based on local search, the main disadvantage is that they may only find a local optimum. Two possibilities to work around this problem are to either use tabu lists to continuously change the neighborhood of all solutions depending on the search history, or to backtrack out of local optima by organizing the solution space into a graph. Both neighborhood structures and algorithms are evaluated with respect to their optimality and computational cost.

Moscibroda and Wattenhofer [81] propose a distributed algorithm to solve the facility location problem and study the tradeoff between communication overhead and quality of the approximation. Their algorithm is based on the assumption of each facility being able to communicate with each client (and vice versa) once in each round of the algorithm.

Distributed Databases

In the field of distributed databases, data replication deals with improving accessibility to data in MANETs. The fundamental idea is that – while for each data item a node can be established as its "owner" – replicating data items mitigates the problems caused by node mobility, e.g., network partitions. Hara summarizes the key problems of data replication as follows [46]:

- **Replica Relocation:** Deals with time, location, method and coordination of allocating replicas of data items, also considering node mobility and load balancing, e.g., with regard to power consumption.
- **Update Management:** Ensures the consistency between the original data item and its replicas, possibly supporting invalidation of replicas and rollbacks of failed updates.
- **Location Management:** Reduces data traffic by choosing appropriate locations for the data items and their replicas in the network with regard to access patterns.

In [80], Mondal et al. propose a scheme for Context and Location-based Efficient Allocation of Replicas (CLEAR). The algorithm collects information from all mobile nodes in a predefined region on a central node within said region. Based on this information, and considering load and mobility of the individual nodes, the central node then selects a node for hosting the replicas. The information which nodes are responsible for which replicas is then distributed to all nodes in the region via broadcast. Finally, queries are processed by each node forwarding the query to the host of a corresponding replica that currently has the lowest load.

2.4.6 Evaluation

As pointed out at the beginning of this chapter, service placement in ad hoc networks is of interest to two communities with different backgrounds. For the researchers of the networking community, service placement is an additional feature for their operating system, middleware, or routing protocol. For researchers with a background in facility location theory, service placement is an opportunity to apply their algorithms to real-world systems. As a consequence proposed solutions differ not only in methodology, but also in their emphasis on different aspects of the resulting system.

The proposals with a background in operating systems, middleware, or routing [11, 14, 37, 48, 49, 66–68, 80, 93] commonly employ an algorithmic approach of iterative migration of existing service instances in combination with some sort of neighborhood exploration [11, 14, 49, 67, 68]. From the theoretical point of view, this procedure resembles a hill climbing algorithm with the inherent disadvantage of potentially only finding locally optimal solutions. Simulative and experimental results, however, indicate that this has only a minor impact in practice [14, 49, 67, 68]. The heuristics employed in the iterative migration process are in some cases tailored to the specific application, e.g., coverage [67, 93], topology [11], group mobility [66], or request volume [85], and would fail for general-purpose service placement. Other approaches rely on centrally collecting data related to the network topology, either of a limited region [80, 93] or the entire network [37, 48]. With regard to the timing of

placement decisions, we can observe that the approaches that rely on local information tend to employ some form of global timing or epochs rather than a timing mechanism related to actual changes in service demand or network topology.

The approaches with a background in facility location theory [1, 36, 62, 81, 85] suffer from the fact that the facility location problem is NP-hard. They work around this by either making assumptions on the structure of the network [85], artificially limiting the size of the problem [62], or by falling back to heuristics [1, 36]. The evaluation of these proposals tends to assume a very simplistic models of ad hoc networks that does not include relevant aspects such as properties of the wireless channel, limited resources on the nodes, or overhead traffic incurred by lower layers of the protocol stack. Generally speaking, the focus is more on the reduction of service-level network traffic, but the traffic required for measurement, signaling, and adaptations is not considered, e.g., [81] requires one network-wide broadcast for each service and each client per round of the algorithm.

Notable exemptions that cross the border between the two communities are [34, 58, 98]. These approaches address either general-purpose service placement directly [34, 58] or as part of placing the components of a hierarchical service overlay [98].

With respect to the three fundamental research questions of service placement raised in Section 1.2.1, it can be concluded that current state-of-the-art research provides answers to the first two questions (*where* to place service instance and *how many* service instances are optimal for the operation of the network). However, there is hardly any quantitative evaluation of these approaches and hence the real benefit of service placement is unknown. The third question (*when* to adapt a service configuration) has, to the best of our knowledge, not been addressed explicitly.

Furthermore, most of the approaches that have been presented in this section do not consider variable service traffic patterns (e.g. regional distribution or changing demand over time) or control traffic overhead in their evaluations. Looking at the bigger picture, the interactions between service placement, service discovery and routing are also widely unexplored.

2.4.7 Candidates for Quantitative Comparison

Given the number of proposals for implementing service placement in ad hoc networks, we have to consider which of the approaches are the most suitable candidates for a quantitative comparison with our own Graph Cost / Single Instance (GCSI) and the Graph Cost / Multiple Instances (GCMI) placement algorithms that we will present in Chapter 5. The ideal candidates should deal with general-purpose service placement, rather than a specialized variant, and operate under similar assumptions about the ad hoc network and the service

Table 2.1: Selection of placement algorithms for the quantitative evaluation

	Centralized placement algorithm	**Distributed placement algorithm**
Centralized service	• LinkPull, PeerPull, and TopoCenter(1) Migration [68] • Tree-topology Migration [85] • SP*i* / GCSI (cf. Sec. 5.4)	n/a
Distributed service	• SP*i* / GCMI (cf. Sec. 5.5)	• ASG / simple and ASG / Event Flow Tree [49]

to be placed as our own work. Furthermore, we need access to an implementation of each placement algorithm that is to be evaluated. This implies that an implementation must either be readily available or it must be feasible to implement the algorithm with reasonable effort. Unfortunately, this criterion rules out some of the more interesting approaches such as [34, 58].

The selection of placement algorithms to be used in our evaluation in Chapter 6 is shown in Table 2.1. For centralized services, we have chosen the LinkPull, PeerPull, and TopoCenter(1) migration [68] placement algorithms as well as the Tree-topology migration [85]. These algorithms make no assumptions about the service and include approaches that do and do not consider the network topology as an input. For distributed services, we have chosen to implement the two placement algorithms presented as part of the ASG system [49]. This choice is primarily due to the very similar focus of this project to our own work, but also due to the simplicity and thorough description of the algorithms. The additional benefit of this choice is that we also get the possibility to compare the performance of distributed algorithms with that of a centralized algorithms such as our own SP*i* / GCMI. We consider other centralized placement algorithms such as LEACH-C [48] to be less interesting since they rely upon the presence of a dedicated entity to run the placement algorithm (cf. Sec. 2.4.1).

2.5 Summary

In this chapter, we have given brief overviews over the fields of ad hoc networking and facility location theory, which are prerequisites for our own work on service placement in ad hoc networks. We have also reviewed and evaluated current approaches to service placement and have motivated our choice for question the which placement algorithms to include in our evaluation in Chapter 6.

Our review of the literature in the field of service placement has shown that this topic is of interest to two distinct communities: networking and middleware research on one side, and facility location theory on the other. We have also observed that both communities lack the tools to implement placement algorithms for side-to-side comparisons. In fact, none of the papers that we reviewed included a comparison of their results with those of any other proposal. For this reason, we took special care to design the SP*i* service placement framework (as presented in Chapter 4) in such a way that it supports implementations of a wide variety of algorithms. We hope that this tool, in particular when combined with our approach to support a variety of evaluation methods (as presented in Chapter 3), will ease the implementation and evaluation of placement algorithms and foster further research in this area.

Chapter 3

Methodology

One of the major challenges in conducting research in the field of ad hoc networking is the dependency on complex and time-consuming experiments to investigate the functional properties of a system or protocol. In order to lessen this burden, the research community frequently resorts to the use of simulations instead of real-world deployments. Simulations have the additional advantage that they make the software development process more flexible due to the possibility to run experiments with simplifying assumptions and the possibility to repeat experiments without any changes to the parameter set. However, the results obtained from simulations are often questionable with regard to their credibility [61].

In order to leverage the advantages of both evaluation methods, i.e., the ease of use of simulations and the credibility of real-world experiments, we propose in this chapter a process that allows us to develop and evaluate a system or protocol using both simulators and real-world platforms. We present a software interface that makes the implementation of a protocol or distributed system portable across network simulators, in particular the network simulator `ns-2`, major operating systems such as Microsoft Windows and Linux, and embedded systems as employed in wireless sensor networks. The two key challenges that we address with our proposal consist of identifying the minimal set of low-level services required for implementing network protocols and of ensuring that the resulting interface can be implemented across all target platforms with identical semantics.

We begin this chapter with a brief overview and motivation in Section 3.1. In Section 3.2, we present the technical details of the proposed software interface and discuss the implementation details specific to two evaluation platforms. Afterwards, we review other approaches that address the same problem and compare them to our own approach in Section 3.3. In Section 3.4, we summarize our findings and conclude.

3.1 Overview

Developing a set of network protocols or, more generally, software components for distributed systems – as required for the implementation of a service placement system – is a complex task. This is especially true if the overall architecture of the system under development is likely to be adapted as more knowledge about the exact tradeoffs in the domain is learned. At the same time, it is our goal for the resulting system to reach a level of maturity and completeness that enables us to evaluate the overall performance in a real-world deployment. Given these two constraints, we need to employ a software development process that is light-weight and flexible, and allows for early and frequent experimentation. Making use of a network simulator to implement, debug, and profile prototypes of the software components adequately matches these needs [16, 47]. However, since the development process should ultimately result in a set of protocols or software components that are deployable on real-world platforms, we need to identify a way to reuse their implementations from the simulator on the real-world evaluation platforms.

For our research project in particular, the impact of service placement on the performance of an ad hoc network can be expected to increase with the size of the network because the placement of service instances gains importance within a growing network topology. Repeatedly deploying an ad hoc network of significant size is impractical, and therefore at least a part of our evaluation will have to be conducted using simulations. In the worst case, the act of switching back and forth between evaluation methods results in the necessity to maintain two implementations of the same software components, one to be used within the simulator and one for the real-world evaluation platform. This problem is similar to the problem of developing software components in simulations and then reusing their implementations in real-world deployments because it once again emphasizes the need to be able to transparently switch between evaluation platforms with minimal overhead as research progresses.

In our previous work [108, 109], we have developed a simulation tool specifically for Wireless Sensor Networks (WSNs) and investigated the differences between simulations and real-world WSN deployments. The key advantage of this simulation tool is that it allows to run applications built from the same unmodified implementation on both the network simulator `ns-2` [31, 84] and on ScatterWeb sensor nodes [97]. As a result, while most development and evaluation still take place using simulations, it is nevertheless possible to verify the results with a real-world deployment without any error-prone reimplementation of algorithms or software components.

The method that we present in this section expands upon this approach. As illustrated in Figure 3.1, we propose to introduce a layer of abstraction between the software components

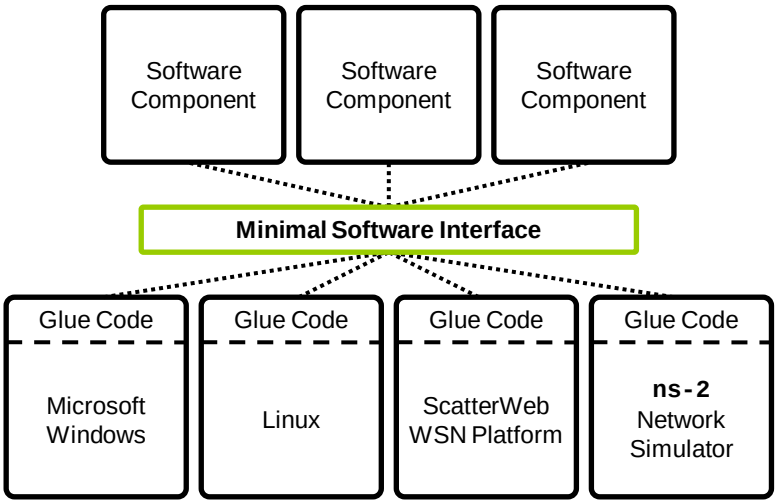


Figure 3.1: Minimal software interface to ensure the portability of software components across evaluation platforms

under development and the evaluation platform. This layer of abstraction exposes a minimal software interface to the software components which has the same semantics across all evaluation platforms. For each evaluation platform, a thin layer of glue code needs to be provided that maps the functionality offered by the interface to the functions and low-level services available on the platform in question.

This approach is relatively straightforward except for the integration of the implementation of a protocol or software component into a simulation environment. The problems that arise in this context are mainly due to the difference in execution models between real-world systems and simulations: A software component usually runs in isolation on real-world systems, either on an embedded device or as a process on top of an operating system. Hence, access to global data structures and control flow within the software component are non-issues. This is, however, not the case for simulations since multiple instances of the same software component are run within the process context of the simulator. Therefore, special care needs to be taken to ensure that the software components are executed in isolation from each other. We discuss these issues in detail and present solutions in Section 3.2.3.

3.2 Portability Across Evaluation Methods

In this section, we present the software interface that serves as a layer of abstraction between the implementation of a network protocol or component of a distributed system and a variety of evaluation platforms. The principal goal of this layer is portability, i.e., it provides exactly the same semantics across a wide variety of evaluation platforms and thus supports

quantitative evaluations of the software components under development on each of them. Our target platforms include major operating systems such as Microsoft Windows and Linux, the firmware of embedded devices such as ScatterWeb sensor nodes, and network simulation tools such as `ns-2`. Due to this wide range of supported platforms, the software interface only provides the bare minimum of functionality required for implementing network protocols. Adding more functionality than the bare minimum would undermine our stated goal of portability since it would be increasingly difficult to implement the layer of glue code in such a way that the semantics of the interface are uniformly preserved across all evaluation platforms.

The only prerequisite for our approach to be applicable is that the software components must be implemented in the C programming language. This is required to support embedded devices with limited resources for which C is the de facto standard programming language. It also facilitates the integration into `ns-2` which is implemented in C++. Given the fact that C is also the de facto standard programming language for major operating systems, this prerequisite does not severely limit the applicability of our approach.

We have previously presented the method for implementing the software interface on sensor nodes of the ScatterWeb WSN platform in [108] and hence focus the discussion of the implementation on the required glue code for POSIX-compliant operating systems and the network simulator `ns-2`.

3.2.1 Software Interface

The software interface encompasses two sets of functions: The first set consists of the functions that must be implemented by the software component under development to be called by the underlying evaluation platform. The second set of functions corresponds to the interface provided for the software component under development to transparently access functionality of the evaluation platform it is currently running on. An overview of all functions is given in Figure 3.2.

The first set of functions, the interface functions to be implemented by the software component under development for calls from the underlying system, is as follows:

`void C_init(C_addr_t local_address, C_addr_t broadcast_address)` – This hook allows the software component to perform initialization and allocation of resources, e.g., global data structures. It also allows to set the local address of this node and the broadcast address of the network.

`void C_start()` – A hook that starts the execution of the software component, e.g., by starting periodic timers.

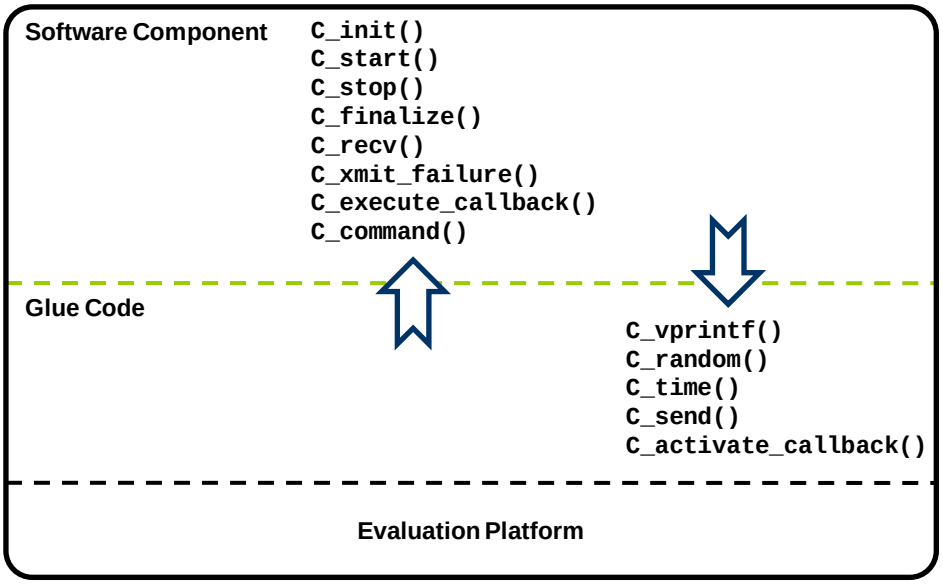


Figure 3.2: Functions of the software interface by architectural component

`void C_stop()` – A hook that stops the execution of the software component, e.g., by stopping periodic timers.

`void C_finalize()` – A hook that allows the software component to deallocate and free resources, e.g., by dropping all remaining packets or requests stored in queues.

`void C_recv(C_header_t* header, const void* payload)` – This function is called whenever a packet is received by the simulated or real network interface of the node. The packet consists of header and payload; the `C_header_t` structure contains the length of the payload.

`void C_xmit_failure(C_header_t* header, const void* payload)` – A call to this function notifies the software component that the transmission of a unicast packet has failed. This may happen due to a simulated packet collision occurring within the network simulator or due to interference between radio transmissions in a real IEEE 802.11 network. The software component may take appropriate responses to link failures, e.g., by purging neighborhood information or updating routing tables. The packet for which the transmission failed is passed as a set of two parameters.

`void C_execute_callback()` – This function is used to implement software timers. It is triggered after a timeout specified by calling `C_activate_callback()` (see below).

`bool C_command(int argc, char** argv)` – A function that is used to pass application-specific commands to the components at run time, e.g., to trigger scripted events

as part of an evaluation run. The parameters correspond to the command and its arguments in a format similar to that of standard command line arguments. The return value of this function indicates whether the command is implemented in this particular software component.

The second set of functions, the functions provided by the underlying system for calls from the software component under development, is as follows:

`void C_vprintf(const char *format, va_list ap)` – A function that resembles the `vprintf()` provided by the C standard library. It implements a platform-specific output mechanisms for formatted strings. For most evaluation platforms, the glue code will map this function directly to `vprintf()`. For simulations, however, it is necessary to direct the output through the logging mechanism of the simulator.

`double C_random(double max)` – This function provides access to the platform-specific random number generator and returns a uniformly distributed random number from the interval $[0, \texttt{max}]$.

`C_time_t C_time()` – This function returns the current time of the evaluation platform. `C_time_t` is a platform-independent representation of time.

`void C_send(C_header_t* header, const void* payload)` – This function allows to transmit a single packet over the simulated or real network interface. Packets are transmitted to all other nodes in the link layer broadcast domain of the current node and are subject to packet loss due to problems with the radio transmission. Packets are not routed automatically over multiple hops, except for very specific setups of `ns-2` that are used to compare the performance of routing protocols (cf. App. A).

`void C_activate_callback(C_time_t time)` – This function specifies the time at which the function `C_execute_callback()` is to be called (see above).

In this list of functions we have omitted the functions that implement the dynamic allocation of memory. We have found that the implementation provided by the C standard library is sufficiently platform-independent, including on simulators, and that a reimplementation in the layer of glue code is not required.

This software interface is designed to allow for the comprehensive implementation of network protocols or components of distributed systems, while requiring only minimal glue code to map its functions to each of the targeted evaluation platforms. It thus comprises only a minimal set of low-level functions. Additional functionality such as periodic timers, packet handling, routing, and reliable signaling between nodes is to be implemented on

top of this interface. Our implementations of WSN-specific experiments [109] and the SP*i* service placement framework (cf. Chp. 4), both of which utilize this interface, proof that the interface is sufficiently comprehensive to support the implementation of non-trivial software components. In the discussion of the platform-specific glue code in the following two sections, we illustrate that support for multiple evaluation platforms is easily feasible.

3.2.2 Integration into POSIX-compliant Operating Systems

The glue code required to support the software interface on a POSIX-compliant operating system is quite straightforward. Instances of the software component run in their own process context and there is no need for bridging between programming languages. Most functions of the interface can be mapped to a function of the C standard library.

The sole difficulty lies in handling the asynchronous arrival of packets that are to be processed. This is achieved by an appropriate parameterization of the `select()` system call. Listing 3.1 contains a simplified version of the main loop of the glue code. The global variable `udp_socket_` is initialized with the socket descriptor of the UDP socket used for transmitting and receiving packets; `callbacks_` is a sorted list that contains information about when callback functions of the software component are to be executed. Depending on whether a callback is pending, `select()` is either called with or without a timeout value (cf. lines 21 and 25). The return value of `select()` indicates whether a packet has been received and needs processing (cf. lines 31 and 32). Any pending callbacks are executed independently of the arrival of packets whenever `select()` returns (cf. lines 34-39).

Note that the correct deadlines for executing callback functions may be missed under very high load. This is acceptable since our interface does not offer real-time guarantees to the software component. The delay until the execution of the next callback is adjusted if it is negative (cf. lines 14 and 15). We do, however, not skip the call to `select()` since we need to avoid dropping packets even when running under high load.

3.2.3 Integration into the Network Simulator `ns-2`

Implementing the layer of glue code to support execution of the same software components on the network simulator `ns-2` is more challenging. This is due to a variety of reasons:

- The core of `ns-2` and most of its components are implemented in C++, while the software components we are dealing with are written in C.
- All components of `ns-2` are statically linked into the `ns-2` binary at compilation time. Hence, we need to link the compiled object code of the software component into the `ns-2` binary, possibly dealing with clashing symbols.

Listing 3.1: Simplified main loop of the glue code for POSIX-compliant operating systems

```
while(running) {
  fd_set incoming_sockets;
  int ret = 0;

  FD_ZERO(&incoming_sockets);
  FD_SET(udp_socket_, &incoming_sockets);

  if(!list_empty(callbacks_)) {
    struct timeval timeout;
    C_time_t time_until_next_callback =
      ((callback_t*) list_head(callbacks_))->time - C_time();

    // Adjust timeout to next callback.
    if(time_until_next_callback < 0.0)
      time_until_next_callback = 0.0;
    timeout.tv_sec = time_until_next_callback;
    timeout.tv_usec =
      (time_until_next_callback - ((C_time_t) timeout.tv_sec)) * 1000000;

    // Wait for either a packet to arrive or for the timeout to expire.
    ret = select(udp_socket_ + 1, &incoming_sockets, NULL, NULL, &timeout);

  } else {
    // Wait for a packet to arrive.
    ret = select(udp_socket_ + 1, &incoming_sockets, NULL, NULL, NULL);
  }

  if(!running)
    return;                           // Return, if aborted.

  if(ret == 1)
    handle_packet();                  // Handle packet, if any.

  if(
    !list_empty(callbacks_)
    && (((callback_t*) list_head(callbacks_))->time < C_time())
  ) {
    callback_expire();                // Handle callback, if any.
  }
}
```

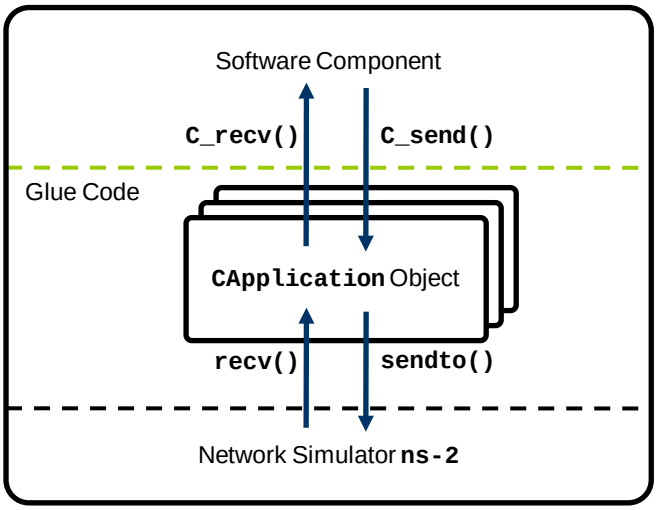


Figure 3.3: Interaction between the software component, the glue code layer, and `ns-2` at run time

- The software components we are dealing with are written to run in isolation from each other, either within their own process context or on a dedicated embedded processor. When simulated on `ns-2`, several instances of a software component find themselves running in the same process context, i.e., in the same address space. Hence the global variables need to be replicated for each instance of the software component and access to the global variables from with the C code must be redirected to the appropriate instance.

We have implemented the layer of glue code between `ns-2` and the software interface as a C++ class. The `CApplication` class implements the software interface in C++ and takes care of the interaction with `ns-2`. Figure 3.3 illustrates the interaction between `ns-2`, the `CApplication` objects, and the implementation of the software component at run time. For the simple example of receiving and sending a packet, the diagram illustrates how `CApplication` translates a call to its `recv()` (which is part of the interface of the `Application` class of `ns-2`) to the appropriate function of the software component. When sending a packet, the software component calls the `C_send()` function of the `CApplication` object, which is in turn translated into the `sendto()` function of the `ns-2 Agent` class.

Linking C Code into `ns-2`

The most difficult part of integrating C code into `ns-2` is the glue code for the `C_send()` function: While implemented as method of the `CApplication` class, it must look just like a C style function from the point of view of the implementation of the software

Listing 3.2: Excerpt from the implementation of the software interface on `ns-2`

```
#include "c_application.h"

using namespace C;

// Variable to point to current instance when accessing fields from C code.
CApplication* CApplication::instance_ = NULL;

void CApplication::recv(Packet* pkt, Handler*) {
  unsigned char* p = ((PacketData*) pkt->userdata())->data();
  C_header_t* header = (C_header_t*) p;
  void* payload = NULL;

  if(header->payload_length > 0)
    payload = p + sizeof(C_header_t);

  // Set pointer to current instance before calling into C code.
  instance_ = this;
  C_recv(header, payload);
}

void CApplication::C_send(C_header_t* header, void* payload) {
  ns_addr_t dst = {header->next_hop_address, agent_->port()};
  unsigned int packet_size = sizeof(C_header_t) + header->payload_length;
  PacketData* data = new PacketData(packet_size);

  // Assemble packet for transmission via the simulator.
  memcpy(data->data(), header, sizeof(C_header_t));
  if(header->payload_length > 0)
    memcpy(data->data() + sizeof(C_header_t), payload, header->
        payload_length);

  agent_->sendto(packet_size, data, NULL, dst);
}

// Redirect calls into C code to C++ class functions.
#define C_recv CApplication::C_recv
[...]

// Redirect calls to interface from C code to C++ instance functions.
#define C_send CApplication::instance_->C_send
[...]

// Redirect accesses to global C variables to C++ instance variables.
#define C_state CApplication::instance_->C_state
[...]

// Include C code files into C++ class definition.
#include "../../interface.c"
[...]
```

component. Further, as there may be multiple `CApplication` objects at run time, the software component needs to call the method on the correct object.

To achieve this goal, we use the C preprocessor. Listing 3.2 shows a shortened version of the `CApplication` class definition. In line 39, we use the `define` directive to convert a call to `C_send()` to a call to the method of the current instance of `CApplication`. The static class variable `instance_` is adjusted to point to the object corresponding to the current node whenever control is passed to the implementation of the software component, e.g., when receiving a packet (cf. lines 17 and 18). Similar conversions are required for all functions of the software interface. Note that the names of the interface functions must be unique strings in the application code as otherwise the C preprocessor would incorrectly replace other strings in the implementation of the software component.

Similar conversions are also required for all global variables of the implementation of the software component (cf. line 43). For function calls from the glue code into the software component it is sufficient to adjust the functions to be included in the class definition (cf. line 35). Once the `define`s for all redirections are in place, the C preprocessor is used to include the C code of the implementation of the software component into the C++ class definition of the `CApplication` class (cf. line 47).

Admittedly, lines 34 to 48 use language features in unusual ways. However, the pattern is quite simple and largely independent from the code of the software component. In this light, we argue that it is an acceptable solution for the problem at hand.

Connecting the Network Stack

Now that we have established the means of interaction between the `ns-2` simulator and the software component, we shift our focus on the details of this interaction taking place: A shortened version of the `C_send()` method of the `CApplication` class is shown in lines 21 to 32 of Listing 3.2. This method is called by the software component in order to transmit a packet over the network interface. The method allocates a `ns-2` data payload object in line 24 and then copies header and payload of the packet from its parameters into the payload of the `ns-2` packet in lines 26 to 29. Line 31 hands the packet down to the core of the `ns-2` simulation.

Lines 8 to 19 of Listing 3.2 show the `recv()` method of the `CApplication` class, which is invoked by the `ns-2` simulation whenever a packet has been received by the lower networking layers of the simulated node. In line 17, the `instance_` variable is set to point to the object of the current `CApplication`. As stated above, `instance_` is a static field of the `CApplication` class, that gives the C code of the application a reference to the object it is part of. Therefore, the `instance_` variable needs to be updated whenever the `ns-2`

simulation transfers control to software component implemented in C. In line 18, the newly received packet is passed to the software component.

With the ability to send and receive packets, the crucial functionality of the glue code is complete. There are a few additional issues, such as dealing with namespaces and predefined data types, for which we omit a discussion for brevity. Our layer of abstraction between software components and evaluation tools is implemented as part of the SP*i* service placement framework, and the interested reader is referred to the full documentation as well as the BSD-licensed source code both available at http://cst.mi.fu-berlin.de/projects/SPi/.

3.3 Comparison with Other Approaches

We have previously reviewed current approaches that ease the integration of implementations written for network simulators into real-world software stacks (and vice versa) in [59]. In this section, we provide a brief summary of our findings. We begin by introducing a set of criteria and requirements to highlight advantages and disadvantages of each approach and then continue with a more in-depth discussion of each of them. Afterwards, we compare the current state of the art with our own proposal.

3.3.1 Requirements and Evaluation Criteria

In order to classify existing approaches to software integration, it makes sense to focus on the user perspective on the framework, i.e., the requirements of the developer or researcher who is interested in implementing and evaluating a new network protocol or component of a distributed system. The major concerns fall into three categories:

Usability: The usability aspects describe how difficult a framework is to use, in particular for new developers. This includes the initial learning curve as well as the repetitive work required for running simulations. The *complexity* of the software interface may range from the very narrow interface of packet and timer handlers to an entire kernel or firmware Application Programming Interface (API). The more complex the API is, the harder it is for new developers to port their system or protocol in reasonable time. To mitigate this effect, one can leverage the *familiarity* of developers with well-known APIs. Reimplementing a well-established API in the glue code is preferable to starting from scratch. The drawback of this approach is that established APIs are commonly platform-specific and tend to be rather complex. The crucial question is, however, that of *integration*: If an existing implementation is to be integrated into a simulator, two problems arise: First, in most cases a number of initial changes need to be applied to

the implementation, and second, an ongoing effort may be required while continuing work on the project. Some frameworks include special-purpose tools to help the developer with these issues.

Correctness: The correctness of a framework depends on in how far the software interface has identical functional properties on all supported platforms and in how far the lower level components, i.e., those below the interface, are parameterized appropriately. One key aspect of correctness is *consistency*, i.e., the semantics of the software interface provided by the glue code on the simulator should be as close as possible to those of the real platform. Consequently, any code running on top of the interface should be completely agnostic about whether it is being executed as part of a simulation or on a real system. A second major aspect of correctness is the *accuracy*. Depending on the focus of the simulation, it may be desirable to evaluate low-level metrics, e.g., the radio signal strength or the number of bytes transmitted. Especially for high-level software interfaces, this requires the framework to properly model and implement all underlying components. Alternatively, frameworks may also decide against providing low-level metrics for the sake of speed and simplicity.

Performance: The performance of a framework relates to how efficient in term of run time overhead the integration of the software component into the simulator is handled. Integrating the implementation of a system or protocol into a simulator usually imposes a run time overhead over a native implementation within the simulator. This is due to only part of facilities provided by the network simulator being exposed to the implementation of the software component via the software interface. Additionally, running several instances of a non-native system or protocol within the simulator may require additional memory management.

The key aspect that influences all of these criteria is the level of abstraction provided by the integration framework, i.e., at which level of abstraction is the glue code inserted to translate between the different evaluation platforms. Obviously, if a rather high level of abstraction is chosen, e.g., the UNIX socket API, then it is more challenging for the framework to guarantee correctness and accuracy of the simulation. In contrast, if a low level of abstraction is chosen, e.g., Data Link Layer (DLL) frames, then inaccuracies induced by the simulated routing and transport layers become a non-issue. The choice of which layer of abstraction (and consequently which framework) to use depends on the focus of the network protocol or distributed system under development.

3.3.2 Exemplary Frameworks

In the following, we briefly describe four software integration frameworks and sketch their focus and internal design.

Network Simulation Cradle The aim of the Network Simulation Cradle (NSC) [52] is to integrate existing kernel-space implementations of networking stacks into the `ns-2` network simulator. The approach is to parse the C code of the network stack, replace the declarations of global variables with per-node instance variables and compile the code as a shared library. As part of this library, kernel-level interfaces are mapped to `ns-2` via a layer of architecture-specific glue code. The library can then be linked against `ns-2` and run simulations of the kernel-space protocol implementations. The network stacks of both Linux, FreeBSD and OpenBSD have been successfully integrated into `ns-2`. Integration is supported in part by the parser that semi-automatically handles global symbols. However, exactly which symbols need to be adapted needs to be set manually for each stack.

TOSSIM TOSSIM [63] is a simulator with the specific goal of transparently running TinyOS [50] applications. It does not follow the approach of integrating existing code into an existing simulator via glue code, but rather implements a new simulator from scratch. The component-oriented software architecture of TinyOS greatly aids integration into a simulator: Hardware abstraction is provided by software components with specific interfaces which are enforced at compile time. For the simulation, these components are replaced with pure software counterparts that model the behavior of the real hardware. The compiler is used to replace global variables with arrays indexed by the node ID in the code that is to be simulated. The simulation is event-driven and radio communication is modeled with bit error rates for each unidirectional link. The key advantage of TOSSIM is its seamless integration with TinyOS and the nesC programming language. As TinyOS applications are already inherently structured into components, it is relatively easy to replace the hardware abstraction layer, i.e., the components that interact with the hardware, with a different, simulated one. On the other hand, the radio model is quite simplistic, e.g., it does not accurately describe interference caused by simultaneous transmissions. The simulator is also missing some other features, e.g., mobility models are not part of TOSSIM.

Avrora Avrora [104] is a full-system emulator that models the hardware of typical sensor nodes such as the Mica2, Mica2dot, and MicaZ motes [112–114], including micro-controller, radio chips and sensors. It is binary compatible with these platforms and executes a sensor node operating system including its device drivers and applications without the need for cross-compilation. Avrora provides highly realistic results in terms of timing and memory

usage, as it executes the binary compiled for the sensor node itself. Overall, full-system emulation provides a very detailed insight into a communication system, operating system, and application.

COOJA While the previous integration frameworks were always tied to one particular level of abstraction, COOJA [86] explicitly supports simulation at different levels and even combining multiple levels in the same simulation. More precisely, COOJA supports systems or protocols implemented in Java specifically for the simulation, code written for the Contiki operating system [28] and machine code compiled for the ScatterWeb ESB sensor node [97]. The simulator core is a in-house development and supports simple unit-disc models for radio propagation as well as ray tracing. The method used for integration depends on the level of abstraction. However, in contrast to other approaches, it does not require changes to the source code. Instead, COOJA swaps the content of the memory region in which the global variables are located based on which node is currently active. By supporting simulations across several levels of abstraction, COOJA allows for additional flexibility during the design process of a new system or protocol. It is up to the user to decide which part of the system should be evaluated at which level of abstraction. While no changes to the implementation of the software component are required, the way global state is handled in COOJA incurs additional run time overhead as compared to the other frameworks for software integration.

3.3.3 Comparison

None of the frameworks presented in the previous section is equally suitable for all use cases. In fact, the choice of framework will in most cases be dictated by which platforms need to be supported. For the sake of discussion, we ignore this fact and present a comparison of all four approaches and our own approach in Table 3.1.

Of all five approaches, Avrora and NSC focus most on correctness, the latter doing so at a comparatively high level of abstraction. This is more challenging to achieve as compared to frameworks that provide a level of abstraction closer to the hardware. Furthermore, NSC and our approach are the only ones to build upon a well established network simulator, namely `ns-2`, as opposed to implementing the simulator and low-level components from scratch. The main goal of TOSSIM is to provide a development sandbox for TinyOS applications. As a hardware emulator, it is comparatively easy for Avrora to achieve very trustworthy results, however, this comes at the price of a significant run time overhead. Finally, COOJA combines the advantages of TOSSIM and Avrora for the Contiki and ScatterWeb ESB platforms and leaves it up to the user to fine-tune the simulation. Our

Table 3.1: Comparison of integration frameworks

	NSC	**TOSSIM**	**Avrora**	**COOJA***	**Our proposal**
Abstraction†	NET	PHY	HW	HW/OS/NET	**NET**
Usability	0	+	+	+	**0**
Complexity	-	0	-	-/0/+	**+**
Familiarity	+	+	0	0/+/-	**-**
Integration	0	++	++	++/+/++	**0**
Correctness	+	0	++	+	**+**
Consistency	++	+	++	+/+/+	**++**
Accuracy	0	-	++	+/0/-	**0**
Performance	+	+	-	--/0/++	**++**
Supported Platforms	Linux FreeBSD OpenBSD `ns-2`	TinyOS simulator‡	Mica2 Mica2dot MicaZ	Contiki ScatterWeb simulator‡	**Windows** **Linux** **`ns-2`** **ScatterWeb**
Maintenance	-	0	-	-	**0**

* Supports variable levels of abstraction. † Level of abstraction of the interface: HW – full system emulation; PHY – physical layer emulation; NET – emulation of a networking subsystem; OS – reimplementation of a complete operating system interface. ‡ Refers to a custom-build simulator that is only used by this particular tool.

approach has the clear advantage that, due to the minimal software interface it provides, it is able to support the widest range of evaluation platforms.

Looking at the internals, NSC, TOSSIM and our approach are alike in that they preprocess the implementation of the software component in order to integrate it into the simulator, while this is not required for Avrora and COOJA. Only TOSSIM and our approach result in a comparatively low maintenance burden on the framework developer, while the other three approaches have to ensure consistency and accuracy for more complex software interfaces.

3.4 Summary

There are numerous advantages of developing a network protocol or distributed system on top of a software interface that isolates the components under development from the evaluation method. This approach allows the researcher to combine the strengths of various evaluation tools without being hindered by their respective weaknesses. For our project, it

eliminates the overhead of switching evaluation platforms and thus expands our flexibility when evaluating the SP*i* service placement framework and its algorithms.

More specifically, the work of setting up a simulation is trivial when compared to the work required for deploying a large number of nodes in order to test a protocol. It is even possible to simulate very large ad hoc networks, for which it would otherwise be impossible to procure enough real nodes. On a smaller scale, the development of small software components is faster because the development cycle of implementing, compiling and testing does not involve the time-consuming act of deploying the binary images onto the nodes. Perhaps even more importantly, debugging of large distributed applications, such as routing or load balancing algorithms, is supported by tracing events in the network and the availability of standard debugging tools.

Software integration frameworks have been used to validate the implementation of low-level components in simulations and to ease the development process of network protocols and distributed systems. In [53], the authors of NSC use their framework to generate traces of the network stacks of Linux, FreeBSD and OpenBSD and compare these traces to those gathered from an emulation testbed running the same stacks natively. In [109], software integration is used to evaluate the accuracy of the radio propagation models as implemented in the `ns-2` network simulator. In Appendix C, we present a comparison of the results specific to our work on service placement that were obtained using different evaluation methods.

Chapter 4

The SP*i* Service Placement Framework

In this chapter, we present the SP*i* service placement framework [111] as our contribution towards enabling service placement in ad hoc networks. Our framework is applicable to the placement of both centralized and distributed services as defined in Section 1.1.3. It supports both the Graph Cost / Single Instance and the Graph Cost / Multiple Instances algorithms for the placement of centralized and distributed services as well as several algorithms as proposed in the literature. Furthermore, we tightly integrate the process of placing service instances with the related problems of service discovery and routing. The overall goal is to reduce the cost of implementing placement decisions, thus making service placement a viable alternative to traditional architectures.

This chapter is structured as follows: In Section 4.1, we begin with a high-level overview of the components and the mode of operation of the SP*i* service placement framework, and then proceed to discuss the fundamental design considerations in Section 4.2. Afterwards, we continue to present the components in more detail in Section 4.3. We then discuss the process of migrating a service instance from one node to the other in Section 4.4 and propose various methods for reducing the communication overhead this process incurs. We conclude this chapter with a brief summary in Section 4.5.

When presenting the components and algorithms, we limit the discussion of the implementation to the most interesting aspects. For more details, the interested reader is referred to the full documentation of the SP*i* service placement framework as well as the BSD-licensed source code both available at `http://cst.mi.fu-berlin.de/projects/SPi/`.

4.1 Overview

The SP*i* service placement framework implements a novel approach to service placement in ad hoc networks. With its two purpose-built placement algorithms, it optimizes the number and the location of service instances based on usage statistics and a partial network topology derived from routing information. The system only requires minimal knowledge about

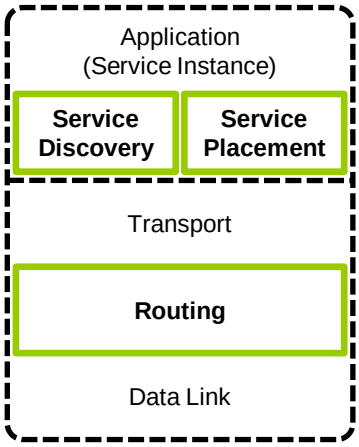


Figure 4.1: Main components of the SPi service placement framework

the service it is tasked with placing in the network. It is unique in explicitly considering the communication between service instances that is required to synchronize shared data. Furthermore, our system implements a cross-layering approach to take advantage of the interdependencies between service placement, service discovery and the routing of service requests to reduce network overhead.

The three main components of SPi are depicted in Figure 4.1. The service placement middleware is active on each node which is currently hosting a service instance. It is tasked with monitoring the usage of the service and with adjusting the service configuration, i.e., the set of nodes to host the service instances, when necessary. The service discovery component locates the most suitable service instance to provide a service for a client. The routing component implements an enhanced routing protocol that also provides information about the current network topology to the service placement middleware. These components are discussed in detail in Section 4.3.

The service placement middleware supports several placement algorithms presented in the literature. Additionally, we propose two new algorithms developed specifically for the SPi framework: For the placement of centralized services, we propose the Graph Cost / Single Instance algorithm (GCSI). It monitors the network traffic between the service instance and its clients, combines it with information about the network topology, and calculates the optimal host for the service instance. For the case of placing a distributed service, we propose the Graph Cost / Multiple Instances algorithm (GCMI). It is executed on a dynamically assigned coordinator node, to which local usage statistics and information regarding the regional network topology are transmitted from all service instances. The algorithm calculates the optimal service configuration, establishes a set of actions required

to adapt the current to the optimal configuration, and issues these actions as commands to the nodes that currently host service instances. Both algorithms rely on accurate and timely information about the current network topology. We present the required infrastructure as part of Section 4.3.1. Both the GCSI and the GCMI algorithms are presented in detail in Chapter 5.

In order to ease the replication and migration of service instances between nodes in an ad hoc network, the SP*i* service placement framework follows a cross-layering approach. This is the fundamental prerequisite for making layer-specific information available to components that traditionally reside on other layers of the ISO/OSI reference model. The prime use of this approach in our framework is the sharing of information related to the network topology between the routing component and the service placement middleware. Other uses are mostly related to protocol optimizations with the goal of reducing the overhead incurred by moving service instances between nodes. Theses uses of cross-layering are discussed in more detail in Section 4.4.4.

4.2 Design Considerations and Rationale

The fundamental design of the SP*i* service placement framework is motivated by two key observations. First, changing the configuration of a service is an operation that is both costly in terms of network traffic as well as macroscopic when compared to other decisions that commonly affect the configuration of ad hoc networks. And second, the signaling overhead required for operating any type of service placement system (SPS) has a non-negligible impact on the overall performance of the network.

The first of these two observations relates to the cost and the relative impact of service placement decisions. For instance, when compared to a routing decision, a service placement decision affects many if not all nodes of the network, while in contrast the choice of a routing path only affects the nodes between the two end points of the route. More precisely, the cost of changing the configuration of a service comprises

1. the network traffic caused by transferring the state and possibly also the binary program file of a service instance from a current host to a new host, including the traffic required for establishing and maintaining routes between old and new hosts,

2. the traffic caused by each client having to locate and select a new service host, and

3. the traffic caused by the client nodes establishing new routes to the service host of their choice.

Since these operations include bulk data transfer as well as partially flooding the network (depending on the choices of service discovery and routing protocols), it is hard to underestimate the cost of this process and its immediate impact on network performance. Similarly, the decision of changing the service configuration results in the reconfiguration of the ad hoc network at a large scale: Information about the location of service instances needs to be updated, new routes have to be found, and new connections between clients and service instances have to be established. This not only affects multiple components across the network stack, but at the same time also multiple, if not all, nodes of the network. Thus, changing the configuration of a service should also be considered as having a very high impact on the overall configuration of the network, especially when compared to other common decisions, e.g., which radio channel to use to communicate between two nodes, on which path to route a packet, or which set of parameters to use for a data transfer.

The second of the two observations is concerned with the cost of operating an SPS, even if no change in the configuration of the service is currently required or underway. As pointed out in Section 1.1.2, one possible goal of an SPS is to reduce the bandwidth required for service provisioning. Obviously, any overhead caused by an SPS itself, may it be for signaling between service instances or for gathering information as input for the placement algorithm, runs contrary to this goal. In fact, there is tradeoff between the quality of the placement decision and the cost (in terms of bandwidth used) incurred by collecting the information to base this decision upon: If little or no information is collected, the placement decision is inexpensive but probably of low quality; if large amounts of information are collected, the placement decision is most likely of high quality, but also very costly.

In light of these two observations, we derive the following three requirements on the mode of operation of any SPS:

Service placement algorithms should aim for few, high-quality decisions. As changes in the configuration of a service are both costly and have a high impact on the operation of an ad hoc network, they should occur as rarely as possible. From this follows that the quality of these placement decisions should be as high as possible, i.e., the choice of the number of service instances and their location should approximate the optimum given the network topology and the current service demand.

Adaptations of the service configuration should be timed with caution. If adaptations of the service configuration are rare, the point in time at which they occur gains importance. It may be undesirable to adapt a service configuration immediately, if there are only minor expected savings. However, if a sudden and large change in either network topology or service demand occurs, there should not be an excessive delay before adapting the service configuration to the new optimum.

Table 4.1: Evaluation of design alternatives for placing a distributed service

	Distributed algorithms		Centralized algorithm
	Local heuristics	Consensus	
Quality of placement decisions	–/o	o/+	+
Timing of placement decisions	–	o/+	+
Required signaling overhead	+	–	o

The signaling of the SPS should be as light-weight as possible. As one of the goals of service placement is to reduce the overall bandwidth used, an SPS must itself require as little bandwidth as possible for its operation. In this context, the signaling includes both the traffic required to communicate among service instances as well as the process of collecting input data for the placement algorithms.

Considering the design space for service placement systems as already discussed in detail in Section 2.4.1, we can now evaluate the three viable design alternatives in light of these requirements. This evaluation is visualized in Table 4.1. As illustrated, systems that merely rely on local information cannot achieve the same level of quality in their placement decisions as systems with a more global view. This is especially true for the question regarding the optimal number of service instances. Similarly, it is safe to assume that systems, which by their design rely on more information being available, can also make more informed and thus more appropriate decisions on the proper timing of service adaptations. This assumption is supported by the fact that the majority of the systems proposed in the literature, which merely rely on local information, employ some form of global timing or epochs rather than a timing mechanism related to actual changes in service demand or network topology (cf. Sec. 2.4.6). Finally, systems relying on local information have their clear advantage in the fact that they cause only minimal signaling and data collection overhead. At the same time, systems based on distributed algorithms have a clear disadvantage in this area, as they require high volumes of information to be exchanged between nodes in order to achieve high-quality placement decisions. Centralized systems, in contrast, can implement a more streamlined process of gathering the required information and thus require less signaling.

Given the advantages, disadvantages and tradeoffs between the fundamental architectural alternatives for service placement systems, we have opted to design the SP*i* service placement

framework with a central controlling entity in mind. In particular, the GCMI algorithm (cf. Sec. 5.5) is implemented with a dynamically assigned coordinator node. This is, however, just one alternative supported by the framework, as fully decentralized placement algorithms are supported as well.

Employing a centralized system to control parts of an ad hoc network has several well-known drawbacks, especially regarding properties such as scalability and resilience against failure of the node that hosts the central entity. Both problems can, however, be mitigated within the architecture we are proposing: Scalability can be achieved by adding one or multiple layers of regional coordinator nodes between the global coordinator and the service instances. Reliability can be improved by electing a new coordinator among the service instances in case the current one fails or leaves the network. For these reasons and in light of the quantitative evaluation that we will present in Chapter 6, we argue that a centralized architecture is indeed the superior approach to designing a service placement system.

4.3 Components

The three principal components of the SPi service placement framework are the service placement middleware, the service discovery component, and the routing component. Service discovery and routing in ad hoc networks are areas of research in their own right, and there is an extensive body of work on each of them, e.g., [12, 17, 19, 55, 70, 87, 93, 95]. The reason for including them in the design of an SPS is twofold: First, any placement algorithm must encode knowledge about the mode of operation of these components, e.g., it must anticipate the routes that will be established between clients and service instances. And second, commonly used approaches to service discovery and routing need to be adapted if used in the context of networks that employ service placement in order to allow for efficient operation. If a service placement system relied on conventional mechanisms for establishing routes and locating services, a change in the configuration of a service would cause major disruptions to the operations of the network. For instance, migrating a service instance from one node to another renders the locally cached information about the location of services invalid and also obsoletes a non-negligible subset of the routes stored in the routing table. In order to counter these effects, traditional approaches to both problems need to be adapted if service placement is to be used viably in an ad hoc network.

In the following, we describe each of the three components of SPi in detail, pursuing a bottom-up approach. An overview of the components with their respective subtasks is depicted in Figure 4.2.

