Diss. ETH No. 17354

Role-based Configuration of Wireless Sensor Networks

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences

presented by
Christian Frank
Diplom-Informatiker, Technische Universität Berlin
born November 3rd, 1978
citizen of Germany

accepted on the recommendation of
Prof. Dr. Friedemann Mattern, examiner

Prof. Dr. Gian Pietro Picco, co-examiner

2007

Abstract

Wireless sensor networks consist of so-called sensor nodes – small computing devices which integrate processing, storage, sensing, and wireless communication capabilities and an autonomous power source. Due to their decreased dependency on a wired infrastructure, wireless sensor networks can be used to monitor environmental phenomena on an unprecedented scale.

This thesis considers approaches for in-situ configuration of wireless sensor networks. In-situ configuration is a prominent challenge of wireless sensor networks: On the one hand, network nodes typically cannot be configured manually due to the large scale of such networks. On the other hand, nodes cannot be configured automatically prior to deployment because their runtime properties are unknown at deployment time and node failures (e.g., due to depleted battery power) require frequent re-configuration of the network during its lifetime.

Consequently, it is common that, right after deployment, a wireless sensor network is in a homogeneous software state. In this property, wireless sensor networks differ from classic infrastructure-based networks in which some nodes are custom-built or pre-configured for a certain task (e.g., a router used for routing). Therefore, a main purpose of configuration is to break the initial symmetry and to realize and maintain the desired structure of the network in face of runtime properties that change with time.

The contributions of this thesis approach such configuration tasks by assigning *roles* to network nodes based on their runtime properties (i.e., based on the nodes' remaining battery power, location, or network neighbors). Such automatic role-based network configuration is part of many applications and also present in classic configuration protocols developed for wireless sensor networks. For example, the *coverage* problem aims to exploit redundancy in the network by assigning the roles *sensing* and *idle* such that the selected *sensing* nodes suffice to collect data on the area of interest. Similarly, clustering aproaches select *cluster leaders* as sole communication partners for *slave* nodes in their vicinity which al-

lows slave nodes to save energy by synchronizing to the communication schedule of their cluster leaders. Finally, the construction of a data gathering tree made up of adequately chosen *aggregator* nodes can also be interpreted as a role-based configuration problem.

In this context, this thesis claims that generic system support for such role-based configuration tasks can significantly reduce the effort required for programming wireless sensor network applications and yet be implemented efficiently. We sustain this claim by providing generic support for role-based configuration problems occuring in a set of heterogeneous application domains and by showing that in each domain adequate system support can significantly simplify the configuration of an application. Further, our evaluations demonstrate that the involved overhead is proportional to the "hardness" of the task that is formulated by the programmer at the provided configuration interface. The observed overhead is sometimes even comparable to application-specific implementations that have been optimized for the specific configuration task.

In a wider context, this thesis' contributions provide advances toward plug and play sensor network systems that can be flexibly parameterized for a certain application-specific task without requiring the expertise of a highly specialized engineer for this purpose. Such out of the box usability, often presumed a key prerequisite to wide-spread usage of wireless sensor networks, would finally enable the economies of scale commonly associated with the wireless sensor network vision.

Kurzfassung

Drahtlose Sensornetze basieren auf miteinander kommunizierenden Sensorknoten, die auf engem Raum einen Mikroprozessor, Speicherplatz, eine autonome Energiequelle und die Möglichkeit zur drahtlosen Kommunikation vereinen. Da Sensornetze nicht von vorhandener Infrastruktur abhängig sind, können sie Umweltphänomene kostengünstiger und grossflächiger als bestehende Methoden erfassen.

Diese Arbeit beschäftigt sich mit der Konfiguration von bereits in einer Arbeitsumgebung installierten Sensornetzen. Solche Konfiguration der Sensorknoten vor Ort ist häufig unumgänglich: Einerseits können wegen der zu erwartenden Grösse solcher Netze die einzelnen Sensorknoten nicht manuell konfiguriert werden, andererseits kann die automatische Konfiguration der Knoten erst nach deren Installation erfolgen, da die für die Konfiguration erforderlichen Knoteneigenschaften (wie zum Beispiel deren Ort) erst zur Laufzeit des Systems bekannt sind. Zudem verlangen häufig auftretende Ausfälle einzelner Knoten, bedingt durch Umwelteinflüsse oder aufgebrauchte Batteriekapazität, Anpassungen in der Konfiguration der Knoten während des laufenden Betriebs.

Deshalb befinden sich die Knoten eines Sensornetzes direkt nach dessen Inbetriebnahme zunächst in einem homogenen Zustand. In dieser Eigenschaft unterscheiden sich Sensornetze von klassischen infrastrukturbasierten Computernetzen, in denen Knoten entweder für eine bestimmte Aufgabe (zum Beispiel, um Nachrichten weiterzuleiten) gebaut wurden oder zumindest vor der Inbetriebnahme auf eine bestimmte Aufgabe hin vorkonfiguriert werden können. Ein Hauptaugenmerk bei der Konfiguration von Sensornetzen muss deshalb sein, den anfänglichen symmetrischen Zustand in die gewünschte Struktur des Netzes zu überführen und diese Struktur trotz Änderungen der Knoteneigenschaften über die Laufzeit des Netzes hinweg aufrechtzuerhalten.

Diese Disseration nimmt sich Konfigurationsaufgaben in Sensornetzen an, indem sie Sensorknoten, basierend auf deren Eigenschaften (wie zum Beispiel deren Ort, deren verbleibende Batteriekapazität oder Eigenschaften der jeweiligen Nachbarn in der Kommunikationstopologie), *Rollen*

zuweist. Solche automatische Konfiguration mittels Zuweisung von Rollen ist Teil vieler klassischer Konfigurationsprotokolle, die für drahtlose Sensornetze entwickelt wurden. Zum Beispiel hat das sogenannte *Coverage*-Problem zum Ziel, die Redundanz von Sensorknoten zu nutzen: Die Rolle *Sensor an* wird an eine Untermenge aller Knoten zugewiesen, die für die Abdeckung des untersuchten Gebietes ausreicht, wohingegen die Knoten mit der Rolle *Sensor aus* in einen energiesparenden Zustand wechseln können. Ähnlich wird im *Clustering*-Problem ein Knoten als *Zentrum* einer Gruppe von Knoten ausgewählt, während die verbleibenden *Randknoten* Energie einsparen, indem sie nur mit dem ihnen zugewiesenen *Zentrum* kommunizieren und daher keine Nachrichten für andere Knoten weiterleiten müssen. Schliesslich lassen sich auch Baumstrukturen, die häufig zur Datengewinnung in Sensornetzen verwendet werden, auch als Rollenzuweisungsproblem interpretieren.