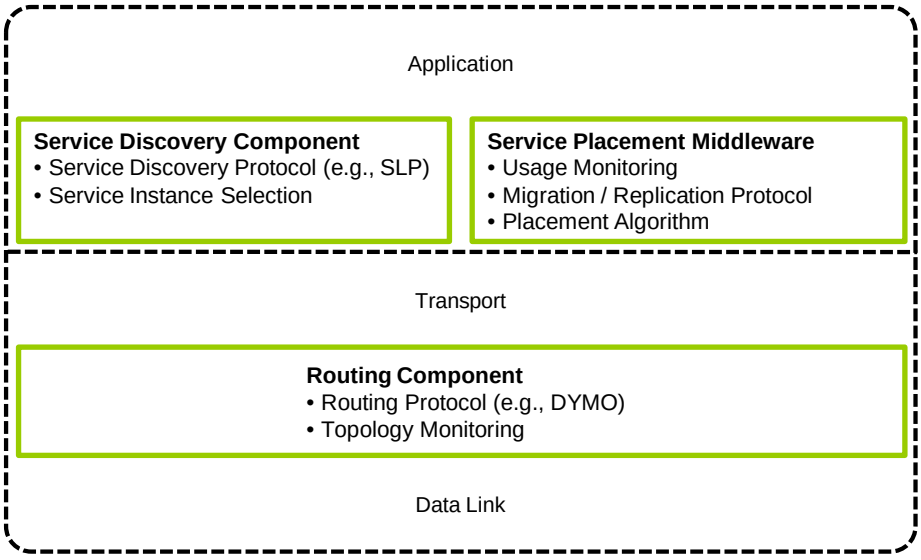


Figure 4.2: Components of the SP*i* service placement framework and their subtasks

4.3.1 Routing Component

The task of the routing component is widely equivalent to that of traditional network layer protocols, i.e., it provides a best-effort datagram service between all nodes of the ad hoc network. This task encompasses the subtasks of locating nodes, establishing a path between sender and receiver, and maintaining or adjusting this path in light of varying link quality and node mobility.

Additionally to mere packet delivery, the routing component employed as part of the SP*i* service placement framework needs to fulfill a second task: Some of the algorithms implemented in the service placement middleware make use of information regarding the network topology, i.e., the set of nodes taking part in the network and the set of radio links between these nodes that are potentially available for routing packets. Hence, the routing component has to map the network and provide this information to the other components in a timely and accurate manner.

Routing Protocol

The basic choice of which routing protocol to employ depends primarily on the envisioned area of application of the ad hoc network. It is generally desirable for an SPS to work with a variety of routing protocols. From an implementation standpoint, there are of course certain types of routing protocols for which the integration with the other components of an SPS is more straightforward.

Routing protocols are generally subdivided into two classes: Proactive protocols such as the Destination-Sequenced Distance Vector (DSDV) protocol [89] or Optimized Link State Routing (OLSR) [19], operate by periodically exchanging neighborhood information between all nodes in the network. These protocols are best suited for networks with low node mobility, sufficient processing and energy resources on each node, and continuous data traffic. In contrast, reactive routing protocols such as Dynamic Source Routing (DSR) [55] or Ad hoc On-Demand Distance Vector (AODV) routing [87], establish routes only on demand. These protocols are more suitable for networks that are subject to a rapidly changing topology, consist of resource-constrained nodes, or run applications with merely sporadic communication patterns. Thus, proactive routing protocols tend to be employed in mesh networks (with deployments ranging from buildings to municipal areas) and in sensor networks tasked with high-fidelity monitoring. Reactive routing protocols are preferred in vehicular networks and sensor networks that respond to rarely occurring external stimuli. An overview of other routing protocols, including hybrid variants of these two classes as well as examples of other, less wide-spread classes is available in [26].

From the perspective of an SPS, it is advantageous if the routing protocol employed in a network automatically gathers information about the network's physical topology, i.e., the availability of radio links between pairs of nodes. This is generally the case for proactive protocols, but not for reactive protocols. Therefore, the integration of a proactive protocol into an SPS requires fewer changes to the implementation of the protocol as compared to a reactive protocol. In fact, the only two changes required for integrating a proactive routing protocol into an SPS are 1) to provide an interface for the other components to access the information about the network topology, and 2) to fine tune the timing parameters to ensure that the information is suitably up-to-date given the requirements of the service placement algorithm.

The changes required to integrate a reactive routing protocol into an SPS are more numerous. Additionally to the two items required for proactive protocols, reactive protocols also need to be extended with the functionality of gathering information about local neighborhoods and routing paths and the means to transfer this information between nodes. Implementing this process involves several interesting tradeoffs which we discuss in the next section.

Network Topology Monitoring

If the functionality of monitoring the network topology is to be added to an existing routing protocol from scratch, as it is generally the case for reactive protocols, there are four questions that need to be addressed:

1. By which means should nodes exchange topology information?
2. Which nodes should participate in gathering information?
3. How much information is to be gathered?
4. How often do nodes need to exchange this information?

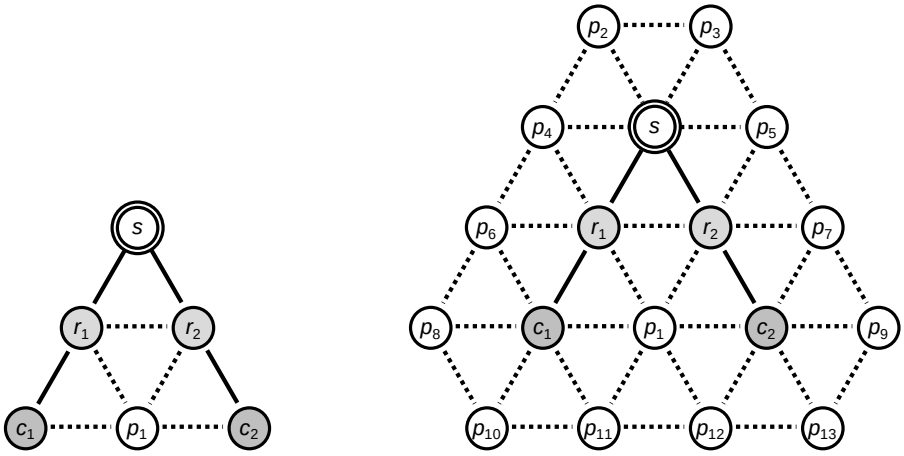
While the first question is rather a matter of implementation, the other three questions share the same underlying basic tradeoff: More timely and more exhaustive information about the network topology allows for higher quality placement decisions, but at the same time the cost of gathering this information reduces the overall benefit of placing service instances.

When answering the latter three questions, it is important to keep in mind that in the context of an SPS, and as opposed to proactive routing protocols, the goal is not to construct a map of the entire network. Instead, we can built upon on the known goal of the SPS of placing the service instances efficiently in the network, i.e., in the close vicinity – or rather within – the regions of the network in which there is client demand for the service that the SPS is tasked with placing. Hence, we are mostly interested in the topology of exactly these regions with high service demand, and can use this knowledge to optimize the process of monitoring the network topology.

By which means should nodes exchange topology information? The two alternatives by which topology information can be exchanged between nodes are either via special-purpose packets or by piggybacking this information on packets that are transmitted when a service is accessed by its clients. Piggybacking is the obvious choice, since it is the alternative which incurs less overhead. The only drawback of piggybacking is that it depends on packets being exchanged between nodes in the first place and that it fails in the absence of this traffic. However, this is not a problem in the context of an SPS, since network traffic exists by definition in regions of high service demand.

Which nodes should participate in gathering information? With respect to a service being offered in an ad hoc network, each node in the network falls into one of three categories: It is either a *client node*, i.e., one of the nodes on which an application with demand for the service is being executed; it is a *routing node*, i.e., there is no demand on the node itself, but it forwards packets between clients and service instances; or it is a *passive node* that is not involved at all with the provisioning of the service in question.

Obviously, client nodes should always provide topology information, at least about themselves but preferably also about their surroundings. The network topology around



(a) Small topology (b) Large topology

Figure 4.3: Roles of nodes in information gathering

clients is very likely to be relevant to an SPS since its goal is to place service instances in relation to the existing clients.

Routing nodes and passive nodes are similar in so far as it is impossible for a node of these two groups to determine locally whether the node itself might be relevant for the overall placement of a service. For illustration, let us consider the extreme case of a passive node as depicted in Figure 4.3a. Node s is the current host of a service instance serving requests from clients on nodes c_1 and c_2. Nodes r_1 and r_2 are routing nodes, node p_1 is a passive node. For simplicity, let us assume that nodes c_1 and c_2 have the same service demand, that all links in the network are of the same quality, and that there is no need to create a second service instance. The best node for hosting the service is the passive node p_1 since it minimizes the sum of the distances to all clients. This fact should, however, not be mistaken as an indication that node p_1 should have provided topology information to other nodes. Consider the same node configuration embedded in a larger topology as depicted in Figure 4.3b. Obviously, there is no point in having *all* passive nodes provide topology information.

There is, however, one distinction between routing and passive nodes in that routing nodes play an active role in the provisioning of the service. As service traffic is routed through these nodes, there is only little overhead involved if routing nodes add their own topology information to the packets that they forward anyway.

In light of these considerations, client nodes should always participate in the process of gathering topology information, routing nodes should participate to a lesser degree, and passive nodes should remain inactive.

How much information is to be gathered? The quantity of topology information gathered by each node depends on how interesting the neighborhood of this node is for the SPS. The metric used for defining the region of the network surrounding a node is the hop count of packets originating from said node, hence the term n-hop neighborhood. The parameter n corresponds to the minimal number of retransmission of a packet that is required before the packet reaches the node in question.

The question of how much information should be gathered can thus be reformulated as a question for the appropriate setting of the parameter n for each node. The tradeoff is, once again, that a larger n for more nodes increases the quality of the overall topology information, but at the same time also increases the overhead of transmitting this information between nodes. It is thus advisable to adapt this parameter at run time depending on the status of a node with respect to the service that is to be placed. More precisely, the parameter n should be larger for client nodes than for routing nodes. It is not applicable to passive nodes since they do not participate in the process of monitoring the network topology.

How often do nodes need to exchange this information? In answering the first question of the four questions mentioned above, we have argued that piggybacking should be employed for transmitting topology information between nodes. As a consequence, the frequency of exchanging information is limited by the flow of regular packets (which we assume to be sufficient, as explained above). The question is thus to which of these packets topology information should be added. The tradeoff is, yet once again, that the more frequent we add information, the more accurate the overall topology information becomes, however, at the cost of also increasing the overhead.

It is easy to conceive sophisticated schemes for selecting when to piggyback topology information on packets passing through a node. For example, topology information could be added only if the node detects that its n-hop neighborhood has changed. However, these advanced schemes incur the disadvantage that they require additional signaling for topology updates, i.e., the reception of the piggybacked information needs to be acknowledged by the destination node. Simply resorting to piggybacking the signaling data is not an option, since there is no guarantee that packets are transmitted in the opposite direction, from service instance to client. And even if this is the case, the returning packets may well be routed via a different path that excludes the destination node of the acknowledgment.

The alternative is to simply piggyback topology information on one packet during a configurable period of time or on every packet. These two options share the advantage that they do not introduce additional complexity into the system and are straightforward to implement. For these reasons, the simpler alternatives should be preferred when deciding how often topology information needs to be exchanged.

Implementation

In the current implementation of the SPi service placement framework, we employ the Dynamic MANET On-demand (DYMO) routing protocol as described in [17]. DYMO is a reactive protocol that has been proposed as a refinement of the Ad hoc On-Demand Distance Vector (AODV) routing protocol [87]. It defines three packet types: `ROUTE REQUEST` (`RREQ`), `ROUTE REPLY` (`RREP`), and `ROUTE ERROR` (`RERR`). Whenever a route needs to be established, the sender floods the network with `RREQ`s, optionally in the form of an expanding ring search in which the Time-To-Live (TTL) of the `RREQ` is incremented iteratively after each unsuccessful request. Upon receiving a `RREQ`, the destination unicasts a `RREP` back to the sender. If a route to the destination node is known to a node that processes a `RREQ`, it may send so-called intermediate `RREP`s to both nodes instead of forwarding the `RREQ`. When processing `RREQ`s and `RREP`s, each node updates its routing table with an entry for the respective source node containing the address of the next hop towards this source and metrics for the quality and the age of the routing information. If a unicast packet is received by a node whose routing table entry does not contain an entry corresponding to the destination node, a `RERR` is sent to the original sender of the unicast, which may establish a new route by flooding a `RREQ` once again.

Our choice for DYMO is due to two reasons: As a reactive protocol, DYMO leaves us full control on exactly how to implement the monitoring of the network topology. No integration or tradeoffs with any existing mechanism, as present in proactive protocols, are required. Furthermore, DYMO favors simplicity over advanced features found in other protocols, e.g., expanding ring search and route repair. Hence, it is straightforward to implement and has less complex interactions with the other components of the SPi framework. We have validated our implementation of the DYMO routing protocol during preliminary experiments which are described in Appendix A.

DYMO does not specify which link metric to use for its routing decisions. While a simple hop count metric works well in simulations, it performs poorly in real-world settings. We have thus extended our evaluation framework to support the Expected Transmission Count (ETX) metric [25]. As above, the choice for this metric is mainly motivated by its simplicity.

Our implementation of DYMO has been extended to support monitoring the network topology. Following the argumentation of the previous section, we have decided to collect the 1-hop neighborhood information for all client nodes and the 0-hop information for all routing nodes. The latter corresponds to the path along which a packet is forwarded. This information is piggybacked on every packet that originates at a client node or is forwarded through a routing node. For the scenarios that we examined during the evaluation of our system, this implementation resulted in an increase in the payload size of the packets by

about 5% on average and corresponds to an average of 35 bytes of each packet being used for exchanging topology information. These values are, however, scenario and architecture dependent, i.e., they are subject to change under varying node densities or address lengths.

Since the configuration of the topology monitoring did not manifest itself as a performance bottleneck during our evaluation of the SP*i* service placement framework, we assume that further optimization will only have a minor impact on the overall performance of the framework. Hence, we leave a more detailed evaluation for future work.

4.3.2 Service Discovery Component

The task of the service discovery component is to locate a node that hosts a suitable instance of an application-specific service for a client node. This comprises the subtasks of assembling a list of all potentially suitable hosts that are available in the ad hoc network at that point in time, and then selecting one of these hosts as being the most suitable one for satisfying the local client's service demand. The first of these subtasks is fulfilled by a service discovery protocol. The second one is noteworthy in its own right because the policy that is implemented needs to match the assumptions encoded within the service placement algorithm.

Service Discovery Protocol

The basic mode of operation of a service discovery protocol consists of two phases: In the first phase, the node hosting the client application floods the network with requests for the service that the client application wants to access. Upon receiving this request, each node that hosts a matching service sends a reply back to the originator of the request.

The first phase is initiated when a client application prepares a service request packet for a certain service. The application inquires the service discovery component for a suitable destination for this service request, which in turn floods the network with request packets for the service in question. There are several alternatives on how the flooding can be implemented. It can be regionally limited in the form of an expanding ring search. Other alternatives include reducing the cost of flooding the network by selectively forwarding the packet to nodes that are expected to provide a good regional coverage, or having several dedicated nodes with service directories spread out in the network. The first of these alternatives is comparatively light-weight; however, it may delay the discovery of distant service instances as each retry can only be considered as unsuccessful after a timeout has expired. The other two alternatives do not share this drawback, but at the cost of introducing additional complexity into the system in terms of differing node behaviors which needs to be administered and controlled.

Once a matching service host (or alternatively a node hosting a directory service) receives a request packet, it sends a reply back to the originator of the request via unicast. Since these unicast packets are sent during a time of high network activity – the network is being flooded with both service discovery requests and most likely also with route requests that try to establish a route back to the originator of the request – a small random delay may be added before sending the reply. As a reply packet is forwarded back to the originator node, it can aggregate various metrics that can be used later on to assess the relative quality of the service instance for the particular client node. These metrics usually include measurements related to the path between client and service instance, e.g., hop count or Round-Trip Time (RTT). As the reply packets reach the originator node, it collects the replies and the associated metrics and stores them for subsequent selection of the service instance that is best suited to provide the service to the local client.

Service Instance Selection

The selection of a service instance is triggered either after the expiry of a timeout that was started at the initiation of the service discovery process, or if a sufficient number of replies have been received from service instances. The second alternative implicitly relies on the assumption that nearby service instances, for which a reply takes less time to arrive, are more suitable for providing a service to a client, and hence ignoring late replies from other hosts does not discard any good service hosts.

When triggered, the metrics of all currently known service instances are compared and the address of the most suitable service instance is returned to the application. It is important to note that there is a dependency between this act of service instance selection and the placement of the service instances. In fact, the placement algorithm needs to anticipate which instances will be selected locally by each client as otherwise the effects of its placement decision would be suboptimal.

As an architectural alternative, one might consider employing centralized coordination for both service placement and service instance assignment, i.e., the process of centrally assigning a service instance to be used by each client. This alternative has however three major drawbacks: For one, it does not scale as well as the decentralized approach as the service coordinator not only has to contact all service instances but also all clients. Further, reliability suffers as client nodes may try to contact the wrong hosts if a service migration fails or takes longer than expected. And finally, in contrast to the service placement problem, and in particular the question for the correct number of hosts, the distributed approach to service discovery finds good solutions without excessive communication overhead.

Implementation

Our implementation of the service discovery protocol is loosely based on the Service Location Protocol (SLP) as specified in RFC 2608 [43]. SLP defines `Service Request` (`SrvRqst`) and `Service Reply` (`SrvRply`) messages for service look-up, either via single broadcast over via expanding ring search. It also specifies naming conventions for services as well as a distributed directory services that consists of so-called *directory agents* located on certain nodes.

In our implementation, we decided to omit the parts of the specification related to service naming and service directories. Naming conventions are not relevant for questions related to the placement of service instances, and service directories would have introduced another component in the system in which information would require timely updates. We have implemented the expanding ring search mechanisms, even though we observed that it introduces delays in the service discovery process which in turn results in lower service recall if the placement of the service instances changes frequently.

We also implemented a cache for known service hosts that is kept up to date as replies to service requests reach the respective client node. Furthermore, we added a rate limit on the number of `SrvRqst`s for the same service in order to make it impossible for a single nodes to flood the network repeatedly looking for a service which is not present.

4.3.3 Service Placement Middleware

The service placement middleware constitutes the core of the SP*i* service placement framework. In contrast to the two previously discussed components, the middleware component is not required to be active on all nodes of the network, but rather only on those nodes that currently host a service instance. It implements three primary functions:

Usage Monitoring: Statistics about the current usage of a service are collected by all service instances, either for local evaluation or for transmission to a dynamically assigned coordinator node for central evaluation.

Migration / Replication Protocol: The replication protocol handles the replication and migration of service instances that take place as part of the adaptation of a service configuration.

Placement Algorithm: The placement algorithm implements the policy according to which service instances are to be placed in the ad hoc network. Different policies may reflect goals such as resource conservation, low access latencies, or reliability. In light of this goal, the placement algorithm evaluates the usage statistics and decides whether the current configuration of the service needs to be adapted.

We will now present the mechanisms used for implementing the usage monitoring and discuss the integration of placement algorithms into the framework. The complete replication protocol including discussions of replication costs and optimizations will be covered afterwards in Section 4.4.

Service Usage Monitoring

The collection of usage statistics is performed locally on each of the nodes that hosts a service instance. It keeps track of several metrics for each client it serves. The questions that arise in this context are which metrics to collect to measure service usage, and how and when to process these statistics. Optionally, it may also be desirable to integrate this process with other aspects of the service placement system. Since the usage statistics constitute part of the input of the placement algorithm, it is not surprising that the answers to these questions depend largely on the choice of the placement algorithm.

Metrics Different placement algorithms rely on different kinds of usage statistics of the service as input. For the eight algorithms that we implemented, a total of eight different metrics are required. Some algorithms make use of multiple metrics when deciding on the optimal service configuration. Table 4.2 contains a complete list of the metrics employed by the algorithms that we implemented. All metrics are extracted from the service requests as they reach the service instance to which they were sent by the client. Additionally to the metrics, the table also describes two fields of miscellaneous information that also serves as input to some placement algorithms.

Three metrics (service request count, service traffic volume, and service traffic data rate) are collected twice for each node, once with the node in the role of the originator of the service requests, and a second time for the node as the last hop on the routing path between client and service instance. The metrics for the originator are employed by algorithms that rely on regional or global topology information; the metrics for the last hop are useful for algorithms that merely work on information from the vicinity of their hosting nodes.

The metrics for routed service request count and sum of hop counts of service requests, as well as the field for the node address of the last hop serve as special-purpose input for the two ASG placement algorithms [49]. Finally, the flag whether a node has refused to host a service instance in the past is used by several algorithms to keep track of which nodes are not in the set of potential replication or migration targets. This flag is set whenever a node refuses a replication or migration requests, i.e., when a `REPLICATION NACK` is received in response to a `REPLICATION REQUEST` during service replication (cf. Sec. 4.4.2).

Table 4.2: Per-node metrics for service usage monitoring

Service request count	Number of service requests that originated on this node
Service traffic volume	Total traffic volume of all service requests that originated on this node (in bytes)
Service traffic data rate	Average data rate of service requests that originated on this node (in B/s)
Service request count (as last hop)	Number of service requests for which this node was the last routing
Service traffic volume (as last hop)	Total traffic volume of all service requests for which this node was the last routing hop (in bytes)
Service traffic data rate (as last hop)	Average data rate of service requests for which this node was the last routing hop (in B/s)
Routed service request count	Number of service requests that were routed via this node
Sum of hop counts of service requests (as last hop)	Running sum of the hop count of all service request for which this node was the last routing hop
Node address of last hop of last service request	Address of the node that was the last routing hop of the last request that originated on this node
Refused to host service instance	Flag whether this node has refused to host a service instance in the past

Exchange of Statistics between Service Instances Just as above, the way in which these measurements are exchanged depends on the placement algorithm. For algorithms dealing with centralized services, i.e., placement algorithms that only decide about the location of a single service instances, the statistics are just processed locally and no exchange with any other entity is necessary. In contrast, algorithms that deal with distributed services require that the statistics are exchanged in one of the following ways as mandated by the design of the placement algorithm:

Exchange of usage statistics between a subset of service instances: Usage statistics are exchanged between service instances that operate in the same region of the ad hoc network in order to support distributed, and yet regionally limited placement decisions.

Exchange of usage statistics between all service instances: Statistics are exchanged between all service instances in the ad hoc network in order to support distributed global placement decisions.

Exchange of usage statistics with a central coordinator: Usage statistics are sent to a coordinator node for centralized evaluation. The GCMI algorithm that we propose as part of the SPi framework (cf. Sec. 5.5) falls into this category.

No exchange of usage statistics: Placement algorithms for distributed services such as ASG / simple and ASG / Event Flow Tree take placement decisions by locally matching rules against the collected service metrics. Thus, they do not require for statistics to be exchanged between nodes.

If the placement algorithm requires that usage statistics are exchanged between multiple entities in the ad hoc network, the question arises when to do so. The predominant method proposed in the literature is to exchange usage statistics periodically, which is especially suitable for placement algorithms that are executed once per preconfigured epoch [49, 68]. While easy to implement, periodically exchanging usage statistics is suboptimal with regard to accuracy and efficiency: If the demand for a service changes rapidly, distributed information about service demand will quickly grow stale, and timely adaptations of the service configuration are not possible. On the other hand, if service demand is constant, periodically sending statistics adds unnecessarily to the cost of running the SPS. Similarly, if the topology of the network changes rapidly due to node mobility or radio interference, this information needs to be communicated promptly.

As a solution to this problem, we propose to adapt so-called *funnel functions* as described by Mühl et al. in [82] to trigger the exchange of usage statistics between service instances. Funnel functions are defined as $|\delta x| > c \cdot (1 - \delta t/d)$, where δx is the change of an observed value since the last trigger event and δt is the time that has passed since this event. c and d are constant parameters of the function for maximal change and maximal duration respectively. The funnel function triggers as soon as a value is sampled that causes the inequality to evaluate as true. In a time series diagram, this can be interpreted as a tilted triangle centered over the current sampling value. As soon as a new value is sampled that lies outside the triangle, the function triggers and a new triangle is created centered around the current value.

The triggering mechanism of a funnel function is depicted in Figure 4.4. It shows the time series of a cost function and a series of triggering events t_{1-10}. We can observe that, as long as the sampled value remains constant, the funnel function only triggers when its maximal duration d has expired. However, if the sampled value changes rapidly, the rate of triggering increases. Hence, funnel functions exhibits the required behavior to be employed for the triggering of the exchange of usage statistics in an SPS.

In order to adapt funnel functions to be used in an SPS, a single metric is required to be used as input for the funnel function. This metric must have to property of reflecting

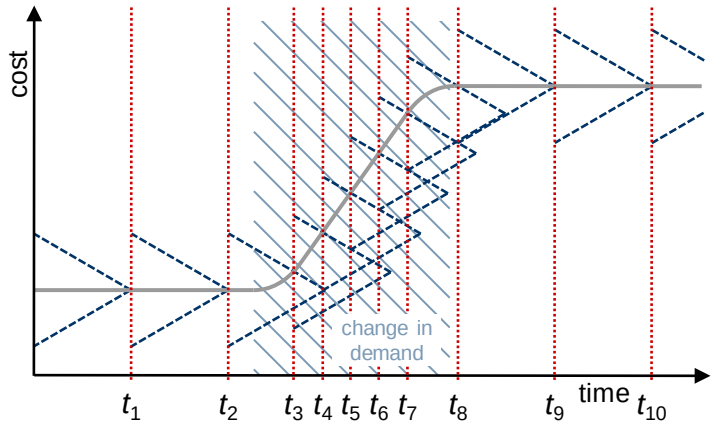


Figure 4.4: Funnel function as trigger mechanism

the change of service demand in the region around the node that calculates it. The service provisioning cost, which we will present in Section 5.2, has this property and hence we use this metric as an input for the funnel function. Furthermore, we parametrize the funnel functions in such a way that the maximal delay until the function is triggered increases the more stable the metric was during the previous interval.

Integration Usage monitoring and the exchange of statistics contribute significantly to the overhead of running an SPS. Hence, this process should be integrated as tightly as possible with the other components.

The first thing to note is that the usage statistics alone are useless to certain placement algorithms without the availability of a map of the corresponding region of the ad hoc network. The algorithms in question internally reconstruct the graph-like nature of the network and associate the service demand with the nodes in this graph. This approach is used both in algorithms that place centralized services, e.g., TopoCenter(1) [68] and GCSI (cf. Sec. 5.4), and algorithms that place distributed services, e.g., GCMI (cf. Sec. 5.5). For algorithms for distributed services that follow this approach, notably for GCMI, it makes sense to jointly exchange usage statistics and the respective regional network map of the service instance in question. This way the communication overhead is kept to a minimum. More importantly however, it also ensures that the usage statistics and the network map are inherently synchronized, i.e., both reflect the same state in time of the region of the ad hoc network with regard to service demand and network topology.

The messages that are exchanged to update usage statistics (and in some cases network topology information) may also serve two additional purposes: For certain distributed placement algorithms, in which individual service instances are not necessarily aware of each

others' existence and location, these messages may be used to share this information among service instances. This may be required either as an input to the placement algorithm, or more prominently to keep track of with which nodes the state of the service has been synchronized when processing client updates. For centralized placement algorithms, the statistics messages can be used as a built-in resilience mechanism to protect against losing the coordinator node under churn. This mechanism can, for instance, trigger a new coordinator election among current service instances if several consecutive statistics update messages remain unacknowledged.

Service Placement Algorithm

The other major part of the service placement middleware is the placement algorithm. From the point of view of the service placement middleware, there are two points to consider: First, the middleware must provide the necessary input data for each algorithm and be able to process its output. Second, for the matter of evaluation, it is also important to consider in how far all algorithms can be implemented against the same API, thus making the algorithms interchangeable.

Input, Output and Triggering Placement algorithms differ in the input they require in order to make the placement decision. For some algorithms it is sufficient to work on the usage statistics of one or multiple service instances, other algorithms also require knowledge about the network topology.

As for the output of the placement algorithms, a clear distinction can be made between algorithms that are tasked with placing a centralized service and those that deal with a distributed service. For centralized services, the only action that a placement algorithm can take is to migrate the single service instance from the current host to another one. For distributed services, the placement algorithm can take one or multiple out of three actions:

Replication: Use the application-level data, and possibly the serialized application binary, from a current service instance to create a new service instance on another node.

Migration: Move an existing service instance, specifically its application-level data and possibly the serialized application binary, from one node to another node.

Shutdown: Stop the execution of a current service instance and remove the associated data, and possibly the application binary, from the node that was hosting the instance.

A list of these actions that transforms the current service configuration into the optimal service configuration is the output of the placement algorithm. A more formal definition of these actions will be given in Section 5.3.

It is worth noting that placement algorithms are not required to issue a list of actions every time they are executed. In fact, in most cases, especially if service demand and network topology have remained stable for a certain period of time, the placement algorithm may well reach the conclusion that the current configuration of the service instances should not be changed. This may either be due to the fact that the configuration is optimal given the current knowledge about the network, or because the available data is insufficient to make a good placement decision. Hence, there is a forth, implicit action available to placement algorithms which causes the service configuration to remain unchanged. This action can be easily implemented by allowing the algorithm to return the empty list.

A final distinction is whether a placement algorithm runs periodically, often once per preconfigured epoch, or whether it is triggered in response to changes in the demand for the service or the network topology. The latter option can be implemented by reusing the funnel functions that also control when usage statistics are exchanged between service instances. In this case, the tradeoff, however, is between timely decisions and the cost of executing the placement algorithm, i.e., the run time complexity.

Placement Algorithm API In order to make service placement algorithms interchangeable at run time, we have created a generic interface for placement algorithms with the goal of allowing for side-to-side comparisons.

The interface of a placement algorithm comprises the following functions, listed in a simplified C notation:

`void start()` – Start the selected placement algorithm for a service instance. This function is called once when the service is started. In the implementation, the algorithm initializes data structures and, if required, registers timers for periodic execution.

`void stop()` – Stop a running placement algorithm of a service instance. This function is called once when the service instance is stopped. The algorithm cancels any pending timers and frees any resources it has allocated.

`void trigger(uint8_t reason_id, void* reason_parameter)` – Trigger a placement algorithm of a service instance. This function is called every time the usage statistics of the service have changed, either after a client's service request has been processed or after receiving usage statistics from another service instance. `reason_id` is the numeric ID for either one of these two circumstances. `reason_parameter` is a pointer to optional, reason-specific data which includes information about the remote node whose packet led to the triggering of the placement algorithm.

`size_t get_data(char** data)` – Serializes the internal data structures of the placement algorithm and pass them to the caller via the pointer `data`. This function is called when a service instances is replicated to a new host and the placement information is required to be replicated as well.

`void set_data(char* data, size_t data_size)` – Initialize internal data structures of the placement algorithm using a serialized representation of this information pointed to by `data`. This function is used when creating a new service instance during a replication.

`bool update_data(list_t* adaptation_actions)` – Update the internal data structures of the placement algorithm based on a list of actions that are performed as part of an adaptation of the service configuration. Actions may include replicating, migrating, and shutting down service instances.

`void report_known_service_instance(addr_t host_addr)` – Inform the placement algorithm about the availability of a new service instance, e.g., after receiving a SLP `Service Announce` packet from a previously unknown service instance (cf. Sec. 4.4.4).

`void report_invalid_service_instance(addr_t host_addr)` – Inform the placement algorithm about the unavailability of a previously known service instance, e.g., after receiving a SLP `Service Purge` packet (cf. Sec. 4.4.4).

`list_t* get_synchronization_hosts(bool inter_host_update)` – Return a list of addresses of hosts with which a modification to the application-level data of the service needs to be synchronized. The flag `inter_host_update` is set if the request to update the service data was received from another service instance (as opposed to from a client). This is required in order to support synchronization schemes that employ specialized synchronization hosts.

It is not required for an algorithm to implement all of these functions, e.g., if there is no internal state that needs to be transferred as part of service replications, or if centralized services do not need to synchronize updates to their application-level data with other hosts.

The placement algorithm may, in turn, make use of the following functions provided by the service placement middleware:

`void lock_service(srv_t* srv)` – Lock service instance `srv`. This method is called by the placement algorithm before beginning to change a service configuration.

`void unlock_service(srv_t* srv)` – Unlock service instance `srv`. This method is called when all changes to a service configuration have been completed.

`void shutdown_service(srv_t* srv)` – Shutdown service instance `srv`.

`void replicate_service(srv_t* srv, addr_t dest_addr)` – Replicate the locally running service instance `srv` to the target node with address `dest_addr`.

`void migrate_service(srv_t* srv, addr_t dest_addr)` – Migrate the locally running service instance `srv` to the target node with address `dest_addr`.

`void adapt_configuration(srv_t* srv, list_t* actions, list_t* dest_addrs)` – Adapt the configuration of the service of instance `srv` by having the nodes with addresses `dest_addrs` implement the list of actions (comprising replications, migrations, and the shutting down of service instances).

These functions allow for easy implementation of the placement decision taken by the placement algorithm. Furthermore, the middleware also provides the following two functions used to deal with corner cases and error conditions that can be detected by the placement algorithm:

`void update_coordinator(srv_t* srv, addr_t dest_addr, addr_t coord_addr)` – For a distributed service with a centralized placement algorithm, this function updates the address of the coordinator node used by the placement algorithm on the remote service host with address `dest_addr`. This may become necessary, if the coordinator is changed during an adaptation, but communication with one of the then current service instances fails and it continues to report to the previous coordinator.

`void shutdown_service_host(srv_t* srv, addr_t dest_addr)` – Shutdown the service instance running on the remote service host with address `dest_addr`. This may become necessary, if a service instance is supposed to be shut down as part of an adaptation, but the communication with the host node fails and the service instance remains active.

With this API in place, one may also consider whether it may be beneficial to exchange placement algorithms at run time in response to a fundamental change in the properties of the service or the network. While this seems technically feasible, we do not explore this question further since the goal of this work is to develop an algorithm that works well in a large variety of scenarios.

Implementation

We have implemented eight service placement algorithms. Six of these algorithms have been proposed in the literature [49, 68, 85] and two have been designed along with the SP*i* service

Table 4.3: Placement algorithms implemented within the SPi framework

Placement Algorithm	Input	Output	Triggering
LinkPull [68]	Local statistics	Migration	Once per epoch*
PeerPull [68]	Local statistics	Migration	Once per epoch*
TopoCenter(1) [68]	Local statistics and network map	Migration	Once per epoch*
Tree-topology [85]	Local statistics	Migration	Not specified†
SPi / GCSI (cf. Sec. 5.4)	Local statistics and network map	Migration	On changing service provisioning cost (cf. Sec. 5.2)
ASG / simple [49]	Local statistics	Per instance replication, migration, or shutdown	Once per epoch*
ASG / Event Flow Tree [49]	Local statistics and partial network map	Per instance replication, migration, or shutdown	Once per epoch*
SPi / GCMI (cf. Sec. 5.5)	Global statistics and network map	Global adaptation plan	On changing service provisioning cost (cf. Sec. 5.2)

* The duration of an epoch is not specified in the publication; we use a value of 60 seconds. † Triggering is not covered in the paper; we trigger the algorithm whenever a service request is received.

placement framework (cf. Secs. 5.4 and 5.5). All algorithms implement the API presented in the previous section and are interchangeable at run time.

An overview of all implemented algorithms is presented in Table 4.3. It summarizes the input and output of each algorithm together with how the algorithm is triggered. All placement algorithms for centralized services have a single migration as output. The output of algorithms for distributed services differs depending on whether the algorithm is centralized, e.g., SPi / GCMI, or distributed, e.g., both ASG algorithms. All algorithms for centralized services take locally collected usage statistics as input, as do both ASG algorithms. Only SPi / GCMI– due to its centralized design – requires that locally collected statistics are aggregated into global statistics. Most algorithms are executed periodically, one time per epoch. However, none of the publications specify the duration of an epoch. We have decided to use a value of 60 seconds for all algorithms, because preliminary evaluations have shown that a shorter epoch leads to volatility in the placement decisions and a longer epoch

results in less timely decisions. Both SP*i* algorithms are different, in that they explicitly monitor the service provisioning cost for changes and execute more frequently in periods of significant changes in service demand or network topology.

4.4 Service Replication and Migration

Service replication and migration are two out of three possible actions that may have to be performed by a current service host as part of the service adaptation process. The third action is shutting down a service instance, but since it involves no complex interaction between nodes we do not discuss it in greater detail.

Looking at service replication and migration, one might be tempted to think of the process of service migration as consisting of the steps of replicating the local service instance to a target node and then shutting down the local instance afterwards. However, splitting up the migration into these two steps leaves the current service host with fewer options in handling client requests during this transitional phase. For this reason, we consider replication and migration separately.

In order to implement service replication and migration, we first introduce the state machine required for the service to support this process and then discuss the replication protocol and possibilities for optimization.

4.4.1 Service States

In order to support service replication, the state model of a service needs to be extended as depicted in Figure 4.5. We start with a simple model consisting of the following three states:

INITIALIZING: Resources for the service instance have been allocated on the node, but the application-level data has yet to be setup. At this point, the service instance does not yet accept incoming service requests from clients.

RUNNING: The service instance is fully operational and handles service requests as they are received.

STOPPED: The service instance has been stopped. It does not handle service requests anymore, and its resources may be reclaimed by the node's operating system.

This model is slightly more complex than required for simple client/server operations, in which start-up and tear-down overhead is negligible and thus the `RUNNING` state would be sufficient. The `INITIALIZING` state reflects the fact that the process of creating new service

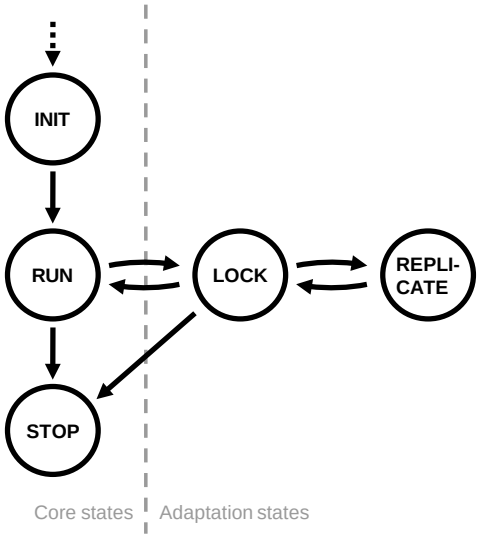


Figure 4.5: Service state machine

instances on remote nodes both takes non-negligible time and may fail due to communication problems. In the latter case, the resources that have already been allocated for the service instance may be freed after a timeout.

In addition to these three basic states, the following two states deal explicitly with the process of service replication:

LOCKED: In the `LOCKED` state, no modifications to the application-level data of the service instance may be performed, i.e., it cannot be updated by clients' service requests. This state is used to ensure data consistency while one or several replications to other hosts take place.

REPLICATING: At least one replication of this service instance is currently in progress. Application-level data and possibly the application binary are being transferred to a new host using the service replication protocol (cf. Sec. 4.4.2).

Note that the separation of locking and replication is necessary because a single replication may be just one step within a more complex adaptation of the service configuration across which the read-only property of the service's application-level data must be preserved.

There are two alternatives when extending existing service implementations to support this state model. The first alternative is to rewrite the implementation of the service to take these states into account and to provide an API which the service placement middleware can use to trigger state transitions. The obvious disadvantage of this approach is the work required for adapting the implementation of services. The other alternative is to build

upon the process abstraction of the underlying operating system (OS). This works well for the first three states (`INITIALIZING`, `RUNNING`, and `STOPPED`), but fails for the latter two states (`LOCKED` and `REPLICATING`) since the service instance is still serving read-only requests in these states and hence the OS-level process needs to be running. One option to work around this problem would be to inspect service request packets within the service placement middleware and only relay appropriate packets, i.e., those that contain read-only requests, to the service process. However, this does not seem to be a viable option since the service placement middleware would require intimate knowledge about the packet format used by the service.

For our implementation, we have decided to implement the state machine as part of the service itself. This decision was mainly motivated by the lower overall system complexity that follows from this approach. For a more long-term solution a proper API between service and placement middleware needs to be developed. However, we consider this to be out of scope of our current work.

4.4.2 Service Replication Protocol

The service replication protocol is in charge of transferring the application-level data and possibly also the serialized executable of a service instance from its current host to the host that is the destination node of the replication. The protocol consists of two phases: In the first phase, an inquiry is sent to the destination node to check whether it has the necessary resources available to host the service. If the response is affirmative, the data is transferred in the second phase. This entire process is depicted in Figure 4.6.

The inquiry begins with a `REPLICATION REQUEST` packet, which contains information regarding the resource requirements of the service. Upon reception of this packet, the target node ensures that the service in question is not already running on this node and that enough local resources (e.g., memory, processing capacity, etc.) are available. If one of the conditions is not fulfilled, the target node replies with a `REPLICATION NACK` packet. Depending on the placement algorithm employed on the current service host, it may then take note of the refusal of this target node, e.g., by removing it from the set of potential replication targets for future placement decisions.

If, however, both conditions are fulfilled, the target node accepts the replication, thereby moving on to the second phase of the replication protocol. Based on the information embedded in the `REPLICATION REQUEST`, the target node initiates the transfer of the service data by sending a `REPLICATION ACK` packet to the current service host. A `REPLICATION ACK` contains fields specifying which segment of the service data the target node expects the current service host to transmit. This may either be one or multiple segments, thereby

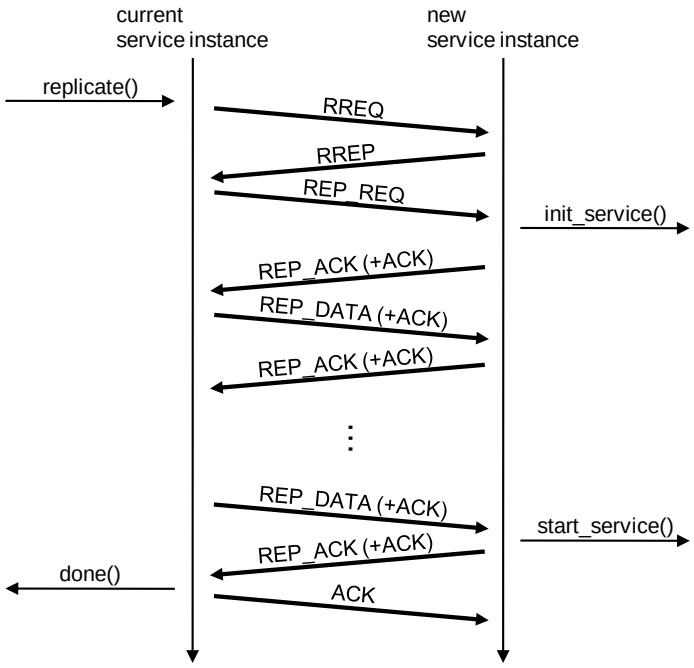


Figure 4.6: Time diagram of service replication protocol

allowing for flow control between target and originator node. If the current service host receives the `REPLICATION ACK` packets, it sends one or multiple `REPLICATION DATA` packets that contain the requested service data. Once the complete service data (as specified in the initial `REPLICATION REQUEST`) has been transferred to the target node, it sends a final `REPLICATION ACK` requesting no further data to the originator node. This indicates to the current service host that the service replication was successful, and that it may proceed to resume normal operation or, in case of an ongoing service adaptation, move on to implement the next action of that adaptation. Finally, the target node proceeds to start its own service instance.

This protocol for service replication leaves the control of the process on the receiving side at the new service host. Once it has agreed to host a new service instance, the target node can thus easily implement flow control and request retransmissions of lost packets. The current service host does not need to hold any state regarding this process, except for a timer to abort and unlock the service instance in case of the target node becoming unavailable before the replication is completed.

4.4.3 Cost of Service Replication and Migration

The cost of service replication or migration corresponds to the data traffic caused by, and as a consequence, to the bandwidth consumed by, the acts of replicating or migrating a service instance form one host to another. They need to be considered separately because service migration requires for the configuration of all current clients of the service instance in question to be updated, while the same update is optional for clients if the instance is merely replicated. This gives raise to the distinction between *direct* and *indirect* cost of service replication and migration: The direct cost corresponds to the cost incurred by the data transfer of the replication. The indirect cost corresponds to the acts of communication required for updating the clients' information on available service instances and routes. By this definition, a replication only incurs its direct cost, while a migration incurs both direct and indirect cost.

The traffic caused by a service replication is dominated by the transfer of the serialized service data from the old to the new host as described in the previous section. Additionally, a service replication may also involve establishing a route between current service host and target node.

In case of a service migration, the original service host shuts down its local service instance after the replication has been completed. As a consequence, the configuration of all clients of this service instance needs to be adapted. This in turn causes network traffic which needs to be counted towards the cost of the service migration. In a naïve implementation, the clients need to perform two steps in order to continue using the service: First, they have to locate a new service instance, and second they have to establish a route to the node that hosts this instance. These two operations are noteworthy, because both of them involve flooding the network, the first with SLP `SrvRqst` packets, the latter with DYMO `ROUTE REQUEST`s.

It should be emphasized that these are merely the cost of a single service replication and migration respectively. If the configuration of a distributed service is being adapted, this process usually requires multiple replications and migrations which may even be performed in parallel. Furthermore, if the service demand or the network topology changes frequently, several adaptations of the service configuration may be required in a short period of time. It is thus important to reduce this cost as much as possible as well as taking the adaptation cost into account when deciding whether to change the service configuration. We discuss the first of these issues in the following section, and leave the second one to Section 5.6 in the next chapter.

4.4.4 Optimizations

In this section we discuss possible ways to reduce the cost of migrating service instances from one host to another. Two of them reduce the necessity for flooding the entire network. A third one is a change in the service state machine that allows for graceful handling of packets that were in transit while the service adaptation is taking place.

Proactive Service Announcements

The key idea behind this optimization is to move the initiative in the task of finding service instances from the clients to the service hosts. Instead of having the clients flood the network with requests for a service, each service instance floods the network announcing which service or services it provides. This data is cached locally on all client nodes. Communication between service instances and clients is handled via two new packet types `Service Announce` (`SrvAnce`) and `Service Purge` (`SrvPrge`).

The process operates as follows: Every time the configuration of a service is changed, each service instance of the new configuration floods the network with a `SrvAnce` packet. This packet contains information about which service is now hosted at the node that initiated the broadcast. If the `SrvAnce` packet is sent as a result of a service migration, it also carries information about which host initiated the migration and has now stopped hosting a service instance. The information is stored for a configurable period of time in a cache in the service discovery component on all clients. Should a client start issuing service requests, the optimal service instance is selected based on the cached information and no broadcasts of `SrvRqst` packets are necessary. If the cached information grows stale and is removed from the cache before any service requests are issued, the client simply falls back to using the regular service discovery mechanism.

In the case of a service instance shutting down (including a shutdown as a result of a service migration), the former host broadcasts a `SrvPrge` packet. This packet notifies the clients that the service will no longer be available on this host and the clients purge the host in question from their service cache.

This optimization of the service discovery process effectively reduces the number of broadcast packets sent as a result of a service adaptation. In comparison to the naïve implementation, clients are generally not required to initiate a broadcast, but can instead rely on the notification they receive from the service instances. Since there are always fewer service instances than clients, the network traffic and thus the adaptation cost are reduced.

One additional noteworthy advantage of this optimization is that other clients, which were not sending their service requests to the original service host, are notified of the presence of a the new service instance. Depending on the metric used for selecting service instances,

these client may decide to direct future service requests to the newly created instance. As a result, changes in the service configuration are reflected immediately in the association of clients to service instances.

Low-priority Routes

Another source of relevant overhead of the service migration process stems from the fact that clients have to establish new routes once a service instance migrates to a new host. The very same problem does also arise, if a client decides to switch service instances. Similarly to service discovery, the naïve implementation of this process requires a network-wide broadcast of `RREQ` packets from each client. Once again, this adds significantly to the cost of migrating a service instance. In order to lessen this overhead, we propose that the routing component should generally overhear other network-wide broadcasts and update the routing table using information extracted from these packets.

The detailed operation is as follows: If a broadcast packet is received that is part of a network-wide broadcast, the routing component checks whether a route to the initiator of the broadcast is known. If this is not the case, the last node that forwarded this broadcast is stored as a next hop destination for packets that need to be routed to the originator. This new routing table entry is added to routing table with lowest possible priority. This way it is ensured that any normally established route will update this routing table entry. Furthermore, if a routing protocol is employed that supports exchanging parts of the routing table between nodes (e.g., DYMO), then routing table entries that were created in the manner described above are not exchanged between nodes.

This mechanism implements an inexpensive way to keep track of routes to destinations for which otherwise no route would exists and a traditional route discovery process would have to be initiated. The low priority of these routes together with the fact the knowledge about them is not shared between nodes ensures that there is no interference with regular route discovery and maintenance. If one of these routes should turn out to be invalid, e.g., via a `RERR` notification, it is discarded and a regular route discovery process is initiated.

Creating low-priority routes really pays off when a node in the network can expect that several other nodes will try to establish routes to it in the near future. This is the case when new service instances are created, and coincidentally, we have proposed in the previous section that new service hosts should proactively announce the availability of their service instance. In fact, the `SrvAnce` packets that are broadcasted in this situation serve as a means to add low priority routes. With this mechanism in place, client nodes do not need to establish routes to new service hosts (independently of which one they end up choosing) since routes to all service hosts are already implicitly available.

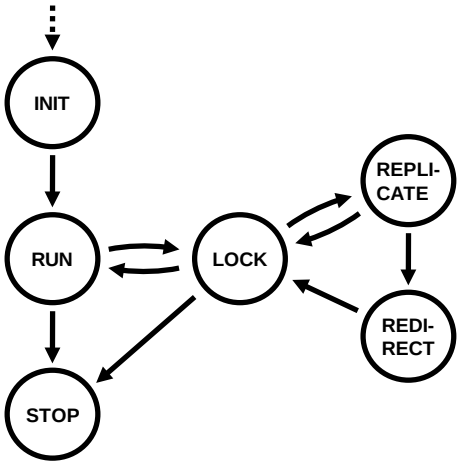


Figure 4.7: Service state machine with support for client redirection

The only remaining network-wide broadcast is required for establishing a route between old and new service hosts. All other broadcasts are eliminated through the combination of this and the previous optimization steps. This results in a reduced cost of service migration, and this operation can be employed more frequently by the service placement algorithm. This in turn allows for the placement of service instances to be adapted more rapidly to changes in service demand or network topology, and thus increases the overall quality of the service configuration.

Client Redirection

Finally, we propose a third technique to reduce the negative impact of service migrations. This one does not deal with the network traffic caused by the migration, but rather focuses on the disruption of the service as perceived by the clients that occurs whenever a service instance is migrated from one node to another. During the short period of time when the migration is already complete, but the client is not yet aware of this change in the service configuration, service requests are erroneously sent from the client to the old service host. In a naïve implementation, these service requests would have to be discarded since the service is not available on this node anymore.