In diesem Zusammenhang vertrete ich in dieser Arbeit die These, dass generische Systemunterstützung für rollenbasierte Konfiguration die Programmierung von Applikationen für drahtlose Sensornetze beträchtlich vereinfacht und dass solche Unterstützung auch effizient implementiert werden kann. Diese These wird durch die Bereitstellung verschiedener rollenbasierter Konfigurationsdienste gestützt, welche in ihrer jeweiligen Anwendungsdomäne die Lösung von Konfigurationsaufgaben signifikant erleichtern. In Experimenten wird aufgezeigt, wie der Kommunikationsaufwand der bereitgestellten Dienste sich proportional zur "Schwere" des vom Programmierer spezifizierten Konfigurationsproblems verhält. Zudem erfordern die generischen Dienste häufig nicht mehr Aufwand als Dienste, die auf spezifische Applikationen hin optimiert wurden.

In einem weiter gefassten Kontext kann diese Dissertation als ein Baustein auf dem Weg zu komponentenbasierten Sensornetzsystemen angesehen werden, in denen eine gewünschte Applikation aus bestehenden Diensten flexibel zusammengesetzt werden kann, ohne dass die Hilfe eines Spezialisten dafür nötig wäre. Solch einfache Nutzbarkeit von Sensornetztechnologien wird oft als eine Schlüsselvoraussetzung für die Anwendung von Sensornetzen in Fachgebieten ausserhalb der Informatik angesehen. Eine grössere Verbreitung von Sensornetzen würde wiederum auch die häufig genannte Vision der Massenproduktion von Sensorknoten und die damit verbundene Wirtschaftlichkeit grosser Sensornetze näher rücken lassen.

Contents

List of Figures

1 Introduction

Wireless sensor networks consist of so-called sensor nodes – small untethered computing devices equipped with sensors, a wireless radio, a processor, and autonomous power supply. Large and dense networks of these devices can be deployed unobtrusively in the physical environment in order to monitor a wide variety of real-world phenomena with unprecedented quality and scale while only marginally disturbing the observed physical processes [ASSC02, KW05].

Various research projects have involved wireless sensor network deployments. These include examining the nesting behavior of otherwise elusive seabirds [MPS+02], monitoring the micro-climate of redwood trees [TPS+05], observing the structural integrity of bridges [XRC+04], and assisting firefighters and help workers with real-time information on their current environment [LMFJ+04].

Each of the deployments, however, has required considerable programming and maintenance effort by highly-specialized engineers and researchers working in the sensor network field. Key for the wide-spread applicability of wireless sensor networks, however, is that non-experts can easily set up a network for a task at hand. To achieve this, system support for the core issues that re-appear in many wireless sensor network applications must be available, in order to provide non-experts high-level levers for specifying the network's behavior.

In this context, the aim of this thesis is to provide specification techniques, algorithms, and tools to support developers with one common need that re-appears in applications for large-scale wireless sensor networks: The *configuration* of the nodes in the network.

1.1 Motivation

Configuration is a prominent challenge in wireless sensor networks based on an intrinsic property in which these networks differ from traditional distributed systems. In classic infrastructure-based networks, many nodes are manufactured and installed with a certain purpose (such as a router,

a hub, a server, or a client device). Typically, the nodes' physical location and their place in the communication topology are pre-determined or manually assigned by network administrators. Moreover, single purpose devices, such as routers, come with pre-installed software dedicated to performing the specific task. Multi-purpose devices, such as servers or clients, can also be set up by hand or in batches by a network administrator who distributes software to a number of nodes from a central location.

In contrast, in wireless sensor networks, many node properties on which configuration decisions are regularly based either cannot be known in advance or change with time. One such property is the nodes' physical location: Some applications require that nodes are deployed at random, i.e., dropped out of the air to the areas of interest [CHP+04]. Others involve mobile sensors attached to animals [JOW+02, BLM05] or carried by people [EAL+06, FBRK07]. Similar considerations apply to other runtime properties such as the nodes' battery power or the quality of their measurements. Moreover, node failures are common as nodes may become dysfunctional due to depleted battery power or environmental damage.

Based on these observations, a sensor node cannot specialize on a certain task before the network is in operation. Instead, the network starts out in a homogeneous software state and the configuration of the network, in terms of specialized roles such as clients, hubs, routers and servers, must be computed and maintained based on the network's runtime parameters.

This thesis provides system support for such in-situ configuration of wireless sensor networks. First, a generic framework which allows to state the desired network structure by means of a simple declarative interface is presented together with efficient algorithms for its implementation.

The interface of the presented framework allows programmers to formulate a variety of network configuration problems focusing on ease of use for application-domain experts. In this regard, the provided interface cannot express every conceivable configuration task.

Therefore, we also elaborate on two specific configuration problems which require more advanced features. The first approach supports developers with computing near-optimal clustering configurations, which involve selecting a few nodes as *hubs* providing certain services to their network neighbors. The second approach supports developers with a specific form of the coverage problem by selecting a set of nodes, which are likely to provide useful data on a phenomenon of interest, as *data sources*, while the remaining nodes may keep their sensors off and thus save power.

We detail each of these contributions in the following three sections.

1.2 Generic Role Assignment

In Chapter 2, we present a programming framework concerned with assigning specific roles to individual sensor nodes if certain conditions are met. These conditions can be formulated in terms of runtime properties of a node (e.g., its location or battery level) and in terms of properties of nodes in the network neighborhood [RFMB04, FR05]. As the network and node properties change over time, role assignments must be updated to reflect these changes. Based on the assigned roles, sensor nodes may adapt their behavior accordingly, establish cooperation with other nodes, or even download specific code for the selected role.