In order to smoothen the transition from one host to another from the clients' perspective, we extend the state machine of the service. This change is depicted in Figure 4.7. The new state `REDIRECTING` is entered when a service migration to a new host has been completed. The service instance remains in this state until a timeout expires and the instance is stopped. In this state, the service instance has already ceased to handle service requests, but the meta information about the service is still available. The meta information includes data

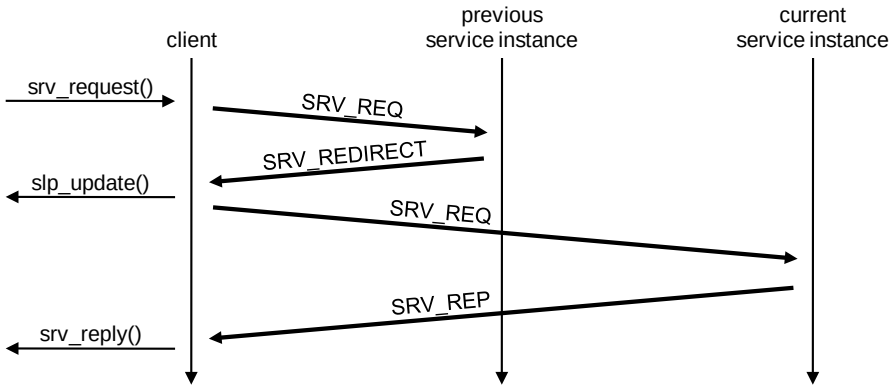


Figure 4.8: Client redirection by updating the client's service cache

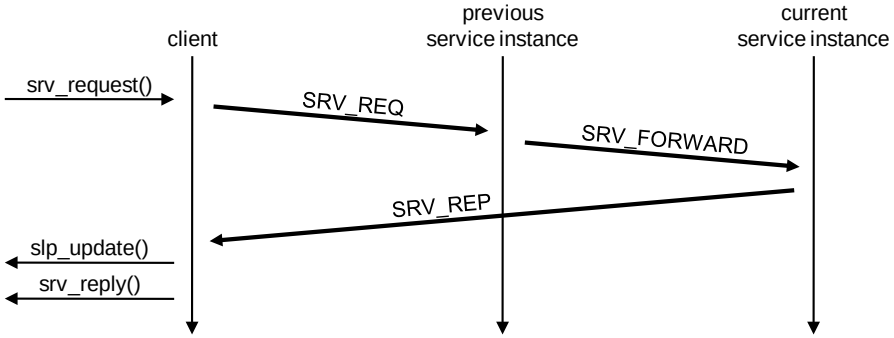
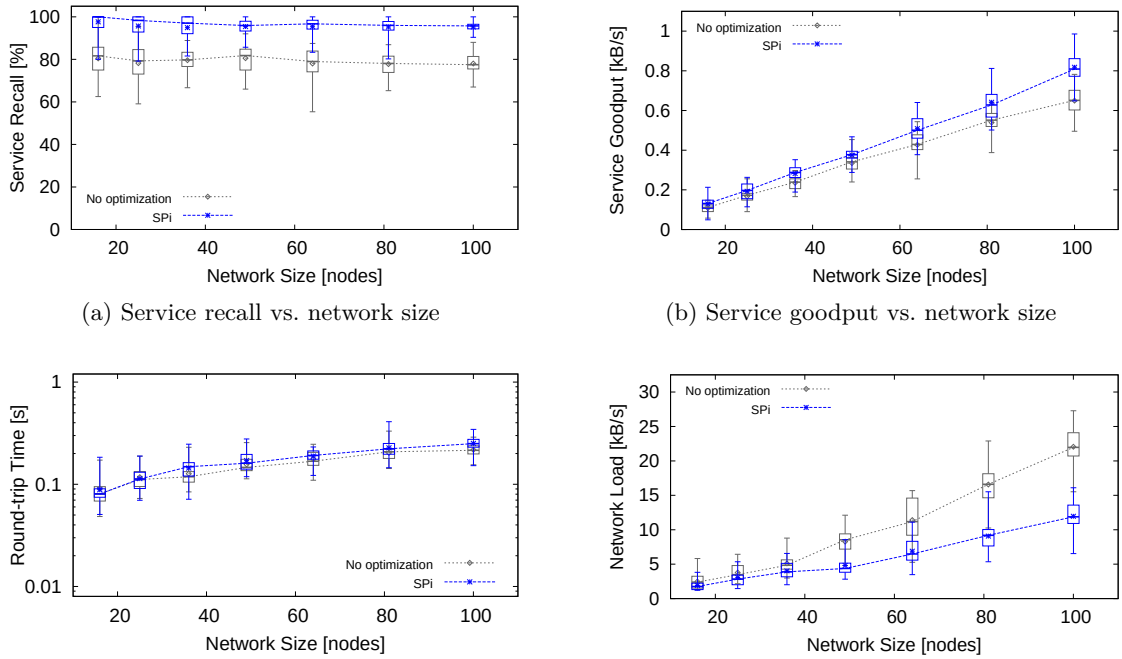


Figure 4.9: Client redirection by forwarding the service request

such as the ID of the service and the target node to which the service instance has been migrated.

While in the `REDIRECTING` state, the node does not simply drop service requests, but instead replies with a packet informing the client about the new host of the service instance. There are two ways in which this can be implemented: The redirecting host can either ask the client to update its service cache, or it can forward the request to the new service instance which then replies to the client. These two alternatives are depicted in Figures 4.8 and 4.9. The redirection by update has the advantage that the client-side information is updated more quickly and further service request are more likely to be sent to the correct node directly. The redirection by forwarding is superior in that it saves one packet exchange between the nodes. In either case, the procedure results in minimizing the disruption of a service as perceived by the client.



(a) Service recall vs. network size

(b) Service goodput vs. network size

(c) Request delay vs. network size

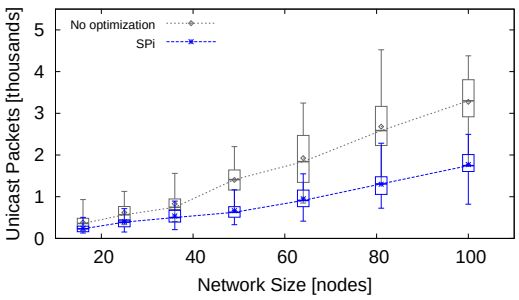
(d) Network load vs. network size

Figure 4.10: Impact of service migration

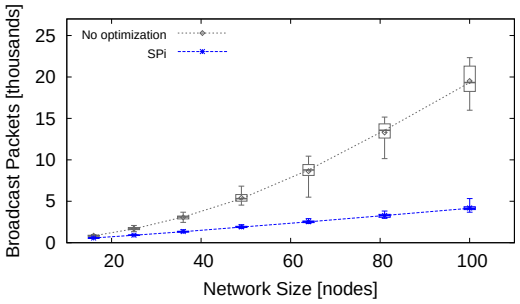
4.4.5 Preliminary Evaluation

We will now briefly examine the impact of the optimizations proposed in the previous section on the negative impact of service migration. We simulate an IEEE 802.11 network with similar simulation parameters as in the overall evaluation of the SPi framework (cf. Sec. 6.3.1). For brevity, we omit a comprehensive discussion of the simulation setup at this point.

The simulated network consists of 100 nodes that are regularly placed in a 10 by 10 grid so that each node has a reliable radio link to each of its direct neighbors (horizontally, vertically, and diagonally). Half of the nodes are randomly chosen as client nodes. The other half remains inactive, except for routing packets and potentially hosting service instances. A centralized service with a single service instance is started on a random node. During the first 60 seconds of the simulation, all client nodes issue one service request to the service instance. The service instance is then migrated to another randomly chosen node. All client nodes remain idle for a delay of 10 seconds to allow for the migration to take place, before issuing a second service request during the following 60 seconds. In total there are thus two service requests per client node and one service migration between two randomly chosen nodes.

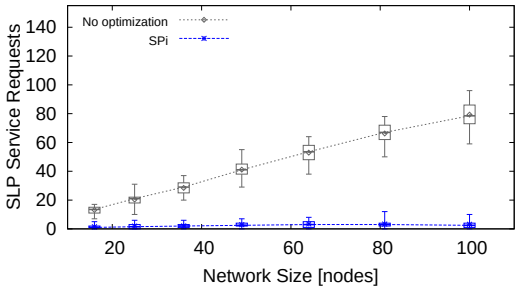


(a) Number of unicast packets vs. network size

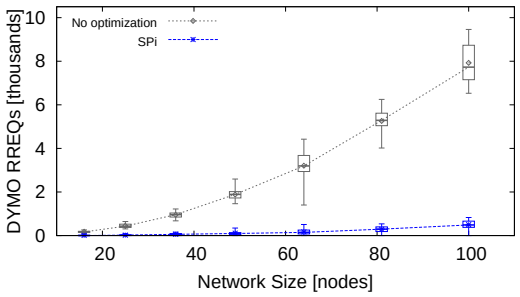


(b) Number of broadcast packets vs. network size

Figure 4.11: Service migration overhead by packet type



(a) Number of SLP `SrvRqst`s vs. network size



(b) Number of DYMO `RREQ`s vs. network size

Figure 4.12: Service migration overhead for service discovery and routing

The results of this simulation are shown in Figures 4.10 to 4.12. Figure 4.10 illustrates the impact of the migration of a service instance on the quality of the service as perceived by a variable number of clients. Once again, we refrain from discussing the metrics in detail (cf. Sec. 6.2), but merely focus on the overall effect. In Figure 4.10a, we can observe that the ratio of successful service requests of the naïve implementation is 20% lower than the almost flawless ratio for the optimized implementation in the SPi framework. We attribute this difference to the fact that SPi gracefully handles service requests that reach the service hosts while a migration is in progress, while the naïve implementation lacks a mechanism for doing so. In Figures 4.10b and 4.10c, there is only a small difference to be observed in the service goodput and no difference in the round-trip time of service requests. In contrast, Figure 4.10d shows a significant difference in network load between the naïve and the optimized implementation.

In Figures 4.11 and 4.12, we examine this difference in network load between the naïve and the optimized implementations in more detail. Figure 4.11 compares the two implementations based on the number of transmitted unicast and broadcast packets. There is a significant difference between the two approaches – especially for the number of broadcast packets –

with the optimized implementation showing superior behavior as the size of the network is scaled up.

The two main processes that broadcast packets in the scenario under consideration are service discovery and route discovery. Figure 4.12 illustrates the differences between these processes for the two implementations. In Figure 4.12a, we plot the number of SLP `SERVICE REQUEST`s against the number of nodes in the network. As can be seen, the optimized implementation has almost no need for this method of locating a service, since the information is readily available to all clients that have received the `SERVICE ANNOUNCE` packet sent by the new service instance. Similarly, Figure 4.12b plots the number of DYMO `ROUTE REQUEST`s against the number of nodes in the network. We can observe that the optimized implementation uses almost no `ROUTE REQUEST`s. This is due to the routing information being already available on all nodes in the form of the low-priority routes that were extracted from the broadcast of the `SERVICE ANNOUNCE` packet.

From this evaluation we can conclude that the optimizations to the processes of service replication and migration not only work, but also have a significant positive impact performance of the ad hoc network. From the perspective of the client nodes, service requests are more likely to be handled successfully even if an adaptation of the service configuration is underway. For the network as a whole, the adaptation process generates less traffic and hence is less likely to cause packet collisions, retransmissions, and network congestion. As a result, replications and migrations of service instances become more lightweight operations. Therefore, they can be used more readily by a service placement algorithm.

4.5 Summary

In this chapter, we have presented the SPi service placement framework, including design considerations and rationale, software components, and mechanisms for supporting the replication and migration of service instances. We have argued that, in light of the high impact of service placement on the logical structure and configuration of an ad hoc network, the framework should be geared towards a centralized placement algorithm that is well integrated with the other components of the framework. In particular, we pointed out that the routing and service discovery components need to be adapted to efficiently support the implementation of placement decisions.

Our framework improves upon the state of the art by providing a generalized API for service placement algorithms. For the first time, it is thus possible to conduct comprehensive side-by-side comparisons of a wide variety of placement algorithms. And although we advocate a centralized placement architecture, the framework equally supports other approaches, such as the two ASG algorithms, that follow other paradigms. The SPi service

placement framework does explicitly *not* implement a placement policy, but rather leaves the placement decisions and the manner in which they come to be entirely up to the placement algorithms.

Furthermore, we present several refinements and optimizations of relevant components that are useful for service placement systems in general. The semantics of the state machine for service instances and the replication protocol may serve as guidelines when designing similar systems. Of particular value, however, are the proposed optimizations to the service replication and migration processes. We have shown that the overhead of implementing placement decisions can be reduced significantly by tightly coupling the placement logic with the service discovery and routing components. These optimizations allow for service placement systems to operate far more efficiently in general, and thereby increase the benefits of employing this technology in ad hoc networks.

In the following Chapter 5, we will now proceed by formalizing some of these concepts and by presenting two placement algorithms that we designed specifically for the SP*i* service placement framework.

Chapter 5

SP*i* Service Placement Algorithms

In this chapter, we present two service placement algorithms, one for centralized and one for distributed services. Both are specifically designed to implement the placement intelligence within the SP*i* framework [111]. The Graph Cost / Single Instance (GCSI) algorithm places the single service instance of centralized services, and the Graph Cost / Multiple Instances (GCMI) algorithm handles distributed services with multiple instances. For the latter case, we employ a dynamically assigned coordinator node to make placement decisions, rather than relying on more distributed approaches that incur the drawbacks of either higher signaling overhead or lower quality placement decisions (cf. Sec. 4.2).

After a brief overview in Section 5.1, we begin in Section 5.2 with an introduction of the concept of service provisioning cost around which both algorithms are built. In Section 5.3, we formalize the actions available for implementing a service adaptation and the associated cost. We then move on to present the two algorithms in Sections 5.4 and 5.5, discussing design considerations and optimizations for real-world deployments. In Section 5.6, we continue to explain the mechanism for deciding at which point in time the service configuration should be adapted. Afterwards, we conclude the presentation of the complete SP*i* service placement framework with a brief example in Section 5.7, and then summarize our findings in Section 5.8.

As in the previous chapter, we limit the discussion of the implementation to the most interesting aspects. The full documentation of the SP*i* service placement framework, including the documentation of both algorithms, as well as the BSD-licensed source code is available at `http://cst.mi.fu-berlin.de/projects/SPi/`.

5.1 Overview

The service placement middleware as presented in the previous chapter employs two different algorithms depending on whether it is tasked with placing a centralized or a distributed service. For the first case, we propose the Graph Cost / Single Instance algorithm (GCSI). It

monitors the network traffic between the service instance and its clients, and uses information about the network topology gathered by the routing component to calculate the optimal host for the service instance. If the optimal host is different from the current host, and if the adaptation cost (in terms of network traffic) is justified in light of the expected savings, the service instance is migrated to the new host. This algorithm is presented in detail in Section 5.4.

For the case of placing a distributed service, we propose the Graph Cost / Multiple Instances algorithm (GCMI). The algorithm approximates the optimal service configuration by solving a variation of the Uncapacitated Facility Location Problem. The network graph and the clients' service demand are used as main inputs for the algorithm, and the cost metric is based on the overall bandwidth required for sustaining the service. With the optimal service configuration as input, the coordinator node establishes a set of actions required for adapting the current to the optimal configuration. Possible actions are the replication of a service instance from a current host to a new host, the migration of a service instance, and shutting down an instance. Once again, if the combined cost of these actions is justified in light of the expected savings, commands for adapting the configuration are issued to the nodes that currently host a service instance. These nodes then distributively proceed with replicating, migrating, or shutting down individual service instances. The complete algorithm is presented in detail in Section 5.5.

The GCSI and the GCMI algorithms are both designed for centralized execution. This fundamental design decision is motivated by the cost of service adaptations and the difficulty of establishing the optimal number of service instances for distributed algorithms (cf. Sec. 4.2). For the GCSI algorithm, the choice of which node should execute the algorithm trivially falls on the node that is currently hosting the single service instance. For the GCMI algorithm, the most suitable node for executing the algorithm, the so-called *coordinator*, needs to be selected at run time. We dynamically assign this role to the most centrally placed service instance, i.e., to the service instance which can reach all other instances with the minimal network traffic.

The fundamental approach of both the GCSI and the GCMI algorithms is to map the service placement problem to a graph problem, in which the vertices of the graph correspond to the nodes of the ad hoc network and the edges of the graph correspond to the radio links between the nodes. The vertices are annotated with the service demand, and the edges are annotated with their capacity of carrying network traffic. We then calculate the optimal service configuration in the graph representation of the network, and issue the necessary commands to adapt the current service configuration to match the calculated optimum. Hence, both algorithms require statistics about the current service demand as well as a graph representation of the network topology as input.

The metric we use for measuring the quality of a service configuration is the *service provisioning cost*. The service provisioning cost corresponds to the total data rate consumed by all acts of communication between nodes that are required in order to satisfy the service demand of all client nodes. The intuition is that the more bandwidth a service consumes, the more costly it is for the ad hoc network to sustain the service in question. The goal of both the GCSI and the GCMI algorithms is to minimize the service provisioning cost. Their means of achieving this is the intelligent manipulation of the placement of service instances. This process should ideally result in the clients' service requests being handled with less overall communication between the nodes.

While adapting the service configuration, the service placement algorithms are free to select the best possible set of nodes to host service instances. For distributed services, both the location and the number of service instances may be adapted. It is, however, important to note that even the best placement decision can do more harm than good if it is not timed correctly. In other words, since adapting the service configuration consumes a non-negligible amount of bandwidth, the placement algorithm needs to ensure that the investment in bandwidth will pay off later on. This kind of decision would, however, require knowledge about the future characteristics of the network topology and the service demand across the network.

Making such a prediction about the future behavior of a complex system is hard if not impossible. The GCSI and the GCMI algorithms rely on the simplifying assumption that the future behavior of the network, with regard to both the topology as well as the service demand, can be deduced from the behavior that was observed in the past. An adaptation of the placement of service instances is thus only performed, if the cost of this adaptation is justified in light of the difference between the service provisioning costs of the current and the optimal service configuration. This is observed over a period of time whose length increases proportionally to the expected cost of the adaptation. This way it is ensured that the service configuration is only adapted if there is sufficient reason to assume that the benefit of this change will outweigh the cost it incurs.

5.2 Service Provisioning Cost

The service provisioning cost is the central metric employed by the SP*i* service placement framework. It is defined as the total data rate of all application-layer acts of communication between nodes that are required in order to satisfy the service demand of all clients in the network. This metric allows for the placement algorithms to organize the service instances in such a way that the overall bandwidth consumed while providing the service to clients is minimized.

The service provisioning cost is the sum of the costs of three individual types of communication. Sorted by relevance, they comprise

1. the exchange of *service data* between clients and service instances,
2. for distributed services, the exchange of *synchronization data* between service instances in order to keep the global state of the service up to date, and
3. for certain distributed placement algorithms, the *signaling overhead* required for the operation of the service placement system.

For now, we focus on the costs of service data and synchronization data. The signaling overhead is dependent on the placement algorithm and hence we discuss it separately as part of the presentation of the GCMI placement algorithm in Section 5.5.2. The GCSI algorithm presented in Section 5.4 deals with placing the single instance of a centralized service and hence no signaling is necessary.

As we will see in the formal definition of the service provisioning cost in Section 5.2.3, the calculation has two time-dependent values as inputs (among other, largely time-independent parameters): a matrix of inter-node distances, i.e., a representation of the network topology, and the service demand on all client nodes. In Section 4.3.3, we have identified these inputs as being relevant for triggering the exchange of usage statistics between service instances. Since the calculation of the service provisioning cost maps these two inputs to a single scalar value, we can employ this value as input for the triggering mechanism, i.e., the funnel function.

5.2.1 Rationale and Alternatives

We have chosen the service provisioning cost as metric for the SPi service placement framework because it allows us to reduce the volume of data that needs to be transmitted across an ad hoc network in order to sustain a service. This in turn results in a reduction in the consumption of two key resources of the ad hoc network: radio bandwidth and energy. In fact, since we propose to have multiple service instances in the network, one can think of our approach as trading storage space and to a lesser degree computational power for radio bandwidth and energy. As one can expect the latter two resources to be scarce in comparison to the first two, this tradeoff makes sense for a large number of ad hoc networking scenarios.

There are however several alternatives that one might consider for other, more specialized use cases:

- For time-critical applications, it may make more sense to deploy as many service instances as possible (given the available bandwidth) in order to optimize the *service*

access time, i.e., the round-trip time that elapses for a client between sending a service request and receiving a reply.

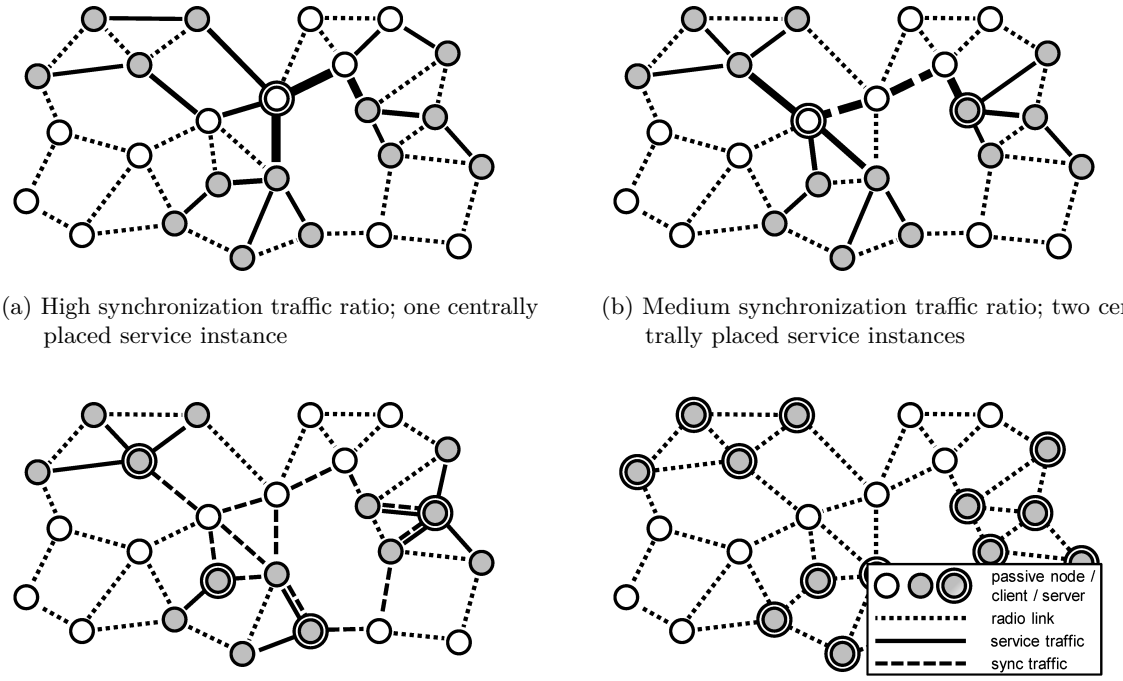
- For services with a focus on availability, one may decide to optimize the *number of service instances located in the immediate vicinity* of all client nodes.
- For services with a focus on reliability, one may chose to optimize the *regional diversity* of the service instances, i.e., have the service instances spread out as far as possible in order to mitigate the risk of the service becoming unavailable should large parts of the network suddenly cease to operate.
- For network with group-based mobility, it may prove optimal to identify the nodes belonging to the same mobility group, and then associate one or multiple service instances with each group.

While certainly interesting, we consider these alternatives to be beyond the scope of our current work and leave them for future work.

5.2.2 Modeling Synchronization Requirements

If a service is provided in a distributed fashion by multiple service instances, these service instances need to synchronize updates and changes to the global state of the service among each other. The traffic required for synchronizing the global state among all service instances varies depending on the type of service, the service request patterns of its clients, and the requirements of the service on data consistency. In order to avoid passing the knowledge about these internal workings of a service to the placement algorithm, we make the simplifying assumption that the synchronization traffic can be expressed as a simple ratio τ of the client traffic received by a service instance. Consider, for example, an instance of a service s serving a set of clients whose service requests consist to 90% of requests that do not modify the service state (i.e., read requests) and to 10% of requests that do modify the state (i.e., write requests). If the policy of the service s was to directly synchronize all write requests with the other service instances, then the synchronization ratio τ_s for this service would be 10%.

In a more sophisticated setting, the synchronization policy of a service may include delayed synchronization, data aggregation and semantic compression. For example, if the service instance in the above scenario was to aggregate all write requests over a preconfigured period of time and then compress the changes with 50% efficiency, the synchronization ratio would drop to 5%. The drawback of this form of delayed synchronization is of course that the updated information does not reach the other service instances as quickly as in the case of direct synchronization. The tradeoffs involved in finding a suitable synchronization



(a) High synchronization traffic ratio; one centrally placed service instance

(b) Medium synchronization traffic ratio; two centrally placed service instances

(c) Low synchronization traffic ratio; four instances, placed according to regional service demand

(d) No synchronization traffic ($\tau_s = 0$); one service instance per client

Figure 5.1: Impact of the synchronization traffic ratio τ_s on the number and distribution of service instances for service s

policy for a specific service are beyond the scope of this work. From our perspective, the synchronization ratio τ_s for a service s is preconfigured based on service-specific expert knowledge or previos measurements. Alternatively, it can also be calculated at run time based on the usage statistics collected by the service placement middleware. τ specifically includes any service-level processing of the service requests, such as delayed processing, aggregation, or compression.

Coming back to an illustration already used in Chapter 1 to motivate the need for intelligently placing service instances, Figure 5.1 highlights the dependency between the synchronization traffic ratio τ_s and the optimal configuration of a service s. Starting with a high value for τ_s in Figure 5.1a, τ_s decreases for each of the following figures until reaching zero in Figure 5.1d. The key observation is that the number of service instances in the optimal service configuration increases as τ_s decreases. The reason for this is that as less traffic is required for synchronization between instances, the overall service provisioning cost can be reduced by placing more service instances closer to the client nodes. We discuss this dependency quantitatively as part of the preliminary evaluation of the GCMI algorithm in Section 5.5.4.

In order to gain more insight into which values to expect for τ for real-world services, it makes sense to consider a few examples: As a first example, assume that the content of a simple web server is updated by the owner at a frequency of one change per hour. Let us further assume that the web server serves 100 client requests during the same period of time. This would yield a value for τ_{web} of about 1%. For a second example, assume that the mapping between domain name and IP address in a DNS server is updated once per day. If this server handles 1000 DNS queries per hour, this results in a value for τ_{dns} of approximately $4.17 \cdot 10^{-6}$. Finally, let us consider a navigation system whose server provides traffic information to its clients. If the traffic information is updated once per minute and accessed by ten clients during this time, we are left with a value for τ_{traffic} of about 10%. From these three examples we can extrapolate that we can expect rather low values for τ for services that either have a comparatively static content or a very large number of clients issuing read-only requests. Since gathering exact data from usage patterns of popular services is beyond the scope of this work, we omit a more detailed investigation but instead evaluate the SP*i* algorithms for a range of values for τ in Section 6.7.

5.2.3 Formalization

The concept of the service provisioning cost can be formalized as follows: For a service s, given the set of nodes that form the ad hoc network N, known distances between nodes $d_{m,n}$ with $m, n \in N$, and a set of nodes hosting service instances $H_s \subseteq N$, the following parameters are used to calculate the service provisioning cost $p(s, H_s)$:

$\delta_s(c)$	Demand for service s on a client node $c \in N$ in terms of required network traffic per unit time, e.g., in kB/s
$\tau_s \geq 0$	Fraction of service traffic processed by an instance of service s that is required for synchronization with the other service instances

The cost of providing service s hosted on node $h \in H_s$ to a given client on node $c \in N$ is the product of the distance between h and c and the service demand at the client $d_{h,c} \cdot \delta_s(c)$. If there is more than one service instance, i.e., $|H_s| > 1$, then the service discovery process can be assumed to result in the closest (network topology-wise) host being used by each client. Thus, the service host used by client c to satisfy service requests for service s is $\phi(H_s, c) = \arg\min_{h \in H_s} d_{h,c}$. Thus, the cost of providing service s to a single client c is $d_{\phi(H_s,c),c} \cdot \delta_s(c)$. With the set of clients of a host h for service s given by $C_{s,h} = \{c \in N \mid \phi(H_s, c) = h\}$, the cost of a service host h can be written as

$$p_{\mathsf{clients}}(s, H_s, h) = \sum_{c \in C_{s,h}} d_{\phi(H_s,c),c} \cdot \delta_s(c)$$

The demand for service s at a host h is $\sum_{c \in C_{s,h}} \delta_s(c)$. Assuming that each host synchronizes its data directly with all other hosts in H_s, the synchronization cost for one service instance of service s hosted on node h is

$$p_{\mathsf{sync}}(s, H_s, h) = \tau_s \sum_{h' \in H_s \setminus \{h\}} d_{h,h'} \sum_{c \in C_{s,h}} \delta_s(c)$$

The service provisioning cost for a service s combines the costs incurred by each service instance $h \in H_s$ while serving client requests and while synchronizing global state and data between all service instances:

$$p(s, H_s) = \sum_{h \in H_s} p_{\mathsf{clients}}(s, H_s, h) + p_{\mathsf{sync}}(s, H_s, h)$$

Note that the service provisioning cost is measured in network traffic per unit time, just as the service demand on a client $\delta(\cdot)$.

Based on this cost function, the goal of our service placement algorithms is to find the optimal service configuration $\hat{H}_s \subseteq N$ of nodes to host a service s that minimizes the service provisioning cost, i.e., $\hat{H}_s = \arg\min_{H_s \subseteq N} p(s, H_s)$.

If used in the context of triggering the transmission of usage statistics among service instances, the service provisioning cost needs to be parametrized as follows: Instead of considering all the nodes in the network N, it only operates on the set of clients $C_{s,h}$ of a service host $h \in H_s$. Similarly, instead of considering all service hosts H_s, only the host h itself is relevant. The input value for the funnel function is thus $p(s, \{h\})$.

5.3 Adapting the Service Configuration

Now that we have defined the optimal service configuration $\hat{H}_s$ for a service s, the next step is to formalize the possible actions for adapting a service configuration. A list of these actions, as calculated by the service placement algorithms, transforms the current service configuration $\bar{H}_s$ into the optimal configuration $\hat{H}_s$.

5.3.1 Formalization of Adaptation Actions

We begin by formalizing the set of available actions which we already introduced informally in Section 4.3.3:

- $\mathsf{REP}(s, \bar{h}, \hat{h})$: Replicate an instance of service s from current service host $\bar{h} \in \bar{H}_s$ to target node $\hat{h} \in \hat{H}_s$.

- $\mathsf{MIG}(s, \bar{h}, \hat{h})$: Migrate an instance of service s from current service host $\bar{h} \in \bar{H}_s$ to target node $\hat{h} \in \hat{H}_s$.
- $\mathsf{STOP}(s, \bar{h}, \cdot)$: Shutdown an instance of service s on current service host $\bar{h} \in \bar{H}_s$.
- $\mathsf{CORD}(s, \cdot, \hat{h})$: Set node $\hat{h} \in \hat{H}_s$ to be the new coordinator for service s.

Note that we have added a forth action to the set of available actions: $\mathsf{CORD}(\cdot)$ is specific to centralized placement algorithms for distributed services, such as GCMI. It allows to control which node should coordinate the placement of service instances in the future, i.e., which node should execute the service placement algorithm. If the coordinator is moved to another node, this needs to be communicated to all service instances, since they have to send their service usage statistics and network topology information to the new coordinator in the future.

5.3.2 Service Adaptation Cost

Similarly to the service provisioning cost defined in the previous section, we now define the *service adaptation cost* that describes how costly it is to transform the current service configuration into the optimal configuration. This cost plays a crucial role when deciding about the appropriate timing of a service adaptation, which we will discuss in Section 5.6.

The service adaptation cost $a(s, \mathfrak{A}, t)$ for a service s is the sum of the cost of all actions $\mathfrak{A}$ that are required for adapting the service configuration. The parameter t corresponds to the time allocated for amortizing the investment in transmitted data required for this change of the service configuration. The intuition behind this parameter is that an adaptation is more costly if the required actions need to pay off, i.e., result in a total reduction of used bandwidth, in less time. Conversely, if there is a less immediate need for the adaptation to pay off, this reduces its cost. Section 5.7 provides an example that illustrates this dependency between adaptation cost and amortization time.

In order to calculate the adaptation cost, we need to be able to calculate the cost of implementing each of the four types of actions listed in the previous section. We begin with the replication $\mathsf{REP}(\cdot)$ and the migration $\mathsf{MIG}(\cdot)$. Given the optimizations of the processes of replicating and migration service instances as presented in Section 4.4.4, we note that the only significant contribution to the cost of these actions stems from transferring the state and application-level data of the service instance from the source to the destination node. We thus introduce a new parameter to express this property of a service instance:

σ_s Amount of data, in bytes, that needs to be transmitted when replicating or migrating an instance of service s

Note that the value for σ_s is the same for all instances of a service s. Since we assume that all instances of a service are synchronized with each other, it follows that the amount of data required for replicating or migrating any of them is the same.

The costs for the actions of replicating and migrating a service instance of service s from a current service host $\bar{h} \in \bar{H}_s$ to a target node $\hat{h} \in \hat{H}_s$ can thus be expressed as follows:

$$\begin{aligned} a_{\mathsf{action}}(\mathsf{REP}(s, \bar{h}, \hat{h}), t) &= d_{\bar{h},\hat{h}} \cdot \sigma_s \,/\, t \\ a_{\mathsf{action}}(\mathsf{MIG}(s, \bar{h}, \hat{h}), t) &= d_{\bar{h},\hat{h}} \cdot \sigma_s \,/\, t \end{aligned}$$

For the actions of shutting down a service instance and setting a new coordinator node, $\mathsf{STOP}(\cdot)$ and $\mathsf{CORD}(\cdot)$, the calculation is trivial, since implementing these actions does not cause any communication between nodes, and thus incurs no cost:

$$\begin{aligned} a_{\mathsf{action}}(\mathsf{STOP}(s, \bar{h}, \cdot), \cdot) &= 0 \\ a_{\mathsf{action}}(\mathsf{CORD}(s, \cdot, \hat{h}), \cdot) &= 0 \end{aligned}$$

With the costs for the individual actions in place, we can now formally define the cost for a complete adaptation of the configuration of a service s as

$$a(s, \mathfrak{A}, t) = \sum_{\mathfrak{a} \in \mathfrak{A}} a_{\mathsf{action}}(\mathfrak{a}, t)$$

where $\mathfrak{A} = [\mathfrak{a}_1, \cdots, \mathfrak{a}_n]$ with $\mathfrak{a}_{1 \cdots n} \in \{\mathsf{REP}(\cdot), \mathsf{MIG}(\cdot), \mathsf{STOP}(\cdot), \mathsf{CORD}(\cdot)\}$ is a list of n actions for adapting the service configuration. This list is calculated by a service placement algorithm.

Note that both the service provisioning cost $p(\cdot)$ and the service adaptation cost $a(\cdot)$ are measured in network traffic per unit time. It is thus possible to compare them directly with another. We make use of this property when deciding about the optimal timing for adapting a service configuration (cf. Sec. 5.6).

5.4 The Graph Cost / Single Instance Algorithm

Like most approaches to the placement of a centralized service, the Graph Cost / Single Instance algorithm (GCSI) is very simple. This is due to the fact that as long as there is only a single service instance that needs to be placed and if service demand and network topology are known, we can efficiently enumerate all possible placements and calculate the cost for each. The solution corresponds to the absolute median of the graph representation of the network (cf. Sec. 2.3.1).

Algorithm 1: Graph Cost / Single Instance (GCSI)

Data: The set of nodes in the network N, the distance $d_{m,n}$ between nodes m and n, the demand $\delta_s(n)$ for the *centralized* service s on node n, the size of the service σ_s, and the current service configuration $\bar{H}_s$.

Result: A list of actions $\mathfrak{A}$ required for transforming $\bar{H}_s$ into the optimal configuration $\hat{H}_s$.

```
1 p̂ ← ∞
2 foreach n ∈ N do
3     p_n ← p(s, {n})      // Calculate the provisioning cost with n as host.
4     if p_n < p̂ then      // Store n and p_n if they improve the optimum.
5         Ĥ_s ← {n}
6         p̂ ← p_n
  /* At this point, Ĥ_s contains the optimal service configuration.  */
7 𝔄 ←[MIG(h̄, ĥ)] where h̄ ∈ H̄_s and ĥ ∈ Ĥ_s
8 return 𝔄
```

5.4.1 Algorithm

The algorithm takes the service demand and the network topology as input and calculates the provisioning cost for all candidate hosts. Since there is by definition only a single service instance, there is no synchronization traffic and we can set $\tau = 0$ independently of the service. The host with the lowest cost is then obviously the optimal candidate for hosting the service instance.

This very simple algorithm is formalized in Algorithm 1. Note that we continue to use the same notation as introduced in Section 5.2.3. In lines 2 to 6, we loop over all known nodes, calculate the service provisioning cost with the current node as host in line 3, and then store the optimal known configuration in lines 4 to 6.

The run time of this algorithm is obviously proportional to the number of nodes $|N|$. However, the cost function relies upon the knowledge about the distances between all nodes $d_{m,n}$ as one of its inputs. This distance matrix is the solution to an all-pairs shortest path problem which can be calculated using the Floyd-Warshall algorithm in a run time proportional to $|N|^3$ [21, p. 693ff.]. Hence, this corresponds to the dominant factor in the run time of the GCSI algorithm. By the same argument, the memory required by the algorithm is proportional to $|N|^2$.

5.4.2 Implementation Considerations

In real networks, in particular in networks with very lossy links, it may not be desirable to migrate a service instance directly to the optimal host. This is especially the case if

the current service host and the new service host are far away from each other (in terms of forwarding hops) and the amount of data that needs to be transferred is large. The reliability of the end-to-end connection between the two nodes involved in the service migration decreases with every additional hop. As a consequence, each data packet is more likely to require multiple retransmissions and the time required until the migration of the service instance is completed is likely to increase. For target nodes that are very far away, it is even possible that the migration is aborted due to timeouts. The cost of service migrations to a distant node is thus harder to predict and hence increases the uncertainty under which the service placement algorithm operates.

In order to work around this problem, we limit the maximal distance between the current service host and the migration target in the presence of lossy links. We do so by introducing a configurable parameter d_{max} that controls how far source and destination of the migration may be apart from each other. This parameter is then used to constrain the set of potential migration targets. Instead of iterating over all nodes in the network N in line 2 of Algorithm 1, we iterate over $\{n \in N \mid \exists_{h \in \bar{H}_s} d_{h,n} < d_{\mathsf{max}}\}$ where $\bar{H}_s$ is the current configuration of service s. The parameter d_{max} can be adapted at run time depending on the measured link quality, the number of retransmissions that were necessary during a service migration, or the number of failed service migrations.

The drawback of this approach is that multiple migrations may become necessary for the service instance to reach the optimal service host. We deem this to be an acceptable tradeoff for the improved predictability of the migration process.

5.5 The Graph Cost / Multiple Instances Algorithm

The Graph Cost / Multiple Instances algorithm (GCMI) is a centralized algorithm that solves the service placement problem for distributed services in ad hoc networks. It is built around the observation that, provided the clients' service discovery component employs a metric that selects the nearest service instance, the mapping of clients to service instances induces a clustering of the network. For a service s, the goal of the algorithm is thus to calculate the clustering with the lowest provisioning cost $p(s, \hat{H}_s)$ in which the optimal service configuration $\hat{H}_s$ corresponds to the set of cluster heads.

The GCMI algorithm is executed on a coordinator node, i.e., there is one node for each service in the ad hoc network that controls the placement and the number of service instances. The coordinator is, however, not a fixed, infrastructural node. Instead, the task of coordinating the configuration of a service is assigned dynamically to the most suitable node. Since the coordinator needs to receive service usage statistics and network topology information from all service instances, we assign this role to the service instance that is

most centrally located. This corresponds to the service instance for which the distance to all other service instances is minimal. More formally, the coordinator of a service s with the service configuration H_s is $\Phi(H_s) = \arg\min_{h \in H_s} \sum_{h' \in H_s \setminus \{h\}} d_{h,h'}$.

5.5.1 Algorithmic Background and Rationale

The GCMI algorithm calculates an approximation to the solution of the NP-hard Uncapacitated Facility Location Problem (UFLP) as introduced in Section 2.3.2. It does so by following a similar approach as the Agglomerative Median method (AM-NNA) presented by Domínguez Merino et al. in [75] which proposes to solve the strongly related p-median problem in the following way: The algorithm starts with $|N|$ clusters, each containing a single node. It then iteratively merges pairs of adjacent clusters that are selected using a cost function, and terminates when the number of clusters reaches p.

With the GCMI algorithm we follow a similar line of thinking. However, since in our case the network may also comprise nodes without demand for the service, we start by only creating clusters with client nodes as cluster heads (instead of all nodes). In a second initialization step, we then add all passive nodes to the cluster whose cluster head is closest. Afterwards, we iteratively merge those pairs of adjacent clusters that have the minimal combined service demand. Instead of continuing this process until the number of clusters reaches the value of a given parameter p, we calculate the service provisioning cost $p(s, H_s)$ after each iteration and return the set of nodes that yields the lowest value of this function. This set of nodes corresponds to the optimal service configuration according to GCMI.

Furthermore, we expand upon this approach by additionally integrating an optimization step as proposed by Santos in [94]. This optimization step addresses the problem that the placement of cluster heads within a cluster may deteriorate as clusters are being merged during each iteration of the algorithm. It is thus suggested that the role of cluster head should be reassigned to the most centrally located node within each cluster after each merger. Additionally to improving the solution of the UFLP, this optimization is of critical importance for the correct operation of our service placement algorithms. It ensures that the placement algorithms properly take the behavior of the service discovery components on the clients into account. Since each client selects the closest service instance, the placement algorithm needs to ensure that the service instance for a specific set of clients is placed in such a way that no other service instance is closer to any of these clients. Otherwise, the clients would select another service instance and the mapping between clients and service instance in the actual network would diverge from the network as modeled by the placement algorithm, thereby leading to an overall reduction of the placement quality. Reassigning the cluster heads to a central node within each cluster effectively mitigates this problem.

When compared to other approaches to solving the UFLP, in particular the greedy heuristics presented in [22, Sec. 3.5], our approach has the advantage that it induces a natural load balancing between the cluster heads, i.e., between the service instances. This is due to the choice of adjacent clusters for merging in the main loop of the GCMI algorithm. Since we always select the pair of clusters with the minimal combined service demand, the clients' service requests are spread out equally among all service instances. This property of the algorithm avoids that a centrally placed service instance serves a significantly higher service demand than others. It thus spreads the network load more equally throughout the entire ad hoc network. As one can assume spatial diversity in the radio communication of multi-hop ad hoc networks, this results in a regionally spread out bandwidth usage across the network. We thereby avoid creating local communication bottlenecks that would cause the quality of the service to deteriorate for those clients that access the service via an overly busy service instance.

5.5.2 Service Provisioning Cost Revisited

Before we continue to discuss the GCMI algorithm in detail, we have to amend the calculation of the service provisioning cost to take the algorithm-specific communication overhead into account. In other words, we develop the calculation of the generic service provisioning cost $p(s, H_s)$ (cf. Sec. 5.2) into algorithm-specific cost $p_{\mathsf{gcmi}}(s, H_s)$.

The centrally-executed GCMI algorithm requires up-to-date information about the service demand and the regional network topology from all service instances as input. We have already discussed in Section 4.3.3 how and when this data can be transmitted. However, as the amount of information depends on the number of nodes in the vicinity of each service instance, and the rate at which this information is transmitted depends on the volatility of the service demand, we need to make simplifying assumptions for the calculation. A conservative estimation is that the required information will not exceed the Maximum Transmission Unit (MTU) of the network stack, i.e., we assume that the signaling information fits into a single packet. About the time interval after which this information is transmitted, we just know that it depends on the parameter d of the funnel function used for triggering the transmission (cf. Sec. 4.3.3). For the sake of clarity, we will refer to this parameter as d_{TRIGGER} in the current context. If the value for d_{TRIGGER} is chosen in such a way that the trigger reacts appropriately to changes in service demand under normal operation of the network, it makes sense to assume it may fire once after an average interval of $d_{\mathsf{TRIGGER}} \,/\, 2$. Thus, the signaling overhead specific to the GCMI algorithm can be expressed as

$$p_{\mathsf{signaling/gcmi}}(s, H_s, h) = d_{h,\Phi(H_s)} \cdot \frac{\mathsf{MTU}}{d_{\mathsf{TRIGGER}} \,/\, 2}$$

Given the assumptions that led to this formula one might be conrned about the accuracy of the result. Fortunately, it is by its magnitude the most insignificant factor in the service provisioning cost $p_{\mathsf{gcmi}}(s, H_s)$ and hence we deem the inaccuracies to be acceptable.

Furthermore, since GCMI is a centralized placement algorithm, we can leverage the presence of a central hub in the network to reduce the cost of synchronization. When calculating $p(s, H_s)$ in Section 5.2, we worked on the assumption that every service instance would synchronize directly with every other instance. With a centrally placed hub, we can reduce this cost by sending the synchronization data to the hub and distributing it from there to all other instances. The synchronization cost for GCMI is thus reduced to

$$p_{\mathsf{sync/gcmi}}(s, H_s, h) = \tau_s \left(d_{h,\Phi(H_s)} + \sum_{h' \in H_s \setminus \{h, \Phi(H_s)\}} d_{\Phi(H_s),h'} \right) \sum_{c \in C_{s,h}} \delta_s(c)$$

One additional benefit of this change, which is not captured in this formalism, is the fact that both signaling and synchronization traffic make use of the same routes, i.e., the overhead with regard to route maintenance is kept to a minimum. A potential drawback is the additional load that is put on the coordinator node and the nearby radio links. In fact, if the volume of synchronization traffic grows large enough, it may even become the bottle neck in the system. In this case, it is more sensible to skip this optimization. Alternatively, more sophisticated solutions for synchronization could to be employed, e.g., having several service instances serve as synchronization hubs in a two-tier hierarchy. We leave this refinement of SP*i* for future work.

With these two changes to the calculation of the service provisioning cost in place, we are left with a total service provisioning cost of

$$p_{\mathsf{gcmi}}(s, H_s) = \sum_{h \in H_s} p_{\mathsf{clients}}(s, H_s, h) + p_{\mathsf{sync/gcmi}}(s, H_s, h) + p_{\mathsf{signaling/gcmi}}(s, H_s, h)$$

This metric is the cost metric that we will use as part of the GCMI algorithm.

5.5.3 Algorithm

The GCMI service placement algorithm requires the topology of the network, the service demand on the client nodes, and the fraction of synchronization traffic τ_s for the service s it is tasked with placing as inputs. The output of the algorithm is a list of actions that change the current service configuration $\bar{H}_s$ into the optimal service configuration $\hat{H}_s$.

As already discussed in Section 5.5.1, we initialize our data structures to the valid but suboptimal configuration of having one service instance per client, i.e., each client node is its own cluster head. Passive nodes are added to the cluster with the closest

cluster head. The main loop iteratively merges the two adjacent clusters with the lowest combined cost $\sum_{c \in C_{s,h}} \delta_s(c)$. At each step of the iteration, we calculate the provisioning cost $p_{\mathsf{gcmi}}(s, H_s)$ and retain the configuration with the lowest cost. Finally, we calculate the least expensive set of actions that transform the current into the optimal configuration and adjust the coordinator if necessary.

The complete GCMI service placement algorithm is given in Algorithm 2. It relies upon two external functions `optimizeClusters` and `calculateAdaptationActions` which we will discuss separately. Furthermore, it should be noted that for the sake of discussion, we present the algorithm in a simplified form, especially concerning overall structure and data representation. The ANSI C implementation of the algorithm amounts to more than 1500 source lines of code, not counting utility functions and basic data structures.

The algorithm begins by initializing the set of all cluster $\mathcal{C}$ in lines 1 to 9. Note that a cluster is a set of nodes C_i where the index i corresponds to the node that is the cluster head. Obviously, all cluster heads must always be contained in their own cluster, i.e., $i \in C_i$ is always true for all clusters. In lines 10 and 11, we then initialize the variables for the best currently known service configuration and service provisioning cost $\hat{H}_s$ and $\hat{p}$. Note that $\hat{H}_s$ is derived from the set of all clusters $\mathcal{C}$ and corresponds to the set of their cluster heads.

From line 12 to line 27 the algorithm then iteratively merges two clusters until there are no clusters left for merging, i.e., until $|\mathcal{C}| = 1$. All adjacent pairs of nodes are considered for merging. Adjacent in this context means that the clusters are directly connected to each other on the data link layer, i.e., there is at least one pair of nodes with one node from each cluster who are exactly one hop away from each other. This property can be expressed as $\exists_{x \in C'_n, y \in C'_m} d_{x,y} = 1$ where C'_n and C'_m are the clusters being tested for adjacency (cf. line 15). Of all these merge candidates, the algorithm selects the pair in which the nodes have the lowest combined service demand $\sum_{x \in C'_n \cup C'_m} \delta_s(x)$ for merging.

The merger of the two cluster is then performed in lines 20 to 22. Merging two clusters consists of two steps: First, all nodes from one cluster are added to the other cluster, and then the first cluster is removed from the set of all clusters $\mathcal{C}$. This procedure does not change the cluster head of the remaining cluster. As more and more clusters are merged, the cluster head, i.e., the service instance, is moved to a suboptimal position within the cluster of its client nodes. In fact, since the service discovery component on each client selects its service instance following a local quality metric (cf. Sec. 4.3.2), it is quite likely that the clients will pick a service instance from another cluster later on. It is thus fundamental for the GCMI algorithm, that the property of each cluster head being the closest to the nodes in its cluster among all cluster heads is always preserved. Formally, the invariant $\forall_{n \in N}\, \phi(H_s, n) = h \Leftrightarrow n \in C_h$ always needs to remain true. Since a merger of two cluster will certainly invalidate this invariant, it needs to be restored by

Algorithm 2: Graph Cost / Multiple Instances (GCMI)

Data: The set of nodes in the network N, the distance $d_{m,n}$ between nodes m and n, the demand $\delta_s(n)$ for the *distributed* service s on node n, the size and the synchronization ratio of the service σ_s and τ_s, and the current service configuration $\bar{H}_s$.

Result: A list of actions $\mathfrak{A}$ required for transforming $\bar{H}_s$ into the optimal configuration $\hat{H}_s$.

$\mathcal{C} \leftarrow \emptyset$
foreach $n \in N$ **do** `// Create a cluster for each client node.`
  if $\delta_s(n) > 0$ **then**
    $C_n \leftarrow \{n\}$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_n\}$
foreach $n \in N$ **do** `// Add remaining nodes to the closest cluster.`
  if $\delta_s(n) = 0$ **then**
    $C_{\mathsf{closest}} \leftarrow \arg\min_{C_x \in \mathcal{C}} d_{n,x}$
    $C_{\mathsf{closest}} \leftarrow C_{\mathsf{closest}} \cup \{n\}$
$\hat{H}_s \leftarrow \{x \mid C_x \in \mathcal{C}\}$ `// Initialize with trivial placement.`
$\hat{p} \leftarrow p_{\mathsf{gcmi}}(s, \hat{H}_s)$
while $|\mathcal{C}| > 1$ **do**
  $\delta_{\mathsf{clusters}} \leftarrow \infty$ `// Find adjacent clusters with minimal demand.`
  foreach $(C'_n, C'_m) \in \mathcal{C} \cdot \mathcal{C}$ **do**
    if $C'_n \neq C'_m \ \wedge \ \exists_{x \in C'_n, y \in C'_m} d_{x,y} = 1$ **then**
      $\delta'_{\mathsf{clusters}} \leftarrow \sum_{x \in C'_n \cup C'_m} \delta_s(x)$
      if $\delta'_{\mathsf{clusters}} < \delta_{\mathsf{clusters}}$ **then**
        $(C_n, C_m) \leftarrow (C'_n, C'_m)$
        $\delta_{\mathsf{clusters}} \leftarrow \delta'_{\mathsf{clusters}}$
  $C_n \leftarrow C_n \cup C_m$ `// Merge these two clusters.`
  $\mathcal{C} \leftarrow \mathcal{C} \setminus \{C_m\}$
  $\mathcal{C} \leftarrow$ `optimizeClusters`$(s, \mathcal{C})$
  $H_s \leftarrow \{x \mid C_x \in \mathcal{C}\}$ `// Recalculate provisioning cost.`
  $p \leftarrow p_{\mathsf{gcmi}}(s, H_s)$
  if $p < \hat{p}$ **then** `// Store improved configuration.`
    $\hat{H}_s \leftarrow H_s$
    $\hat{p} \leftarrow p$
`/*` $\hat{H}_s$ `now contains an approximation of the optimal configuration. */`
$\mathfrak{A} \leftarrow$ `calculateAdaptationActions`$(s, \bar{H}_s, \hat{H}_s)$
return $\mathfrak{A}$

Function `optimizeClusters`$(s, \mathcal{C})$

Data: The distance $d_{m,n}$ between nodes m and n, the demand $\delta_s(n)$ for service s on node n, and a clustering $\mathcal{C}$.

Result: An optimization $\hat{\mathcal{C}}$ of the clustering $\mathcal{C}$ in which nodes are moved between clusters depending on the nearest cluster head and cluster heads are centered within their cluster.

```
1  Ĉ ← C
2  repeat                                  // Repeat until cluster heads remain stable.
3      cluster_heads_changed ← false
4      H_s ← {x | C_x ∈ C}
5      foreach C_h ∈ Ĉ do                              // Iterate over all clusters.
6          foreach n ∈ C_h do
7              h_closest ← φ(H_s, n)    // Find closest cluster head for each node.
8              if h ≠ h_closest then
9                  C_h ← C_h \ {n}      // Move node to cluster with closest head.
10                 C_{h_closest} ← C_{h_closest} ∪ {n}
11         h_center ← arg min_{n∈C_h} Σ_{n'∈C_h\{n}} d_{n,n'} · δ_s(n')      // Find central node.
12         if h ≠ h_center then
13             Ĉ ← Ĉ \ {C_h}                // Move cluster head to central node.
14             C_{h_center} ← C_h
15             Ĉ ← Ĉ ∪ {C_{h_center}}
16             cluster_heads_changed ← true
17         if cluster_heads_changed then
18             break    // Restart iterations if cluster heads have changed.
19 until ¬cluster_heads_changed
20 return Ĉ
```

reassigning the role of cluster head appropriately in each cluster. This is implemented in the function `optimizeClusters` which will be discussed later on.

In lines 23 to 27, the algorithm recalculates the service provisioning cost and, if it improves the best currently known solution, updates the optimal service configuration $\hat{H}_s$. Finally, in lines 28, the algorithm calculates the adaptation plan and its cost by calling the function calculateAdaptationActions. The adaptation plan is then returned to the SPi middleware and the run of GCMI is complete.

The GCMI service placement algorithm employs two subroutines, `optimizeClusters` and `calculateAdaptationActions`. The task of `optimizeClusters` is to ensure that the invariant of each cluster head being the closest to the nodes in its cluster among all cluster heads is always preserved. The two operations that are employed in order to achieve this

Function `calculateAdaptationActions`$(s, \bar{H}_s, \hat{H}_s)$

Data: The distance $d_{m,n}$ between nodes m and n, the current and the optimal configuration $\bar{H}_s$ and $\hat{H}_s$ of service s, and size of the data of this service σ_s that needs to be transferred when replication.

Result: A list of actions $\mathfrak{A}$ required for changing $\bar{H}_s$ into $\hat{H}_s$.

```
1  𝔄 ← []
2  𝔄 ← append(𝔄, CORD(Φ(Ĥ_s)))                    // Establish new coordinator node.
3  foreach ĥ ∈ Ĥ_s do                    // Find replication sources for all target hosts.
4      h̄ ← arg min_{h∈H̄_s} d_{h,ĥ}
5      if h̄ ≠ ĥ then 𝔄 ← append(𝔄, REP(h̄, ĥ))
6  foreach h̄ ∈ H̄_s do                          // Convert last replication into migration.
7      if h̄ ∉ Ĥ_s ∧ REP(h̄, ·) ∈ 𝔄 then
8          foreach REP(s, d) ∈ {𝔞 ∈ 𝔄 | 𝔞 = REP(·)} do
9              if s = h̄ then h_last ← d
10         𝔄 ← replace(𝔄, REP(h̄, h_last), MIG(h̄, h_last))
11 foreach h̄ ∈ H̄_s do                                   // Shutdown remaining hosts.
12     if h̄ ∉ Ĥ_s ∧ MIG(h̄, ∗) ∉ 𝔄 then 𝔄 ← append(𝔄, STOP(h̄))
13 return 𝔄
```

are selectively moving nodes from a cluster to an adjacent cluster and reassigning the role of cluster head within a cluster.

In the implementation of `optimizeClusters`, these two steps are handled in lines 6 to 10 and lines 11 to 16 respectively, and applied to all clusters by the surrounding **for**-loop. Note that a flag is set in line 16, if the role of cluster head has been reassigned to another node. This flag is used to restart the iterations over all clusters because a change in the location of one of the cluster heads may invalidate the previous assignment of nodes to the respective closest cluster head. Hence, we have to repeat the process until no changes in the location of cluster heads are required. The function terminates because with each iteration of the outer loop less nodes need to be moved between clusters and hence the cluster heads stabilize. Even in atypical scenarios such as uniform service demand across regularly placed nodes in a simulation, we have never observed more than very few, if any, iterations of the outer loop of the function.

As the final piece of the GCMI algorithm, the function `calculateAdaptationActions` calculates the list of actions which are necessary in order to transform the current service configuration $\bar{H}_s$ into the optimal service configuration $\hat{H}_s$. The function begins with appending the command for adjusting the coordinator node to the list of actions $\mathfrak{A}$ in line 2. This is done independently of whether the coordinator needs to be changed or not, because

the case of an unchanged coordinator, i.e., $\Phi(\bar{H}_s) = \Phi(\hat{H}_s)$, does not require any special treatment on the service instances.

In the first loop from line 3 to line 5, a replication is added to the list of actions for each new service instance. The source nodes for these replications are the closest service instances of the respective target node. In the following loop from line 6 to line 10, the function finds the last replication from each current host that is not part of the optimal configuration. This replication is then replaced with a migration that retains the same source and destination nodes. In the last loop in lines 11 and 12, the algorithm stops those service instances of the current configuration that are neither contained in the optimal configuration nor the source of a previously added migration. Finally, the complete list of the actions required for the adaptation of the service is returned to the caller.

The calculation of the run time of the GCMI algorithm is complicated by the circumstance that the size of the set of all clusters $\mathcal{C}$ decreases throughout the execution of the algorithm, while the sizes of all the clusters contained in this set increase on average. For the sake of simplicity, we introduce two new variables C_{avg} and H_{avg} that correspond in their size to the average cluster and to the average service configuration during the execution of the algorithm. An obvious upper bound for the size of all of these variables is the total number of nodes $|N|$.

The function `calculateAdaptationActions` does not contribute significantly to the run time of GCMI since it merely consists of three iterations over $|H_{\mathsf{avg}}|$ elements. The run time of `optimizeClusters` is more significant. It is proportional to $|\mathcal{C}| \cdot |C_{\mathsf{avg}}| \cdot |H_{\mathsf{avg}}|$ due to its two nested loops and the calculation of $\phi(\cdot)$. An upper bound for the run time of this function is thus $|N|^3$. In the main part of the GCMI algorithm, the run time is dominated by the selection of the optimal pair of clusters for merging. The calculation has a run time proportional to $|\mathcal{C}|^2 \cdot |C_{\mathsf{avg}}|^2$, where $|C_{\mathsf{avg}}|^2$ is an upper bound for the steps required for checking whether two clusters are adjacent. This can be reduced by augmenting the data structure of each cluster with an additional set storing its adjacent clusters in the current configuration. As a result, the run time drops to $|\mathcal{C}|^2 \cdot |C_{\mathsf{avg}}|$, for which once again $|N|^3$ is an upper bound. As the outer **while**-loop of the algorithm is executed $|N|$ times, this leaves us with an upper bound for the run time of the GCMI service placement algorithm proportional to $|N|^4$.

5.5.4 Preliminary Evaluation

We will now briefly examine the interdependencies between service provisioning cost, number of service instances, and synchronization traffic ratio. To this end, we employ the same simulation parameters as already used during the preliminary evaluation of the cost of

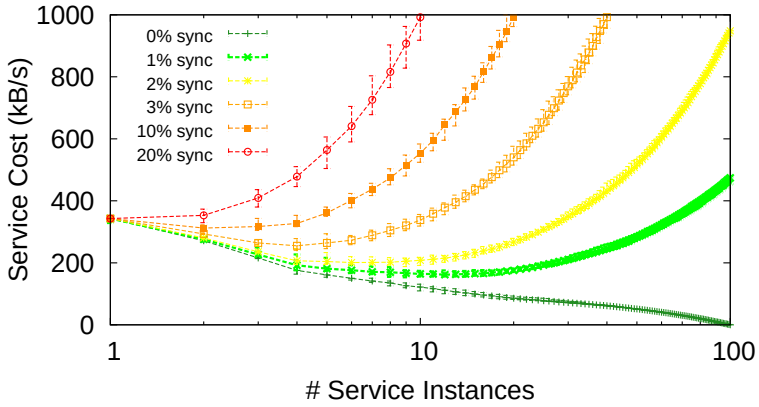


Figure 5.2: Service provisioning cost $p_{\mathsf{gcmi}}(s, H_s)$ for different service configurations

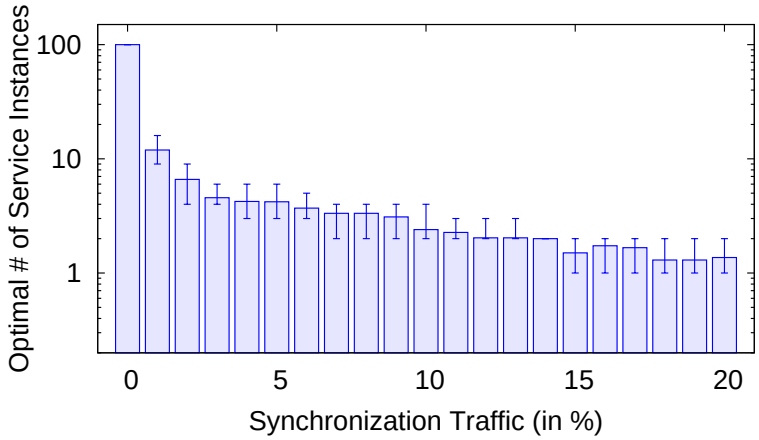


Figure 5.3: Optimal number of service instances $|\hat{H}_s|$ depending on the synchronization traffic ratio τ_s

migrating service instances in Section 4.4.5. A comprehensive discussion of the simulation setup can be found in Section 6.3.1.

Once again, the simulated network consists of 100 nodes that are regularly placed in a 10 by 10 grid so that each node has a reliable radio link to each of its direct neighbors. A centralized service with a single service instance is started on one of the four central nodes. All 100 nodes were programmed to locate the current service host and then request one 1 KB data item per second. We simulated the network for a time of 15 minutes after a 1-minute setup period and aggregated data from 30 runs of the simulation.

Figure 5.2 shows the service provisioning cost $p_{\mathsf{gcmi}}(s, H_s)$ for different numbers of service instances $|H_s|$ and different values for the ratio of synchronization traffic τ_s. The corner

case of $\tau_s = 0$ is clearly visible as the only graph that is decreasing monotonously with its minimum in the configuration that corresponds to having one service instance per client. For the values of τ_s between 1% and 20%, the cost function has a minimum between 13 and 1 service instances respectively. For a service with $\tau_s = 1\%$, the optimal configuration with 13 instances reduces the service provisioning cost compared to a configuration with a single instance from averaged 342.4 kB/s to 162.8 kB/s, i.e., by more than 50%.

Figure 5.3 shows the number of service instances of the optimal configuration for different values of τ_s. We observe that the prediction based on the illustration in Figure 5.1 was indeed correct, and the number of service instances in the optimal service configuration increases as the synchronization traffic ratio decreases.

5.5.5 Implementation Considerations

When implementing the GCMI algorithm in real networks, there are two facts worth considering. First, the run time of the algorithm is non-negligible and should be reduced if possible. And second, the network load caused by service replication and migration may prove too high for some nodes and thus needs to be limited.

Reduction of Run Time by Randomized Initialization In order to reduce the run time of the GCMI algorithm, we begin with the observation that each iteration of the GCMI algorithm results in an intermediate service configuration that comprises one service instance less than the configuration of the previous step. Since the algorithm is initialized with a configuration that comprises one service instance for each client, one can visualize this process as generating the service provisioning cost curve (cf. Fig. 5.2) from the maximum to the minimum number of clients, i.e., moving from right to left on the x-axis of the plot. We can furthermore observe that for most scenarios the number of service instances in relation to the total number of nodes is comparatively small. These two observations lead to the assumption that the major part of the iterations of the GCMI main loop, i.e., those that occur before reaching the region of the global minimum, contribute little to the overall quality of the solution.

We can now leverage this assumption to reduce the run time of the algorithm, by changing the initialization to create randomly a predefined number of cluster and then begin to merge them from this starting point. If the number of the initial clusters is chosen correctly, the algorithm will eventually reach the same global minimum just as if it had been initialized with one cluster per client node. The obvious problem with this approach is that there is no way to know in advance whether the global minimum will be reached based on a given number of initial clusters. Therefore, we reiterate this process several times, each time with a significantly increased number of initial clusters. If two consecutive iterations result in

the same service configuration, we take this as an indication that the globally minimal cost and thus the optimal service configuration has been found.

This approach works well for scenarios with very few service instances (in relation to the number of client nodes), i.e., for scenarios with a comparatively large τ_s. For scenarios that require a large number of service instances, it results in an increase in run time since the new outer loop needs several iterations until it reaches the correct value for the number of initial clusters. Since one can assume that the relation between clients and the optimal number of service instances does not change abruptly in a real-world deployment, it makes thus sense to only make use of this optimization in case the aforementioned ratio is below a configurable threshold.

Controlling Adaptation Overhead by Limiting Replication Targets Just as already pointed out when discussing the implementation of the GCSI algorithm, lossy links adversely affect the predictability of the replication of service instances from one node to the other (cf. Sec. 5.4.2). The same is obviously also true for the GCMI algorithm, and it thus makes sense to limit the distance between current service instances and replication targets by using the same parameter d_{max}.

Following the same line of thought, it makes also sense to limit number of replications per service instance. If the number of replications per service instance was unlimited, then a service instance could remain in the `LOCKED` state for a potentially long period of time. Similar to the limit on replication distance, the limit on the number of replications may impede that the optimal service configuration is reached directly. This problem, however, can once again be mitigated by further improving the service configuration in the future after the current service adaptation has been completed.

In order to limit the number of replications per service instance, we introduce a new parameter $\mathsf{REP}_{\mathsf{max}}$ and extend line 3 to 5 of Function calculateAdaptationActions in such a way, that a current service host is removed from the set of potential replication sources if it is already the source node of more than $\mathsf{REP}_{\mathsf{max}}$ replications.

5.6 Deciding on the Timing of Service Adaptations

Finding the optimal configuration for a service is only part of the solution to the service placement problem in ad hoc networks. Especially in scenarios with dynamically changing service demand or node mobility, it is furthermore crucial to find the correct moment at which the adaptation from the current service configuration to the optimal configuration (as established by the placement algorithm at this particular point in time) should be performed.

Our approach to solve this problem is to keep track of the difference in cost between the current and the optimal service configuration. If this difference is big enough for a sufficiently long period of time, then the service configuration is adapted. In this calculation we explicitly consider the service adaptation cost, i.e., the bandwidth used by the data transfers required for implementing the adaptation (cf. Sec. 5.3.2). As a result, we get a dependency between the size of the serialized service data and the confidence required for adapting the configuration of large services.

5.6.1 Motivation and Preconsiderations

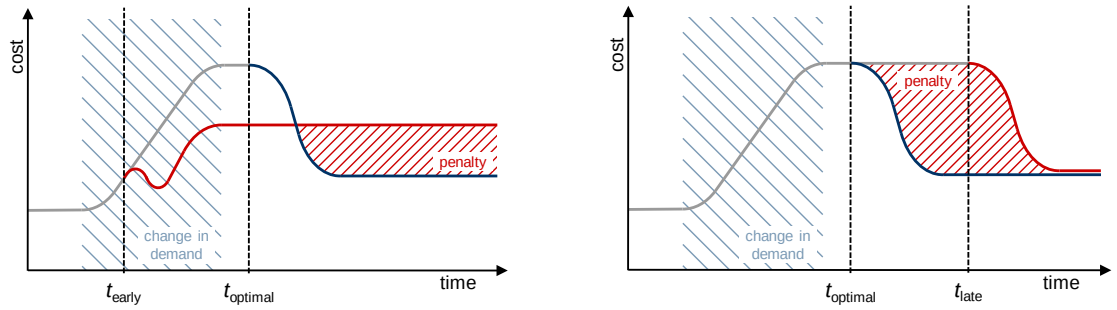
The timing of service adaptations is non-trivial. Since a service adaptation incurs the cost of transferring the service data and updating the configuration of all client nodes (cf. Sec. 4.4), the service adaptation can essentially be thought of as an investment under uncertainty. This mainly stems from the fact that the future behavior of the clients and the network is unknown. In order to answer the question of when to adapt the service configuration, it is thus required to make predictions about the upcoming behavior of the network.

The quality of these predictions is important. Consider, for instance, the two cases that may occur if a service configuration is adapted at the wrong point in time: If the service adaptation has, in retrospect, been performed too early, then this implies that the service placement algorithm would have made a different and superior choice at the latter point in time. Hence, the actual return on investment (with regard to the service adaptation cost) is not as high as it could have been if the system had waited for more information to become available to the placement algorithm.

This situation is illustrated in Figure 5.4a. As the service demand or the network topology changes, the service provisioning cost increases. The gray line corresponds to the service provisioning cost as long as no placement adaptation is performed. The blue line corresponds to the result of a service adaptation at the correct time; the red line corresponds to a service adaptation that was performed prematurely. As can be seen, the premature adaptation does not reach the same low level of service provisioning cost as the adaptation that was performed at the optimal time. The difference between these two timing decisions is highlighted in red and corresponds to the penalty of a premature adaptation.

Similarly, an adaptation that is performed too late also incurs unnecessary cost. This is illustrated in Figure 5.4b. As can be seen, the provisioning cost remains at a high level for a longer period of time while all information for establishing the optimal service configuration is in fact already available. Once again, the unnecessary cost is highlighted in red.

Two conclusions can be drawn from these two examples: First, and already pointed out, the timing of service adaptations is crucial to keep the service provisioning cost at a



(a) Early adaptation decision

(b) Late adaptation decision

Figure 5.4: Adaptation timing and service provisioning cost

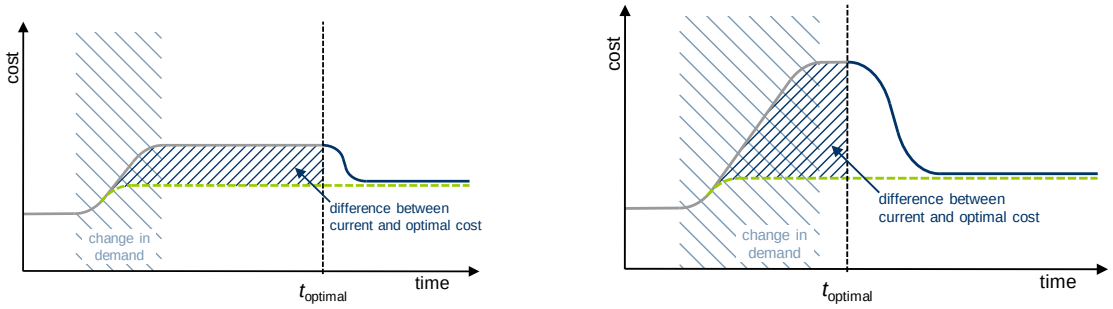
minimum. And second, a premature adaptation is slightly worse than an adaptation that occurs too late. This is due to the fact that the excessive cost is only temporary for a late adaptation, but for a premature adaptation it remains in place until corrected by a second adaptation.

One architectural alternative to attempting to predict the future behavior would be to have the clients transmit information about their expected service usage as part of their service requests or even during the service discovery process. This could possibly take a similar form as suggested for protocols that deal with distributed resource allocation in computer networks, e.g., the Resource ReSerVation Protocol (RSVP) [15]. Together with advanced mobility and radio models it might be possible to make acceptable predictions about the future state of the network. A full implementation and evaluation of this approach is, however, beyond the scope of our current work.

5.6.2 Service Adaptation Condition

The timing decision used for the SP*i* service placement algorithms relies on a very simple assumption: We assume that the overall behavior of the ad hoc network, including client demand and changes in the topology, in the near future will match the behavior of the recent past. In other words, if our placement algorithm has been able to continuously find a superior service configuration $\hat{H}$ for a past time interval of duration t, then we assume that this configuration $\hat{H}$ will remain superior over the current configuration $\bar{H}$ for at least another (future) time interval of the same duration t. One of the key reason for choosing this approach is its simplicity over other approaches that attempt to predict the future behavior of the network as discussed in the previous section.

Based on this assumption, we can now formulate the condition for adapting a service configuration: We consider an adaptation of the configuration of a service s worth while,



(a) Small change in service demand

(b) Large change in service demand

Figure 5.5: Adaptation timing decision

if the difference in provisioning cost between the current and the optimal configuration $p(s, \bar{H}_s) - p(s, \hat{H}_s)$ exceeds the currently estimated service adaptation cost $a(s, \mathfrak{A}, t)$. If we adapt the service configuration at the point in time when this condition becomes true, and if the behavior of the service does not change to our disadvantage for the same period in the future (as stipulated by our assumption), then we know that an investment corresponding the service adaptation cost will at least break even. In other words, if the network continues to operate as assumed, then the future savings in service provisioning cost can safely be expected to pay off for the current investment in service adaptation cost.

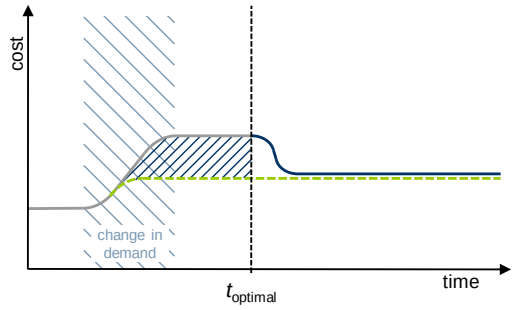
Formally, an adaptation of the configuration of a service s, being placed with either the GCSI or the GCMI service placement algorithm, is initiated as soon as the following statement becomes true:

$$a(s, \mathfrak{A}, t_{\mathsf{current}} - t_{\mathsf{superior}}) \; < \; p(s, \bar{H}_s) - p(s, \hat{H}_s)$$

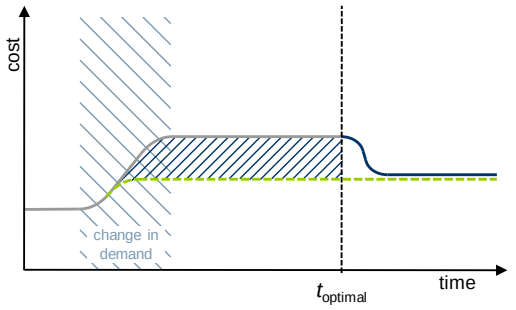
where $\mathfrak{A}$ is list of adaptation actions as calculated by the placement algorithm, $\bar{H}_s$ and $\hat{H}_s$ are the current and the optimal service configuration, t_{current} is the current time, and t_{superior} is the point in time a since which a superior configuration has been known, i.e., since which $p(s, \hat{H}_s) < p(s, \bar{H}_s)$ has been true for all runs of the placement algorithm.

5.6.3 Discussion

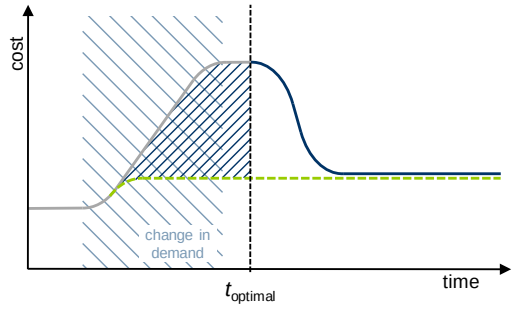
The condition for triggering the adaptation of a service s has the important property that it is equally applicable to services of varying sizes σ_s and in situations of varying differences between current and optimal service provisioning cost $p(s, \bar{H}_s) - p(s, \hat{H}_s)$. For illustration, we have plotted two scenarios in Figure 5.5, one with a small change in service demand and one with a large change. For the sake of illustration and without losing generality, let us assume that the service adaptation cost is the same for both scenarios.



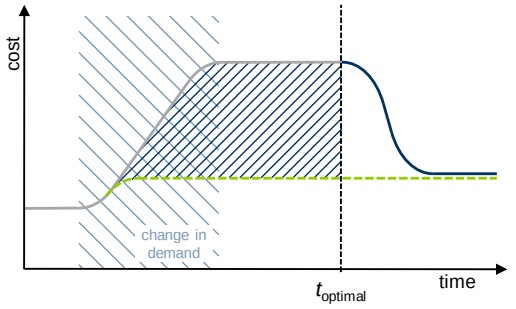
(a) Small service, small change in service demand



(b) Medium service, small change in service demand



(c) Medium service, large change in service demand



(d) Large service, large change in service demand

Figure 5.6: Adaptation timing decision for different service sizes

In both cases, the service adaptation takes place as soon as the expected payoff exceeds the expected adaptation cost. This is reflected in the fact that the highlighted area between current and optimal service cost is of the same size for the two scenarios. However, as the difference in provisioning cost is smaller in Figure 5.5a, it takes a longer time until the adaptation is deemed worth while. In contrast, the large expected savings highlighted in Figure 5.5b warrant a quick adaptation. This illustrates that the triggering mechanism for service adaptations reacts faster when much can be gained by adapting the service configuration, and shows more inertia if the expected savings might not be worth while.

The mechanism works equally well for services of different sizes σ. In Figure 5.6, we illustrate four scenarios: Figure 5.6a depicts a small service faced with a small change in service demand, Figures 5.6b and 5.6c depict a medium sized service faced first with a small and then with a large change in service demand, and Figure 5.6d depicts a large service faced with a large change in service demand. Since the value for σ is different for all three services, it follows that the adaptation cost $a(s, \mathfrak{A}, t)$ differs as well. Assuming that the number of replications and migrations as well as their respective distances between source and destination nodes in $\mathfrak{A}$ is the same for all three services, we can state that if

$\sigma_{\mathsf{small}} < \sigma_{\mathsf{medium}} < \sigma_{\mathsf{large}}$ then $a(\mathsf{small}, \mathfrak{A}, t) < a(\mathsf{medium}, \mathfrak{A}, t) < a(\mathsf{large}, \mathfrak{A}, t)$ for the same value t.

From this follows that, as time passes by, a small service will sooner satisfy the adaptation condition than a larger service. For this reason the configuration of the small service in Figure 5.6a is adapted sooner than that of the medium service in Figure 5.6b, even thought both face the same small difference between current and optimal service provisioning cost. The same is true for the medium and the large service in Figures 5.6c and 5.6d, both facing the same large difference between current and optimal provisioning cost. This illustrates that our triggering mechanism, in addition to considering the potential gain of a service adaptation, is also more conservative when making decisions about the placement about larger services and takes more risks when placing smaller services.

5.7 Example

Now that we have covered both the SPi service placement framework and the placement algorithms, we provide a brief example to illustrate how the overall service placement system operates. To this end, we simulated the GCMI algorithm in a network of 100 nodes placed randomly in a square area that was setup in such a way that the average node degree of the network approximates the same value as in the preliminary evaluations in Sections 4.4.5 and 5.5.4 (cf. Sec. 6.3.1 for an in-depth discussion of simulation parameters).

We ran this setup for a simulated duration of 20 minutes. During the first 5 minutes, half of the nodes were set to begin to issue service requests at a rate of one 1 KB request per second. We then waited for another 5 minutes to allow for the service configuration to stabilize. 10 minutes into the simulation, we then change the service demand pattern, by randomly choosing a new set of nodes to issue service requests. The first and the second set of client nodes were allowed to overlap. After the change in demand, we waited for another 10 minutes to allow for the service configuration to stabilize once again.

The goal of this simulation is to convey a detailed understanding of the workings of the SPi framework when used in combination with the GCMI placement algorithm. We thus begin in Figure 5.7 by looking at the adaptation activity over time. The areas highlighted in light blue are the phases during the simulation in which the service demand changes. The phases result in the current service configuration becoming suboptimal and thus cause GCMI to adapt the number and the location of service instances. The points in time at which GCMI decides to adapt the service configuration are highlighted with a vertical blue line. The same highlights are used throughout Figures 5.8 to 5.11 as well.

As we can see in Figure 5.7, there is a strong correlation between changes in service demand and adaptations of the service configuration. The service configuration is adapted

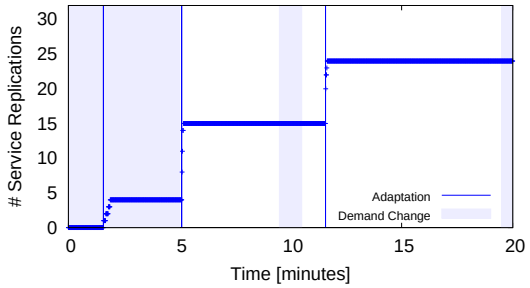


Figure 5.7: Number of service replications vs. time

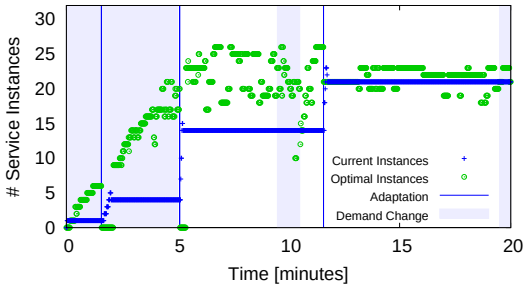


Figure 5.8: Current and optimal number of service instances vs. time

to the changing service demand three times during the simulation, twice during the initial setup phase and once slightly after the reconfiguration phase.

Plotted in blue in Figure 5.8, is the current number of service instances in the network. Plotted in green is the currently optimal number of service instances as calculated by GCMI based on information about regional service demand and network topology that was sent to the coordinator node by all service instances. At the start of the simulation, there is a single service instance, but the number increases every time an adaptation of the service configuration takes place. The optimal number of service instances increases steadily during the setup phase. This reflects the fact that as more client nodes become active, a service configuration with more service instances is better suited to provide the service. In the period between 5 and 10 minutes of the simulation, the optimal number of service instances fluctuates with an average that is greater than the current number of service instances. Yet, the service configuration is not adapted creating more instances. We will explain this behavior in the discussion of Figure 5.9 below. Finally, after the third adaptation, the number of optimal service instances closely matches that of the current service instances. This is an indication that the service configuration is optimal and the data in Figure 5.9 confirms this.

In Figure 5.9, the current service provisioning cost $p_{\mathsf{gcmi}}(s, \bar{H}_s)$, as calculated by the coordinator node based on reports from the service instances, is plotted in blue, and the optimal service provisioning cost $p_{\mathsf{gcmi}}(s, \hat{H}_s)$ is plotted in green. We can observe that current cost increases during phases of changing service demand as the current service configuration grows more and more inadequate to provide the service. The cost of the optimal service configuration, corresponding to the optimal number of service instances in Figure 5.8, remains lower than the current cost at all times. The difference between these two cost functions at any point in time corresponds to the projected savings if the current configuration was changed to the optimal configuration. We further observe that the current cost closely matches the previously calculated optimal cost immediately after

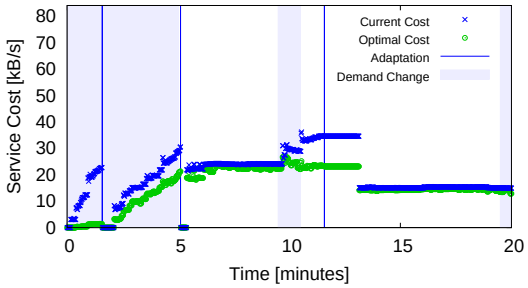


Figure 5.9: Current and optimal service provisioning cost vs. time

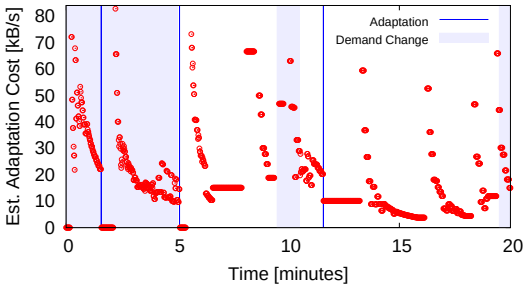


Figure 5.10: Estimated adaptation cost vs. time

the first two adaptations. After the third adaptation, there is a delay before this occurs. This is due to one of the newly created service instances not being able to contact the coordinator node immediately. The coordinator node reacts by suspending the calculation of the cost function during this time. Finally, we can now also explain why the service configuration was not adapted in the time between 5 and 10 minutes, as Figure 5.8 might have lead us to expect given its high value for the optimal number of service instances. As we can see in Figure 5.9, the difference in cost between the current and the optimal configuration, i.e., the benefit of performing an adaptation, is almost non-existent. Hence, even though a better service configuration is known, it does not make sense to adapt the current configuration because the cost of performing the adaptation is greater than the expected savings in service provisioning cost.

This is illustrated in detail in Figure 5.10 which plots the estimated service adaptation cost $a(s, \mathfrak{A}, t)$. Just as in the definition of the adaptation cost in Section 5.6.2, the parameter t corresponds to the difference between the current time t_{current} and the point in time t_{superior} since which a superior configuration has been known. t_{superior} is clearly visible in Figure 5.9 as current and optimal costs begin to diverge under changing service demand. As time progresses, the value $t = t_{\mathsf{current}} - t_{\mathsf{superior}}$ increases, and the value for $a(s, \mathfrak{A}, t)$ decreases as a result. This steady decrease in the estimated adaptation cost is clearly visible several times in the plot of Figure 5.10. When considering both Figures 5.9 and 5.10 together, it becomes also apparent that all three service adaptations are triggered exactly when the adaptation cost $a(s, \mathfrak{A}, t)$ grows smaller than the difference between current and optimal provisioning costs $p_{\mathsf{gcmi}}(s, \bar{H}_s) - p_{\mathsf{gcmi}}(s, \hat{H}_s)$. If a superior configuration is known, but the adaptation condition is not met, the current service configuration remains unchanged.

The overall result of employing GCMI as a service placement algorithm is shown in Figure 5.11, which plots the total layer 3 network load over time. There are several interesting things to note about this plot: First, the network load always raises during periods of changing service demand, once again, as a result of the respective current service

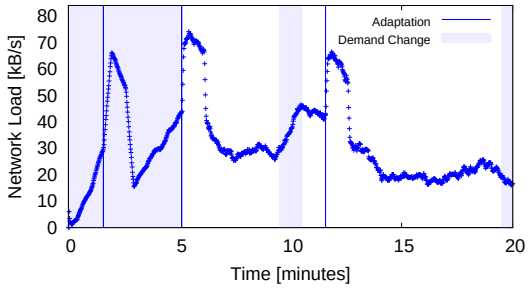


Figure 5.11: Network load vs. time

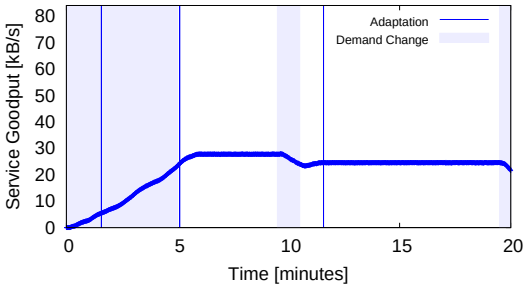


Figure 5.12: Service goodput vs. time

configuration becoming more and more inadequate to handle the demand. Second, the adaptations of the service configuration are clearly visible as temporary increases in network load that follow immediately after an adaptation is initiated. These values (after subtracting the load due to service traffic) correspond to the cost of the adaptation. Thirdly, it can be seen that after each adaptation is completed, the network load drops to a level below the one it had before the adaptation took place. And forth, when comparing Figures 5.11 and 5.9, it becomes apparent that the current service provisioning cost in Figure 5.9 closely tracks the network load in Figure 5.11. From this we can conclude that the model representation used by the SP*i* framework and the GCMI algorithm to calculate the service provisioning cost is indeed a good approximation of the real network.

Finally, Figure 5.12 shows the service goodput, i.e., the application-level throughput between clients and service instances, over time. Apart from the obvious characteristics such as increasing goodput during the setup phase and a slight drop in goodput while the service demand changes, there is one interesting thing to note about this plot when compared to Figure 5.11: After an adaptation has been completed and the network has stabilized, the goodput is equal to or actually lower than the network load. This shows that a service is being provided to a set of clients and the data rate as perceived by the clients is actually higher than the data rate that needs to be transmitted over the network.

5.8 Summary

In this chapter, we have introduced the service provisioning and service adaptation cost, thereby formalizing some of the concepts presented in Chapter 4. We then continued with a detailed description of the Graph Cost / Single Instance (GCSI) and Graph Cost / Multiple Instances (GCMI) service placement algorithms that build upon these concepts. Afterwards, we discussed at which point in time a service configuration should be adapted and how this decision can be integrated with the placement algorithms. Finally, we have given an example to illustrate how these concepts and algorithms interact.

The main contributions of this chapter are the service placement algorithms, in particular the GCMI algorithm for distributed services with multiple instances, as well as the triggering mechanism for initiating service adaptations. When discussing the overall design of the SP*i* service placement system in Section 4.2, we concluded that a centralized placement system has distinctive advantages over other approaches. The GCMI algorithm built on top of the SP*i* framework is such a centralized placement system. To the best of our knowledge, we are the first to propose and implement this kind of a system.

Our preliminary evaluation in the preceding Section 5.7 has already yielded some noteworthy results: By the means of employing a service placement system, in this case SP*i* with the GCMI placement algorithm, the ad hoc network is able to satisfy the demand for a service at an overall bandwidth utilization that is lower than the combined service demand of all clients. Therefore, one can not only expect a service placement system to reduce network load in unloaded ad hoc networks, but also to increase service availability and reliability in networks under high load. In the next Chapter 6, we will quantitatively investigate these and other network properties that result from employing service placement systems such as SP*i* / GCMI.

Chapter 6

Evaluation

In this chapter, we present a quantitative evaluation of the SP*i* service placement framework [111] running the Graph Cost / Single Instance and the Graph Cost / Multiple Instances service placement algorithms. We compare the performance of our approach with both a traditional client/server architecture without active service placement as well as several service placement algorithms proposed in the literature. In this evaluation, we employ the network simulator `ns-2` for scenarios requiring large numbers of nodes or prolonged evaluation time. For the evaluation with regard to link reliability, we make use of a wired IEEE 802.3 network. Finally, we evaluate the performance of the framework and its algorithms under real-world conditions using a wireless IEEE 802.11 testbed.

This chapter is structured as follows: After a brief overview in Section 6.1, we begin with discussions of both the metrics used in the evaluation in Section 6.2 and the three evaluation setups for simulation, emulation, and real-world testbed in Section 6.3. We then continue with an evaluation regarding the scalability of service placement approaches with regard to network size and load, first for centralized services in Section 6.4 and then for distributed services in Section 6.5. In Section 6.6, we proceed to evaluate the performance of the placement algorithms in scenarios with varying regional distribution and volatility of the service demand. In Section 6.7, we examine the behavior of the network for services with varying synchronization requirements. In Section 6.8, we evaluate the impact of variations in the quality of the communication links, and in Section 6.9, we present the results of an evaluation under real-world conditions. Finally, we summarize our findings in Section 6.10.

Some evaluation results are omitted at this point for brevity and presented in the appendices. In Appendix A, we evaluate the performance of our implementation of the DYMO routing protocol and verify its correct operation. Appendix B contains a detailed evaluation of the performance differences of several placement algorithms from the literature that we implemented within the SP*i* framework. Finally, Appendix C investigates the differences in the results obtained with the three evaluation tools, simulation, emulation, and the real-world testbed.

6.1 Overview

The evaluation that we present in this chapter discusses the performance of the SP*i* framework in conjunction with the the Graph Cost / Single Instance (GCSI, cf. Sec. 5.4) and the Graph Cost / Multiple Instances (GCMI, cf. Sec. 5.5) service placement algorithms in a wide variety of scenarios. The goal of this evaluation is to gain a general understanding of the effects of and tradeoffs related to employing service placement systems in ad hoc network. Of course, we are particularly interested in the performance of the GCSI and GCMI algorithms when compared to the current state of the art. Our evaluation therefore covers scenarios with varying network sizes and loads, varying service demand and synchronization requirements, and varying link reliability on the Data Link Layer.

In our evaluation, we place the focus on distributed services rather than centralized services, since we consider distributed services to be the far more promising approach to general service provisioning in ad hoc networks. In fact, one can think of a centralized service as merely a special case of a distributed service with the synchronization ratio τ set to a value that makes it a waste of resources to create more than one service instance. Furthermore, we expect, and in fact we will find this confirmed by our evaluation, that employing service placement in conjunction with a distributed service has a much larger positive impact on the overall performance of the ad hoc network. Hence, this approach is architecturally superior and thus warrants a more detailed evaluation.

The evaluations for each of the scenarios covered in the chapter follows the same basic pattern: We always begin with a look into the inner workings of the placement algorithms in the given circumstances, and explain why a specific placement algorithm is behaving in a certain way in each scenario. Afterwards, we shift our focus on the effects that each approach has on the overall performance of the network and the quality of the service as perceived by the clients. To this end, we employ a set of metrics that we will present in detail in the following section.

6.2 Metrics

A comprehensive list of all the metrics used in this evaluation is shown in Table 6.1. The metrics are subdivided into two groups with the first group of four metrics describing the internal operation of a placement algorithm. The second group of four metrics deals with the impact that the service placement has on the overall operation of the network, in particular on the quality of the service as perceived by the clients.

Looking at the first group, the *number of service requests* is used in conjunction with the hop count of each service request in order to gain an insight into the distribution of distances

Table 6.1: Metrics used in the evaluation

Name	Description	Unit
# Service Requests	Total number of service requests received by all services instances during an experiment	n/a
# Service Instances	Average (mean) number of service instance that were available in the ad hoc network during an experiment	n/a
Request Distance	Average (mean) distance that service requests had to travel before reaching a service instance	hops
# Replications	Total number of replications of service instances during an experiment (successful or failed, and including migrations)	n/a
Service Recall	Average (mean) ratio of successful service request over all service requests during an experiment	n/a
Service Goodput	Average (mean) application-layer data rate of service traffic over all clients during an experiment	kB/s
Round-trip Time	Average (mean) time between a client transmitting a service request and receiving a reply from a service instance (only counting successful service requests)	s
Network Load	Average (mean) network-layer data rate of traffic caused by the communication between clients and service instances	kB/s

between clients and service instances. The *number of service instances* is interesting for two reasons: First, it is worth examining in which situations a placement algorithm chooses different numbers of service instances. And second, when considered in conjunction with the *request distance*, it is interesting whether a placement algorithm manages to place additional service instances in such a way that the distance between clients and service instances is reduced. The combination of these two metrics can indeed be used as a way to estimate the overall quality of the placement decisions of an algorithm. Finally, the *number of replications* serves as an indication on the activity of a placement algorithm. Given the cost of service replications, a placement algorithms should aim to achieve a high-quality placement of service instances with as few replications as possible. Furthermore, this metric is also an indication of whether a placement algorithm manages to find a stable service configuration or whether the configuration is being adapted continuously throughout the run of the experiment.

When measuring the effects of service placement on the overall operation of an ad hoc network, one has to note that there is no single metric that can be used to comprehensively evaluate a service placement system or a placement algorithm. Service placement can be expected to improve the quality of the service as perceived by the clients and, at the same time, reduce the network traffic that is generated by the communication between the two parties. This expected behavior is especially relevant in scenarios with very low or very high network load. In scenarios with very low network load, we expect only minor differences in the metrics that measure quality and instead need to focus on the network load alone. In scenarios with very high network load, i.e., in situations in which the communication channel is saturated, there will obviously be little difference in the network load and we thus need to focus on the service quality.

Therefore, we use four metrics to measure the impact of a service placement algorithm: The *service recall* captures how successful the clients are at communicating with the service instances. This ratio is sensitive to situations of high network load and lossy links, and we can expect a good service placement algorithm to improve this metric in these situations. Similarly, we also expect an improved behavior with regard to *service goodput*, and a reduction in *round-trip time*. The latter one will in fact prove to be very sensitive to the quality of a service configuration with measurements varying across several orders of magnitude. Finally, the *network load* corresponds to the overall cost of providing a service in the network. A good placement algorithm can be expected to reduce this load in all scenarios. This is especially noteworthy since it implies that some approaches may still operate normally while others have already reached the point of saturation of the communication link. These situations are particularly interesting because the measurements for all metrics concerning the service quality can be expected to diverge strongly depending on the placement algorithm.

In order to present our results in a statistically meaningful way, we plot each data point as a combination of median, mean, first and third interquartile, and minimum and maximum value. For easy comparison of several approaches in the same diagram, we connect the medians of related data points with dashed lines. The number of measurements used to generate such a data point depends on the method that was used for measuring them. In scenarios that were evaluated using simulations, we used 30 measurements per data point. Due to excessively long run times of the experiments, we reduced this number to 15 and 10 measurements respectively when using the emulation setup and the wireless testbed.

6.3 Evaluation Setup

We make use of three different tools for the evaluation of the service placement algorithms implemented on top of the SP*i* framework: the `ns-2` network simulator, a wired IEEE 802.3 network, and a wireless IEEE 802.11 network. In this section, we present the setup of each of these evaluation setups.

6.3.1 Simulation Setup

There are several reasons for a simulation-based evaluation of protocols and systems for ad hoc networks. Among them is the comparative low overhead required for setting up experiments and the reproducibility of complex interactions between system components. We made extensive use of simulations during the prototyping and implementation phases of this project, and were in fact able to port the SP*i* framework from the simulation platform to real-world system with relative ease. The method we used to achieve this is described in detail in Chapter 3. The most important reason for using simulations is however the ability to scale the evaluation up to numerous scenarios and large numbers of individual experiments. We estimate that conducting the same number of experiments in real-time on real-world systems correspond to a pure run time of approximately one year.

One of the drawbacks of conducting an evaluation based on simulations is the lack of credibility of the results as opposed to those obtained using more realistic deployments. We mitigate this risk by basing our conclusions not only on the results from simulations, but also on emulations and real-world deployments. In Appendix C, we discuss in how far the results differ depending on the method that was used for measuring them.

For our simulations, we used version 2.33 of the `ns-2` network simulator [31, 84], released on March 31, 2008. `ns-2` is the most widely used tool for the simulation-based evaluation of protocols and systems for ad hoc networks [61]. The source code of `ns-2` is publicly available under a variety of GNU GPL-compatible licenses. `ns-2` is implemented in the C++ programming language with support for the scripting of complex simulation setups in MIT Object Tcl (OTcl), an extension to Tcl/Tk for object-oriented programming. It runs natively on POSIX-compliant system, in particular on Linux.

We simulated networks of different sizes (in terms of number of nodes) and under different load scenarios. Unless otherwise indicated, the nodes were randomly placed in an area whose size was changed depending on the number of nodes. The median node degree thus corresponds to the node degree of the regular grid layout that we used in our preliminary evaluations (cf. Secs. 4.4.5, 5.5.4 and 5.7). We also ensured that all network topologies are connected, i.e., at least once during each experiment a path exists between each pair of nodes. We discarded the results and repeated the experiment if this was not the case.

Table 6.2: Network stack used in ns-2 simulations

Channel type	`Channel/WirelessChannel`
Radio-propagation model	`Propagation/TwoRayGround`
Antenna model	`Antenna/OmniAntenna`
Network interface	`Phy/WirelessPhy`
MAC sublayer	`Mac/802_11`
LLC sublayer	`LL`
Interface queue	`Queue/DropTail/PriQueue`
Routing agent	`DumbAgent`

Table 6.3: Parameters used in ns-2 simulations

Parameter	Value	Comment
`Phy/WirelessPhy CSThresh_`	6.30957e-12	Carrier sense threshold according to [24] (6.30957e-12 W = -82 dBm)
`Phy/WirelessPhy RXThresh_`	1.17974e-09	Receive power threshold based on radio range for semi-open scenarios [24] (1.17974e-09 W corresponds to 50 m)
`Phy/WirelessPhy bandwidth_`	11Mb	Bandwidth corresponding to maximal data rate (11 Mbit/s)
`Phy/WirelessPhy Pt_`	0.031622777	Transmit power according to [24] (0.031622777 W = 15 dBm)
`Phy/WirelessPhy freq_`	2.472e9	IEEE 802.11b using channel 13 (2.472 GHz)
`Mac/802_11 basicRate_`	1Mb	Data rate for IEEE 802.11b control frames (1 Mbit/s)
`Mac/802_11 dataRate_`	11Mb	Data rate for IEEE 802.11b data frames (11 Mbit/s)

We configured the network simulator according to the parameters given in Tables 6.2 and 6.3. The components of the network stack as listed in Table 6.2 were selected following the recommendations from [115]. Note that we do not make use of any of the routing agents that are provided as part of `ns-2` but instead employ our own implementation of the Dynamic MANET On-demand (DYMO) [17] routing protocol. This is due to two reasons: First, the SP*i* framework relies on a tight integration between the routing component and the rest of the framework. And second, since we intend to employ the same implementation of the same routing protocol in all of our simulations, emulations and real-world experiments, we require a routing component that is portable across all three evaluation environments. None of the existing routing agents in `ns-2` satisfies these two conditions and thus we had to implement a routing protocol from scratch. In Appendix A, we verify the correct operation of our implementation of DYMO.

The network interface was configured using the data from the datasheet of the ORiNOCO 802.11b Client PC card [24]. The values for each parameter together with a short explanation is given in Table 6.3. These values differ from the default values used in `ns-2`, for which we could not establish the source they are based on. A notable choice is to calculate the power threshold for a successful packet reception based on the radio range for semi-open scenarios. For this scenario, the suggested value for the transmission radius is 50 m.