A number of research projects have stated the need for such role-based configuration in sensor networks (e.g., [HMCP04, MLM+05, UWMG05]). Moreover, even classic network configuration problems can be considered instances of generic role assignment. To illustrate the concept consider the coverage problem [XHE01]. Here, two roles `on` and `off` are defined such that every geographic spot falls within the observation range of at least one `on` node. Based on this definition, `off` nodes do not contribute to sensor coverage and thus may switch to a power-saving sleep mode. Once an `on` node fails, e.g., due to depleted battery power, redundant `off` nodes would switch their role to `on` such that the coverage condition is satisfied.

Similar roles and conditions for their assignment can be found for other network configuration problems. For example, to obtain a clustered configuration [KG02], the roles `clusterhead` and `slave` are assigned to each node such that clusterheads can act as hubs and represent sole communication partners for associated `slave` nodes, while `gateway` nodes forward data between clusterheads. Similarly, a data aggregation tree can be implemented by assigning the roles `source` and `aggregator` in a manner that allows aggregators to efficiently compress the data provided by source nodes while forwarding it to the network base station.

While a number of specialized algorithms for these problems have been developed, these are typically hard to adapt to different applications, where varying criteria for assigning the above roles may have to be applied. Driven by these observations, the presented role assignment framework provides specification techniques and algorithms that support *generic role assignment*, a programming abstraction applicable to a wide variety of role-assignment problems similar to the ones described above.

Moreover, the results of our work may be integrated as a fundamental service into programming frameworks such as [HMCP04, MLM+05].

Generic role assignment can be considered a *programming abstraction* that partially shields application developers from the complexity of programming sensor networks at the system level. Rather than implementing low-level protocols and node functions, the developer can now specify parts of the system behavior using a high-level configuration language. Such programming abstractions have recently gained significant attention (e.g., [ABC+04, WM04]) and can be interpreted as a step towards making sensor networks more accessible for users who are not experienced system-level programmers (e.g., typical application-domain experts).

The presented role assignment framework consists of three elements. To configure a sensor network, the programmer may issue a set of *role specifications* containing a set of roles and conditions for their assignment. At the network base station, a *role compiler* translates the specification into a compact abstract representation. This representation may be either pre-installed on sensor nodes prior to deployment or injected into the network by the network base station. On the nodes, a *property directory* provides transparent access to node properties and capabilities such as a node's location or remaining battery power. Based on the nodes' properties and a given specification, a decentralized *role assignment algorithm* assigns roles to sensor nodes. The assignment of a given role may then trigger role-specific code to be executed, for example, enabling a certain routing component once the node has become `clusterhead` or trigger a download of additional role-specific code from the network base station. Finally, the role compiler has been enhanced with a tool for *offline specification analysis* [FR06] which allows to examine various aspects of role specifications, such as feasibility and optimality, before distributing them to the network.

In our evaluation, we show that the overhead induced by the distributed role assignment algorithm is small and moreover proportional to the hardness of the specified configuration problem. Moreover, we discuss how the semantics of role specifications executed by our generic algorithms are comparable to highly specialized algorithms designed for the same problems. Finally, we demonstrate how offline specification analysis can provide further valuable insights into the runtime behavior of certain role specifications.

1.3 Distributed Facility Location

The general focus of generic role assignment is more on ease of use than on computing an optimized configuration. In particular, while the distributed role assignment algorithms attempt to find a *feasible* configuration based on a given role specification, they do not employ any means for choosing an *optimal* role assignment (for example, with a minimum number of `clusterhead` nodes) among multiple assignments that are feasible.

We therefore examine an important subclass of role-assignment problems for which optimality guarantees can be provided in Chapter 3. This subclass requires that certain nodes are selected as *servers* such that every network node can access a server node within a (preferably) small network distance from itself. Several application examples mentioned in the previous section belong to this class, for example *clustering* and *aggregator placement*. We use the *facility location problem* to model such network configuration decisions which require choosing a subset of *server* nodes (also known as facilities) to act as service providers for their network neighborhood (the remaining nodes are called *clients*).

The facility location problem deals with finding an optimal trade-off between two different costs of network operation. On the one hand, so-called *opening costs* model the costs for operating server nodes. For example, these may represent a node's increased communication load if it were to forward traffic for its neighbors or, generally, a measure of the node's suitability as a clusterhead or service provider. On the other hand, *communication costs* model the overhead involved when clients communicate with their closest server, for example, based on the quality of the observed network paths. Given a graph and a set of opening and communication costs, the distributed facility location algorithms described in Chapter 3 assign the roles `server` and `client` to network nodes with the objective to obtain a configuration whose sum of opening and communication costs is minimal.

While the presented algorithms address a smaller set of configuration problems than generic role assignment, they provide near-optimal solutions for problems that are part of this subset. Compared to existing clustering approaches, the presented algorithm is more generic. Specifically, by defining the opening costs of each node and respective communication costs between nodes, the presented approach can be parameterized to compute a minimum dominating set [Mos07] as well as configura-

tions in which clients and servers may be an arbitrary number of hops apart [Fra07]. Moreover, the communication cost parameters, which are usually derived from link quality indicators observed in the network, may even account for asymmetric links.

Further, integration with the generic role assignment abstraction is possible: First, the generic role assignment system of Chapter 2 can determine the *suitability* of a node to assume a `server` role based on local properties of the node (e.g., its battery level or its location). The node's suitability can then be encoded into the respective node's opening cost parameter and used as an input to the facility location algorithm. The algorithm will then obtain a configuration in which the most suitable nodes are selected as servers but at the same time total connection costs between servers and clients remain low.

Based on an existing centralized algorithm [JMM+03], we devise equivalent distributed formulations which, to our knowledge, represent the first distributed approximations of the facility location problem that can be practicably implemented in multi-hop networks. Although the distributed re-formulation requires a high worst-case runtime, we demonstrate through experimental evaluation that in typical instances derived from sensor-network configuration problems the algorithms terminate in only few communication rounds, the runtime does not increase with the network size, and finally, that our implementation requires only local communication confined to small network neighborhoods [FR07].