The amount of data required for replicating a service instance σ was set to 100 kB. A larger value for σ would lead to longer delays before an adaptation occurs (cf. Sec. 5.6.2), which results in a longer time before the service configuration stabilizes and thus would necessitate longer simulation runs. However, as far as our placement algorithms are concerned, the amount of data required for replications σ is merely significant for the decision at which point an adaptation of the service configuration should take place. More importantly, the optimal service configuration does not depend on this value. For this reason, we argue that evaluating our system with a fixed value of $\sigma = 100$ kB is acceptable. Another reason for not using a larger value is that it would increase the dependency of our evaluation on the quality of the implementation of the transport layer protocol. Implementing, verifying and tuning a transport protocol for wireless ad hoc networks is, however, beyond the scope of this work and thus we avoid excessively large data transfers.

For most scenarios, the fraction of synchronization traffic τ was set to 1%. In light of the discussion in Section 5.2.2, we argue that this is a sensible value. However, we are also interested in the capability of a service placement system to deal with different synchronization requirements and hence evaluate our algorithms across a range of value for τ in Section 6.7.

Each evaluation run lasted for simulated 20 minutes, the initial 5 minutes of which were used for setup and initialization. The simulated nodes became active at uniformly

distributed points in time during the initialization phase and started issuing service requests with a payload size of 1024 bytes. By starting the simulation in this manner, we avoided network congestion during initial route setup and service discovery. The initial host of the single service instance was assigned randomly at the beginning of the simulation. Unless otherwise indicated, the role of client nodes, i.e., those nodes that would issue service requests, was assigned to a randomly chosen subset of the total node population comprising 50% of the total number of nodes. This was done to appropriately reflect the fact that not all nodes in a network are necessarily interested in making use of a service. Hence, the scenarios also include a reasonably large population of passive nodes that contributed to the complexity of the network topology but not to the service demand.

For each of the data points plotted in the following diagrams, we conducted 30 runs of the simulation. In each of these runs, we parameterized the internal random number generators of `ns-2` with different hashed values of the current system time at which the simulation run was started. Depending on the complexity of the simulated scenario, a single simulation run would need between 15 minutes and 4 hours to complete on the hardware at our disposal. Assuming a value of 2 hours for the average simulation time, this leaves us with one data point calculated every 2.5 days. Most of our diagrams consist of 15 of these data points. In order to achieve statistically significant results in acceptable time, we ran up to 45 simulations in parallel on the publicly available computers at the Institute of Computer Science, Freie Universität Berlin.

6.3.2 Emulation Setup

A discrete event simulator such as `ns-2` does not properly capture all properties of a wireless ad hoc network. In fact, there is a fundamental tradeoff between the complexity of the models used in the simulations and their implementation and run time overheads. One major point of criticism on network simulators is the model for the wireless communication channel [83], and we address this in Section 6.3.3. However, before evaluating the SP*i* framework and its algorithms on a real-world *wireless* testbed, we take an intermediate step and evaluate SP*i* on a real-world *wired* testbed.

The goal of this part of the evaluation is to show that our algorithms perform as expected when executed on real-world hardware instead of within a simulator. In fact, we discovered and fixed several bugs in the implementation that only become visible when our framework was run on the wired testbed. Most of these bugs were due to the implicit assumption that no time passes while an event is being processed. This is true in discrete event simulations, but not in reality. We note that uncovering and isolating these bugs would have been far more difficult if we had skipped this intermediate step.

Furthermore, we are also interested in the effect of service placement on the performance of the network under controlled variations of the quality of layer 2 links. Our hypothesis is that since service placement reduces the distance between clients and the instances of a distributed service, a lower link quality will have less of a negative impact.

We emulated the ad hoc network using 36 desktop computers running Linux 2.6.30 that were interconnected with an IEEE 802.3ab (Gigabit Ethernet) network. The computers were subject to real-world load both in terms of CPU and network utilization during the experiments. We ignore these factors in our evaluation since the requirements of the SP*i* framework on these resources are very small in comparison to the available resources. All computers were located in the same broadcast domain at the network layer. On top of this physical topology, we created configurable artificial topology by selectively ignoring packets from certain hosts. We generated these topologies in such a way that their approximate node degree matches that of the evaluations using the `ns-2` network simulator. In order to add controllable packet loss to this setup, we generate a random number from an uniform distribution in the interval $[0, 1]$ for each packet and discard the packet if the value of this number is below a configurable threshold. We leave an evaluation with more sophisticated models for packet loss to future work.

Where applicable, the remaining parameters of the system were set to match those of the setup described in previous section. For each of the data points that we generated using this setup, we conducted 15 runs of the emulated network. This reduction in the number of samples in comparison to the simulations is due to the fact that only one run of the emulation can be conducted concurrently. Therefore, it proves to be more time consuming to scale to a large number of measurements.

6.3.3 Testbed Setup

As the third method for evaluating the SP*i* service placement framework and its algorithms, we employed the wireless IEEE 802.11 Distributed Embedded Systems Testbed (DES-Testbed) [13,41,42]. The goal of this setup is to provide the means for evaluating SP*i* under real-world conditions and for verifying the findings of the other parts of this evaluation.

The DES-Testbed currently consists of 90 nodes, 43 of which are located in the offices across all three floors of the main building of the Institute for Computer Science, Freie Universität Berlin, and another 47 nodes in adjacent buildings. Each node is built upon a PC Engines ALIX.2c2/2d2/3d2 embedded PC board [40] and runs a customized version of the Debian GNU/Linux operation system with version 2.6.34 of the Linux kernel. A typical node is equipped with three wireless network interface cards (NICs), one IEEE 802.11b/g-compliant LogiLink WL0025 USB NIC [71] and two IEEE 802.11a/b/g-compliant Compex

WLM54SAG Mini PCI NICs [72]. Since our work does not cover multi-antenna systems, we merely use the LogiLink WL0025 USB NIC set to operate in ad hoc mode at a frequency of 2.472 GHz (IEEE 802.11b, channel 13).

In our experiments, we used 36 out of the 43 nodes located in the main building of the Institute for Computer Science. This conservative choice in the number of nodes is due to several reasons: Most importantly, since our measurements require several weeks to complete, we wanted to avoid using radio links whose quality is strongly dependent on weather conditions. Furthermore, since we observed individual nodes to have intermittent failures during certain time slots, we wanted to avoid depending on the complete set of nodes being operational for each run. In fact, the results that we report in this section are based on different subsets of the set of 43 nodes. We also chose the number of 36 nodes in order to make the results from the testbed comparable with the results from simulation and emulation, both of which were run (in some scenarios) with this exact number of nodes.

In order to minimize the impact of people moving in the building on the quality of the radio links, we ran all experiments across multiple 12 hour time slots between 19:00 o'clock and 7:00 o'clock. The link layer RTS/CTS mechanism of IEEE 802.11 was enabled for unicast packets independently of their size. Since the ETX link metric used in our implementation of the DYMO routing protocol employs broadcast packets that are sent at a fixed data rate of 1 Mbit/s to estimate the link quality between nodes, we reduced the data rate of unicast packets from a variable data rate of up to 11 Mbit/s to the same fixed data rate of 1 Mbit/s. This is necessary because the link quality is highly sensitive to the various modulation schemes used at the different data rates. Implementing a mechanism for estimating link quality that takes the impact of changing data rates during a transmission into account is beyond the scope of our work.

In order to compensate for the reduced maximal data rate, we reduced the service size σ by a similar factor from 100 kB to 10 kB. Furthermore, we observed in preliminary experiments that the success rate of single-hop packet transmissions is highly dependent on the size of the packet and drops sharply as the payload size of packet approaches 1 KB. In order to enhance the reliability of single-hop transmission, we reduced the size of the Maximum Transmission Unit from 1500 B to 512 B, and the payload size of service requests from 1024 B to 256 B.

We also noticed that the performance of our implementation of the DYMO routing protocol decreases under very light network load. We attribute this to the fact that DYMO automatically purges supposedly unused routes from its routing table if no packet has been forwarded along a route for a certain amount of time. This design choice causes routes to be wrongly discarded in situations of light traffic combined with bursty packet loss. As a result, the service recall for scenarios with light service demand suffered. Scenarios with a

distributed service are more susceptible to this than those with a centralized service, since the light traffic is spread out more equally across the network and hence individual links are even less utilized. In order to solve this problem, we disabled the mechanism for purging unused routes from the DYMO routing table.

Where applicable, the remaining parameters of the system were set to match those of the two setups described in previous sections. We collected ten measurements for each of the data points that we generated using the testbed. This additional reduction in the number of samples is due to the fact that the DES-Testbed is shared between multiple experiments, each of which requires exclusive access. Hence, time for experimentation is very limited.

6.4 Placement of Centralized Services

We begin our evaluation with a look at different placement algorithms for centralized services, i.e., services for which the number of service instances is fixed to exactly one. Fixing the number of service instances greatly simplifies the service placement problem and hence the algorithm tend to be rather simple.

The five algorithms that we evaluate for placing a centralized service are:

LinkPull Migration – A placement algorithm proposed in [68] that migrates the service instance to the neighboring node over which most service traffic was forwarded at the end of each epoch.

PeerPull Migration – A placement algorithm proposed in [68] that migrates the service instance to the client node from which most service traffic originated at the end of each epoch.

TopoCenter(1) Migration – A placement algorithm proposed in [68] that migrates the service instance, at the end of each epoch, to the node that minimizes the sum of the migration costs (proportional to the length of the shortest path from the current host to the target node) and the estimated future communication cost (based on the usage statistics of the last epoch and a partial network map).

Tree-topology Migration – A placement algorithm proposed in [85] that migrates the service instance to the neighboring node over which more than half of the service requests are being forwarded, if any.

SP*i* / GCSI – A placement algorithm proposed in Section 5.4 of this work that migrates the service instance to the node that minimizes the service provisioning cost (based on usage statistics and a network map) as soon as the expected payoff exceeds the migration cost.

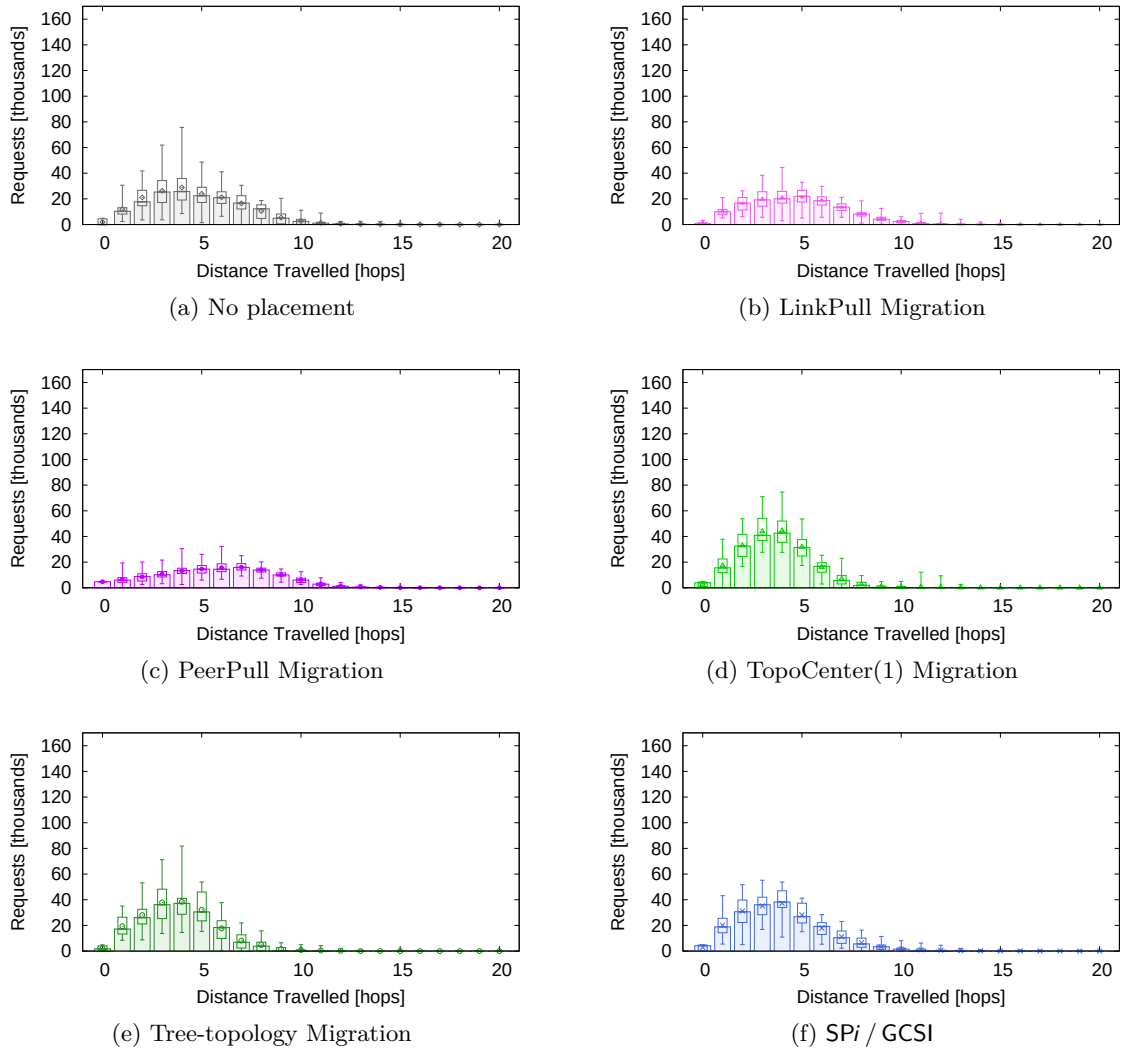
All of these placement algorithms (except SPi / GCSI) are described in greater detail in Section 2.4. For reference, we also include the results of the same setup without any active service placement taking place.

In this part of the evaluation, we focus on the scalability of the ad hoc network when employing the five placement algorithms. We thus vary the size of the network in terms of participating number of nodes, in the range between 16 and 100 nodes, and the rate at which service requests are issued, in the range between 0.1 and 5 requests per second for each client. These ranges were selected in order to cover scenarios in which the network shows significant qualitative changes in its behavior for each of the placement algorithms. When varying the size of the network, the request frequency was set to 5 requests per second per client. When varying the request frequency, the network size was fixed to 100 nodes. All other parameters correspond to those of the general setup for simulations as presented in Section 6.3.1.

In Figure 6.1, we plot histograms of the distances traveled by the service requests before reaching the service instance for all five placement algorithms and for the same network without service placement. The simulated network consists of 100 nodes and clients issue service requests at a rate of 2 requests per second. These parameters were chosen in order to avoid skewed results due to increasing package loss under heavy load that affects some placement algorithms (cf. Fig. 6.4). We can observe that the maximal value in the histogram as well as the spread of values is more or less pronounced depending on the placement algorithm. In particular, a more or less pronounced peak is visible for smaller distances in Figures 6.1d, 6.1e, and 6.1f for networks employing the TopoCenter(1) Migration, Tree-topology Migration, and SPi / GCSI placement algorithms respectively. Figures 6.1a, 6.1b, and 6.1c showing the situation in networks with no placement, LinkPull Migration, and PeerPull Migration do not exhibit this characteristic. Especially for PeerPull Migration in Figure 6.1c, the values are distributed quite evenly.

We interpret this as a first indication of the quality of the placement of the service instance achieved by the algorithm. TopoCenter(1) Migration, Tree-topology Migration, and SPi / GCSI are clearly capable of reducing the average distance between clients and the service instance when compared to the setup without service placement. The same cannot be said about LinkPull and PeerPull Migration. PeerPull Migration seems to perform particularly bad. This can be easily explained by the migration rule of this algorithm: If the service instance is always migrated to the node with the highest service demand in the past epoch, this results in largely unstable behavior in scenarios with identical service demand on several client nodes.

In Appendix B, we evaluate all placement algorithms proposed in the literature in more detail. The result of this evaluation is that the Tree-topology Migration algorithm achieves



(a) No placement

(b) LinkPull Migration

(c) PeerPull Migration

(d) TopoCenter(1) Migration

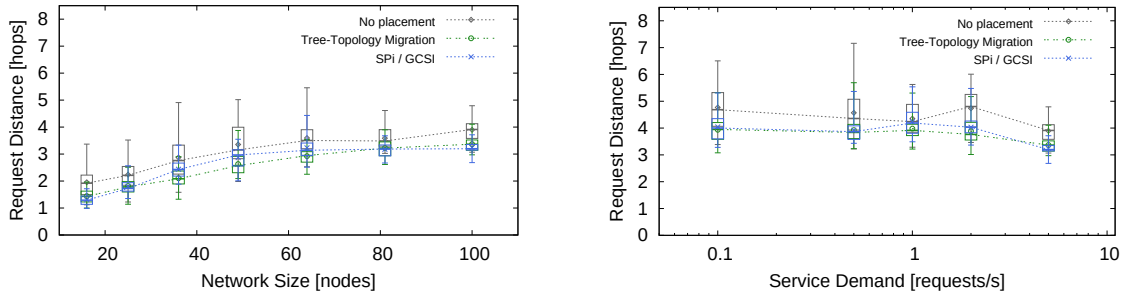
(e) Tree-topology Migration

(f) SP*i* / GCSI

Figure 6.1: Service requests vs. distance (100 nodes, 2 requests/s)

the best performance in the scenarios that we examine. For this reason and in order to improve the readability of subsequent figures, we focus on the Tree-topology Migration algorithm and SP*i* / GCSI in the remainder of this evaluation. For reference, we continue to include the results achieved by the setup without service placement.

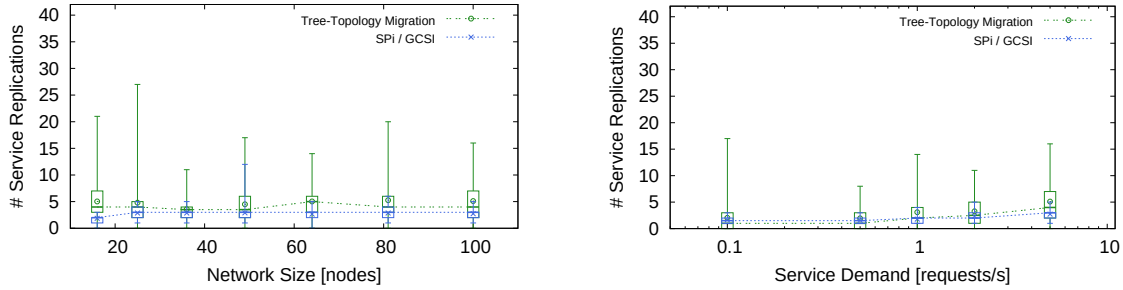
In Figure 6.2, we plot the distances that service requests have to travel between client node and service instance against varying network sizes and service demands. As one would expect, the distance increases slightly for all approaches as the network size grows. For varying service demands, the distances remain nearly constant. A noteworthy observation about these plots, however, is that both placement algorithms, Tree-topology Migration and SP*i* / GCSI, improve the predictability of the system, i.e., they reduce the variance in



(a) Request distance vs. network size

(b) Request distance vs. service demand

Figure 6.2: Placement characteristics for centralized services



(a) Number of replications vs. network size

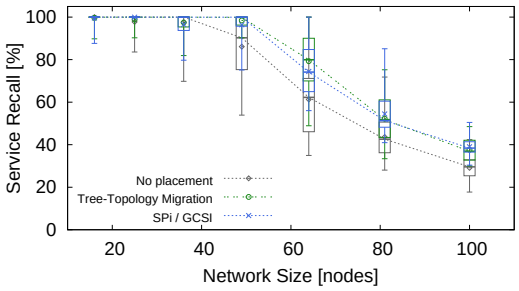
(b) Number of replications vs. service demand

Figure 6.3: Placement activity for centralized services

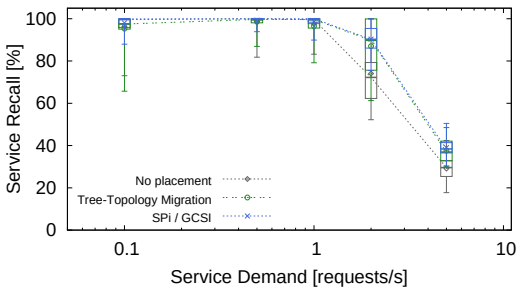
distance between clients and service instance when compared to the setup without service placement. This observation reflects the fact that, in a setup without service placement, the distances between clients and service instance depend only on the randomly chosen location of the service instance. In both of the other two cases, a bad initial placement can be corrected by the service placement algorithm.

In Figure 6.3, we plot the number of replications undertaken by each of the two placement algorithms in the same scenarios as above. The average results of the two algorithms are very similar. However, we note that the Tree-topology migration algorithm seems to have problems converging at the optimal solution in some scenarios. SP*i* / GCSI in contrast shows fewer outliers with large numbers of used replications. This reflects the fact that SP*i* / GCSI explicitly considers the benefit of migrating the service instance, while the Tree-topology migration algorithm does not. However, one should think of this misbehavior of the Tree-topology migration algorithm as merely a minor problem which should be easily fixable by adding a hysteresis to the migration condition.

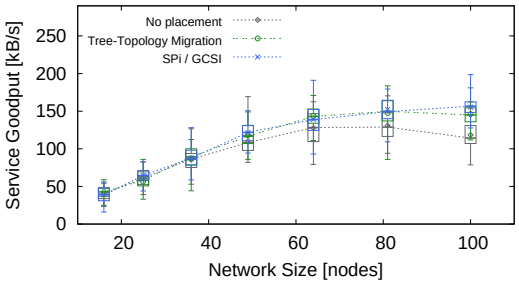
Finally, we move on to evaluate the effect that these two service placement algorithms have on the performance of the ad hoc network. In Figure 6.4, we plot service recall, service



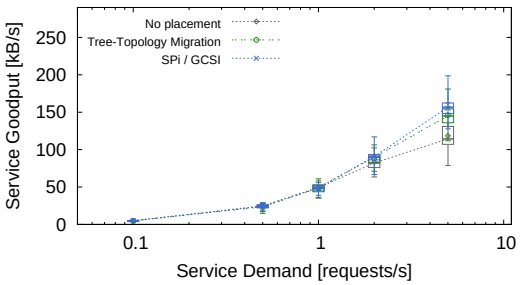
(a) Service recall vs. network size



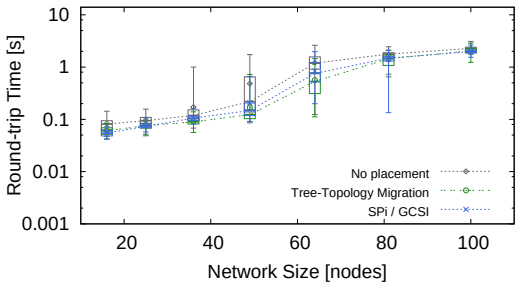
(b) Service recall vs. service demand



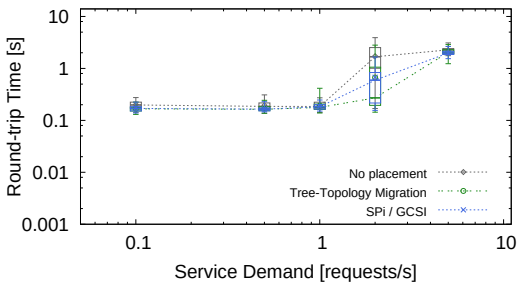
(c) Service goodput vs. network size



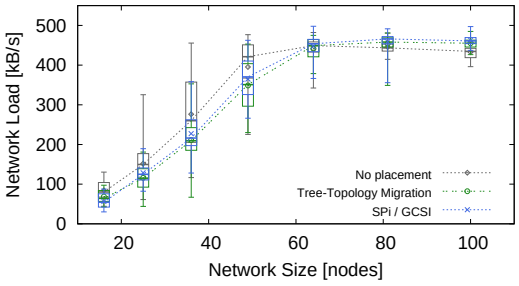
(d) Service goodput vs. service demand



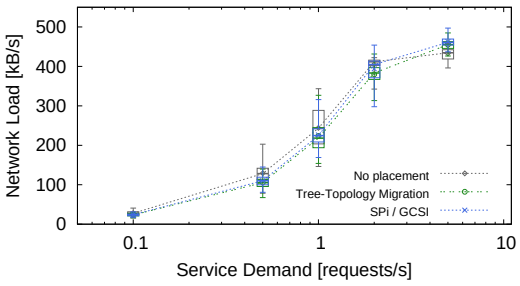
(e) Round-trip time vs. network size



(f) Round-trip time vs. service demand



(g) Network load vs. network size



(h) Network load vs. service demand

Figure 6.4: Placement effect for centralized services

goodput, round-trip time of service requests, and network load for the same scenarios as above. The first thing to note is that the service recall drops with both increasing network sizes and service demand. This is due to the traffic on the wireless channel reaching the maximal capacity of the channel, especially in the proximity of the service instance. This increases the probability of packet collisions and thus reduces the service recall. It should be noted, however, that both placement algorithms achieve slightly better results than a network without service placement. The same analysis is equally true for the service goodput. For the round-trip time, the differences between service placement and no service placement are less visible. Instead, it is interesting to note that there is a sudden increase by one order of magnitude in the round-trip time as the the wireless channel becomes saturated. Finally, when looking at the network load, we can see a steady increase for larger network sizes and service demands. This increase eventually levels off as the network reaches its maximal capacity and any additional packets are lost due to collisions. For the scenario with varying network sizes, it can be noted that the increase in load is slightly, but not significantly, lower if service placement is employed.

There are three facts that can be learned from this evaluation of service placement for a centralized service: First, if the number of service instances is fixed to one, then there are only minor advantages of employing service placement as opposed to a traditional client/server architecture. Second, the only qualitative change in the behavior of the network is the increased predictability of the performance as a bad initial placement of the service instance can be corrected. And third, we observe that there is little difference in the performance between the Tree-topology Migration algorithm and SP*i* / GCSI. Given the evaluation of the other placement algorithms for centralized services in Appendix B, in which the Tree-topology Migration algorithm proved to be superior over all other algorithms proposed in the literature, we can conclude that both the Tree-topology Migration algorithm and SP*i* / GCSI equally represent the state of the art for placing a centralized service.

6.5 Placement of Distributed Services

In this section, we repeat a similar evaluation as in the previous section, but shift our focus from centralized to distributed services. Algorithms that solve the service placement problem for distributed services tend to be more complex than their counterparts for centralized services. In some cases, they are built upon other mechanisms, e.g., the algorithm proposed in [58] employs a separate mechanism for distributed majority voting.

As the field of research into service placement in ad hoc networks is relatively young, there are, as of this writing, no publicly available implementations of distributed placement algorithms at our disposal. We do not have the resources required for implementing multiple

of the proposed algorithms, and hence selected the two algorithms of the Ad hoc Service Grid (ASG) described in [49] to be used for comparison in this evaluation. These two algorithms, ASG / simple and ASG / Event Flow Tree (EFT), implement a distributed, rule-based heuristic for placing a distributed service. These algorithms are good candidates for our comparisons since their assumptions about their target platform and service model closely resemble our own. Further, they follow an architecturally different approach than we do with SP*i* / GCMI, i.e., the ASG algorithms are fully distributed, while SP*i* / GCMI is centralized. This is particularly interesting, since this way our evaluation can shed light onto the question whether a centralized or a distributed approach is architecturally superior when implementing service placement in ad hoc networks.

The three algorithms that we evaluate for placing a distributed service are:

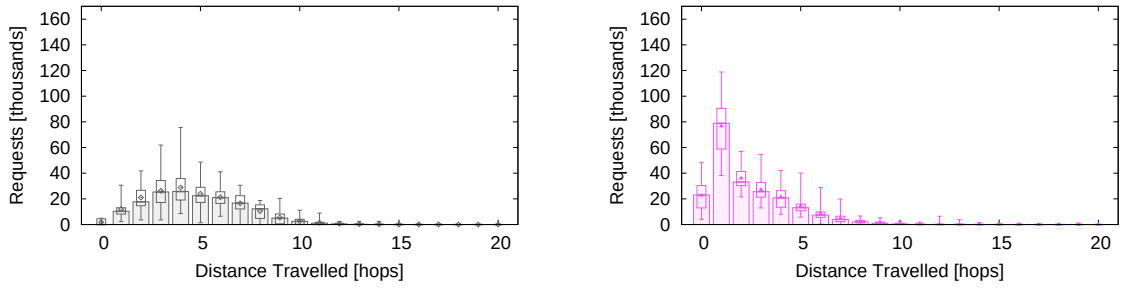
ASG / simple – A distributed placement algorithm proposed in [49] that migrates and replicates service instances to neighboring nodes according to a fixed set of rules. Migrations are triggered if more service requests are received from one neighbor than from all other neighbors and the current service host together. Replications are triggered if the service requests that have been forwarded by a migration target have traveled more than a preconfigured number of hops. Service instances are shut down if the service demand they serve falls below a threshold.

ASG / EFT – A distributed placement algorithm proposed in [49] that migrates and replicates service instances to optimal nodes in a tree-like structure with the current service host as root (i.e., based on a partial network map) following similar rules as those implemented in ASG / simple. The main difference when compared to ASG / simple is that ASG / EFT explicitly considers distant nodes as potential targets for replications and migrations while ASG / simple only considers direct neighbors.

SP*i* / GCMI – A centralized placement algorithm proposed in Section 5.5 of this work that adapts the service configuration in order to minimize the service provisioning cost (based on usage statistics and a network map) as soon as the expected payoff exceeds the adaptation cost.

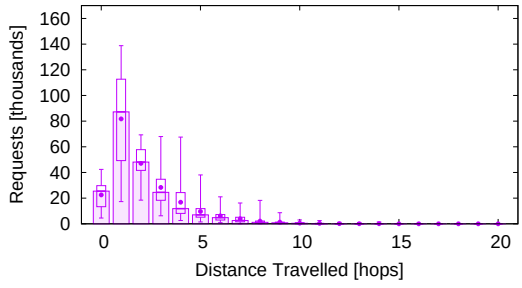
Both ASG algorithms are described in greater detail in Section 2.4. For reference, we once again include the results of the same setup without any service placement taking place.

In this part of the evaluation, we repeat simulations for the same scenarios as already described in Section 6.4. We vary the size of the number of nodes in the range between 16 and 100 with a fixed request frequency of 5 requests per second, and the rate of issuing service requests in the range between 0.1 and 5 requests per second with a fixed network

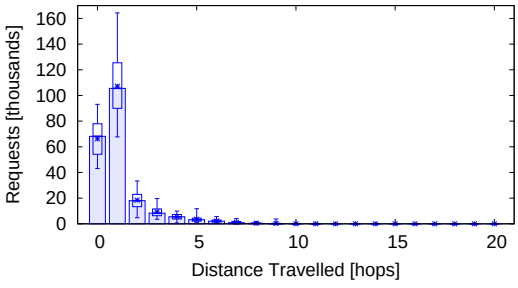


(a) No placement

(b) ASG / simple



(c) ASG / Event Flow Tree



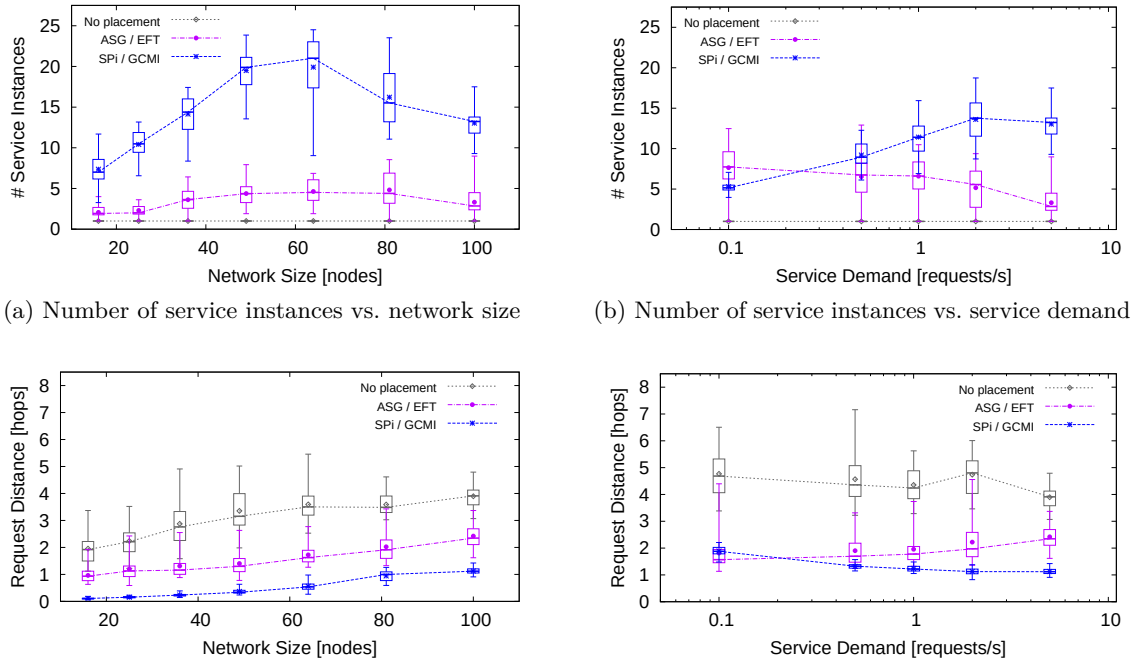
(d) SP*i* / GCMI

Figure 6.5: Service requests vs. distance (100 nodes, 2 requests/s)

size of 100 nodes. All other parameters once again correspond to those of the general setup for simulations as presented in Section 6.3.1.

Like before, we begin our evaluation with histograms of the distances that service requests need to travel from the client nodes to the closest service instance. These histograms are shown in Figure 6.5. Comparing the results of the placement algorithms for distributed services with the setup without service placement, we can observe a substantial change in the behavior of the network. All three placement algorithms adapt the service configuration in such a way that most service requests have to cover a significantly smaller distance in order to reach a service instance. In fact, the peak of the histogram values for all algorithms lies at a distance of 1. This means that for most client nodes, a service instance is merely a single hop away. For SP*i* / GCMI, there is even a significant number of clients that host a service instance on the same node and hence does not require any communication over the network at all.

Just like the previous evaluation of placement algorithms for centralized services, we omit a detailed comparison of the two algorithms proposed in the literature at this point. This comparison is available in Appendix B and results in the fact that the ASG / EFT algorithm is the superior of the two ASG algorithms. Hence, we focus on a comparison between ASG / EFT and SP*i* / GCMI in the remainder of this section.



(a) Number of service instances vs. network size

(b) Number of service instances vs. service demand

(c) Request distance vs. network size

(d) Request distance vs. service demand

Figure 6.6: Placement characteristics for distributed services

In Figure 6.6, we have a first look at the placement characteristics of the two algorithms. Looking at the number of service instances in Figures 6.6a and 6.6b, we observe that SPi / GCMI employs more service instances than ASG / EFT in most cases. The number of service instances used by SPi / GCMI increases steadily with network size and service demand until reaching a plateau and then leveling off. For very large networks, the number of service instances falls. As the number of service instances in SPi / GCMI is dictated by the global minimum of the service provisioning cost (cf. Sec. 5.2), this indicates that different factors of the service provisioning cost come into play: For smaller networks, the traffic between clients and service instances is the significant factor and hence the number of service instances increases as the network grows in size. Once the network reaches a certain size and the clients spread out, the synchronization cost between service instances gains weight and the optimal service configuration includes less service instances. Furthermore, there are less replications (cf. Fig. 6.8b) and thus less service instances in scenarios with low service demand because the expected payoff of adapting a service configuration is smaller. Hence, service adaptations are performed more rarely over a given period of time. In comparison, the number of service instances employed by ASG / EFT remains fairly constant. It drops a bit in large networks and under heavy load, but this must be attributed to the fact that the wireless channel has reached its maximal capacity at this point (cf. Fig. 6.9).

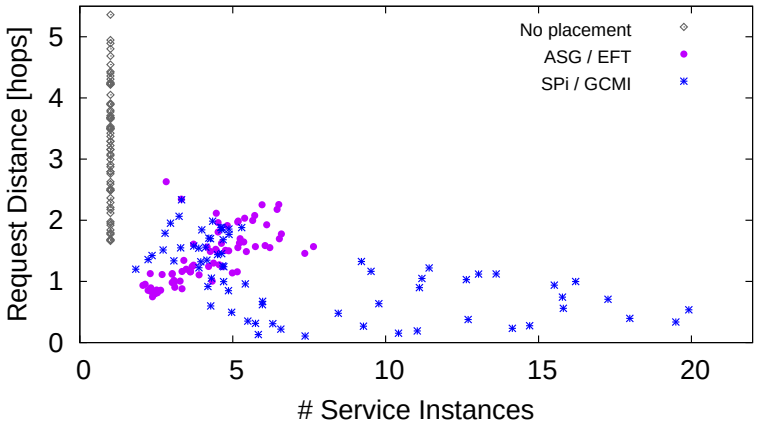
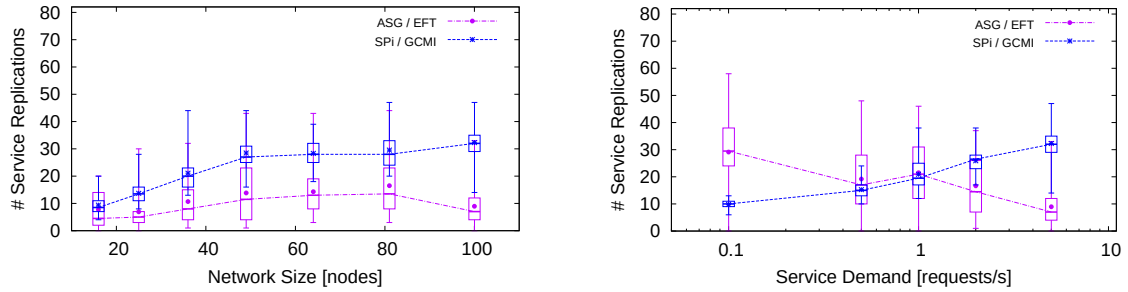


Figure 6.7: Placement quality for distributed services

Looking at the placement characteristics with respect to the distance between clients and service instances in Figures 6.6c and 6.6d, we can observe that SPi / GCMI is able to reduce this distance more than ASG / EFT for almost all scenarios except for the one with very low service demand. Coincidentally, this is the only scenario in which ASG / EFT employs more service instances than SPi / GCMI. This indicates that a higher number of service instances is in fact the better choice for these scenarios. It is also noteworthy that especially for small networks SPi / GCMI achieves a distance between clients and service instances well below 1, i.e., there is a service instance located directly on the client node for most clients. Finally, we also note that both algorithms perform significantly better across all scenarios than the setup without service placement.

In order to examine the quality of the service configuration that each of the placement algorithms is able to achieve, we plot the median number of service instances against the corresponding median distance between clients and the nearest service instance in Figure 6.7. The intuition behind this plot is that if a placement algorithm employs more service instances, this should ideally result in a reduction of the distance between clients and service instances, thereby improving the quality of the service configuration. The data shown in this figure once again illustrates that SPi / GCMI employs more service instances than ASG / EFT. More importantly, however, it also shows that the additional service instances created by SPi / GCMI are placed in such a way that the distance between clients and service instances decreases on average. For ASG / EFT, this is not the case.

Looking at the placement activity for these scenarios in Figure 6.8, we find that SPi / GCMI generally uses more replications to achieve the optimal service configuration than ASG / EFT. This was to be expected since SPi / GCMI employs more service instances in most scenarios. Just as in Figure 6.6, the value decreases for ASG / EFT for large networks and heavy



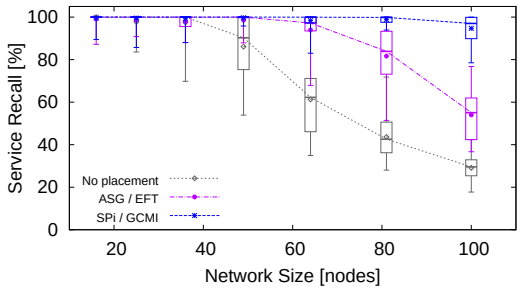
(a) Number of replications vs. network size

(b) Number of replications vs. service demand

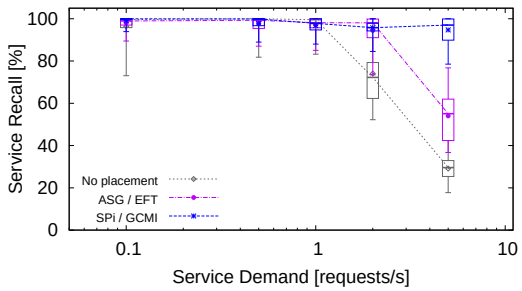
Figure 6.8: Placement activity for distributed services

load. Like above, we attribute this behavior to the saturation of the wireless channel. The noteworthy data point in these figures is the high number of replications under very low service demand in Figure 6.8b. This data point indicates that the high number of service instances seen in Figure 6.6b can indeed be attributed to the ASG / EFT algorithm failing to converge to a stable service configuration under low load. Furthermore, we note that the results for SP*i* / GCMI show a smaller variance than those of ASG / EFT. We interpret this as an indication that the process of adapting a service configuration as implement in SP*i* / GCMI is more controlled and thus less susceptible to the characteristics of the network topology than that of ASG / EFT.

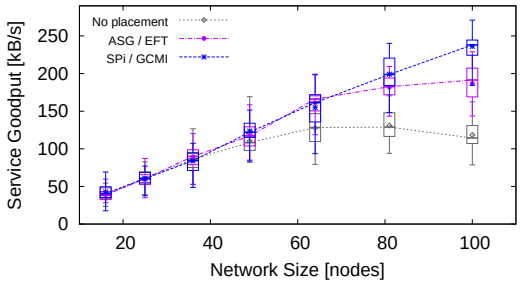
Finally, we have a look at the effects of the service placement algorithms on the performance of the network in Figure 6.9. For the three setups, placement with SP*i* / GCMI, placement with ASG / EFT, and no service placement, we observe substantially different performance characteristics. Starting with a look at the network load in Figure 6.9g and 6.9h, we can observe that SP*i* / GCMI requires significantly less network traffic than both ASG / EFT and the traditional client/server architecture without service placement. In fact, while the latter two approaches eventually saturate the wireless channel, this is not the case for SP*i* / GCMI. This behavior of the network is due to the superior service configuration established by SP*i* / GCMI that causes less traffic to be exchanged between nodes. The immediate result is that a network running SP*i* / GCMI for service placement is able to sustain a high levels of recall and goodput even in large networks and under heavy load (cf. Figs. 6.9a to 6.9d). The other two approaches are not able to provide the service to the clients with the same quality. As expected, the setup without service placement performs poorly when compared to the other two. A very large difference can also be observed in the round-trip time of service requests in Figures 6.9e and 6.9f. Since the network employing SP*i* / GCMI does not reach the limits of its capacity, the clients can benefit from shorter access times to the service. Especially when run under high load, i.e., 5 service requests per second, for all



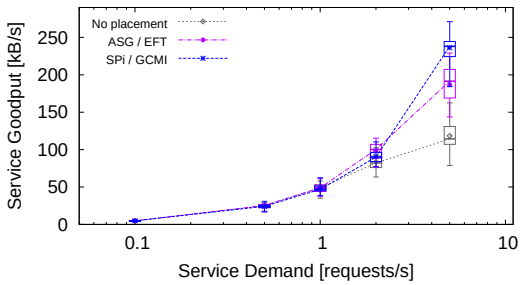
(a) Service recall vs. network size



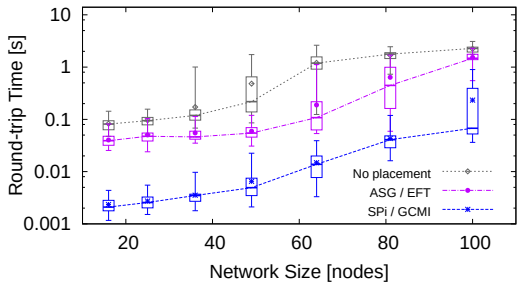
(b) Service recall vs. service demand



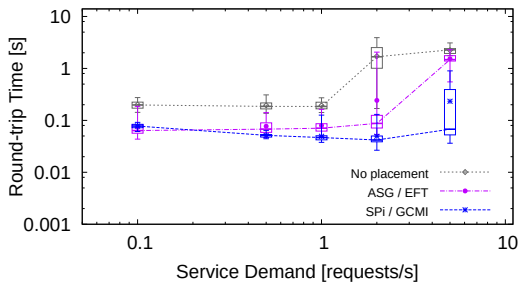
(c) Service goodput vs. network size



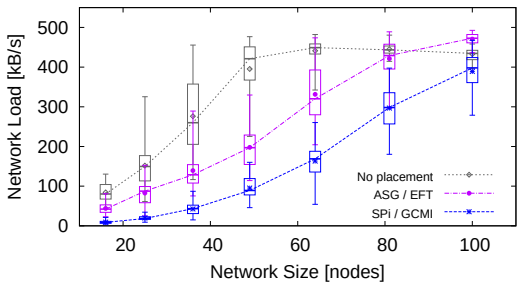
(d) Service goodput vs. service demand



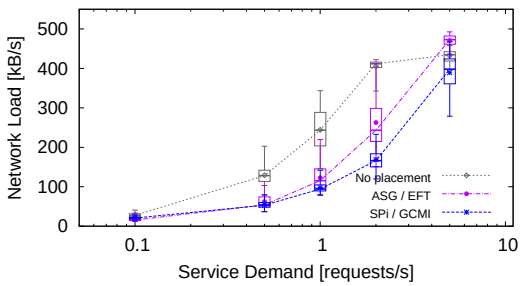
(e) Round-trip time vs. network size



(f) Round-trip time vs. service demand



(g) Network load vs. network size



(h) Network load vs. service demand

Figure 6.9: Placement effect for distributed services

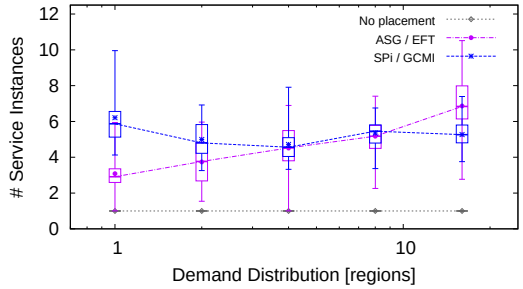
the network sizes in Figure 6.9e the access time differs by between one and two orders of magnitude.

The evaluation presented in this section has for the first time shown the real benefit of employing service placement in ad hoc networks. Compared to a traditional client/server architecture without service placement, our SP*i* / GCMI placement algorithm is able to substantially improve the quality of the service as perceived by the clients (in terms of availability, throughput, and access times) and at the same time reduce the network traffic required for doing so. We also note that a network with a service configuration established by SP*i* / GCMI outperforms a network with a service configuration established by ASG / EFT. This can be attributed to the superior quality of the service configuration established by SP*i* / GCMI. Furthermore, we also note that the architectural weakness of SP*i* / GCMI, i.e., the communication overhead due to the signaling between the service instances (cf. Sec. 4.2), is more than compensated for by the quality of the service configuration. We take this as a first indication that a centralized approach to service placement in ad hoc networks is indeed superior to a distributed approach. However, we will examine this in more detail in the following Sections 6.6 and 6.7.

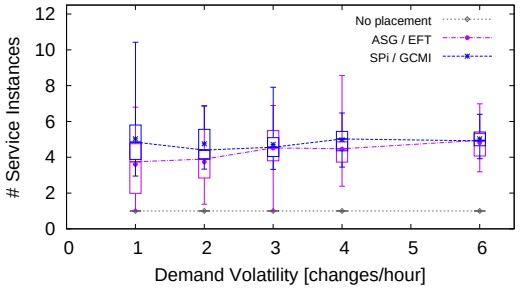
6.6 Service Placement under Varying Service Demand

Now that we have evaluated the impact of service placement on general scalability of ad hoc networks, we proceed by investigating in how far placement algorithms are able to cope with varying service demand on the client nodes. In contrast to the previous section, we do, however, not vary the overall volume of the service demand, but rather the regional and temporal patterns of demand on the client nodes. More precisely, we look at the *demand distribution* in the ad hoc network, i.e., whether service demand is present in specific regions of the network or whether it originates on randomly placed nodes. Furthermore, we also look into the *demand volatility*, i.e., how frequently the demand pattern changes during a deployment.

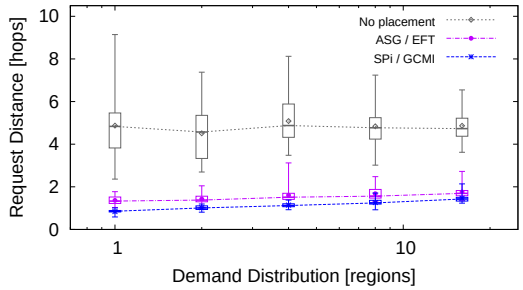
These simulations follow the same basic setup as presented in Section 6.3.1. The simulated networks consist of 100 randomly placed nodes, 16 of which are client nodes, i.e., nodes with service demand. The comparatively low number of client nodes is due to the fact that we intend to maximize the impact of the location of these nodes within the network topology. With a more balanced ratio between client nodes and passive nodes, the importance of the location of the individual client nodes on the behavior of the network would decrease. Furthermore, we changed the synchronization ratio from $\tau = 1\%$ to $\tau = 5\%$ as otherwise SP*i* / GCMI would select the trivial service configuration of placing one service instance on each client node.



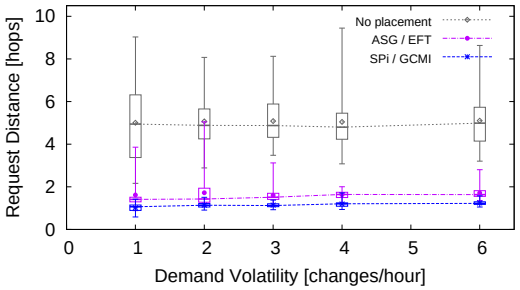
(a) Number of service instances vs. demand distribution



(b) Number of service instances vs. demand volatility



(c) Request distance vs. demand distribution



(d) Request distance vs. demand volatility

Figure 6.10: Placement characteristics under varying service demand

We varied the demand distribution by creating a variable number of demand regions. For each region to be created, we randomly chose a point in the simulated area as the respective center of the circular regions. We then increase the radius of each of these regions until the sum of the nodes contained in them reaches the desired value, in our case 16 nodes. If the number of regions is set to 1, this results in a single cluster of client nodes. If the number of regions is set to 16, then 16 randomly chosen nodes act as clients. This method for constructing regional demand works correctly because the two-ray ground reflection model, which we use to simulate the propagation characteristics of the radio signals, maps close physical proximity between nodes to a good wireless communication channel between these nodes. Since we ran these simulations with 16 client nodes, we varied the number of regions between 1 (one big demand cluster) and 16 (randomly placed client nodes).

To vary the demand volatility, we introduce a new parameter for the number of changes to the set of client nodes over the course of a simulation run. A value of 1 implies that there is no change after the initial selection of client nodes. With higher values the regional demand clusters change more and more rapidly. We ran these experiments for a simulated time of one hour and varied the number of changes to demand clusters between 1 and 6, i.e., we have a change in the regional demand every ten minutes under the highest setting.