1.4 Query Scoping

Similar to Chapter 3, Chapter 4 also deals with a specific subclass of role assignment problems. This class requires selecting a set of nodes as data sources which are likely to provide useful data on a phenomenon of interest.

More specifically, in the considered applications, configuration must take place every time users issue a query for certain information to the network. The approach is based on the observation that, in such settings, it is prohibitively inefficient to send a user query to all nodes of a large network or, alternatively, to let all nodes report all their readings to the network base station to be queried later by users. Instead, a certain subset of nodes must be activated for a given sensing task, nodes which, based on their runtime properties, are likely sources of the desired information [FRNK06]. For such settings, we propose a *query scoping* system

which chooses a set of nodes which are expected to adequately cover a phenomenon of interest.

Note the analogy of query scoping and the *coverage* problem which we used as an introductory role-assignment example: While the latter chooses a subset of all nodes to cover a selected *area*, the former chooses a subset of all nodes that are likely to cover the desired *information*, for example, by determining the area that must be covered to obtain this information.

In terms of a role-based configuration problem, it is required that the roles `sensing` and `idle` are assigned to certain nodes based on their runtime properties. Once a query scope is defined, e.g., consisting of nodes with role `sensing`, it can be used to limit the propagation of certain queries.

Compared to other role assignment tasks, which are performed in a distributed manner by every node inside the network, query scoping is particularly beneficial if it can be performed *offline* at the network base station, which allows to leave `idle` nodes completely unscathed by the query (as otherwise, the effort of letting every network node evaluate whether it is a `sensing` node would be comparable to distributing the query to all nodes).

As a motivation, in Chapter 4, we present an object search system in which query scoping is a prominent challenge [FBRK07]. The application allows users to locate everyday items by means of a large array of sensor nodes. Sensor nodes are implemented by means of mobile phones, which are augmented with the capability to detect the presence of electronically tagged items in their vicinity, and thus act as *object sensors*.

In the context of this application, we provide an approach for using an arbitrary dataset of application knowledge to assign the roles `sensing` and `idle` to sensor nodes. Criteria for selecting sensing nodes include past reports on certain objects or the mobile network cell to which nodes (implemented by mobile phones) are associated at a certain point in time. In the considered case study, this information is in fact available at the network base station and therefore it is feasible to perform query scoping offline before a query is sent.

The presented approach exports yet another parameterization interface to the application developer or user. The query scoping algorithm takes as input a data model of the application domain (e.g., consisting of users, their current and previous locations, associations between befriended users, or previous object “sightings”) and uses them to return a set of

`sensing` nodes which are likely to find a certain object. By means of weight parameters that can be annotated to the edges of the data model, the user may specify which strategies the algorithm should use for query scoping.

In this contribution, a configuration abstraction that is custom-tailored to the application domain was required. Yet, a suitable abstraction level, lying close to the application domain, and effective and efficient means for its implementation were found. Although rather specialized, the presented contribution underlines the claim that role-based configuration is a prominent problem class in wireless sensor networks for which adequate system support should be defined.

1.5 Background

In this section, we place the three contributions of this thesis, summarized above, in the context of related literature. Although specific related work will be discussed next to each contribution in the respective chapter, in this section we attempt a more high-level view on classes of related literature. Depending on the taken point of view, these may be associated with various research domains as shown in Figure 1.1.

On the one hand, generic role assignment provides a *programming framework* for wireless sensor networks. We will discuss programming frameworks in detail in Section 1.5.1. At the same time, it can be viewed as a highly parameterizable *network configuration algorithm*, for which the role assignment specification provides a succinct parameterization interface. While all three contributions share this dual view, we will argue that the provided contributions are more *generic* than existing literature on configuration algorithms in Section 1.5.2. Finally, as the assignment of a role to a node usually implies assigning a certain task to the node (or, vice versa, allocating the node's resources for a task), our contributions on role-based configuration are related to approaches for task assignment and resource allocation, as we will describe in Sections 1.5.3 and 1.5.4.

1.5.1 Programming Frameworks

We subclass approaches that provide programming support for wireless sensor networks using two attributes: The first is their *comprehensiveness*. *Comprehensive* programming frameworks focus on replacing existing low-level programming approaches. In this regard, their aim is to

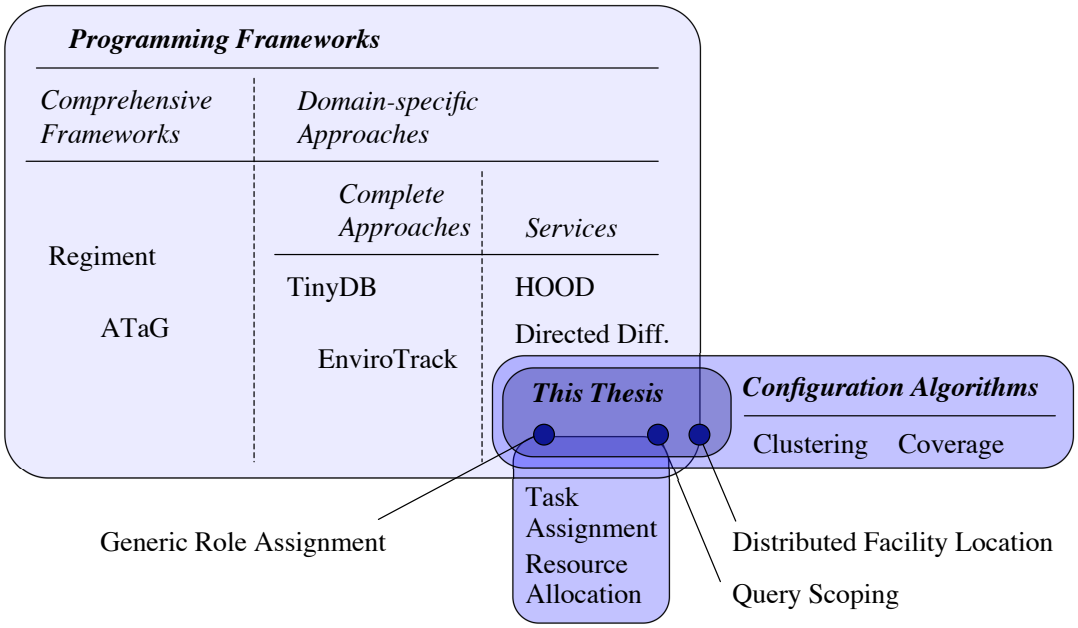


Figure 1.1: Classes of background literature

allow developers to formulate every conceivable application. In contrast, *domain-specific* approaches are custom-tailored to a certain sub-class of applications or functionality which they support.