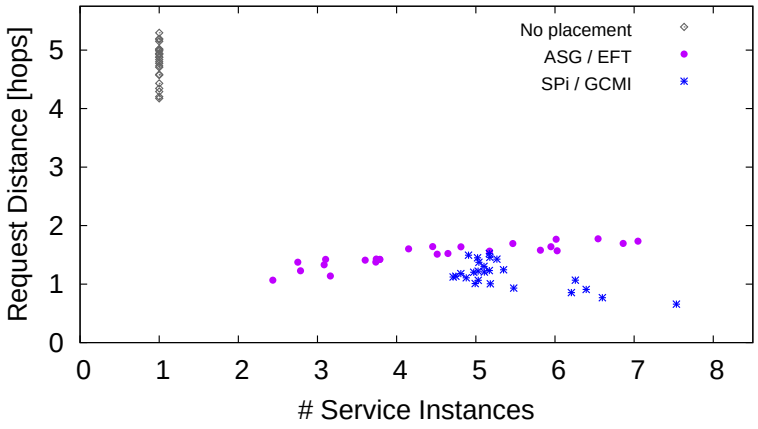
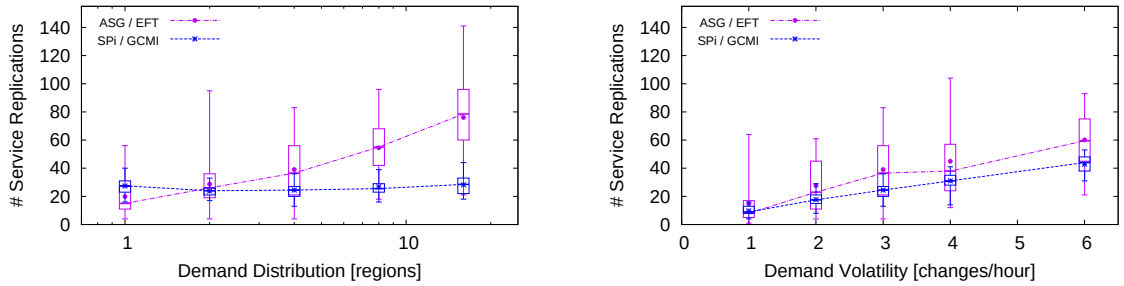


Figure 6.11: Placement quality under varying service demand

The following figures show variations of demand distribution and demand volatility. For the evaluation of the demand distribution, we fixed the demand volatility at three changes per hour, i.e., one change every 20 minutes. This corresponds to the total duration of the simulations in the previous two sections. For the evaluation of the demand volatility, we set the demand distribution to four regions.

In the presentation of the results, we follow the same pattern as in the previous two sections. In Figure 6.10, we have a look at the inner workings of the SP*i* / GCMI and the ASG / EFT placement algorithms. We note that in contrast to SP*i* / GCMI, the number of service instances employed by ASG / EFT increases with the number of demand regions. Since the size of the network remains constant, this behavior needs to be attributed to the increasing distances between the clients. In fact, the replication rule of ASG / EFT creates new service instances if service requests are received from far away nodes. Looking at the number of service instances in light of changing demand volatility, we note that it remains fairly constant. The service configuration that both algorithms establish in all scenarios reduces significantly the distances between clients and service instances as compared to the traditional client/server architecture. The results also indicate that the service configuration found by SP*i* / GCMI is marginally superior according to this metric.

Plotting the median distance a service request has to travel before reaching a service instance against the median number of service instances for these scenarios in Figure 6.11, we note once again that SP*i* / GCMI tends to employ a slightly larger number of service instances. More importantly, and reaffirming the results from Section 6.5, SP*i* / GCMI manages once again to find better configurations with the same number of service instances, i.e., configurations that result in a reduction in the distance between clients and service instances.



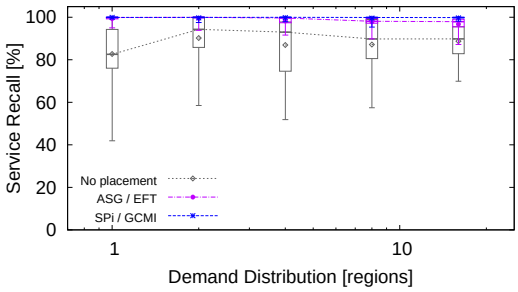
(a) Number of replications vs. demand distribution

(b) Number of replications vs. demand volatility

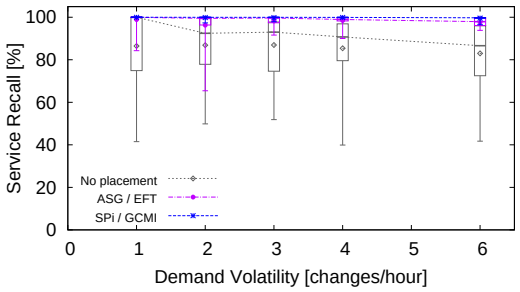
Figure 6.12: Placement activity under varying service demand

Looking at the activity of the two placement algorithms as depicted in Figure 6.12, we can observe that SP*i* / GCMI shows a significantly different behavior than ASG / EFT. For ASG / EFT, the number of replications increases as the service demand is distributed more randomly across the network. Once again, this can be attributed to the structure of its replication rule. For SP*i* / GCMI in contrast, the number of replications remains nearly constant. Looking at the activity under changing demand volatility, the two algorithms show a similar behavior. More importantly, however, we note that SP*i* / GCMI exhibits a far lower variance in the number of replications than ASG / EFT. This reaffirms our observation from Section 6.6 that the service placement process as implemented in SP*i* / GCMI operates reliably across a variety of different scenarios. More generally, these results indicate that a centralized placement algorithm as implemented in SP*i* / GCMI is less subject to changing conditions of the scenario than a distributed approach as implemented in ASG / EFT.

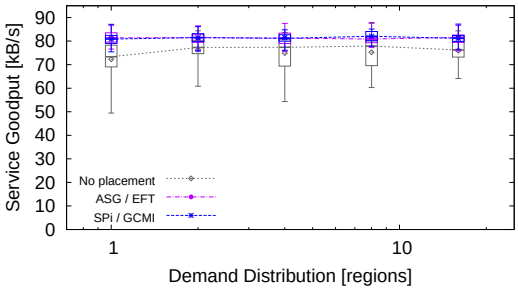
Before concluding this part of the evaluation, we now examine the effect that the two placement algorithms have on the overall performance of the ad hoc network. Figure 6.13 once again plots the same metrics as in the two preceding sections. We note that while service recall and goodput do not differ significantly between SP*i* / GCMI and ASG / EFT, both algorithm show a noteworthy improvement over the performance of a traditional client/server architecture without service placement. This improvement becomes even more apparent when looking at the service access times in Figure 6.13e and 6.13f. Both placement algorithms improve the round-trip time of service requests by more than one order of magnitude as compared to the client/server architecture. Similarly, both algorithms also significantly reduce the network load. Looking at this last metric, we also note the effect of the slightly superior service configuration found by SP*i* / GCMI (cf. Fig. 6.11). The overall traffic required by a network with SP*i* / GCMI for providing the same quality of service is approximately 50 kB/s lower in all scenarios than that of a network employing ASG / EFT. Depending on the scenario, this corresponds to a reduction in traffic by up to 30%.



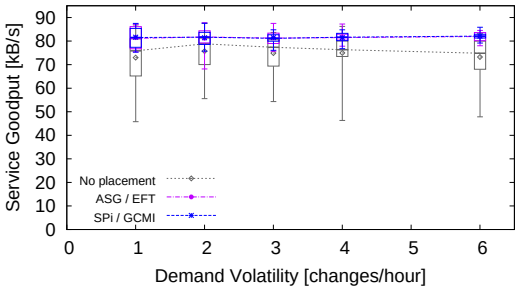
(a) Service recall vs. demand distribution



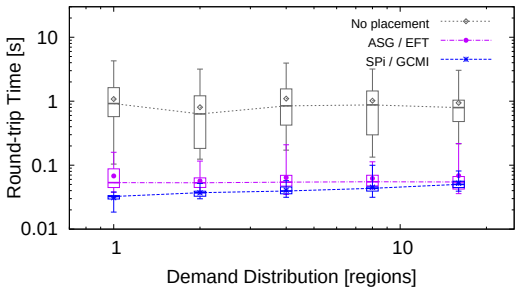
(b) Service recall vs. demand volatility



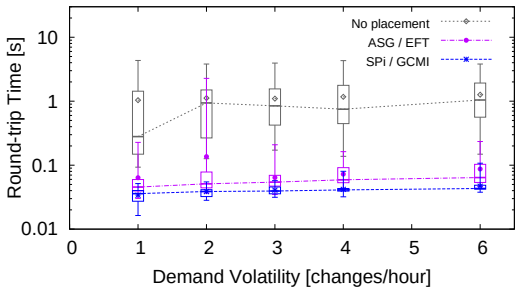
(c) Service goodput vs. demand distribution



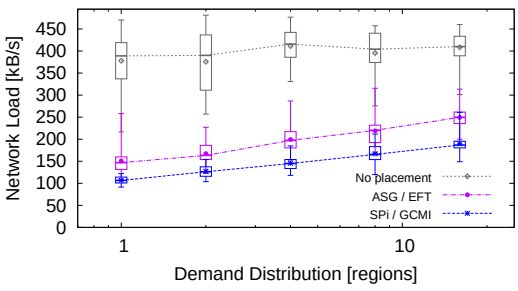
(d) Service goodput vs. demand volatility



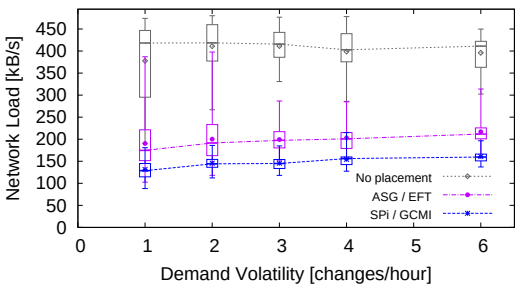
(e) Round-trip time vs. demand distribution



(f) Round-trip time vs. demand volatility



(g) Network load vs. demand distribution



(h) Network load vs. demand volatility

Figure 6.13: Placement effect under varying service demand

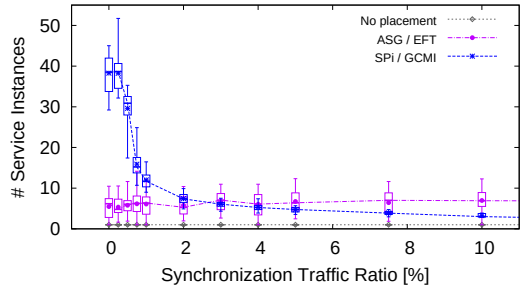
The evaluation of the two placement algorithms under varying service demand has shown that both algorithms cope well with the varying scenarios and perform significantly better than a setup without service placement. The service configurations established by SP*i* / GCMI prove to be quantitatively superior over those found by ASG / EFT in that they are able to provide the same high level of service quality but at less network traffic. Furthermore, this evaluation has shown that the service placement process implemented in SP*i* / GCMI is more predictable and less susceptible to variations of the scenario. We interpret this as an indication that service placement, when coordinated by a centralized placement algorithm, is more likely to reach a stable service configuration.

6.7 Service Placement under Varying Synchronization Requirements

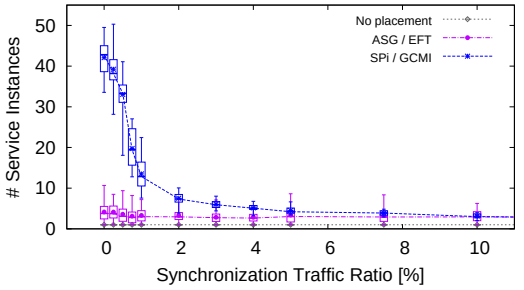
As a final evaluation step that we conduct using the `ns-2` network simulator, we investigate in how far the SP*i* / GCMI and the ASG / EFT placement algorithms are capable of handling different synchronization requirements between the service instances. As we have motivated in Section 5.2.2, the need for keeping the global state of the service synchronized differs depending on the type of service and on the traffic patterns of the clients. As the need for synchronization increases, so does the required traffic between the services instances. A service placement algorithm that takes this fact into account reduces the number of service instances with increasing synchronization requirements.

We once again simulate a network consisting of 100 nodes following the setup as described in Section 6.3.1. In this setup, we varied the ratio of the type of service requests that necessitates updates between service instances. This effectively manipulates the value for the synchronization ratio τ (cf. Sec. 5.2.2). We simulated scenarios with values for τ ranging between 0% and 10%. As we are interested in the behavior of the network both under normal operation and under heavy load, we simulated these scenarios with request frequencies of both two and five requests per second.

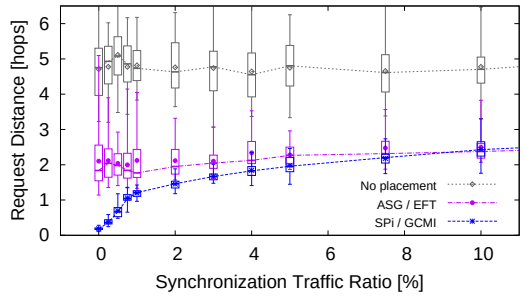
In Figure 6.14, we plot once again the number of service instances and the distance between clients and service instances. Under both normal and high load, there is a substantial difference in the behavior of SP*i* / GCMI and ASG / EFT: While the number of service instances remains near-constant for ASG / EFT, it increases sharply for lower values of τ with SP*i* / GCMI. As pointed out above, this is the expected behavior since lower synchronization requirements imply that more service instances can be viably deployed. The increased number of service instances deployed by SP*i* / GCMI for small values of τ results in significant reduction in the distance service requests have to travel before reaching a service instance. Further, as we have already come to expect from the previous evaluations, both



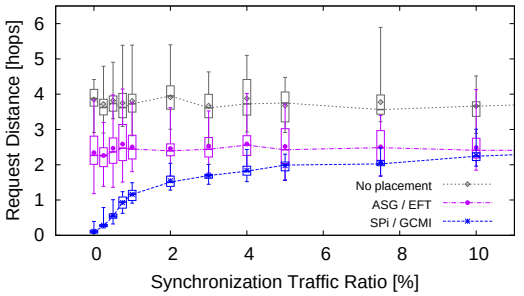
(a) Number of service instances vs. synchronization traffic ratio (2 requests/s)



(b) Number of service instances vs. synchronization traffic ratio (5 requests/s)



(c) Request distance vs. synchronization traffic ratio (2 requests/s)



(d) Request distance vs. synchronization traffic ratio (5 requests/s)

Figure 6.14: Placement characteristics under varying synchronization requirements

placement algorithms significantly reduce this distance in comparison to a setup without service placement.

The difference in behavior of the two placement algorithms becomes even more obvious when plotting the median number of service instances against the median hop count of service requests for all scenarios. As depicted in Figure 6.15, the service configurations established by ASG / EFT are not affected by the variations in synchronization requirements. SPi / GCMI, in contrast, makes use of the full range of possible service configurations. On one extreme, if multiple instances are prohibitively expensive, it approximates the behavior of the traditional client/server architecture with a single service instance. On the other extreme, if no synchronization is necessary, it places one service instance on each client node.

Looking at the placement activity as shown in Figure 6.16 we can observe the expected increase in number of replications for small values of τ for SPi / GCMI. These replications are required for creating the large number of service instances that we discussed above. ASG / EFT does not exhibit this behavior and the number of replications remains fairly constant both under normal and high load.

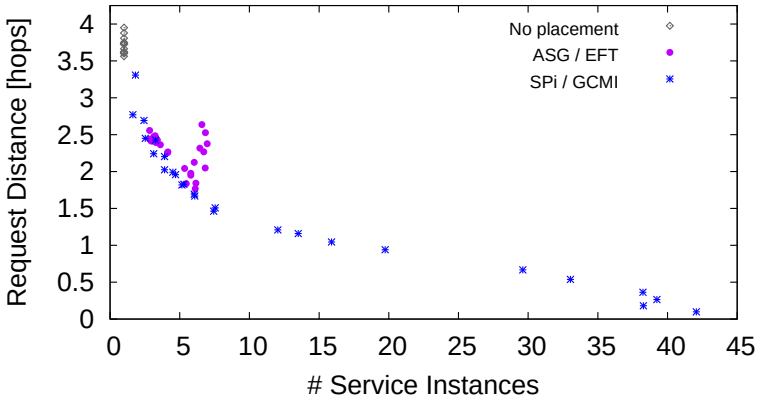
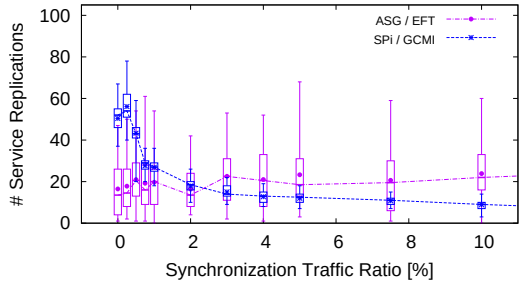
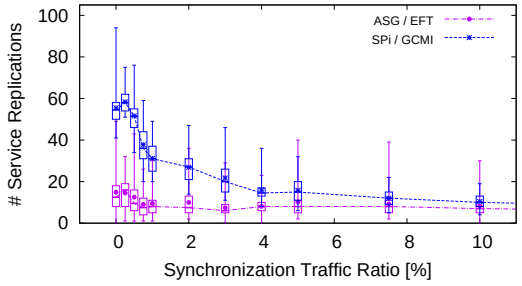


Figure 6.15: Placement quality under varying synchronization requirements



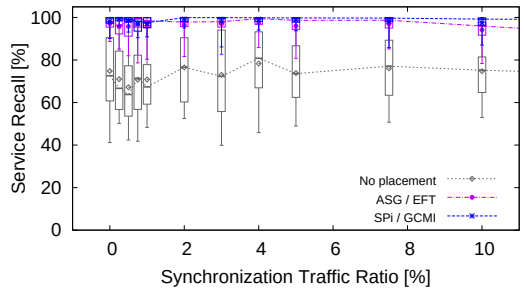
(a) Number of replications vs. synchronization traffic ratio (2 requests/s)



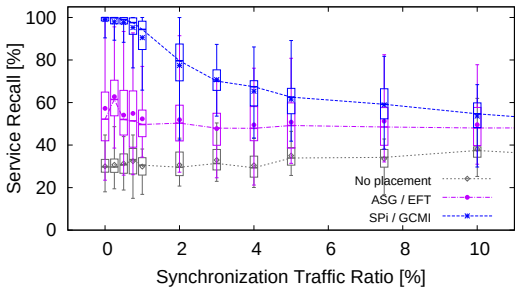
(b) Number of replications vs. synchronization traffic ratio (5 requests/s)

Figure 6.16: Placement activity under varying synchronization requirements

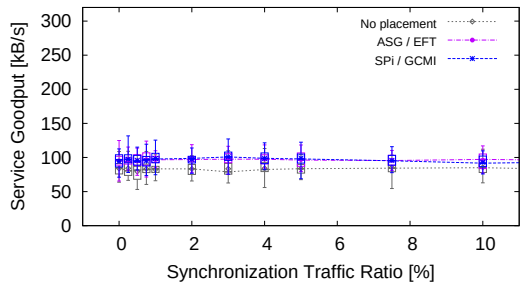
Figure 6.17 shows once again the effect that the placement decisions have on the overall performance on the network. The results for normal load (on the left in Figs. 6.17a, 6.17c, 6.17e, and 6.17g) differ significantly from those scenarios in which the capacity network is exceeded (on the right in Figs. 6.17b, 6.17d, 6.17f, and 6.17h). In the scenarios with normal load, there is little difference between the two algorithms in service recall and goodput. The service access time of SP*i* / GCMI improves over that of ASG / EFT for $\tau \leq 1\%$. And, once again, the network with SP*i* / GCMI generates less overall traffic than that with ASG / EFT. For the saturated network, network load and round-trip time of ASG / EFT match that of a setup with service placement. Service recall and goodput is equally poor, ranging at approximately 50% of the optimum. The same is true for SP*i* / GCMI, but only for scenarios with high synchronization requirements. As soon as the synchronization requirements drop, SP*i* / GCMI creates additional instances of the service which dramatically reduce the network traffic. In fact, the traffic volume approaches zero if no synchronization between instances is necessary. As the traffic decreases, the overall quality of the service, i.e., recall, goodput,



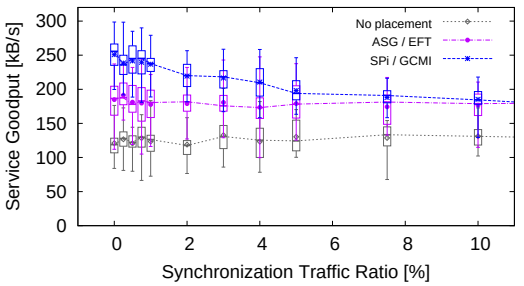
(a) Service recall vs. synchronization traffic ratio (2 requests/s)



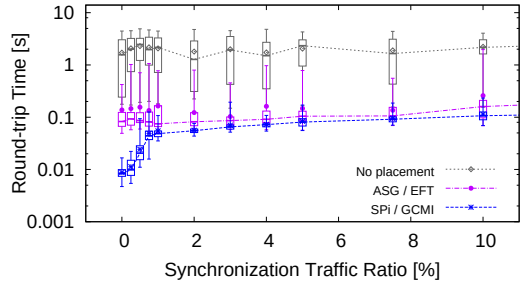
(b) Service recall vs. synchronization traffic ratio (5 requests/s)



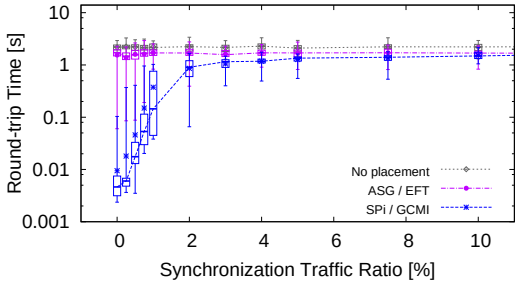
(c) Service goodput vs. synchronization traffic ratio (2 requests/s)



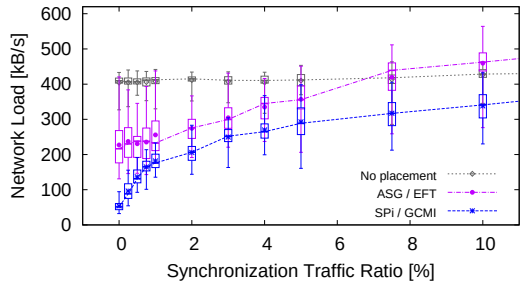
(d) Service goodput vs. synchronization traffic ratio (5 requests/s)



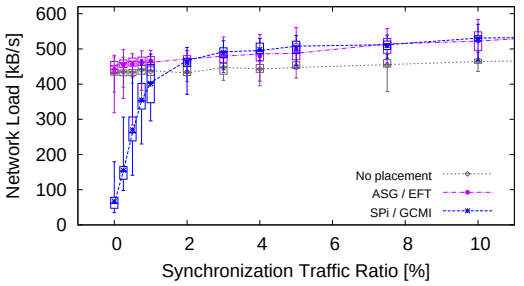
(e) Round-trip time vs. synchronization traffic ratio (2 requests/s)



(f) Round-trip time vs. synchronization traffic ratio (5 requests/s)



(g) Network load vs. synchronization traffic ratio (2 requests/s)



(h) Network load vs. synchronization traffic ratio (5 requests/s)

Figure 6.17: Placement effect under varying synchronization requirements

and access time, improves and reaches acceptable levels that are unattainable for the other placement algorithms let alone a traditional client/server architecture.

The evaluation in this section has shown that SP*i* / GCMI gracefully handles scenarios and services with varying synchronization requirements. SP*i* / GCMI correctly adapts the number of service instances depending on how much synchronization traffic is to be expected. ASG / EFT, in contrast, does not adapt its placement strategy to the synchronization requirements. This should, however, not be mistaken as a general property of distributed placement algorithms. In fact, it is conceivable that ASG / EFT could be amended to take this aspect of service placement into account. Doing so would certainly be interesting, but is beyond the scope of our current work. Instead, we conclude that SP*i* / GCMI is the superior placement algorithm for scenarios with varying synchronization requirements.

6.8 Service Placement under Varying Link Quality

Having evaluated the characteristics of both the ASG / EFT and the SP*i* / GCMI placement algorithms for distributed services in the previous sections, we now once again shift the focus of our evaluation. In this section, we investigate in how far service placement algorithms are able to mitigate the impact of lossy layer 2 links on the performance of an ad hoc network. This part of the evaluation is based on results obtained using the emulation setup as described in Section 6.3.2.

In this and the subsequent parts of the evaluation, we limit ourselves to investigating the behavior of ad hoc networks that employ either the SP*i* / GCSI or the SP*i* / GCMI placement algorithms. This is mostly due to the time required to conduct these experiments: One measurement takes approximately 25 minutes to complete (including 20 minutes for the experiment and another 4 to 5 minutes for setup and data collection). For three algorithms measured across five settings for service demand, four settings for link reliability, and 15 measurements per data point, this amounts to a total time required to conduct these experiments of 375 hours, or slightly more than 15.5 days (not counting runs for establishing appropriate ranges for some of the parameters). Given the fact that we have shown that both SP*i* / GCSI and SP*i* / GCMI match or exceed the performance of other placement algorithms in the previous sections, we argue that it is justified to focus on these two algorithms for very time consuming experiments. For reference, we continue to perform measurements for a setup without service placement in all scenarios.

The new aspect that we introduce into the evaluation in this section is the reliability of the communication links between the nodes. As pointed out in Section 6.3.2, our hypothesis is that the reduction in the average distance between clients and service instance, in particular as achieved by SP*i* / GCMI, will mitigate the negative impact of a reduction in link quality

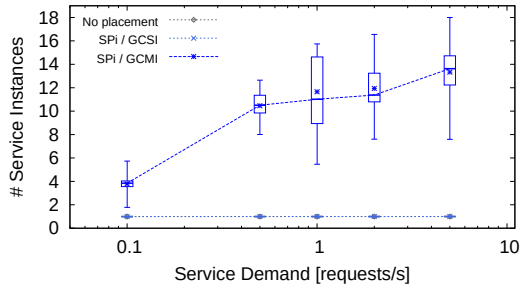
on the overall performance of the network. With the 36 nodes at our disposal, we thus evaluated the same scenarios for service demand (with request frequencies ranging between 0.1 and 5 requests per second for each client node) with artificially introduced packet loss. For the Packet Loss Rate (PLR) we used for values ranging from 0% to 30%. These values were chosen to cover the entire range of interesting effects of the placement algorithms.

Where applicable, we continue to use the same parameters as in the simulation-based evaluation (cf. Sec. 6.3.1). The only noteworthy change to the setup is that we enabled the optimizations for lossy links of the SP*i* framework (cf. Sec. 4.4.4). As a result, only immediate neighbors of current service hosts are considered as potential replication or migration targets. This is done to avoid transfers of large amounts of data over multiple unreliable hops which, in combination with timeouts in the transport layer protocol, would result in a less reliable adaptation process.

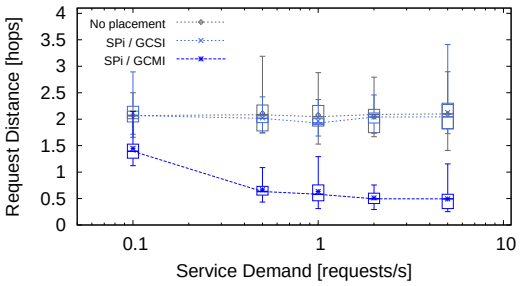
Starting as in the previous parts of the evaluation with a look at the inner workings of the placement algorithms in Figure 6.18, we note that changes in PLR have in fact a significant impact on the behavior of the placement algorithms. Looking at the diagrams from top to bottom, we can see on the left-hand side in Figures 6.18a, 6.18c, 6.18e, and 6.18g how the number of service instances employed by SP*i* / GCMI decreases as packet loss increases. As a result, and plotted on the right-hand side in Figures 6.18b, 6.18d, 6.18f, and 6.18h, the reduction in distance between clients and service instances achieved by SP*i* / GCMI becomes less and less significant. For SP*i* / GCSI and the setup without service placement, this metric remains unaffected by the changes in PLR since only successful service requests are counted.

The reason for this change in behavior of SP*i* / GCMI can be seen in Figure 6.19. As the PLR increases from Figure 6.19a to 6.19d, the number of replications initiated by SP*i* / GCMI decreases across all scenarios. This is due to the fact that as links become less and less reliable, more timeouts and retransmissions of packets on the transport layer are required to successfully transfer the state of a service instance from one node to another. As a result, the duration of service adaptations is more and more dominated by the timeouts of the transport layer protocol. Rather than completing successfully as soon as all service instances have reported to the coordinator node, the adaptations of the service configuration tend to be aborted due to timeouts. Hence, the total number of adaptations during the experiment drops and with it the number of service replications. We can thus conclude that the underlying mechanisms of service placement start to fail as link quality drops. Therefore, the interesting question is whether service placement can mitigate the effects of lossy links as perceived by the client nodes while the replication mechanism is still operational.

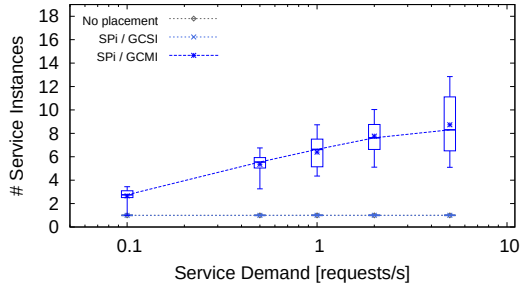
We answer this question in Figures 6.20 and Figure 6.21. Note that in these figures, each vertical column of four diagrams contains the same metrics that we have used for the evaluation of the overall impact of service placement on a network in the previous sections.



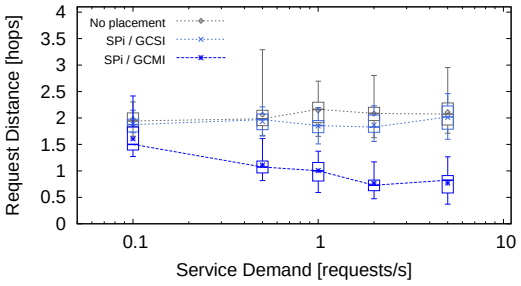
(a) Number of service instances vs. service demand (0% PLR)



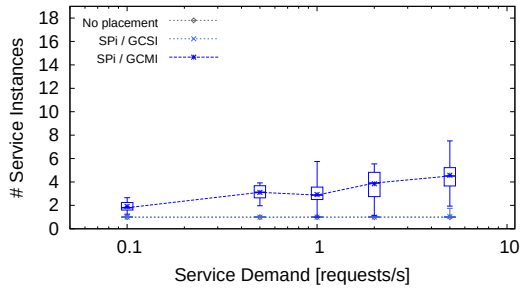
(b) Request distance vs. service demand (0% PLR)



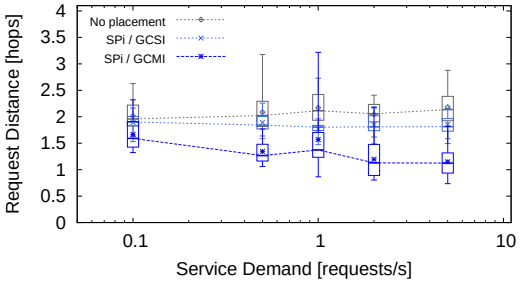
(c) Number of service instances vs. service demand (10% PLR)



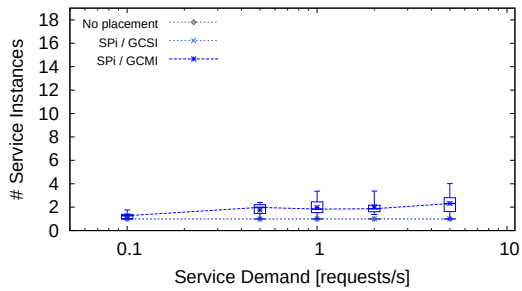
(d) Request distance vs. service demand (10% PLR)



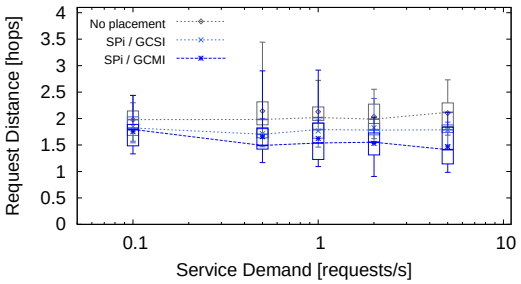
(e) Number of service instances vs. service demand (20% PLR)



(f) Request distance vs. service demand (20% PLR)

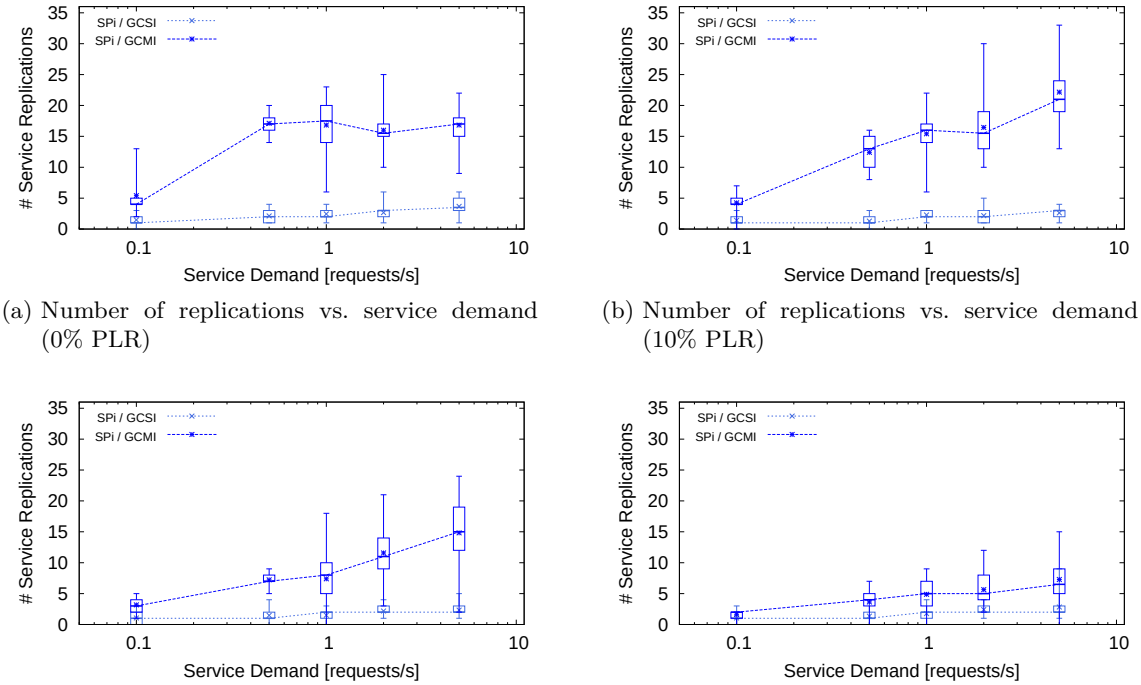


(g) Number of service instances vs. service demand (30% PLR)



(h) Request distance vs. service demand (30% PLR)

Figure 6.18: Placement characteristics under varying link quality



(a) Number of replications vs. service demand (0% PLR)

(b) Number of replications vs. service demand (10% PLR)

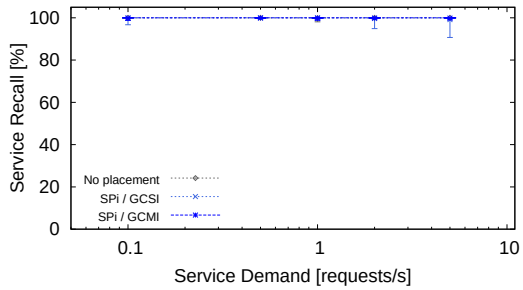
(c) Number of replications vs. service demand (20% PLR)

(d) Number of replications vs. service demand (30% PLR)

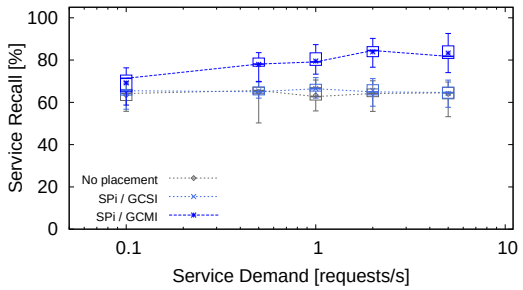
Figure 6.19: Placement activity under varying link quality

Looking at the four columns from left to right, the PLR increases, i.e., the quality of the communication links between nodes decreases. Beginning with the diagrams on the left in Figures 6.20a, 6.20c, 6.20e, and 6.20g, i.e., the results of the emulation setup without artificial packet loss, we observe the behavior of the network that we have come to expect when employing the two placement algorithms: SPi / GCSI only shows minor improvements over the setup without service placement because the network only consists of 36 nodes and a bad initial placement of the service instance has thus only minimal effect. SPi / GCMI achieves the same flawless service recall and goodput as the other two algorithms, but at significantly reduced access times and network load. We note that this effect of our service placement system as observed in this emulated setup matches that of the prior simulations.

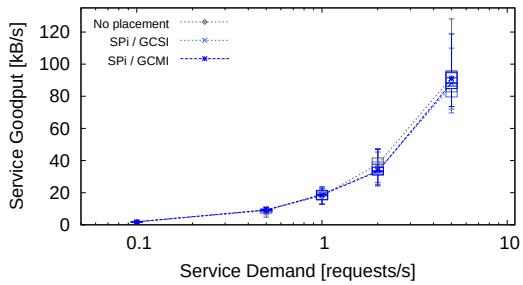
Looking at the impact of lossy links on the metrics on by one, we note that the service recall for all three setups drops from 100% at 0% PLR in Figure 6.20a to a median of approximately 20% at 30% PLR in Figure 6.21b. The average distance between clients and service instances in the latter scenario is approximately 2 (cf. Fig. 6.18h), i.e., a service request is forwarded 4 times on its way from the client to the service instance and back. With 30% PLR, the probability of a successful transmission of a service requests is



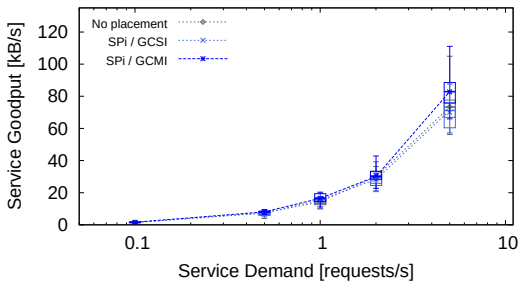
(a) Service recall vs. service demand (0% PLR)



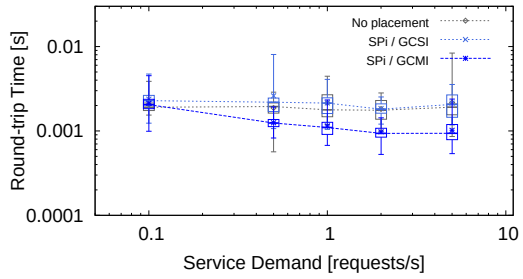
(b) Service recall vs. service demand (10% PLR)



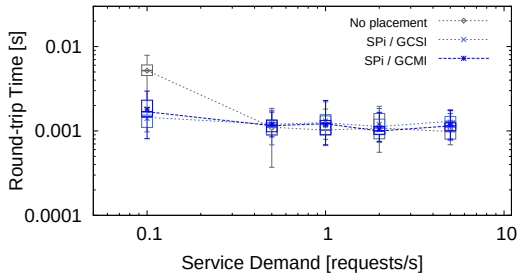
(c) Service goodput vs. service demand (0% PLR)



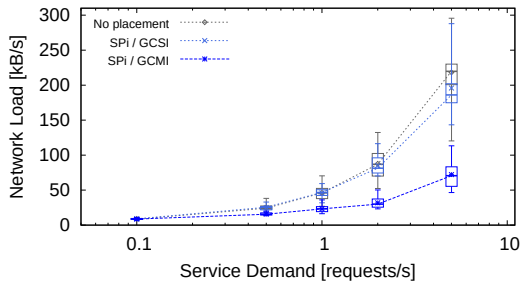
(d) Service goodput vs. service demand (10% PLR)



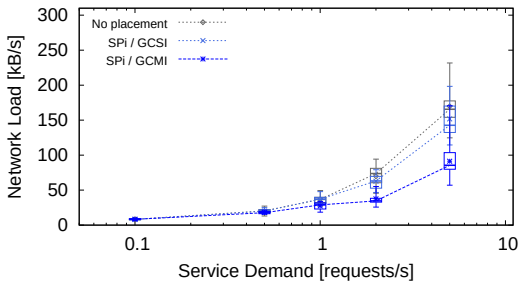
(e) Round-trip time vs. service demand (0% PLR)



(f) Round-trip time vs. service demand (10% PLR)

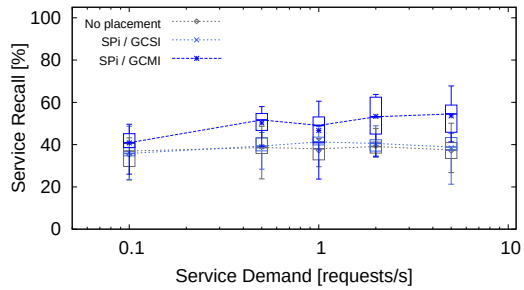


(g) Network load vs. service demand (0% PLR)

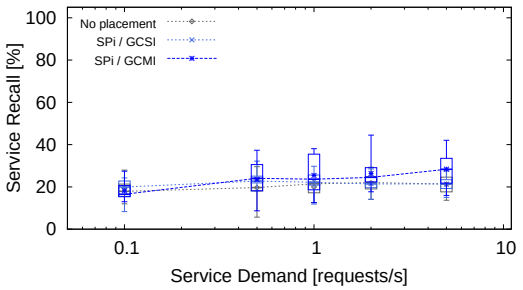


(h) Network load vs. service demand (10% PLR)

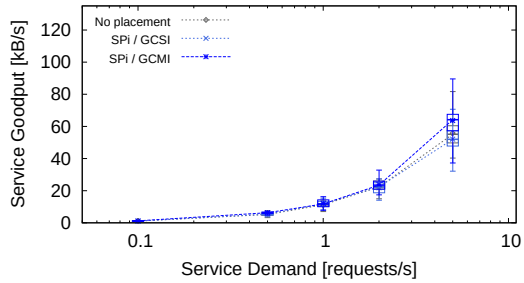
Figure 6.20: Placement effect under varying link quality (0% and 10% PLR)



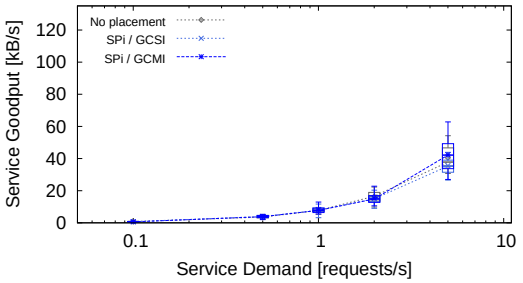
(a) Service recall vs. service demand (20% PLR)



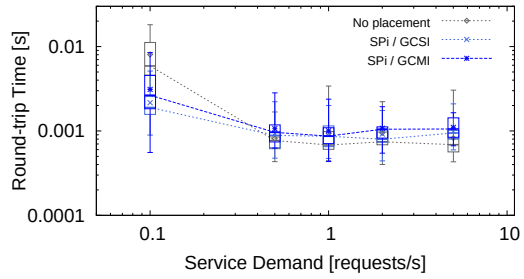
(b) Service recall vs. service demand (30% PLR)



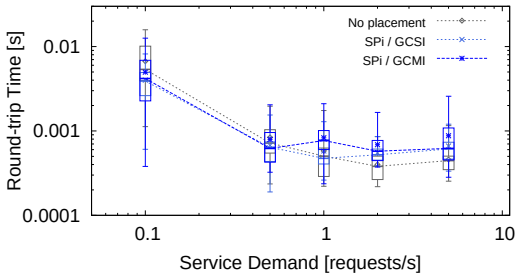
(c) Service goodput vs. service demand (20% PLR)



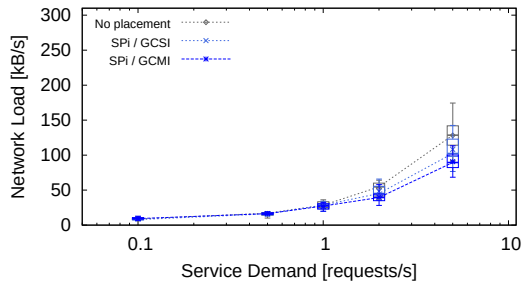
(d) Service goodput vs. service demand (30% PLR)



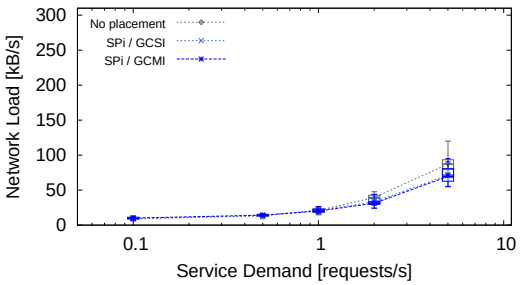
(e) Round-trip time vs. service demand (20% PLR)



(f) Round-trip time vs. service demand (30% PLR)



(g) Network load vs. service demand (20% PLR)



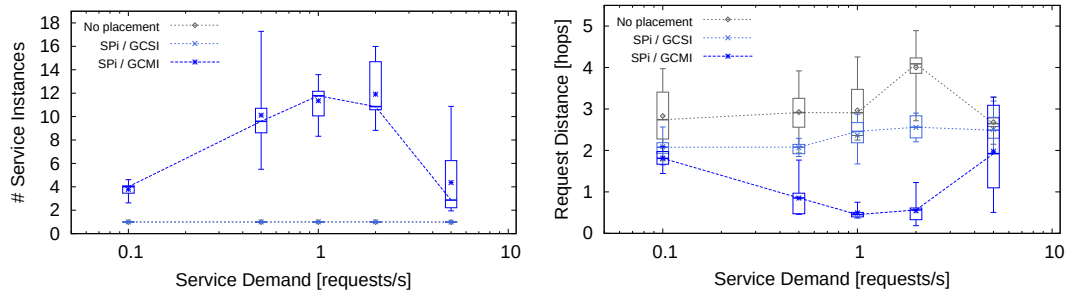
(h) Network load vs. service demand (30% PLR)

Figure 6.21: Placement effect under varying link quality (20% and 30% PLR)

thus $(1 - 0.3)^4 = 0.2401$ which approximates the measured value. Hence, our mechanism for generating artificial packet loss works correctly. More interestingly, however, are the results for 10% and 20% PLR shown in Figures 6.20b and 6.21a. In these figures, we see that SP*i* / GCMI shows a significantly higher recall than SP*i* / GCSI and the setup without placement. This difference is not present for 0% and 20% PLR shown in Figures 6.20a and 6.21b. We conclude from this that a distributed service placed with SP*i* / GCMI is in fact less susceptible to the effects of lossy links than a centralized service.

Of the other three metrics, the service goodput and the network load behave as one would expect as both decrease uniformly for all placement algorithms as the PLR increases. The effects of lossy links on the round-trip time are, however, somewhat counter-intuitive and warrant additional explanation. Our observation is twofold: First, for the scenarios with more than 0.1 requests per second, the round-trip time decreases. As we have pointed out before, this is due to the fact the only successful service requests count towards this metric and hence this metric is skewed in favor of those requests that only have to cover short distances. The second observation is that the round-trip is significantly larger at 0.1 requests per second. This is due to the fact that as the data rate of the service requests is so low that our implementation of the DYMO routing protocol, conforming to the required behavior described in [17], purges the supposedly unused routes from its routing tables after a timeout. The routes need to be reestablished before transmitting the next service request and hence incur a significant delay. For emulation-based part of the evaluation we decided to ignore this effect, but we disabled this mechanism for the evaluation on the real-world testbed (cf. Sec. 6.3.3).

In conclusion, we can state that this part of the evaluation has shown two things: First, the behavior of a real-world network using the SP*i* framework and its placement algorithms matches our expectation based on the results from the simulations. We interpret this as a first indication that the qualitative conclusions that we have drawn from the simulation-based part of the evaluation hold true in reality. We will further support this claim in Section 6.9. And second, we see our hypothesis confirmed that a properly placed distributed service is less affected by link loss than a centralized service. Furthermore, we note that the results presented in this section describe the performance of a service without any form of transport layer protection against packet loss. If the service was to use a reliable transport layer protocol, the effects would be even more pronounced, especially with regard to service recall and round-trip time.



(a) Number of service instances vs. service demand (b) Request distance vs. service demand

Figure 6.22: Placement characteristics on an IEEE 802.11 ad hoc network

6.9 Service Placement in Reality

In this final part of our evaluation, we present the results of a performance evaluation of the SP*i* / GCSI and SP*i* / GCMI service placement algorithms on a real-world IEEE 802.11 ad hoc network. The goal of these experiments is to verify whether a real-world network employing service placement exhibits the same behavior that we have observed in the simulation-based and emulation-based experiments. If this is the case, we can regard it as strong evidence that the conclusion we have drawn so far in this evaluation are actually applicable to real-world deployments. A secondary goal of this evaluation is to give proof that our implementation of the SP*i* service placement framework and its associated algorithms are refined enough to operate correctly on a real ad hoc network, without any simplifying assumptions.

A full description of the setup of these experiments is available in Section 6.3.3. Following the same argument as given in Section 6.8 regarding the time required to run these experiments, we once again limit our evaluation to the SP*i* / GCSI and SP*i* / GCMI placement algorithms, together with a setup without service placement for reference. Like above, we use 36 nodes in these experiments, but we do not perform any form of topology control. The network is measured for varying service demand with request frequencies ranging between 0.1 and 5 requests per second for each client node. Setup and duration of an experiment match those as described as part of the emulation-based evaluation (cf. Sec. 6.8).