Domain-specific approaches have two subclasses, regarding their *completeness*: *Complete* approaches allow to specify an entire application in all of its parts (comprehensive frameworks are typically complete). In contrast, *services* only support certain functionality that re-occurs in a range of applications – yet, an application cannot be composed of a properly parameterized service alone.

We will in the following give examples from the resulting three classes of literature: Comprehensive (and complete) programming frameworks, complete domain-specific approaches, and services.

An orthogonal dimension, discerning approaches in each of the above classes, is the provided level of abstraction, that is, the degree to which implementation details are hidden from the application developer by high-level primitives. A high level of abstraction has the purpose of a reduced programming effort, but (usually) comes at the cost of lower *expressiveness* of the provided programming interface. For example, the fact that message passing details of an application are dealt with by a programming framework might involve the drawback that application-specific optimizations of the messaging protocol cannot be expressed in such frameworks. This is why the provided level of abstraction level varies among the frameworks mentioned in the literature.

Comprehensive Programming Frameworks

The focus of comprehensive programming frameworks is to replace existing sensor network programming approaches, and, typically, allow developers to implement every conceivable application using the provided framework.

The front-end of such programming frameworks are *programming abstractions*, which define the set of primitives that can be used by developers to formulate the task at hand. The user program is then translated, either by a compiler or by adequate run-time support, into node-level instructions that aim to implement the semantics of the program efficiently.

At the highest level of abstraction, so-called *macroprogramming* approaches shield developers from the complexity of programming individual nodes. Instead, developers issue high-level programs that specify the desired global behavior of the application in terms of resources of the network. Typically, such programs make use of various high-level constructs such as data structures managing sets of nodes, variables implicitly shared among nodes, and specifications of the global data flow required for the application, which, so the authors argue, can more effectively express the desired application logic than node-level programs. These programs are then translated into a set of commands for individual nodes.

Typical macroprogramming systems are Regiment [NW04, NMW07] – which uses functional programming constructs that operate over individual events, over streams of events, and over groups of nodes that generate them – and the *Kairos* [GGG05] system, which provides a procedural interface for specifying global application behavior. Other examples are ATaG [BPRL05], whose specification combines custom-implemented modular tasks with a graph-based specification of the flow of information between tasks, and *spatial programming* [BIK+04] which allows programmers to express the desired computation in terms of geospatial references to the resources of the network.

At lower levels of abstraction, programming frameworks typically provide a greater expressiveness but come with increased programming complexity. In such frameworks, it is often required to program on the level of individual nodes, for example, one must specify which data a node should sample and which messages to send or receive. One such example is SensorWare [BS03], whose input are node-level scripts supported by a script migration and replication middleware. A second example is FACTS [TWS06], in which developers specify data-processing rules con-

sisting of trigger conditions (usually in terms of incoming data) and associated actions (a set of operations manipulating or sending data once the rule fires).

On the lowest and most expressive abstraction level, several approaches provide novel node-level programming paradigms [KR05], intermediate languages [NAW05], or virtual machines [LC02] that improve over certain deficiencies that are present in the most wide-spread node-level operating systems such as [TOS]. Such approaches may also serve as compilation targets onto which compilers map more high-level abstractions. Other examples, Impala [LSZM04], SOS [HKS+05], or Contiki [DGV04], provide "enhanced" operating systems integrating node-level programs with middleware support, e.g., for component management and code-deployment.

Complete Domain-specific Approaches

Domain-specific approaches, in turn, can be divided up in two subclasses (cf. Figure 1.1) . In the first class, *complete* approaches allow to specify applications in all their parts.

Perhaps the most wide-spread members of this class provide a database view on sensor networks and on the data they sample (e.g., Cougar [BGS01] or TinyDB [MFHH03, MFHH05]). By means of a declarative query interface, these systems support various classic data gathering applications by allowing users to subscribe to a desired stream of sensor data.

Complementary systems focus on supporting various other application classes. For example, DSWare [LSS03] allows users to specify events that should be detected by a network (such as a fire), IrisNet [GKK+03] adds functionality for managing data generated by sensor networks interconnected by a wide-area backbone, and, finally, EnviroTrack [ABC+04] provides primitives for implementing applications that deal with tracking moving real-world phenomena. Recently, EnviroTrack has been embedded into a more comprehensive programming framework called *EnviroSuite* whose abstractions revolve around real-world entities observed by the network [LAHS06].

Services

In contrast, a second subclass of domain-specific systems, which we refer to as *services*, supports certain *functionality* that manifests itself in many

applications. While a wider range of applications may benefit from such services, these cannot be used to specify an application *completely*. This is the case even when a service is rather generic and exports a certain programming interface for specifying its behavior. Consequently, some parts of the application must either be implemented on the operating system level or provided by complementary services.

Compared to the previous two classes, the benefit of employing services for developing wireless sensor network applications is twofold. On the one hand, composing an application from a choice of services allows application developers to benefit from high-level abstractions for addressing certain functionality (if the service's abstraction semantics match the developer's intention well) but at the same time they are free to revert to any lower but more expressive level of abstraction for other domains of functionality (e.g., implementing a novel localization protocol that is particularly suitable in the application or a driver for a specific sensor board). That is, integrating services into an application (or, alternatively, into a programming framework) provides developers with increased flexibility, as applications can be programmed on a *heterogeneous* abstraction level, custom-tailored to an application's needs.

Generic role assignment belongs to this class: It supports developers solely with network configuration tasks. By means of role specifications, developers may specify the desired network structure. The rest of the application is expected to be implemented by custom-made software components. An event-based architecture provides the glue between the role assignment system and other application components executed on the nodes.