Figure 6.22 shows the metrics regarding the inner working of the placement algorithms in a real-world setting. The number of service instances exhibits a similar behavior as that of the emulation-based experiment (cf. Fig. 6.18a), except for very high volumes of service request. For these scenarios, the number of service instances employed by SP*i* / GCMI drops significantly. We can observe the corresponding trend in the distances between clients and service instances: SP*i* / GCMI manages to significantly reduce these distances when compared to SP*i* / GCSI and the traditional client/server architecture without active service placement,

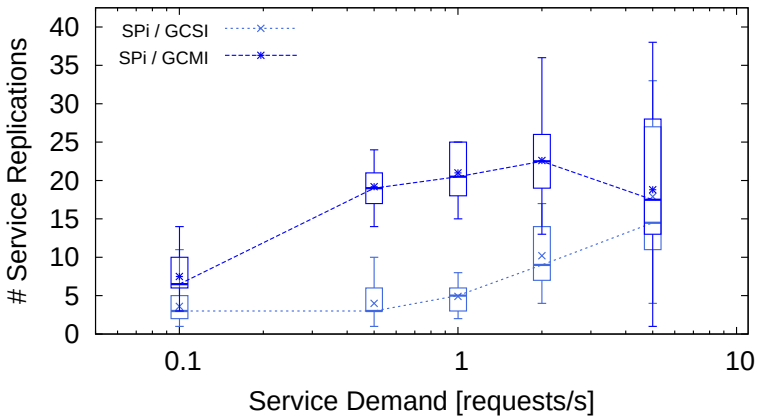
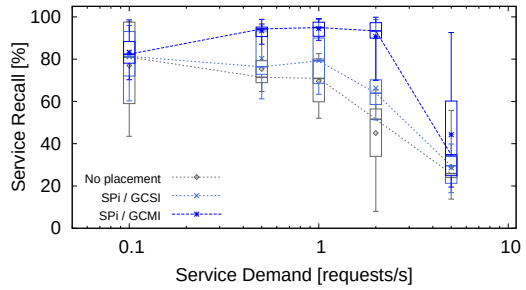


Figure 6.23: Placement activity on an IEEE 802.11 ad hoc network

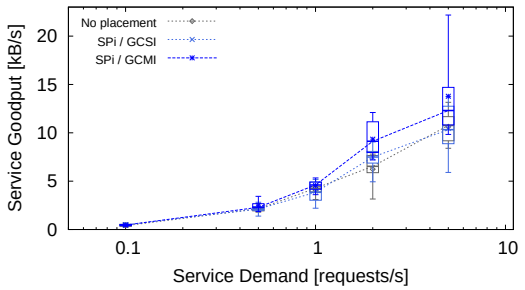
but fails to do so under very high load. Furthermore, we also note that SP*i* / GCSI, although not being able to adapt the number of service instances, manages to reduce the request distance when compared to the setup without placement.

Figure 6.23 shows the activity of the placement algorithm that led to these placement characteristics. Just like the number of service instance, the results shown in this diagram are similar to those of the emulation-based evaluation (cf. Fig. 6.19a). Like above, the results differ for scenarios with high volumes of service demand. For SP*i* / GCSI, the number of replications increases, which indicates that multiple unsuccessful replications attempts are made. For SP*i* / GCMI, the number of replications in these scenarios drops slightly as adaptations fail and time out. All of these observations are indications for the fact that the wireless channel of the ad hoc network is operating at its maximum capacity and many acts of communication fail due to packet collisions.

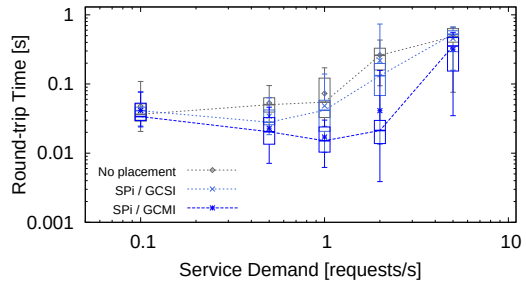
Finally, Figure 6.24 depicts the overall behavior of a real ad hoc network employing the different service placement algorithms. Given the sharp increase in network load for scenarios with high service demand in Figure 6.24d, the drop in service recall in Figure 6.24a, and the rise in round-trip time in Figure 6.24c, we can confirm our assumption that the wireless channel is effectively saturated at service demands above two service requests per second per client. More importantly, when looking at all scenarios together, we can once again see the same overall behavior of a network employing service placement: Placement of a centralized service using SP*i* / GCSI slightly improves the recall over the setup without active service placement, and adaptations of the configuration of a distributed service using SP*i* / GCMI outperforms both other approaches. Furthermore, we can also see how a network using SP*i* / GCMI is able to operate at a lower network load as service demand increases. This in turn results in slightly higher service goodput and significantly lower round-trip times.



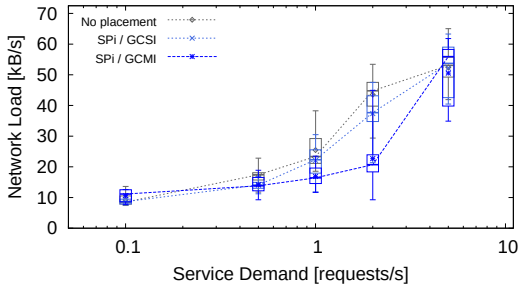
(a) Service recall vs. service demand



(b) Service goodput vs. service demand



(c) Round-trip time vs. service demand



(d) Network load vs. service demand

Figure 6.24: Placement effect on an IEEE 802.11 ad hoc network

Since these measurements were conducted on a real-world system without any simplifying assumption, it is also interesting to look at the exact figures of the results rather than just the general trends. Table 6.4 shows the mean values of all 50 measurements per setup for all four metrics that describe the overall performance of the network. We use the setup without service placement as base case and calculate in the relative improvement that SP*i* / GCSI and SP*i* / GCMI were able to achieve. As already established in our discussion of Figure 6.24, SP*i* / GCSI has only minor impact on the service goodput. However, it improves the service recall by 13.49% and reduces the rount-trip time of service requests by 13.29%, while at the same time causing 9.02% less network load. The overall impact achieved by SP*i* / GCMI placing a distributed service is far more significant: Service recall and goodput increase by 37.20% and 30.16% respectively, while round-trip time is reduced by 52.13%. These major improvements in the quality of the service come at no extra cost. Instead, the overall network load is reduced by 22.19%.

Furthermore, Figure 6.24 also shows how distributed service provisioning with SP*i* / GCMI can mitigate the problem of network congestion due to excessive load in scenarios in which centralized service provisioning begins to fail. This is particularly visible in the scenario with a service demand of two service requests per second. The medians of the ten measurements for this scenario are shown in Table 6.5. As we can see, SP*i* / GCSI already shows very

Table 6.4: Means of evaluation results for the IEEE 802.11 testbed (all scenarios)

	No placement	SP*i* / GCSI absolute	SP*i* / GCSI relative	SP*i* / GCMI absolute	SP*i* / GCMI relative
Service Recall [%]	59.21	67.20	**13.49%**	81.23	**37.20%**
Service Goodput [kB/s]	4.70	4.89	**4.06%**	6.12	**30.16%**
Round-trip Time [s]	0.1840	0.1596	**-13.29%**	0.0881	**-52.13%**
Network Load [kB/s]	29.53	26.87	**-9.02%**	22.98	**-22.19%**

Table 6.5: Medians of evaluation results for the IEEE 802.11 testbed (2 requests/s)

	No placement	SP*i* / GCSI absolute	SP*i* / GCSI relative	SP*i* / GCMI absolute	SP*i* / GCMI relative
Service Recall [%]	51.55	63.91	**23.97%**	93.38	**81.14%**
Service Goodput [kB/s]	6.53	7.49	**14.66%**	9.13	**39.72%**
Round-trip Time [s]	0.2601	0.1319	**-49.29%**	0.0213	**-91.82%**
Network Load [kB/s]	44.79	37.65	**-15.94%**	20.69	**-53.80%**

good improvements in service recall and goodput by 23.97% and 14.66%, while reducing round-trip times and network load by 49.29% and 15.94% respectively. With a distributed service, however, the network exhibits a fundamentally different behavior: SP*i* / GCMI achieves increases in service recall and goodput of 81.14% and 39.72% and a reduction of round-trip time by 91.82%. At the same time, the required network traffic drops by 53.80%. While certainly representing the optimal case for SP*i* / GCMI, these figures underline the fact that substantial improvements – that by far exceeds those of the average case – are possible in certain scenarios.

With this part of the evaluation, we have not only demonstrated that the quality of the implementation of the SP*i* framework and its algorithms is sufficient for them to operate under real-world conditions. We have also provided strong evidence to support our claim that service placement, and in particular the SP*i* / GCMI placement algorithm, has a significant, positive impact on the real-world performance of ad hoc networks. Our results show that major improvements in the quality of service provisioning can be achieved, while at the same time the cost, i.e., the network traffic, is reduced. Furthermore, we note that the positive impact of service placement tends to increase with the size of the network as the complexity of the network topology increases and the impact of a good placement of service instances grows (cf. Sec. 6.5). For this reason we expect the improvements that we have measured for a 36-node network to be equally applicable, if not more, to networks more participating nodes.

6.10 Summary

In this chapter, we have evaluated the SP*i* service placement framework as presented in Chapter 4 together with its two placement algorithms SP*i* / GCSI and SP*i* / GCMI as described in Chapter 5. We have compared these algorithms with several other algorithms proposed in the literature, both for centralized and for distributed services. In our comparisons, we have covered a wide variety of scenarios, including variable network sizes, volume and regional characteristics of service demand, synchronization requirements of the service, and impact of unreliable communication links. Depending on the specific part of the evaluation, we have employed both simulations using the `ns-2` network simulator and emulations using a wired IEEE 802.3 network. Furthermore, we have verified our findings with additional measurements using a wireless IEEE 802.11 ad hoc network.

First of all, this evaluation has demonstrated that we have reached our design goals with the SP*i* service placement framework. The framework has proven to be a suitable basis for the implementation of several placement algorithms, both centralized and distributed. All of these algorithms were shown to operate according to expectation. Furthermore, we

have shown that the framework is capable of efficiently bridging the gap between different evaluation methods, as we were able to perform simulations, emulations as well as real-world experiments using the same code base.

Furthermore, our results have shown that the placement of a centralized service with SP*i* / GCSI or other algorithms leads to a more predictable behavior of the ad hoc network when compared to a traditional client/server architecture without service placement. Minor improvements in the overall quality of the service as perceived by the clients, in particular with regard to service recall and access times, are also possible.

The real advantages of employing service placement in an ad hoc network, however, manifest themselves if service placement is used in conjunction with a distributed service, i.e., if the placement algorithm is allowed to control the number of service instances as well as their location. In this area, our SP*i* / GCMI placement algorithm has proven to be the most versatile algorithm, capable of establishing superior service configurations than any other algorithm. In part this can be attributed to the fact that SP*i* / GCMI pursues a different architectural approach to the placement of distributed services by employing a dynamically assigned coordinator node that centrally makes placement decisions. As a result, SP*i* / GCMI outperformed the distributed placement algorithms in our evaluation. While not a definitive proof, this may be regarded as an indication that service placement should be tackled with centralized rather than distributed algorithms.

The most interesting result of this evaluation is the qualitatively different behavior of an ad hoc network employing a distributed service with service placement, e.g., as implemented in SP*i* / GCMI, when compared with a traditional client/server architecture without active service placement. Our results have shown that a network running with SP*i* / GCMI significantly improves the quality of the service as perceived by the clients, i.e., the service recall, goodput, and access time, while actually incurring *less* network traffic. In other words, we have shown that service placement, as implemented in SP*i*, reduces the overall cost of service provisioning while at the same time improving the quality of the service. Since we have demonstrated the correct operation of our system on a real-world ad hoc network, we argue that even the increase in system complexity is justified for wide-spread services like DNS, FTP, or WWW. Therefore, we can conclude that whenever a service is deployed in an ad hoc network, this deployment should *not* take the form of a traditional client/server architecture, but should rather follow the approach of a distributed service employing active service placement, e.g., with SP*i* / GCMI.

Chapter 7

Conclusion

In this work, we have presented the service placement problem in ad hoc networks and motivated that a purpose-built service placement system should be employed to actively control which nodes in the network are to provide services to other nodes. We gave an introduction to the background of service placement in the fields of ad hoc networking and facility location theory, and reviewed current proposals that address the service placement problem. As our contribution in this field, we have proposed the SP*i* service placement framework that for the first time allows side-by-side comparisons between different approaches to service placement across several evaluation platforms. We have also proposed the Graph Cost / Single Instance and Graph Cost / Multiple Instances placement algorithms that build upon this framework and support the placement of both centralized and distributed services.

We conducted extensive evaluations, employing simulations, emulations, and real-world experiments, and were able to demonstrate that our placement algorithms, in particular the Graph Cost / Multiple Instances algorithm, outperform other approaches in a variety of scenarios. When compared to the performance of a service implemented in the form of a traditional client/server architecture, i.e., without active service placement, our approach significantly improves the quality with which the service is provided to clients while at the same time reducing the bandwidth required to do so.

7.1 Contributions

This work comprises four major contributions in the fields of ad hoc networking and service placement. On the methodological side, we have developed the following two technical approaches:

- We propose a **light-weight programming abstraction** to support research into ad hoc networking. In combination with a set of coding conventions, it allows to conduct experiments with the same code base across a variety of evaluation platforms, specifically including network simulators such as `ns-2`. This approach enables the

rapid prototyping of network protocols and, more generally, software components for distributed systems. Furthermore, it eliminates the overhead of moving between evaluation platforms thereby allowing the researcher to pick the most suitable tool for each part of an evaluation.

Our contribution improves upon the state of the art through the simplicity of its programming abstraction. This enables us to support a wider range of evaluation platforms than other approaches, currently including major operating systems (Microsoft Windows and Linux), the `ns-2` network simulator, and the ScatterWeb platform for wireless sensor networks. Furthermore, our approach features a very low run time overhead when executing real-world code within a simulator.

- We also propose the **SP*i* service placement framework** as a tool that implements the fundamental functionality required for supporting service placement in ad hoc networks. This functionality comprises the monitoring of the service demand to aggregate usage statistics, the monitoring of the network topology to build a network map, and various ways for adapting the service configuration. The framework also provides a fine grained API against which a wide variety of service placement algorithms can be implemented. Currently, we have implemented eight placement algorithms on top of the SP*i* framework. These placement algorithms account for all major architectural possibilities for implementing service placement in ad hoc networks.

  With the SP*i* framework, we improve upon the state of the art by enabling researchers to conduct meaningful side-by-side comparison of placement algorithms. We also lessen the burden of developing and evaluating new placement algorithms since our framework supports various evaluation tools and provides a well-tested API to build upon.

These two contributions have in common that they improve upon the current tools and methods of the research community. The following two contributions are more focused and directly address the service placement problem in ad hoc networks.

- Our **Graph Cost / Single Instance (GCSI) and Graph Cost / Multiple Instances (GCMI) service placement algorithms** operate by collecting usage statistics about the service and information about the network topology. Using this information they adapt the configuration of the service. GCSI places migrates the single instance of a centralized service, while GCMI adapts the number and location of the service instances of a distributed service.

  Motivated by the insight that the adaptation of a service configuration is a very costly operation in an ad hoc network, these two algorithms, in particular GCMI, take a

novel architectural approach by implementing a centralized placement algorithm. This allows for an unprecedented level of control that encompasses not only the location of service instance but also their number and the optimal timing of adaptations of the service configuration.

- In our **quantitative evaluation of service placement algorithms**, we not only verify the correct operation of the SP*i* framework but, more importantly, evaluate our and several other placement algorithms with regard to their capabilities for improving the overall performance of ad hoc networks. Our evaluations cover scalability with regard to both network size and volume of service demand, handling of different regional demand patterns, and resilience against poor link quality. Results obtained used simulations and emulations are verified by the means of real-world experiment on an IEEE 802.11 wireless testbed.

  Our results show that the GCSI and GCMI service placement algorithms match or improve other approaches to service placement. Furthermore, our results show that employing a distributed service with active service placement outperforms traditional client/server architectures. From this we conclude that service placement – as implemented in the SP*i* framework and the GCSI and GCMI algorithms – is highly viable as an architectural option for service provisioning in ad hoc networks.

Given these advantages of an architecture built around the principle of service placement, we note that a service placement system relies on certain capabilities being present in lower-level components for efficient operation (cf. Sec. 4.4.4). These capabilities are, in particular:

- **Pro-active route seeding:** Once a new service instance has been created, it is advantageous for it to be able to seed the routing tables of the nodes in its vicinity with the required information for reaching it. This reduces the overhead compared to the alternative of each client nodes establishing a route independently. In order to support this process, reactive routing protocols should offer a mechanism for actively seeding routing information.

- **Topology monitoring:** The majority of placement algorithms makes use of information about the topology of the ad hoc network. If this information is available in the routing component, either in the form of neighborhood lists, routing tables, or (parts of) the network graph, an interface should be provided for other components to access this information.

- **Pro-active service announcements:** Instead of waiting for client nodes to locate newly created service instances, the new service host should be able to proactively

announcing the availability of a new service instance, thereby reducing the overhead of a reconfiguration following an adaptation. Therefore, designers of service discovery protocols for ad hoc networks should consider adding the possibility to issue host-initiated service announcements.

When designing these components and protocols in the future, and if they are likely to be employed in a network with active service placement, then the inclusion of these capabilities will lead to an overall improvement of network performance.

7.2 Future Work

The SP*i* service placement system presented in this work addresses the major concerns of placing a service in an ad hoc network, i.e., the calculation of an optimal service configuration, adaptation mechanisms, and the integration with other components. Given that the system has proven itself in a variety of experiments, it can be expected to serve as a strong basis to investigate the questions listed in this section.

7.2.1 Extensions of the Architecture

While addressing the placement of centralized and distributed, monolithic services, the SP*i* framework in its current form does not cover the placement of composite services (cf. Sec. 1.1.3). The placement of composite service is more complex than the placement of monolithic services since the number and location of instances for each subservice needs to be adapted. This requires more sophisticated placement algorithms which include the flow of information between instances of subservices into their model of the network. In order to achieve this, one could either employ a formal specification of the interaction between subservices or learn how the subservices interact with each other through observation at run time. Since none of these alternatives is particularly light-weight, it is our opinion that an in-depth investigation of these questions should be postponed until service decomposition in general has proven to be a promising approach in the field of ad hoc networking.

The placement of multiple monolithic services, however, seems like a much more worthwhile focus for future research. In this work, we have treated each service as if it was the only service on the network. Since the placement of service instances influences the regional load characteristics of the network, this assumption may lead to suboptimal performance if multiple services are being placed without coordination or even knowledge about each other. As a consequence, our formulation of the service placement problem needs to be adapted to reflect the fact that multiple services, each with multiple instances, are to be placed. A possible way to tackle this problem would be by extending the SP*i* / GCMI placement

algorithm to take node properties into account when making placement decisions. These properties may include information such as whether a node is already hosting instances of other services or how much unrelated traffic is currently being forwarded through the node.

Furthermore, as we have pointed out in Section 5.2.1, our current system implements a very basic placement policy. Support for more advanced placement policies would allow the service placement system to tune its placement decisions to the specific requirements of the network. Wang and Li [105] have proposed a special-purpose placement policy for mobile and vehicular ad hoc networks that deals with group-based node mobility. It would be interesting to implement and evaluate this placement policy using the SP*i* framework. Similarly, deployments of wireless sensor networks are usually highly application-specific and, as such, interesting candidates for special-purpose service placement policies.

7.2.2 Refinements of the Implementation

When presenting the SP*i* framework and its placement algorithms, we left a few optimizations and refinements of the system to future work since we do not expect them to have an significant impact on the results that we have discussed in this work. However, this does not imply that slight improvements are not possible. Hence, we propose to revisit our implementation of the topology monitoring functionality (cf. Sec. 4.3.1) in order to systematically reevaluate the accuracy and the timeliness of the topology information in light of changing network conditions.

Focusing more on the engineering aspects of distributed service provisioning, it would certainly be interesting to extend the implementation of a widely used client/server package, e.g., the Apache HTTP server [5], to operate in a distributed fashion and to support replication and migration of service instances. The insights gained while implementing the required changes can be expected to lead to a programming interface for the service placement system to interact with the service, e.g., in order to signal an imminent change in the service configuration. Further, this would also allow us to conduct evaluations on how to efficiently implement the synchronization between service instances (cf. Sec. 5.5.2). A possible starting point for this may be the synchronization mechanism proposed by Herrmann [49, p. 183ff.].

7.2.3 Security

Distributed service provisioning is obviously problematic from a security perspective. Instead of only having to trust a single central instance, each client essentially has to extend its trust to all nodes of the ad hoc network since each node is a potential service host. We have not addressed this significant aspect of the service placement problem in our current

work since our goal is to gain a general insight into the impact of service placement on the performance of an ad hoc network. We are, nevertheless, aware of the fact that the security aspects need to be solved before service placement is to gain any kind of acceptance.

Fundamentally, we see two architectural options for establishing trust between the clients and the dynamically allocated service instances: The first option is to establish a hierarchy between service instances, with one master instance remaining on its initial, trustworthy host. It would be up to this master instance to issue temporary certificates to other instances and to control whether service requests are being processed correctly. In this option, the host of the master instance would also be the natural choice to run the placement algorithm. Alternatively, one could envision a system that employs distributed consistency checks between all service instances and uses majority voting to identify malicious service hosts. However, just like with distributed approaches to placement algorithms, special attention would have to be paid to the network overhead incurred by these security checks since they may outweigh the utility of employing a service placement system in the first place.

7.3 Concluding Remarks

This work has shown that distributed service provisioning with active placement of service instances has superior scalability properties than a traditional client/server architecture. With the SP*i* service placement framework and the Graph Cost / Single Instance and Graph Cost / Multiple Instances placement algorithms we have made an important step towards real-world services benefiting from this advantage.

Distributed services with active service placement represents an interesting architectural alternative to service provisioning in general. Looking at the big architectural alternatives in this field, we observe that systems that employ a client/server architecture run into scalability problems in ad hoc networks. The bottleneck in this case is the available bandwidth at the node that hosts the single service instance. Peer-to-Peer (P2P) networks, on the other hand, require a radically different programming model to implement services.

Service provisioning using distributed services and active service placement offers a new alternative between these two extremes. It is more flexible and more scalable than a traditional client/server architecture, yet its programming model still resembles that of a classical server rather than that of a P2P network. As such, this architecture with its combination of scalability and simple programming model may in fact represent a viable alternative between the two established architectures.

Appendix A

Evaluation of the DYMO Routing Protocol

As pointed out in Section 4.3.1, we have chosen the Dynamic MANET On-demand (DYMO) routing protocol as described in [17] as core for the implementation of the routing component of the SP*i* service placement framework. The correct operation of the SP*i* framework, and therefore also the validity of the evaluation presented in Chapter 6, depends largely on the correctness of our implementation of DYMO. We have thus compared our implementation of DYMO with other implementations of routing protocols that are available for the `ns-2` network simulator. The goal of this comparison is to ensure that our implementation of DYMO is functionally complete and performs reasonably well.

In this chapter, we briefly discuss the exact configuration of our implementation of DYMO within the SP*i* framework. We then present results from simulations that compare DYMO / SP*i* to three other implementations of reactive routing protocols: the native implementation of the Ad hoc On-Demand Distance Vector (AODV) routing protocol [87] that is bundled with `ns-2` [84], the implementation of AODV from Uppsala University (AODV-UU) [4], and the implementation of DYMO from the University of Murcia (DYMO-UM) [29].

A.1 Setup

We configured our implementation of the DYMO routing protocol using the values in the rightmost column of Table A.1. We established these values based on the standard documents for AODV [87] and DYMO [17] as well as through inspection of implementations of these protocols [4, 29].

This process was inconclusive for some parameters. In these cases, the rationale for the parameter value that we used is as follows:

- The value for the maximal hop limit (`DYMO_MSG_HOPLIMIT`) of AODV is more than double that of DYMO. We use a value of 20, which is sufficient to reach all nodes in all topologies under consideration.

Table A.1: Parameters of routing protocols according to standard documents and implementations

Parameter	AODV RFC [87]	AODV ns-2 [84]	AODV -UU [4]	DYMO RFC [17]	DYMO -UM [29]	DYMO SP*i*
DYMO_MSG_HOPLIMIT	35	30	35	10	10	**20**
DYMO_ROUTE_TIMEOUT [s]	n/a	n/a	n/a	5	3	**5**
DYMO_ROUTE_AGE_MIN_TIMEOUT [s]	n/a	n/a	n/a	1	n/a	**1**
DYMO_ROUTE_SEQNUM_AGE_ MAX_TIMEOUT [s]	n/a	n/a	n/a	60	n/a	**300**
DYMO_ROUTE_USED_TIMEOUT [s]	$5 \cdot 3$	n/a	10	5^*	3	$\mathbf{5}^*$
DYMO_ROUTE_DELETE_TIMEOUT [s]	$3^\dagger$	10	$10^\dagger$	$2 \cdot 5^*$	$5 \cdot 5^*$	$\mathbf{5 \cdot 5}^*$
DYMO_ROUTE_RREQ_WAIT_TIME [s]	$2 \cdot 0.04 \cdot 35$	10	$2 \cdot 0.04 \cdot (\mathrm{ttl}^\ddagger + 2)$	2	1	$\mathbf{2 \cdot 2 \cdot 0.04 \cdot 20}$
DYMO_RREQ_TRIES	2	3	2	3	3	**3**
AODV_NET_DIAMETER [hops]	35	30	35	n/a	10	**20**
AODV_NODE_TRAVERSAL_TIME [s]	0.04	0.03	0.04	n/a	n/a	**0.04**
AODV_NET_TRAVERSAL_TIME¶ [s]	$2 \cdot 0.04 \cdot 35$	10	$2 \cdot 0.04 \cdot 35$	n/a	n/a	$\mathbf{2 \cdot 0.04 \cdot 20}$
AODV_TIMEOUT_BUFFER	2	n/a	2	n/a	n/a	**2**
AODV_TTL_START	1	5	1	n/a	n/a	**1**
AODV_TTL_INCREMENT	2	2	2	n/a	n/a	**2**
AODV_TTL_THRESHOLD	7	7	7	n/a	n/a	**7**
PACKET_BUFFER_SIZE [packets]	n/a	64	512	n/a	512	**512**

* Corresponds to the value of DYMO_ROUTE_TIMEOUT. † The value is doubled for RREP packets. ‡ Corresponds to the TTL value of the RREQ packet.
¶ Corresponds to $2 \cdot$ AODV_NODE_TRAVERSAL_TIME $\cdot$ AODV_NET_DIAMETER.

- For the sequence number timeout (`DYMO_ROUTE_SEQNUM_AGE_MAX_TIMEOUT`), we found that the default value leads to bad performance under light load. While DYMO-UM does not implement this timeout at all, we decided to increase the value in order to achieve a reasonably good performance. For similar reasons, we disabled the timeout for unused routes (`DYMO_ROUTE_USED_TIMEOUT`) when running experiments on the IEEE 802.11 testbed.

- In setting the timeout for `ROUTE REQUEST` packets (`DYMO_ROUTE_RREQ_WAIT_TIME`), we followed the setting of AODV and made it dependent on the size of the network, rather than a constant as suggested by other DYMO sources. However, instead of using the value for network traversal directly, we doubled this value to take both the transmissions of `ROUTE REQUEST` and `ROUTE REPLY` into account.

A.2 Evaluation

We have conducted this evaluation using the `ns-2` network simulator in a similar setup to that described in Section 6.3.1. 100 simulated nodes were placed in a fixed 10-by-10 grid layout. Like in the previous simulations, the distance between the nodes was adjusted in such a way that each node could communicate with its direct neighbors, horizontally, vertically, and diagonally. We performed 30 measurements for each data point and plot median, mean, minimum, maximum, and first and third interquartile. Where applicable, other parameters were set to match the simulations described in Sections 6.4 and 6.5.

A.2.1 Multiple Sources / Single Destination

In the first experiment, we configured all nodes to transmit packets with a payload of 1024 bytes at varying time intervals to one centrally located destination node. The frequency with which packets are sent is varied in the range between 0.1 and 5 packets per second for each node. The nodes are started over a time interval of one minute and then continue to transmit for 15 minutes. We measured the Packet Delivery Ratio (PDR), i.e., the ratio of received to transmitted packets, and the network load at the MAC layer in terms of packets and data volume.

In Figure A.1, we plot the PDR against the offered load. We can see that the PDR decreases for all routing protocols, which can be attributed to network congestion and corresponds to the expected behavior. While DYMO-UM has problems at low data rates due to timeouts, our implementation of DYMO performs just as well as the two implementations of AODV.

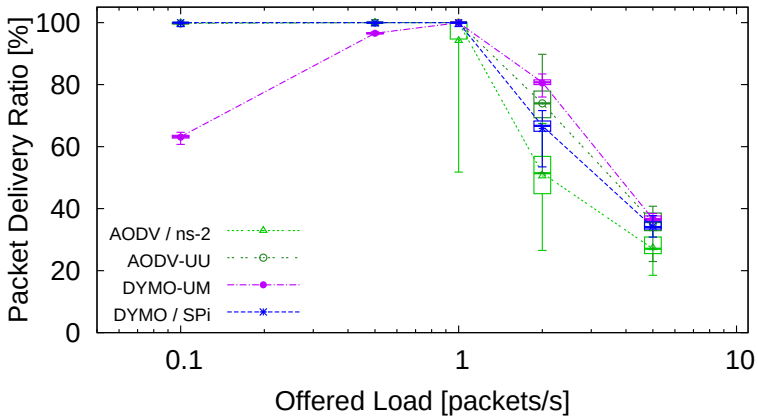
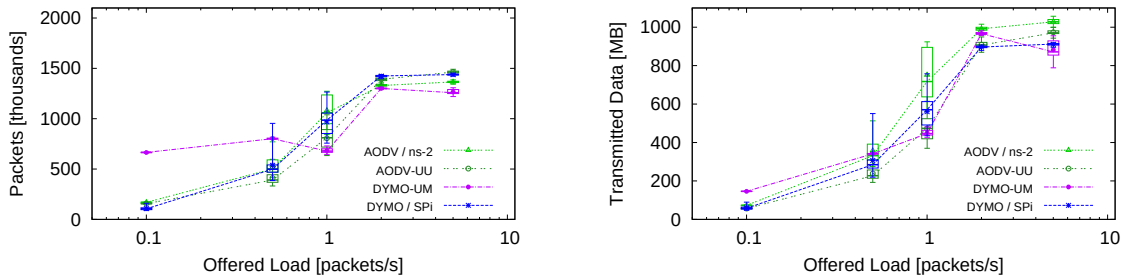


Figure A.1: Network performance for routing to a single destination (packet delivery ratio vs. offered load)



(a) Unicast packets vs. offered load

(b) Transmitted data vs. offered load

Figure A.2: Network load for routing to a single destination

Figure A.2 examines the network load, both in terms of transmitted packets and overall volume of traffic. We observe that except for the misbehavior of DYMO-UM under light load, all routing protocols perform similarly well.

A.2.2 Multiple Sources / Multiple Destinations

For a second experiment, we changed the setup to transmit each packet to a randomly chosen node instead of to a fixed destination. All other parameters remained identical.

In this scenario, we observe some more surprising results. As shown in Figure A.3, our implementation of DYMO is able to support a high PDR under heavier load than all other protocols. While we have in fact spent considerable time debugging our implementation, we still find it surprising that apparently we are able to outperform the other implementations.

Looking at the total volume of network traffic in Figure A.4b, we can see that our implementation reaches the point of saturation as far as the volume of network traffic is

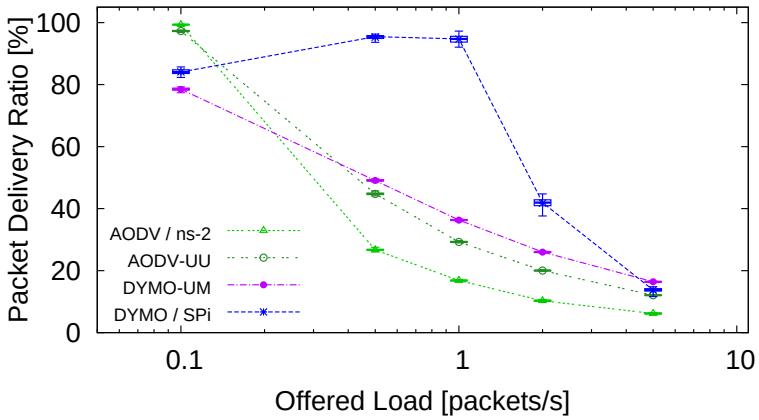
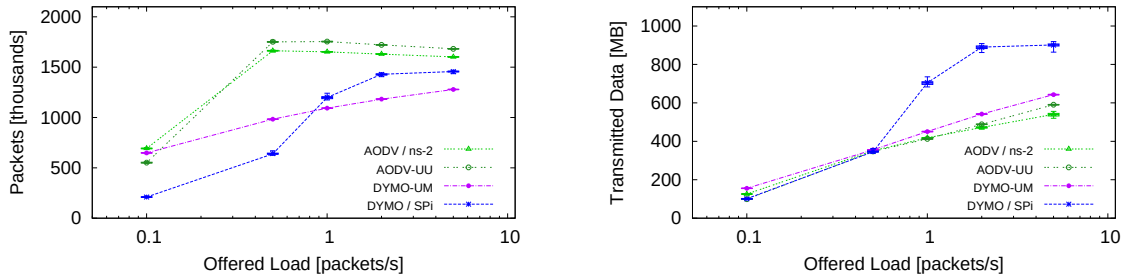


Figure A.3: Network performance for routing to multiple destinations (packet delivery ratio vs. offered load)



(a) Unicast packets vs. offered load

(b) Transmitted data vs. offered load

Figure A.4: Network load for routing to multiple destinations

concerned (cf. Fig. A.2b), while the other implementations do not. In other words, these implementations drop packets although bandwidth for packet transmissions is still available. Interestingly, this correlates with the number of transmitted packets reaching a maximum in Figure A.4a. We take this as indication that some kind of traffic shaping is in place to limit the undoubtedly high number of `ROUTE REQUEST` packets. Since our implementation is not affected by these performance problems, we omit a more detailed investigation.

In this evaluation, we have seen that the performance of our implementation of DYMO either matches or surpasses that of other implementations of the AODV and DYMO routing protocols for `ns-2`. In fact, we have seen that in some scenarios our implementation of DYMO performs significantly better than the other implementations. We did not investigate the differences in performance in more detail. For the purpose of validating the implementation of DYMO as used in the SP*i* service placement framework, it is sufficient that we can conclude that our implementation works correctly and performs reasonably well.

Appendix B

Evaluation of Current Approaches to Service Placement

In this chapter, we provide the results of a detailed comparison of six approaches to service placement proposed in the literature. We omitted these results from the evaluation in Chapter 6 to improve its readability, and merely pointed out that the Tree-topology Migration and the ASG / Event Flow Tree (EFT) placement algorithms represent the state of the art in the placement of centralized and distributed services respectively. With the additional results presented in this chapter, we back up this statement.

B.1 Placement of Centralized Services

Like in Chapter 6, we begin with a look at placement algorithms that deal with centralized services. We cover the following four algorithms (cf. Sec. 6.4):

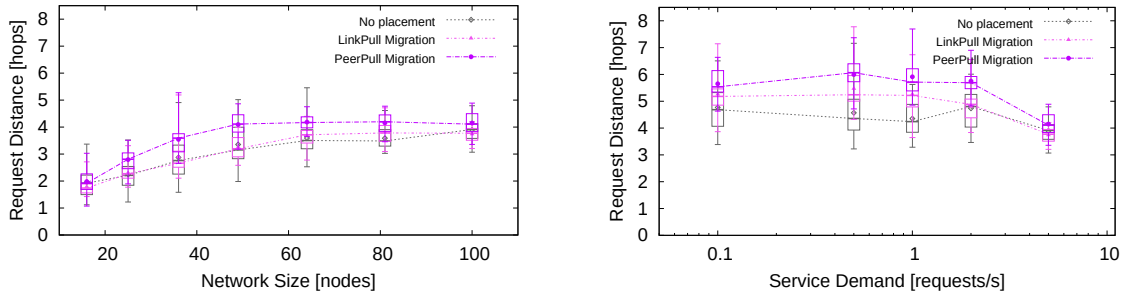
LinkPull Migration – This placement algorithm migrates the service instance to the neighboring node over which most service traffic was forwarded at the end of each epoch [68].

PeerPull Migration – This placement algorithm migrates the service instance to the client node from which most service traffic originated at the end of each epoch [68].

TopoCenter(1) Migration – This placement algorithm migrates the service instance, at the end of each epoch, to the node that minimizes the sum of the migration traffic and the estimated future traffic required for service provisioning [68].

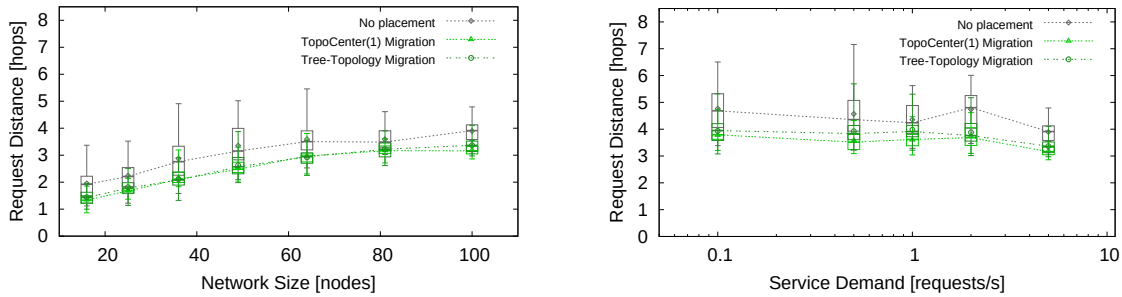
Tree-topology Migration – This placement algorithm migrates the service instance to the neighboring node over which more than half of the service requests are being forwarded, if any [85].

For readability of the plots, we subdivide these four algorithms into *node-centric* and *topology-aware* algorithms when plotting the measurement results. Node-centric placement



(a) Request distance vs. network size (b) Request distance vs. service demand

Figure B.1: Placement characteristics for centralized services (node-centric algorithms)

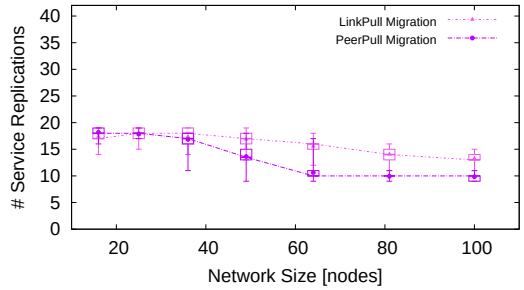


(a) Request distance vs. network size (b) Request distance vs. service demand

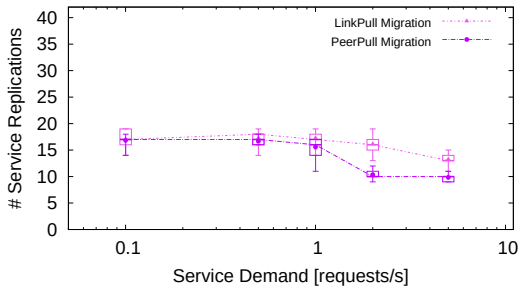
Figure B.2: Placement characteristics for centralized services (topology-aware algorithms)

algorithms base their placement decision on usage statistics of individual nodes. Topology-aware algorithms consider topological adjacent groups of nodes, either explicitly by the means of constructing a graph representation of the network or through implicit assumptions that are embedded in the algorithm logic. For comparison, we also plot the results of a network without active service placement. The details of the setup of these simulations correspond exactly to those used in the simulations presented in Section 6.4.

Like in our previous evaluation, we begin with a look at the average distance between clients and the single service instance over varying network sizes and volumes of service demand. In Figures B.1 and B.2, we can observe that this distance increases for larger networks and remains constant over varying service demand. In large networks and under high demand, this increase levels off while the constant value drops slightly. Both effects are due to the saturation of the wireless communication channel (cf. Figs. B.5g, B.5h, B.6g, and B.6h) and the resulting loss of packets that are forwarded over routes with large hop counts. This corresponds to the expected behavior of the system. More interesting, however, is the fact that the topology-aware placement algorithms manage to reduce the distance between clients and service instances for all scenarios, while the same value uniformly increases

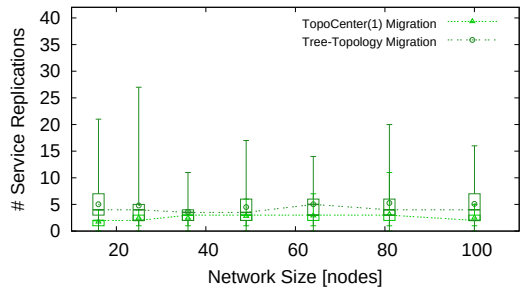


(a) Number of replications vs. network size

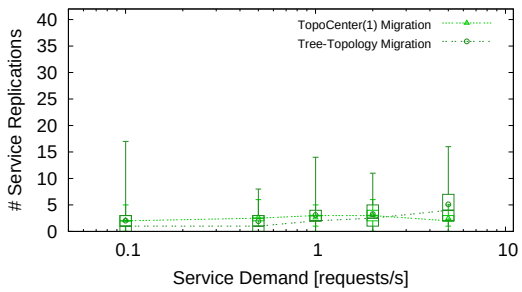


(b) Number of replications vs. service demand

Figure B.3: Placement activity for centralized services (node-centric algorithms)



(a) Number of replications vs. network size

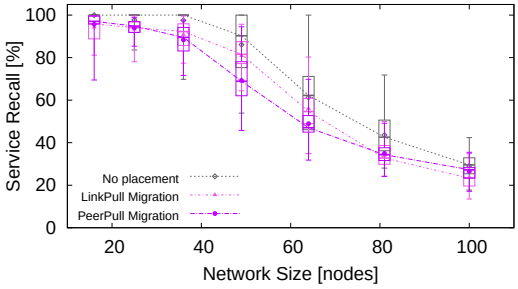


(b) Number of replications vs. service demand

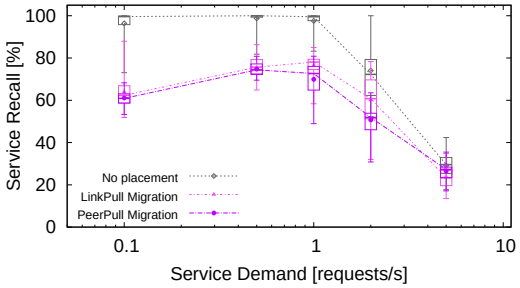
Figure B.4: Placement activity for centralized services (topology-aware algorithms)

for node-centric algorithms. This behavior is surprising, since it indicates that the two node-centric placement algorithms actually perform worse than a network without service placement.

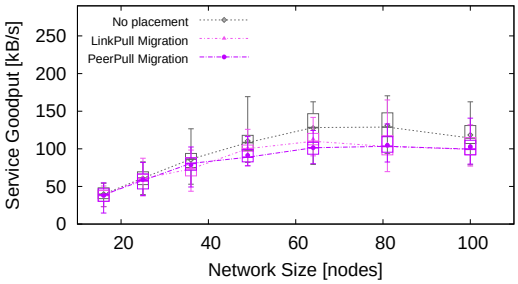
This observation can be explained by evaluating the placement activity shown in Figures B.3 and B.4. The number of replications for node-centric algorithms is consistently higher than that of topology-aware algorithms and only drops for large networks and high volumes of service demand, i.e., when packet transmissions begin to fail. The PeerPull Migration algorithm is more susceptible to this problem as it supports migrations to nodes that may be further away. In scenarios in which the wireless communication channel is not saturated, we observe that both algorithms perform between 15 and 20 migrations during each simulation run. Since the duration of an epoch within these algorithms is set to one minute, this corresponds to one migration per epoch. In other words, the two node-centric algorithms fail to converge. This can be explained by all nodes exhibiting the same level of service demand and thus node-centric placement decisions being bound to fluctuate in their results. Adding appropriate hystereses to the placement logic of the LinkPull and PeerPull Migration algorithms is likely to fix this problem. In contrast, the topology-aware



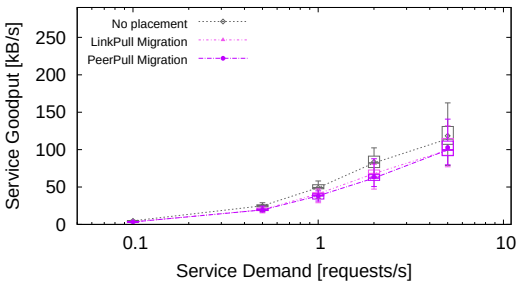
(a) Service recall vs. network size



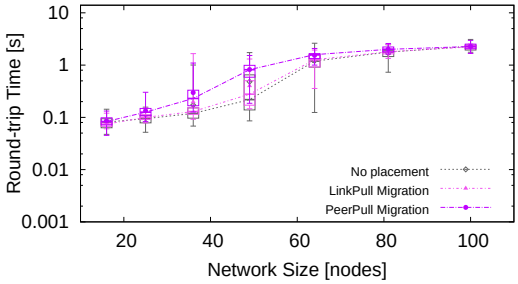
(b) Service recall vs. service demand



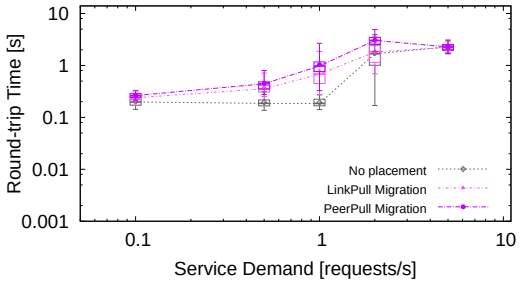
(c) Service goodput vs. network size



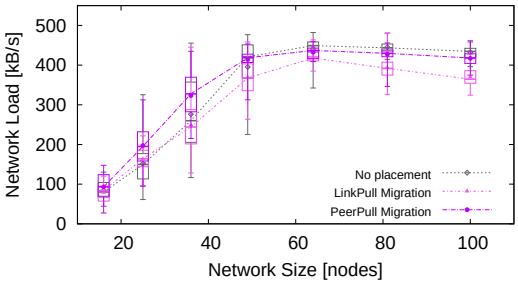
(d) Service goodput vs. service demand



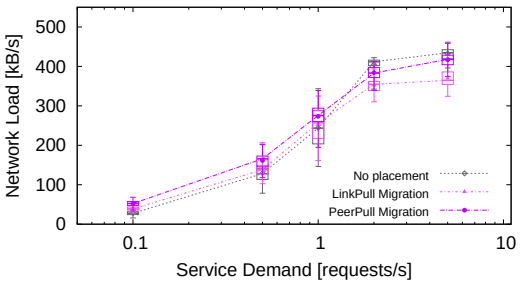
(e) Round-trip time vs. network size



(f) Round-trip time vs. service demand

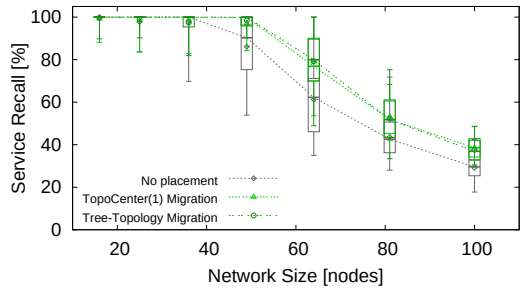


(g) Network load vs. network size

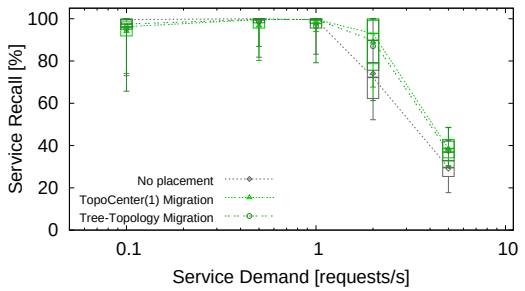


(h) Network load vs. service demand

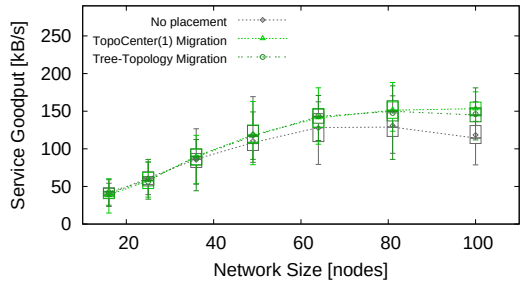
Figure B.5: Placement effect for centralized services (node-centric algorithms)



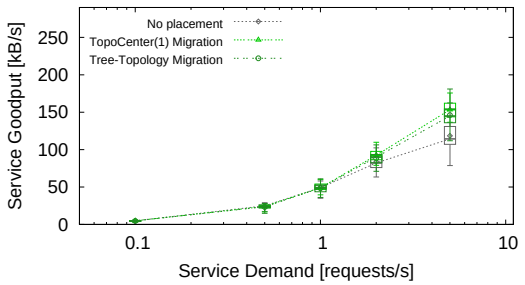
(a) Service recall vs. network size



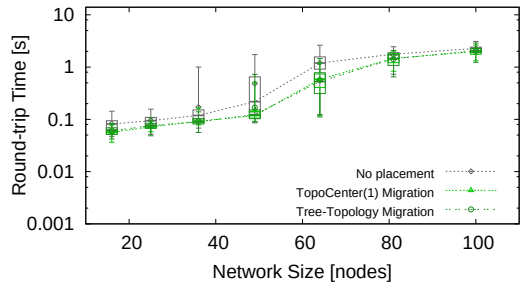
(b) Service recall vs. service demand



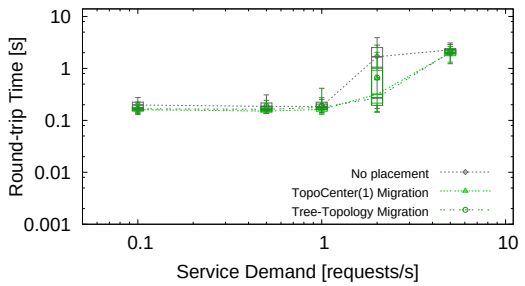
(c) Service goodput vs. network size



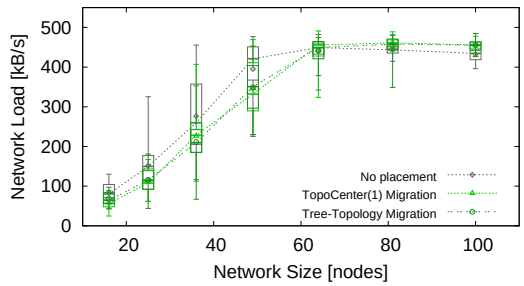
(d) Service goodput vs. service demand



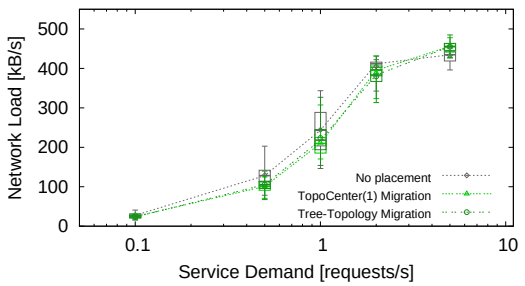
(e) Round-trip time vs. network size



(f) Round-trip time vs. service demand



(g) Network load vs. network size



(h) Network load vs. service demand

Figure B.6: Placement effect for centralized services (topology-aware algorithms)

algorithms do not suffer from the same systematic problem. It is, however, worth pointing out that the Tree-topology Migration algorithm regularly exhibits outliers in the number of replications which hint at it also having convergence problems in certain scenarios.

Figures B.5 and B.6 illustrate the performance of ad hoc networks employing these four placement algorithms. Overall, there are only minor differences to the reference network without service placement. As we have come to expect from the discussion of Figures B.1 and B.2, networks employing one of the two node-centric placement algorithms perform slightly worse than networks without service placement, while networks with topology-aware algorithms perform slightly better across all metrics. Noteworthy is the comparatively low service recall of node-centric algorithms which is caused by the ongoing migrations initiated by these algorithms. When comparing the two topology-aware algorithms, we find that their performance is almost identical.

One obvious conclusion from this evaluation is that the use of hystereses really is a must for placement algorithms built upon simple metrics such as per-node service demand. More interesting, however, is the conclusion that both the TopoCenter(1) Migration and the Tree-topology Migration algorithms equally represent the state of the art for the placement of a centralized service, at least as far as performance is concerned. Still, we consider the Tree-topology Migration algorithm to be superior to the TopoCenter(1) Migration algorithm because it reaches the same level of performance while relying on less information about the network. More precisely, TopoCenter(1) Migration needs to maintain neighborhood lists of all client nodes, while Tree-topology Migration merely relies on the assumptions about the network topology encoded into its placement logic. Furthermore, Tree-topology Migration allows for a dynamic timing of placement decisions while TopoCenter(1) Migration makes use of fixed-length epochs. Due to these two reasons, we consider Tree-topology Migration to be superior to TopoCenter(1) Migration and use it as a representative for placement algorithms for centralized services when evaluating our own Graph Cost / Single Instance algorithm in Section 6.4.

B.2 Placement of Distributed Services

We now proceed to the evaluation of placement algorithm for distributed services. As already motivated in Section 6.5, we chose the two placement algorithms proposed as part of the ASG framework [49] as potential candidates due to the similarity of their focus to our own work and the complementary architectural decisions. These two algorithms are (cf. Sec. 6.5):

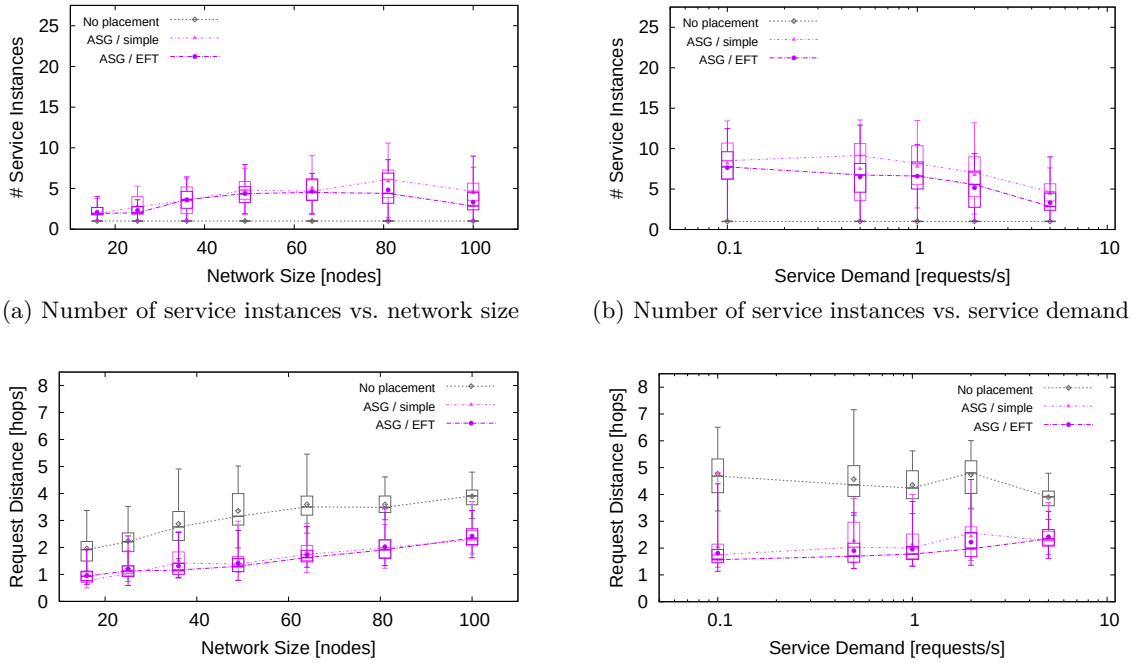
ASG / simple – This placement algorithm migrates and replicates service instances to neighboring nodes according to a fixed set of rules [49]. Migrations are triggered if more service requests are received from one neighbor than from all other neighbors and the current service host together. Replications are triggered if the service requests that have been forwarded by a migration target have traveled more than a preconfigured number of hops. Service instances are shut down if the service demand they serve falls below a threshold.

ASG / EFT – This placement algorithm migrates and replicates service instances to optimal nodes in a tree-like structure with the current service host as root following similar rules as those implemented in ASG / simple [49]. The main difference when compared to ASG / simple is that ASG / EFT explicitly considers distant nodes as potential targets for replications and migrations while ASG / simple only considers direct neighbors.

For comparison, we once again include the results for a network without service placements in the plots. The details of the experimental setup match those described in Section 6.5.

The placement characteristics of the two ASG algorithms are depicted in Figure B.7. Both algorithms show nearly identical behavior. The number of service instances remains low for small networks, since no client is sufficiently distant to trigger a replication. For larger networks, the number first increases and then drops slightly as service requests from distant nodes are increasingly lost due to network congestion (cf. Fig. B.10g). For the same reason, the number of service instances remains constant and then decreases steadily under increasing service demand. This may hint at a weakness of the replication rule, which only triggers if service requests are received from distant nodes. However, request packets from these nodes are the most likely to be lost due to packet collisions since they are forwarded over the most hops. As depicted in Figures B.7c and B.7d, the overall replication behavior of the two placement algorithms results in a slow increase in the distance between clients and available service instances for larger networks and in scenarios with higher volumes of service demand.

In Figure B.8, we plot the request distance against the number of service instances. This serves as an indication of the overall quality of the placement algorithm since one would



(a) Number of service instances vs. network size

(b) Number of service instances vs. service demand

(c) Request distance vs. network size

(d) Request distance vs. service demand

Figure B.7: Placement characteristics for distributed services (ASG algorithms)

expect that additional service instances result in a decrease in the distance between clients and their nearest service instance. This is, however, not the case for the two ASG algorithms as additional service instances are only deployed in scenarios which exhibit larger request distances. Once again, there is little difference between the two algorithms.

Looking at the number of replications in Figure B.9, we observe for the first time a major difference between ASG / simple and ASG / EFT. Across all scenarios, ASG / EFT utilizes less replications than ASG / simple. In fact, ASG / EFT is designed to work around the limitation of ASG / simple which is only capable of replicating and migrating service instances to immediate neighbors. The results illustrate that ASG / EFT successfully manages to find more distant target nodes for replications and migrations. Other than that, the two algorithms exhibit a placement activity that matches the number of service instances that they create (cf. Figs. B.7a and B.7b).

In the performance evaluation of networks employing both ASG placement algorithms as shown in Figure B.10, the two placement algorithms result in a very similar performance of the ad hoc network. Both algorithms enable larger networks to sustain high levels of service recall and goodput even under increasing service demand. They also reduce the round-trip time of service requests, which remains largely constant until the wireless communication channel becomes saturated. At the same time, the network load observed in networks that

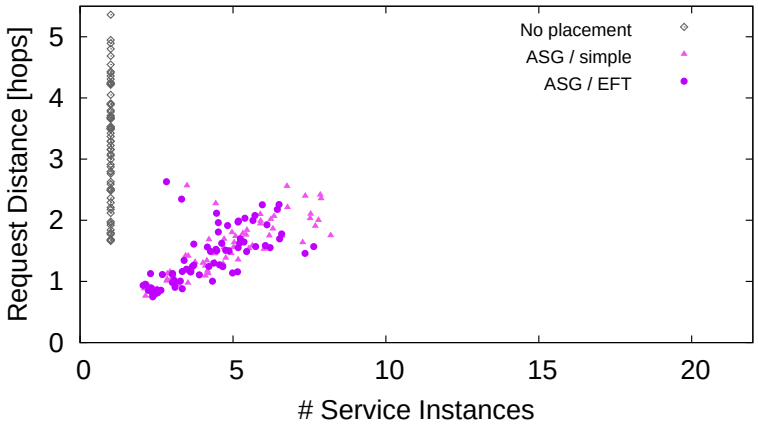
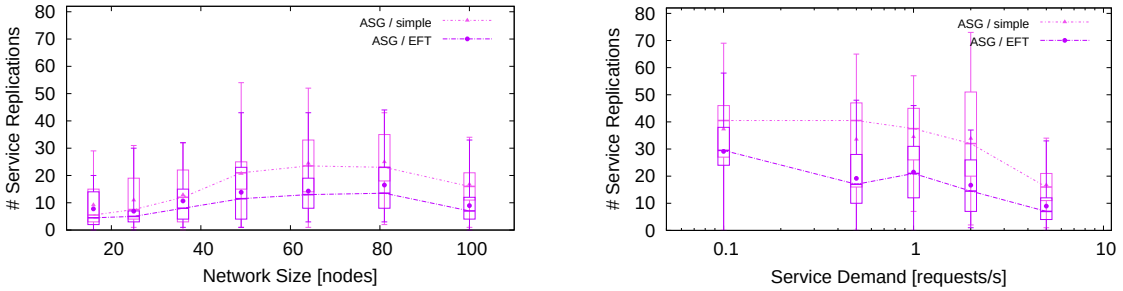


Figure B.8: Placement quality for distributed services (ASG algorithms)



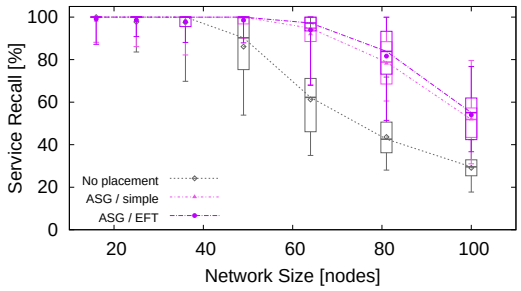
(a) Number of replications vs. network size

(b) Number of replications vs. service demand

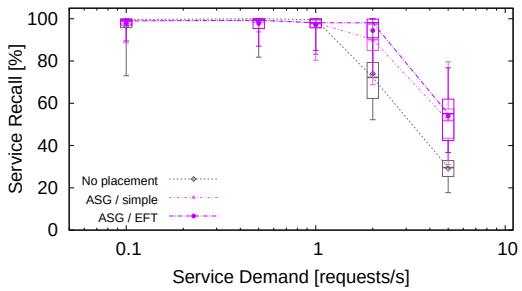
Figure B.9: Placement activity for distributed services (ASG algorithms)

employ the ASG algorithms is lower than that in networks without active service placement.

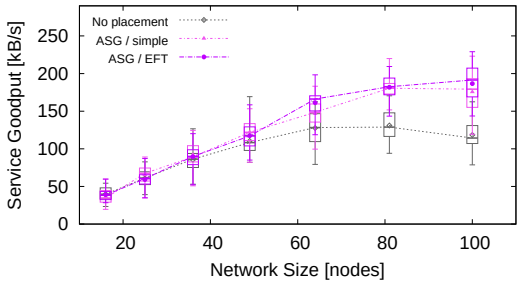
This evaluation has once again shown that distributed service provisioning is superior to a centralized client/server architecture. Furthermore, while the two placement algorithms proposed as part of the ASG framework result in comparable network performance, the ASG / EFT algorithm is able to achieve this with less replications of service instances than ASG / simple. For this reason, we regard ASG / EFT as the superior placement algorithm for distributed services and use it as a reference in the evaluation of our Graph Cost / Multiple Instances algorithm in Sections 6.5 to 6.7.



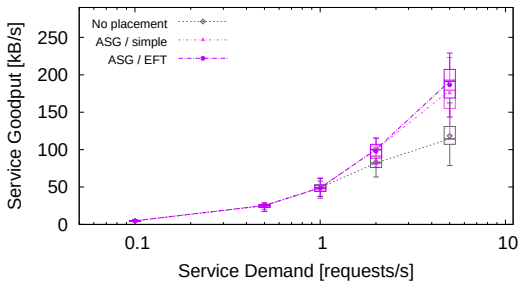
(a) Service recall vs. network size



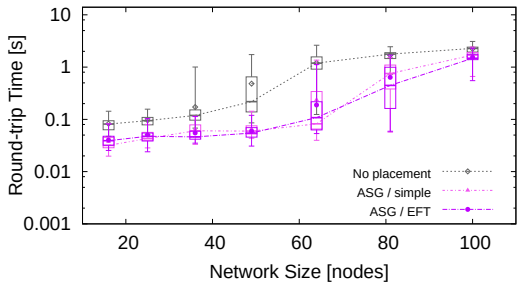
(b) Service recall vs. service demand



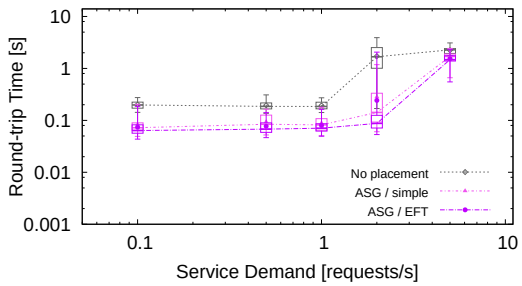
(c) Service goodput vs. network size



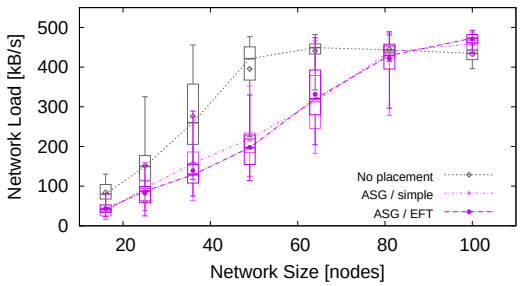
(d) Service goodput vs. service demand



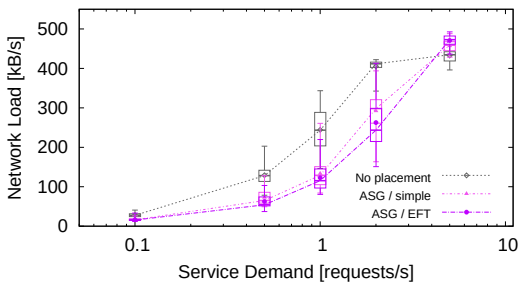
(e) Round-trip time vs. network size



(f) Round-trip time vs. service demand



(g) Network load vs. network size



(h) Network load vs. service demand

Figure B.10: Placement effect for distributed services (current algorithms)

Appendix C

Comparison of Evaluation Methods

This chapter presents a quantitative comparison of the three evaluation methods that we used to measure the results presented in Chapter 6. The goal is to gain an insight into how much the results obtained with each of these tools complement each other and, more importantly, under which circumstances they differ. This will allow us to add credibility to the results presented in Chapter 6 and to judge whether our overall approach to evaluation as proposed in Chapter 3 is valid.

C.1 Differences in Setup

Before looking at the results in detail, it is important to understand in which areas similarities and differences are to be expected and which subtleties may cause them. The main difference between simulations, emulations, and real-world experiments is the (model of the) communication channel between the nodes that form the ad hoc network. Therefore, it makes sense to briefly review the key differences in the setup of the communication channel in each of the evaluation tools:

- **Wireless testbed:** For experiments, the DES-Testbed was configured to run its IEEE 802.11b-compliant network interface cards at 1 Mbit/s with RTS/CTS enabled for all packets (cf. Sec. 6.3.3). To compensate for the low maximal data rate we reduced the service size to 10 kB and the size of the service requests to 256 B.

  The results obtained from these real-world experiments represent the base line for this comparison. If the results from the other evaluation methods differ, then these differences need to be explainable as effects of the differences in setup for the results to claim validity.

- **Emulation setup:** The emulation runs were conducted on a wired IEEE 802.3ab network operating at 1 Gbit/s. The topology was manipulated by selectively dropping packets from preconfigured hosts (cf. Sec. 6.3.2). Similarly, link quality was reduced

artificially by dropping packets with a configurable probability. Since the data rate is higher than in the real-world setup, the service size and the service request size are set to 100 kB and 1024 B respectively.

In this setup, we can expect the packet loss rate to be virtually independent of the network load because the capacity of the wired links by far exceed the offered load. Furthermore, the delay for transmitting a packet can be expected to be shorter than on the wireless testbed since no RTS/CTS procedure is required.

- **Simulation setup:** The `ns-2` network simulator was configured to simulate an IEEE 802.11 network at running at 11 Mbit/s on top of the two-ray ground reflection radio propagation model (cf. Sec. 6.3.1). Like above, the service size and the service request size are set to 100 kB and 1024 B.

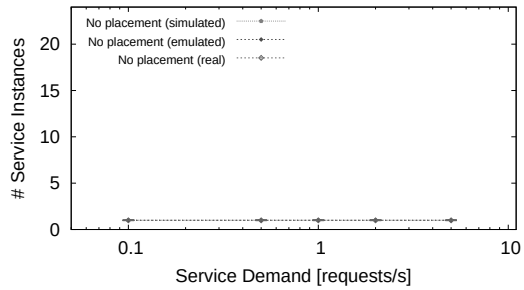
  Employing this radio propagation model in the absence of node mobility results in no inherent packet loss in communication links. The only reason for packet loss are thus packet collisions due to high load.

In light of these differences in the setup, we cannot expect the results of the three evaluation methods to be exact matches. However, it should be possible to explain all differences in the results with the differences in the setup listed above. Any unexplainable difference would indicate a problem with the setup of the respective experiment.

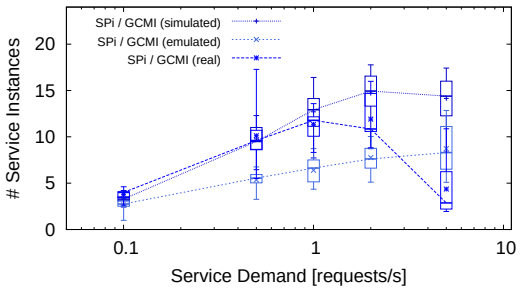
C.2 Comparison

In the comparison of the evaluation methods, we follow the same pattern that we used in the evaluation of placement algorithms and also make use the same metrics (cf. Sec. 6.2). The basic setup is a 36-node network for which we vary the rate at which service requests are issued in the range between 0.1 and 5 requests per second for each client. This corresponds to the setup used in Sections 6.8 and 6.9. For the emulation setup, we present the results that correspond to an artificial packet loss rate of 10%, since these results are the best approximation to the results from the real-world experiments. In order to avoid a too narrow focus in this comparison, we compare the results from both a network without service placement and from a network employing the SP*i* / GCMI placement algorithm.

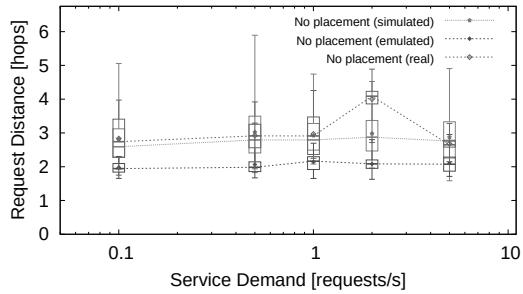
In Figure C.1, we plot the placement activity measurements taken using simulations, emulations, and real-world experiments. For networks without service placement, there is obviously no difference in the number of service instances used since the service configuration remains unchanged throughout the experiments. The request distance, however, differs between simulations and real-world experiments on one side and emulations on the other



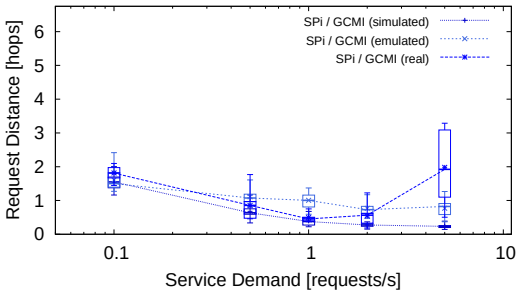
(a) Number of service instances vs. service demand (no placement)



(b) Number of service instances vs. service demand (SP*i* / GCMI)



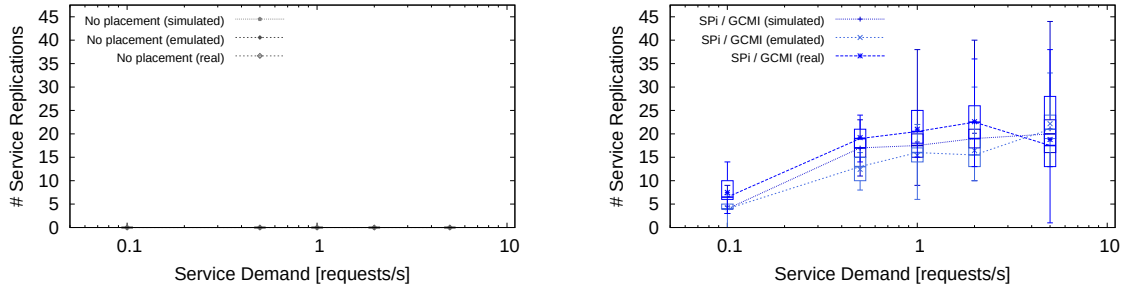
(c) Request distance vs. service demand (no placement)



(d) Request distance vs. service demand (SP*i* / GCMI)

Figure C.1: Placement quality according to different evaluation techniques

side. For emulations, the requests distance is consistently lower than for the other two evaluation methods. This indicates that the average distance between all pairs of nodes is smaller in the artificial network topology that we generated for the emulations than in the simulations and in the testbed. Since the number of nodes is the same for all three setups, we conclude from this that more communication links are available in the emulated topologies. Furthermore, we observe that the request distance suddenly increases in the real-world experiments at a service demand of two requests per second. This peak coincides with the scenario in which the network begins to experience congestion (as can be deduced from the sudden increase in round-trip time in Figure C.3e). Since congestion has the highest impact in the vicinity of the single service instance, we can attribute this increase in request distance to the last hops of some routes to the service instance failing while less frequently used, longer routes still operate normally. Other than that, there is no significant difference in the request distances measured using simulations and real-world experiments. This indicates that we have, by chance, configured the simulations in such a way that the resulting network topology is similar to that of the DES-Testbed as far as average path length is concerned.



(a) Number of replications vs. service demand (no placement)

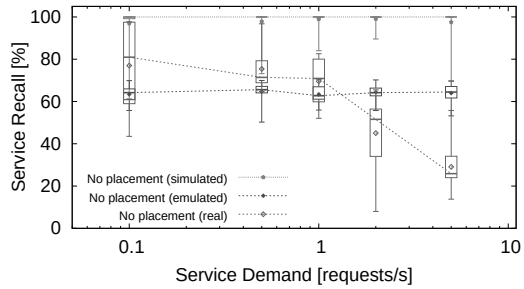
(b) Number of replications vs. service demand (SP*i* / GCMI)

Figure C.2: Placement activity according to different evaluation techniques

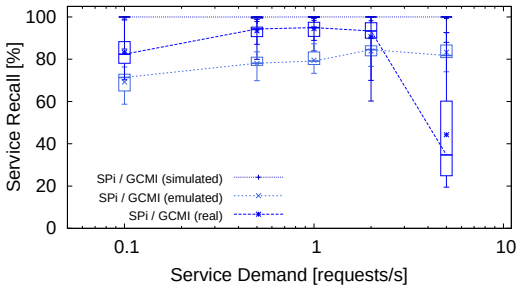
For networks that employ the SP*i* / GCMI placement algorithm (cf. Figs.C.1b and C.1d), we can observe two major differences between the number of service instances: First, the emulations result in a lower number of service instances than the other two methods. This, once again, hints at the artificial topology resulting in denser network graphs than those used in the simulations or in reality. Second, we observe that the number of service instances decreases under high service demand in the real-world experiments. This is due to less replications taking place as the signaling of the service placement system breaks down because of network congestion. After all, the communication channel in the real-world experiments has a lower capacity than that employed in the simulations (by approximately one order of magnitude) and in the emulations (by three orders of magnitude). The request distance, in turn, directly reflects the availability of service instances.

Looking at the replication activity in Figure C.2, we note that there is obviously no placement activity to be expected in a network without active service placement. For SP*i* / GCMI, the number of replications corresponds to our interpretation of the availability of service instances shown in Figure C.1b. In particular, there are less replications in emulations since the placement algorithm utilizes less service instances due to the more densely meshed network. For real-world experiments, the number of replications decreases under high-service demand as signaling packets of the service placement system are more likely to get lost due to network congestion.

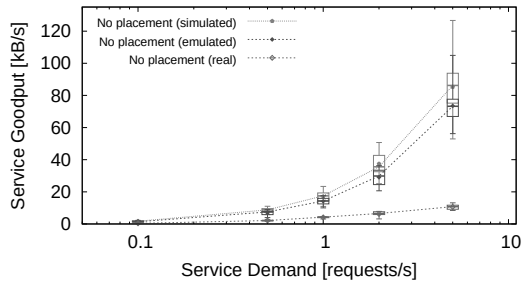
In the metrics that describe the effects of the placement algorithm on the network as shown in Figure C.3, we can observe several similarities between networks that employ no service placement and those that run SP*i* / GCMI. The service recall, for instance, is almost perfect in the simulation setup, constant at a lower level for the emulation setup, and decreasing under high service demand for the real-world experiments. These results directly reflect the different properties of the communication channels used in each of the setups. In simulations, there is no packet loss except for that caused by packet collisions.



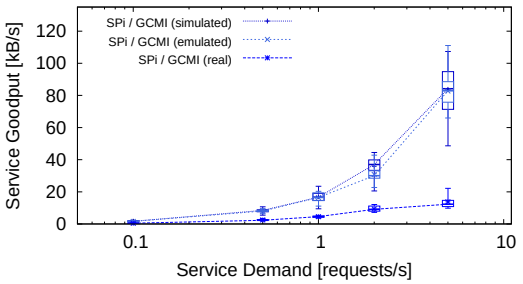
(a) Service recall vs. service demand (no placement)



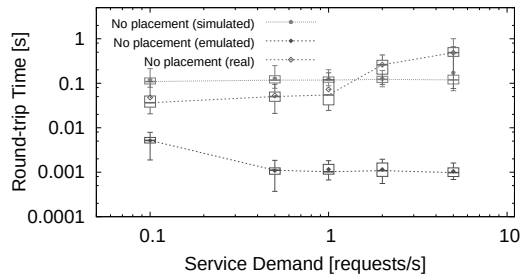
(b) Service recall vs. service demand (SP*i* / GCMI)



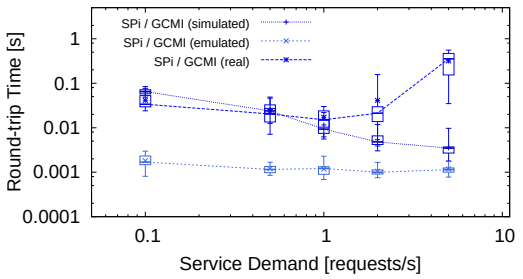
(c) Service goodput vs. service demand (no placement)



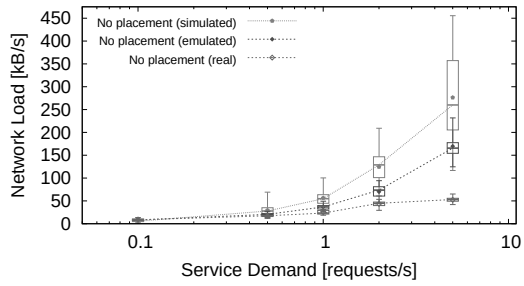
(d) Service goodput vs. service demand (SP*i* / GCMI)



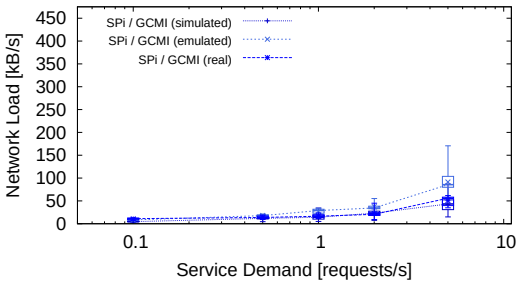
(e) Round-trip time vs. service demand (no placement)



(f) Round-trip time vs. service demand (SP*i* / GCMI)



(g) Network load vs. service demand (no placement)



(h) Network load vs. service demand (SP*i* / GCMI)

Figure C.3: Placement effect according to different evaluation techniques

However, packet collisions only occur in the presence of higher overall volumes of traffic due to the larger capacity of the communication channel (cf. Figs. 6.4a and 6.9a). In emulations, the near-constant recall reflects the fact that this setup is configured with a static packet loss rate of 10%. For the real-world experiments, we can observe that recall decreases as the overall traffic reaches the maximal capacity of wireless communication channel.

The service goodput is significantly lower for the real-world experiments than for the other two evaluation methods. This is due to two factors: First, the service recall suffers most in the real-world experiments and hence the goodput decreases. Second, and more importantly, the size of the service requests in the real-world experiments is only one fourth of that used in the other two methods. Hence, even under ideal conditions, only a quarter of the service goodput measured with the other methods is to be expected.

For the round-trip time, we note that it is significantly lower for the emulation setup than for the other two evaluation methods. As pointed out above, this is due to the fact that no RTS/CTS mechanism is required on the wired IEEE 802.3 network that we used for emulations. Further, there is a discrepancy between simulations and real-world measurements for SP*i* / GCMI under high service demand. This is due to the fact that there are less service instances available in these scenarios in the real-world experiments and thus the distance between clients and service instances is larger (cf. Figs. C.1b and C.1d). Other than that, we note that the round-trip times in simulations and real-world experiments are reasonably good matches. This can be attributed to both the radio propagation model of `ns-2` and the value for `AODV_NODE_TRAVERSAL_TIME` parameter from the AODV standard [87] which we used to simulate processing time on the node in the simulation.

Finally, looking at the overall network load for the networks without service placement in Figure C.3g, we find it to be highest for the simulations, followed by the emulations and then the real-world experiments. This matches the service recall observed using the various methods (cf. Fig. C.3a). The network load of networks employing SP*i* / GCMI shown in Figure C.3h is largely influenced by the operation of the placement algorithm. Therefore, only little can be deduced about the adequacy of the evaluation methods. For example, the load is highest for the emulations due of the lower number of service instances (cf. Fig. C.1b).

In conclusion, we can say that our comparison of the results from the three evaluation methods we used in this work has resulted in but one unexpected difference: The artificial topology used in the emulations seems to result in a denser network graph, which in turn results in a comparatively lower average path length and fewer service instances being created by SP*i* / GCMI. On the other hand, we found that the properties of the network topology used in the simulations closely resemble those of DES-Testbed. This was nothing that we were able to plan when we began to simulate ad hoc networks with service placement

algorithms because the DES-Testbed was not operational at that time. Overall, we can safely argue that the properties of the network topology used in the emulations do not undermine the conclusions that we have drawn in Section 6.8 regarding correct operation of the SP*i* framework and the impact of lossy links.

Our discussion of the other differences in the results has shown that they can be explained satisfactorily by the differences of the communication channel employed by each evaluation tool. We take this as an indication that our evaluation of the SP*i* service placement framework and its algorithms as presented in Chapter 6 is indeed valid.

List of Figures

List of Tables

Bibliography

[1] Zoë Abrams and Jie Liu. Greedy is Good: On Service Tree Placement for In-Network Stream Processing. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*, Lisboa, Portugal, July 2006.

[2] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A Survey on Sensor Networks. *IEEE Communications Magazine*, 40(8):102–116, August 2002.

[3] Ian F. Akyildiz, Xudong Wang, and Weilin Wang. Wireless Mesh Networks: A Survey. *Computer Networks*, 47(4):445–487, March 2005.

[4] Website of the AODV-UU Project. http://core.it.uu.se/core/index.php/AODV-UU.

[5] Website of the Apache HTTP Server Project. http://httpd.apache.org/.

[6] Karl J. Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, February 2009.

[7] Website of the Athens Wireless Network Project. http://wind.awmn.net/.

[8] Paolo Bellavista, Antonio Corradi, and Eugenio Magistretti. Comparing and Evaluating Lightweight Solutions for Replica Dissemination and Retrieval in Dense MANETs. In *Proceedings of the Tenth IEEE International Symposium on Computers and Communications (ISCC '05)*, Cartagena, Spain, June 2005.

[9] Paolo Bellavista, Antonio Corradi, and Eugenio Magistretti. Lightweight Replication Middleware for Data and Service Components in Dense MANETs. In *Proceedings of the First IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM '05)*, Taormina, Italy, June 2005.

[10] Paolo Bellavista, Antonio Corradi, and Eugenio Magistretti. REDMAN: A Decentralized Middleware Solution for Cooperative Replication in Dense MANETs. In *Proceedings of the 2nd International Workshop on Middleware Support for Pervasive Computing (PerWare '05)*, Kauai, HI, USA, March 2005.

[11] Paolo Bellavista, Antonio Corradi, and Eugenio Magistretti. REDMAN: An Optimistic Replication Middleware for Read-only Resources in Dense MANETs. *Journal on Pervasive and Mobile Computing*, 1(3):279–310, August 2005.

[12] Christian Bettstetter and Christoph Renner. A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol. In *Proceedings of the EUNICE Open European Summer School*, Twente, Netherlands, September 2000.

[13] Bastian Blywis, Mesut Günes, Felix Juraschek, and Jochen Schiller. Trends, Advances, and Challenges in Testbed-based Wireless Mesh Network Research. *Mobile Networks and Applications*, 15(3):315–329, June 2010.

[14] Boris Jan Bonfils and Philippe Bonnet. Adaptive and Decentralized Operator Placement for In-Network Query Processing. In *Proceedings of the Second International Workshop on Information Processing in Sensor Networks (IPSN '03)*, Palo Alto, CA, USA, April 2003.

[15] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP). RFC 2205, September 1997.

[16] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huand, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in Network Simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[17] Ian D. Chakeres and Charles E. Perkins. Dynamic MANET On-demand (DYMO) Routing. Internet-Draft, March 2009.

[18] Inc. Cisco Systems. Website of the Linksys WRT54G Wireless Broadband Router. http://homesupport.cisco.com/en-us/wireless/lbc/WRT54G.

[19] Thomas H. Clausen and Philippe Jacquet (eds.). Optimized Link State Routing Protocol (OLSR). RFC 3626, October 2003.

[20] Marco Conti and Silvia Giordano. Multihop Ad Hoc Networking: The Reality. *IEEE Communications Magazine*, 45(4):88–95, April 2007.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, September 2009.

[22] G. Corneújols, G. L. Nemhauser, and L. A. Wolsey. The Uncapacitated Facility Location Problem. In Pitu B. Mirchandani and Richard L. Francis, editors, *Discrete Location Theory*, chapter 3. Wiley-Interscience, December 1990.

[23] HTC Corp. Website of the Google Nexus One Smartphone. http://www.htc.com/www/product/nexusone/overview.html.

[24] Proxim Corporation. ORiNOCO 11b Client PC Card. Datasheet, 2003.

[25] Douglas S. J. De Couto, Daniel Aguayo, John Bicket, and Robert Morris. A High-Throughput Path Metric for Multi-Hop Wireless Routing. In *Proceedings of the Ninth Annual International Conference on Mobile Computing and Networking (MobiCom '03)*, San Diego, CA, USA, September 2003.

[26] Pengfei Di, Matthias Wählisch, and Georg Wittenburg. Modeling the Network Layer and Routing Protocols. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*. Springer, 2010. (in print).

[27] Falko Dressler. Self-Organization in Ad Hoc Networks: Overview and Classification. Technical Report 02/06, University of Erlangen, Department of Computer Science 7, Erlangen, Germany, March 2006.

[28] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, FL, USA, November 2004.

[29] Website of the DYMO-UM Project. http://masimum.dif.um.es/?Software:DYMOUM.

[30] Norman Dziengel. Verteilte Ereigniserkennung in Sensornetzen. Master's thesis, Department of Mathematics and Computer Science, Freie Universität Berlin, October 2007.

[31] Kevin Fall and Kannan Varadhan. *The ns Manual*. The VINT Project, May 2007.

[32] WiMAX Forum. Website of the WiMAX Forum. http://www.wimaxforum.org/.

[33] OLPC Foundation. Website of the One Laptop Per Child Project. http://laptop.org/.

[34] Christian Frank and Kay Römer. Distributed Facility Location Algorithms for Flexible Configuration of Wireless Sensor Networks. In *Proceedings of the 3rd IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS '07)*, Santa Fe, NM, USA, June 2007.

[35] Website of the Freifunk Berlin Project. http://berlin.freifunk.net/.

[36] Takehiro Furuta, Hajime Miyazawa, Fumio Ishizaki, Mihiro Sasaki, and Atsuo Suzuki. A Heuristic Method for Clustering a Large-scale Sensor Network. In *Proceedings of Wireless Telecommunications Symposium (WTS '07)*, Pomona, CA, USA, April 2007.

[37] Takehiro Furuta, Mihiro Sasaki, Fumio Ishizaki, Atsuo Suzuki, and Hajime Miyazawa. A New Clustering Algorithm Using Facility Location Theory for Wireless Sensor Networks. Technical Report NANZAN-TR-2006-04, Nanzan Academic Society, March 2007.

[38] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, January 1979.

[39] Diptesh Ghosh. Neighborhood Search Heuristics for the Uncapacitated Facility Location Problem. *European Journal of Operational Research*, 150(1):150–162, October 2003.

[40] PC Engines GmbH. *ALIX.2 / ALIX.3 / ALIX.6 Series System Boards*, May 2010.

[41] Mesut Günes, Bastian Blywis, and Felix Juraschek. Concept and Design of the Hybrid Distributed Embedded Systems Testbed. Technical Report TR-B-08-10, Freie Universität Berlin, Berlin, Germany, August 2008.

[42] Mesut Günes, Bastian Blywis, Felix Juraschek, and Philipp Schmidt. Practical Issues of Implementing a Hybrid Multi-NIC Wireless Mesh-Network. Technical Report TR-B-08-11, Freie Universität Berlin, Berlin, Germany, August 2008.

[43] E. Guttman, Charles E. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. RFC 2608, June 1999.

[44] S. L. Hakimi. Optimum Locations of Switching Centers and the Absolute Centers and Medians of a Graph. *Operations Research*, 12(3):450–459, May 1964.

[45] S. L. Hakimi. Optimum Distribution of Switching Centers in a Communication Network and Some Related Graph Theoretic Problems. *Operations Research*, 13(3):462–475, May 1965.

[46] Takahiro Hara. Data Replication and Update Management in Mobile Ad Hoc Networks. In *Proceedings of the International Workshop on Data Management in Global Data Repositories (GRep '06)*, pages 622–626, Krakow, Poland, September 2006.

[47] John Heidemann, Kevin Mills, and Sri Kumar. Expanding Confidence in Network Simulation. *IEEE Network Magazine*, 15(5):58–63, September/October 2001.

[48] Wendi B. Heinzelman, Anantha P. Chandrakasan, and Hari Balakrishnan. An Application-Specific Protocol Architecture for Wireless Microsensor. *IEEE Transactions on Wireless Networking*, pages 660–670, October 2002.

[49] Klaus Herrmann. *Self-Organizing Infrastructures for Ambient Services*. PhD thesis, Berlin University of Technology, Berlin, Germany, July 2006.

[50] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, Cambridge, MA, USA, November 2000.

[51] Ian Jacobs and Norman Walsh (eds.). Architecture of the World Wide Web, Volume One. W3C Recommendation, December 2004.

[52] Sam Jansen and Anthony McGregor. Simulation with Real World Network Stacks. In *Proceedings of the 2005 Winter Simulation Conference*, December 2005.

[53] Sam Jansen and Anthony McGregor. Validation of Simulated Real World TCP Stacks. In *Proceedings of the 2007 Winter Simulation Conference*, 2007.

[54] David B. Johnson, Yih-Chun Hu, and David A. Maltz. The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4. RFC 4728, February 2007.

[55] David B. Johnson, David A. Maltz, and Josh Broch. *Ad Hoc Networking*, chapter 5: DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks, pages 139–172. Addison-Wesley, 2001.

[56] O. Kariv and S. L. Hakimi. An Algorithmic Approach to Network Location Problems. Part II: The p-medians. *SIAM Journal on Applied Mathematics*, 37(3):539–560, 1979.

[57] Denis Krivitski, Assaf Schuster, and Ran Wolff. A Local Facility Location Algorithm for Sensor Networks. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS '05)*, Marina del Rey, CA, USA, June 2005.

[58] Denis Krivitski, Assaf Schuster, and Ran Wolff. A Local Facility Location Algorithm for Large-Scale Distributed Systems. *Journal of Grid Computing*, 2006.

[59] Georg Kunz, Olaf Landsiedel, and Georg Wittenburg. From Simulations to Deployments. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*. Springer, 2010. (in print).

[60] Mauri Kuorilehto, Marko Hännikäinen, and Timo D. Hämäläinen. A Survey of Application Distribution in Wireless Sensor Networks. *EURASIP Journal on Wireless Communications and Networking*, 2005(5):774–788, December 2005.

[61] Stuart Kurkowski, Tracy Camp, and Michael Colagrosso. MANET Simulation Studies: The Incredibles. *ACM SIGMOBILE Mobile Computing and Communications Review*, 9(4):50–61, October 2005.

[62] Nikolaos Laoutaris, Georgios Smaragdakis, Konstantinos Oikonomou, Ioannis Stavrakakis, and Azer Bestavros. Distributed Placement of Service Facilities in Large-Scale Networks. In *Proceedings of the 26th Annual IEEE Conference on Computer Communications (IEEE INFOCOM '07)*, Anchorage, AK, USA, May 2007. To appear, published as technical report.

[63] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys '03)*, 2003.

[64] Baochun Li. QoS-Aware Adaptive Services in Mobile Ad-Hoc Networks. In *Proceedings of the 9th International Workshop on Quality of Service (IWQoS '01)*, pages 251–268, Karlsruhe, Germany, June 2001.

[65] Baochun Li. On Increasing Service Accessibility and Efficiency in Wireless Ad-hoc Networks with Group Mobility. *Wireless Personal Communications, Special Issue on Multimedia Networking and Enabling Radio Technologies*, 21(1):105–123, April 2002.

[66] Baochun Li and Karen H. Wang. NonStop: Continuous Multimedia Streaming in Wireless Ad Hoc Networks with Node Mobility. *IEEE Journal on Selected Areas in Communications, Special Issue on Recent Advances in Wireless Multimedia*, 21(10):1627–1641, December 2003.

[67] Martin Lipphardt, Jana Neumann, Sven Groppe, and Christian Werner. DySSCo - A Protocol for Dynamic Self-organizing Service Coverage. In *Proceedings of the 3rd International Workshop on Self-Organizing Systems (IWSOS '08)*, Vienna, Austria, December 2008.

[68] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin Gün Sirer. Design and Implementation of a Single System Image Operating System for Ad Hoc Networks. In *Proceedings of the Third International Conference on Mobile Systems, Applications, and Services (MobiSys '05)*, Seattle, WA, USA, June 2005.

[69] Jie Liu and Feng Zhao. Towards Semantic Services for Sensor-Rich Information Systems. In *Proceedings of the Second IEEE/CreateNet International Workshop on Broadband Advanced Sensor Networks (Basenets '05)*, Boston, MA, USA, October 2005.

[70] Jinshan Liu and Valérie Issarny. Signal Strength based Service Discovery (S3D) in Mobile Ad Hoc Networks. In *Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC)*, September 2005.

[71] LogiLink. Wireless LAN USB Stick 54 Mbit 802.11g. Datasheet.

[72] Compex Systems Pte Ltd. WLM Mini PCI Series - WLM54G(G) / WLM54G(SUPERG) / WLM54AG(AG) / WLM54AG(SUPERAG). Datasheet, 2006.

[73] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, Rebekah Metz, and Booz Allen Hamilton (eds.). Reference Model for Service Oriented Architecture 1.0. OASIS Standard, October 2006.

[74] Satish Menon and Carter L. Horney. Smartphone & Chip Market Opportunities. Forward Concepts Co., Report No: 9010, February 2009.

[75] E. Domínguez Merino, J. Mu noz Pérez, and J. Jerez Aragonés. Neural Network Algorithms for the p-Median Problem. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN '03)*, pages 385–391, Bruges, Belgium, April 2003.

[76] Kevin L. Mills. A Survey of Self-Organization in Wireless Networks. April 2006.

[77] Pitu B. Mirchandani and Richard L. Francis, editors. *Discrete Location Theory*. Wiley-Interscience, December 1990.

[78] P. Mockapetris. Domain Names - Concepts and Facilities. RFC 1034, November 1987.

[79] P. Mockapetris. Domain Names - Implementation and Specefication. RFC 1035, November 1987.

[80] Anirban Mondal, Sanjay Kumar Madria, and Masaru Kitsuregawa. CLEAR: An Efficient Context and Location-based Dynamic Replication Scheme for Mobile-P2P Networks. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA '06)*, Krakow, Poland, September 2006.

[81] Thomas Moscibroda and Roger Wattenhofer. Facility Location: Distributed Approximation. In *Proceedings of the 24th ACM Symposium on the Principles of Distributed Computing (PODC '05)*, Las Vegas, NV, USA, July 2005.

[82] Gero Mühl, Andreas Ulbrich, and Hartmut Ritter. Content Evolution Driven Data Propagation in Wireless Sensor Networks. In *Proceedings of the 2. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, Karlsruhe, Germany, February 2004.

[83] Calvin Newport, David Kotz, Yougu Yuan, Robert S. Gray, Jason Liu, and Chip Elliott. Experimental Evaluation of Wireless Simulation Assumptions. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 83(9):643–661, September 2007.

[84] Website of the Network Simulator `ns-2`. http://nsnam.isi.edu/nsnam/index.php/Main_Page.

[85] Konstantinos Oikonomou and Ioannis Stavrakakis. Scalable Service Migration: The Tree Topology Case. In *Proceedings of the Fifth Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net '06)*, Lipari, Italy, June 2006.

[86] Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-Level Sensor Network Simulation with COOJA. In *Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp '06)*, Tampa, FL, USA, November 2006.

[87] Charles E. Perkins. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561, July 2003.

[88] Charles E. Perkins. *Ad Hoc Networking*. Addison-Wesley Professional, July 2008.

[89] Charles E. Perkins and Pravin Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *Proceedings of SIGCOMM'94*, 1994.

[90] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc On Demand Distance Vector Routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, 1999.

[91] Josh Reese. Solution Methods for the p-Median Problem: An Annotated Bibliography. *Networks*, 48(3):125–142, August 2006.

[92] C. ReVelle and R. Swain. Central Facilities Location. *Geographical Analysis*, 2:30–42, 1970.

[93] Françoise Sailhan and Valerie Issarny. Scalable Service Discovery for MANET. In *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications (PerCom '05)*, pages 235–244, Kauai Island, HI, USA, March 2005.

[94] Andréa Cynthia Santos. Solving Large p-Median Problems using a Lagrangean Heuristic. Technical Report RR-09-03, Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes, Université Blaise Pascal, Aubière, France, August 2009.

[95] Gregor Schiele, Christian Becker, and Kurt Rothermel. Energy-Efficient Cluster-based Service Discovery for Ubiquitous Computing. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[96] Jochen Schiller. *Mobile Communications*. Addison-Wesley, second edition, November 2003.

[97] Jochen Schiller, Achim Liers, and Hartmut Ritter. ScatterWeb: A Wireless Sensornet Platform for Research and Teaching. *Computer Communications*, 28:1545–1551, April 2005.

[98] Balasubramanian Seshasayee, Ripal Nathuji, and Karsten Schwan. Energy-Aware Mobile Service Overlays: Cooperative Dynamic Power Management in Distributed Mobile Systems. In *Proceedings of the Fourth International Conference on Autonomic Computing (ICAC '07)*, Jacksonville, FL, USA, June 2007.

[99] Balasubramanian Seshasayee and Karsten Schwan. Mobile Service Overlays: Reconfigurable Middleware for MANETs. In *Proceedings of the First International Workshop on Decentralized Resource Sharing in Mobile Computing and Networking (MobiShare '06)*, pages 30–35, Los Angeles, CA, USA, September 2006.

[100] IEEE Computer Society. IEEE Standard for Information Technology – Telecommunications and Information Exchange Between Systems – Local and Metropolitan Area Networks – Specific Requirements – Part 15.1: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs). IEEE 802.15.1-2005, June 2005.

[101] IEEE Computer Society. IEEE Standard for Information Technology – Telecommunications and Information Exchange Between Systems – Local and Metropolitan Area Networks – Specific Requirements – Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs). IEEE 802.15.4-2006, September 2006.

[102] IEEE Computer Society. IEEE Standard for Information Technology – Telecommunications and Information Exchange Between Systems – Local and Metropolitan Area Networks – Specific Requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE 802.11-2007, June 2007.

[103] IEEE Computer Society. IEEE Standard for Local and Metropolitan Area Networks – Part 16: Air Interface for Broadband Wireless Access Systems. IEEE 802.16-2009, May 2009.

[104] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN '05)*, pages 477–482, Los Angeles, CA, USA, April 2005.

[105] Karen H. Wang and Baochun Li. Efficient and Guaranteed Service Coverage in Partitionable Mobile Ad-hoc Networks. In *Proceedings of IEEE INFOCOM '02*, volume 2, pages 1089–1098, New York City, NY, USA, June 2002.

[106] Karen H. Wang and Baochun Li. Group Mobility and Partition Prediction in Wireless Ad-hoc Networks. In *Proceedings of IEEE International Conference on Communications (ICC '02)*, volume 2, pages 1017–1021, New York City, NY, USA, April 2002.

[107] T. Winter and P. Thubert (eds.). RPL: IPv6 Routing Protocol for Low power and Lossy Networks. Internet-Draft, July 2010.

[108] Georg Wittenburg and Jochen Schiller. Running Real-World Software on Simulated Wireless Sensor Nodes. In *Proceedings of the ACM Workshop on Real-World Wireless Sensor Networks (REALWSN '06)*, pages 7–11, Uppsala, Sweden, June 2006.

[109] Georg Wittenburg and Jochen Schiller. A Quantitative Evaluation of the Simulation Accuracy of Wireless Sensor Networks. In *Proceedings of 6. Fachgespräch "Drahtlose Sensornetze" der GI/ITG-Fachgruppe "Kommunikation und Verteilte Systeme"*, pages 23–26, Aachen, Germany, July 2007.

[110] Georg Wittenburg and Jochen Schiller. A Survey of Current Directions in Service Placement in Mobile Ad-hoc Networks. In *Proceedings of the Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom '08, Middleware Support for Pervasive Computing Workshop)*, pages 548–553, Hong Kong, March 2008.

[111] Georg Wittenburg and Jochen Schiller. Service Placement in Ad Hoc Networks. *PIK - Praxis der Informationsverarbeitung und Kommunikation*, 33(1):21–25, January 2010.

[112] MICA2 Wireless Measurement System. Crossbow Technologies, Inc., 2003.

[113] MICA2DOT Wireless Microsensor Mote. Crossbow Technologies, Inc., 2005.

[114] MICAz Wireless Measurement System. Crossbow Technologies, Inc., 2008.

[115] Wu Xiuchao. Simulate 802.11b Channel within NS2. Technical report, National University of Singapore, Singapore, 2004.