As a member of this class, generic role assignment co-exists with other approaches that provide services addressing various domains of functionality on various abstraction levels. For example, neighborhood management services such as HOOD [WSBC04], but also [WM04, MP06, SFCB04], alleviate programmers from the details of message passing and data sharing within certain local network neighborhoods. Each of the above includes functionality to manage the membership in these neighborhoods, a functionality which is often referred to as *scoping*.

Such scoping is often provided as a distinct service, which can be implemented either topologically, e.g., by constraining neighborhoods to a certain number of hops in the network communication graph or by providing custom-implemented neighborhood definitions such as trees [WM04], or generically, e.g., based on conditions formulated in terms of various other

node properties that define a node's membership in a certain neighborhood [SFCB04, MP06]. The fact that the latter approaches use techniques that are similar to role assignment to evaluate a node's membership in the scope (dual to the way role assignment evaluates the node's membership in the set of nodes which are assigned a certain role) implies that generic role assignment can also be viewed as a scoping approach. Our query scoping contribution (described in Chapter 4) provides a similar service but for a different class of applications which require that scoping decisions are performed *before* network nodes (in particular nodes outside of the computed scope) are involved in any communication or computation.

Other examples of this class are the TinyLime [CGG+05] middleware providing the notion of a transparent tuple space in which sensor data can be stored. Similarly, geographic hash tables provide for in-network storage of gathered data [GEG+03]. The seminal Directed Diffusion [IGE00] system provides a content-based routing service allowing nodes to subscribe to arbitrary data provided by the network. A subscription, so-called *interest*, specifies the data that should be delivered at the node by means of attribute value pairs.

Numerous other services exist in the literature supporting various functionality. We used the above services as examples, as they (like generic role assignment) export programming abstractions which can be used to implement a certain behavior of the service. These abstractions, from a different point of view, provide efficient and succinct interfaces for parameterizing the execution of the provided service. The underlying implementation is a highly parameterizable configuration algorithm.

1.5.2 Configuration Algorithms

In contrast to our generic approaches, many algorithms address *specific* role-based network configuration problems. We have mentioned clustering [Lub85, HCB00, KG02], data aggregation [BB03], and coverage [XHE01, MKPS01, GZDG05].

Apart from role-based configuration, many other specialized configuration algorithms exist. Examples include channel assignment [Ram99] (based on distributed graph coloring approaches) and topology control algorithms [WZ04, San05] (which regulate the transmit power used when communicating with neighbors).

Some of these algorithms provide interfaces to parameterize their execution. These interfaces can differ in their expressiveness. For example,

[KG02] allows to set two parameters which determine the number of gateway nodes which are used to join clusterheads into a connected topology.

Compared to existing role-based configuration algorithms, the three contributions of this thesis provide more powerful parameterization interfaces which make them applicable to a wider range of configuration tasks. Making use of the provided interfaces, developers may install one algorithm and afterwards rapidly re-configure the network's behavior in a variety of ways by issuing new parameters to these algorithms.

Each contribution targets a different application domain and therefore provides a different parameterization interface. The generic role assignment framework, described in Chapter 2, extracts the parameters determining algorithm execution from a given role assignment specification. The distributed facility location algorithm, described in Chapter 3, is perhaps the most specific configuration algorithm. Its parameterization is performed by adequately chosen opening costs (assigned to nodes) and connection costs (assigned to links). Finally, the query scoping system of Chapter 4 uses an annotated data model for parameterizing its execution.

1.5.3 Task Assignment

Last but not least, we would like to mention a couple of adjacent research areas out of which some aspects are addressed by the contributions provided in this thesis.

For example, we mentioned in the above that it is sometimes desirable to obtain a role assignment that minimizes the overall costs of communication between nodes of different roles. Our distributed facility location algorithms, for instance, select a cost-optimal set of *servers* while taking into account the cost of communication between clients and their closest server. More generally, one might consider taking larger structures of dependencies between roles into account. For example, the placement of different aggregator roles should minimize total communication cost of the whole aggregation tree.

In the distributed systems field, this problem setting is known as *task assignment*. The task assignment problem is motivated by the need to distribute a set of program modules (or computation tasks) to a set of processors while minimizing the cost of communication between them. Our systems also assign modules to sensor nodes by means of the role abstraction: The assignment of a role to the local node typically triggers the execution of program modules previously associated with the role.

Solving the task assignment problem for arbitrary communication dependencies between different modules is hard. An early approach [Sto77] described a set of centralized network-flow-based algorithms that solved task assignment in polynomial time for two processor nodes (processors correspond to sensor nodes in our models). While flow-based techniques were extended to three processors, for four and more processors the problem was shown to be NP-hard [Bok87]. What proved to be easier is solving task assignment when communication dependencies between different modules have the form of a tree. Here, a centralized algorithm [Bok81] solves the task assignment problem based on finding shortest paths in a graph generated from the network communication graph and the dependency graph between modules.

An example task assignment approach adapted for wireless sensor networks is [BB03], which assigns a tree of query operators (generated from a declarative database query) to network nodes. The approach attempts to minimize data traffic between operators – and therefore provide for efficient query execution. It is based on a hill-climbing technique that, given an initial placement of the operator tree, moves operators to neighboring nodes if these can perform the operator function more efficiently.

Similarly, the MagnetOS system [LRW+05], aims to assign the modules of a global program to the nodes of a network graph. The approach evaluates various heuristics to place modules. Such heuristics include to move a module to the neighbor with which it communicates most, or to a more suitable node based on a partial view of the network aggregated at the local node.

1.5.4 Resource Allocation

The *resource allocation* problem [Lyn81] goes beyond selecting nodes for executing certain modules but allocates goods (such as sensor time on a node, node computation, or bandwidth) for a certain global task. Similarly, role assignment can be considered to allocate a node's processing power, sensors, and transceiver to a certain task implied by the role.

One example of a resource allocation approach [MPW05] for sensor networks uses virtual-market oriented heuristics for this purpose building on similar approaches for distributed systems [Cle96]. Here, nodes attempt certain actions (such as listen, sense, or send) each of which are associated with a certain cost in terms of consumed energy – and a certain reward if the taken action was successful. Developers may parameterize

the reward for each action, which, as nodes aim to optimize their reward based previous actions, provide high-level levers for controlling the network's behavior.

Summing up, both, *task assignment* and *resource allocation* approaches explore functionality complementing the role-based configuration approaches considered in this thesis. Task assignment additionally optimizes the configuration based on the communication involved between nodes with different roles. Resource allocation expands the considered functionality in the time dimension, as it allows to assign node resources to different tasks over time.

While such functionality is intriguing, both problem classes are very demanding in terms of computing complexity. For example, the above approaches for wireless sensor networks [MPW05, BB03] both involve resource-intensive trial phases in which improvements of the current solution are explored. In [BB03], all neighbors of an operator simulate the operator's function (including the involved communication) in order to evaluate the benefit of a possible re-location of the operator. In [MPW05], nodes "attempt" certain actions at random in order to evaluate whether the execution of such actions is beneficial.

1.6 Summary

In this chapter, we have provided an introduction to the three contributions of this thesis and overviewed the corresponding background literature.

In summary, we claim that role-based network configuration is an important challenge which re-appears in heterogeneous wireless sensor network applications and network models. For the domain at hand, suitable configuration algorithms must be found. This thesis therefore provides effective system support for such role-based configuration, first in a generic approach and then for two subclasses of problems. In its contributions, this work sheds light on a range of configuration approaches, each consisting of a parameterization interface and an underlying implementation. In this, the presented thesis examines three different points in the trade-off between, on the one hand, the simplicity and the expressiveness of the configuration interface and, on the other hand, the performance and the efficiency of the underlying implementation. Taken together, the provided configuration support represents a step forward on the path towards plug and play sensor network systems. Such improved usability, often

presumed a key prerequisite to wide-spread usage of wireless sensor networks, would finally enable the economies of scale commonly associated with the wireless sensor network vision.

Parts of this thesis have been published in proceedings of international conferences and workshops, most notably in [RFMB04, FR05, FR06, FR07, FRNK06, FBRK07].

1.7 Outline

This thesis is structured as follows. We first present the role assignment abstraction, its implementation, and its extensions in Chapter 2. In the following two chapters, we then address two subclasses of role assignment problems. In Chapter 3, we describe an approach concerned with computing near-optimal clustering decisions in a distributed manner. In Chapter 4, we discuss an approach for offline query scoping in a large-scale mobile-phone-based sensor network.

A set of aspects will re-appear in each of these chapters: The *interface* of the configuration service, *examples* for its use, the algorithms that constitute the *implementation* of this interface, and an *evaluation* regarding the efficiency and feasibility of the approach.

Based on the contributions detailed in Chapters 2, 3 and 4, we extract common guidelines for the development of future programming support for wireless sensor networks and conclude our work in Chapter 5.

2 Generic Role Assignment

In this chapter, we present a programming framework called *generic role assignment* that allows users to define a list of roles which will be assigned to sensor nodes if certain conditions are met. Above, we mentioned that several classic network configuration problems can be viewed as instances of generic role assignment. As an introduction to this chapter we sketch three role assignment problems and list the involved roles together with possible conditions for their assignment.

Coverage. A certain area is said to be covered if every physical spot falls within the observation range of at least one sensor node. In dense networks, each physical spot may be covered by many equivalent nodes. The lifetime of the sensor network can be extended by turning off these redundant nodes and by switching them on again when previously active nodes run out of battery power [XHE01]. Essentially, this requires proper assignment of the roles `ON` and `OFF` to sensor nodes.

Clustering. Clustering is a common technique to improve the efficiency of data delivery (e.g., flooding, routing) [KG02]. With clustering, one of the three roles `CLUSTERHEAD, GATEWAY, SLAVE` is assigned to each node. A clusterhead acts as a hub for slaves in its neighborhood such that slaves directly communicate with their clusterhead only. Gateways are slaves of more than one cluster and interconnect multiple clusters by forwarding messages between them.

In-network Aggregation. Due to the scarcity of energy and the high energy cost of wireless communication, reducing data communication is an important design goal in sensor networks. One common form of data reduction is *in-network data aggregation*, where certain nodes in the network aggregate sensory data from many sources [IGE00]. For this, sensor nodes must be assigned the roles `SOURCE` (generate sensory data), `AGGREGATOR` (aggregate data), and `SINK` (consume aggregated data) roles. In order to achieve a significant network traffic reduction, aggregator nodes should be located close to the data sources they aggregate.

While a number of specialized algorithms for these problems have been developed, these are typically hard to adapt to different applications,

where varying criteria for assigning the above roles may have to be applied. Driven by these observations, our aim is to provide a *generic* role assignment service that is applicable to a wide variety of role assignment problems. We first give an overview of the various components involved in generic role assignment. These components will be described in detail in the remainder of this chapter.

2.1 Overview

Figure 2.1 gives a sketch of the envisioned use case and its core elements. To setup or re-configure the sensor network, the user/developer provides a *role specification* that defines possible roles and rules for how to assign roles to nodes. This specification is distributed to the whole network via a network base station. On the nodes, a *property directory* provides transparent access to node properties and capabilities. A distributed *role assignment algorithm* assigns roles to sensor nodes, taking into account role specifications and node properties. Finally, applications on the node access node properties (including the node's role), which may trigger execution of role-specific code. For example, the assignment of the role `clusterhead` would trigger the start of an additional routing component on the node.

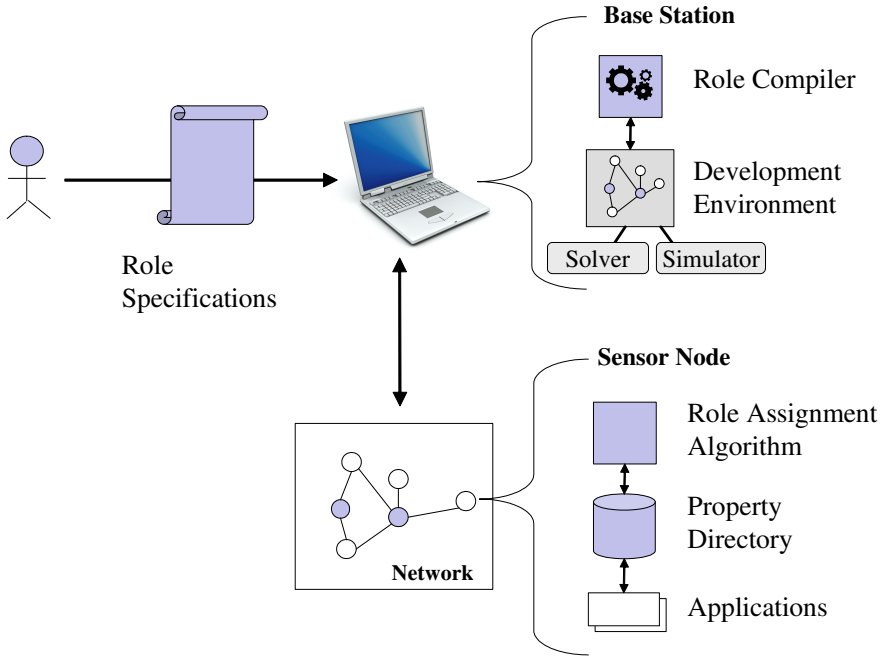


Figure 2.1: Core elements of generic role assignment

Property	Value
battery	50%
pos	(12.3, 3.4)
temp-sensor	true
neighbors	7
role	ON

Table 2.1: Example contents of the property directory

Property Directory. Properties of individual sensor nodes include the available sensors (e.g., temperature) and their characteristics (e.g., resolution), other hardware features (e.g., memory size, processing power, communication bandwidth), the remaining battery power, and the nodes' physical location and orientation. Some properties are static, some may change over the lifetime of the network. However, we assume that properties are not subject to frequent significant changes. This reflects the understanding that a particular configuration is valid for a certain minimum amount of time. Depending on their nature, properties may be defined at production time, by hardware introspection, or by actual sensor data. The property directory provides a unified interface for accessing property values. There is one such directory on each sensor node, which is independent of the directories on other nodes.

As described, the property directory is not specific to the role assignment task, but a general component facilitating cross-layer interaction among software components. In our implementation, the property directory exports property values as a list of name-value pairs. Moreover, it can issue asynchronous notifications to other software components on the node whenever property values change. Our implementation supports numeric and Boolean types, node positions, and sets of node identifiers.

A set of example contents of the property directory is shown in Table 2.1. The current *role* of the node, and other information acquired during the role assignment process (e.g., topology information such as number of neighbors), are also treated as node properties. Applications can subscribe to role changes or to other properties of interest and customize their behaviour based on the assigned role. Inversely, applications may update entries in the property directory, which would notify the role assignment algorithm to adapt assigned roles accordingly.

Role Specification. In its basic form, a role is an identifier (e.g., `CH` for clusterhead, `GW` for gateway). The role specification is a list of role-rule pairs. For each possible role, the associated rule specifies the conditions for assigning this role. Rules are Boolean expressions that may contain

predicates over the local properties of a sensor node and predicates over the properties of well-defined sets of nodes in the neighborhood of a sensor node. All nodes in the network have a copy of the same role specification. This reflects the understanding that all sensor nodes are in the same initial software state. Detailed role specification examples will be given in Section 2.3.

Role Assignment Algorithm. This component assigns roles to sensor nodes, taking into account role specifications and node properties. Depending on the specific problem instance, it might be useful to allow the assignment of multiple roles to one node. For example, a single sensor node might act both as a data source and as an aggregator. Property changes and node failures may necessitate re-assignment of roles.

A separate instance of the role assignment algorithm is executing on each sensor node. Triggered by property and role changes at nodes in the network neighborhood, the algorithm evaluates the rules contained in the role specification. If a rule evaluates to true, the associated role is assigned. We discuss such algorithms in Section 2.4.

Role Compiler. The above algorithm can be considered a template that must be properly parameterized for a specific role assignment task. This parameterization is carried out by a compiler at the network base station, which reads a role specification and outputs appropriate parameters for the role assignment algorithm. These parameters are then encoded in a role specification message that is sent to all nodes in the network.

Development Environment. Integrated with the role compiler, the network base station executes a centralized development environment allowing developers to analyze role specifications before these are distributed to a network in operation. On the one hand, the development environment includes a network simulator which can be used to test the distributed algorithm's performance with a given specification. On the other hand, it contains a solver component, which may be used to check the feasibility of a given specification and, further, to compute *optimal* configurations in which the use of certain roles is either maximized or minimized.

The solver component is mainly used for specification analysis. While it may as well compute role assignments for actual networks (as a replacement of the distributed algorithm), this is only sensible for small networks with static properties, as it requires that a global view on the network topology and node properties is collected at the network base station.

Basic Services. A number of basic services such as node localization, neighbor/topology discovery, or time synchronization may add valuable information to the property directory. The availability of such services could also be a node property. These services are decoupled from the rest of the system through the property directory.

The remainder of this chapter is organized as follows. After surveying related background literature in Section 2.2, we introduce a language for specifying roles in Section 2.3. Section 2.4 presents distributed algorithms for role assignment. The development environment and the implementation of the distributed algorithms is discussed in Section 2.5. We evaluate the efficiency of these algorithms quantitatively in Section 2.6 and qualitatively, by comparing the semantics of our role assignment system to specialized role assignment implementations from the literature, in Section 2.7. Finally, we present the role assignment solver in Section 2.8 and provide an outlook to an extended role specification syntax in Section 2.9.

2.2 Related Work

Self-configuration in ad hoc and sensor networks has been an active research topic in the recent past. Various other approaches for solving *specific* self-configuration problems have been devised. Examples include coverage [SP01]; aggregator placement [BB03]; clustering, routing and addressing [KSG03, SAGP00, SK00]. [KSG03] uses a fixed set of roles to build a network-wide backbone infrastructure. However, none of these approaches are *generic* frameworks that support the assignment of user-defined roles in an application-specific manner.

The concept of role assignment has been mentioned in various research projects related to wireless sensor networks. In [HMCP04], a middleware called MiLAN is outlined that controls the allocation of functions to sensor nodes in order to meet certain quality-of-service requirements specified by the user. In [MLM+05], a cross-layer framework called TinyCubus is presented that uses the notion of roles to implement efficient code deployment. In [UWMG05], a high-level programming approach for sensor networks is presented, where a high-level task specification is compiled into a set of node-level programs that must be properly allocated to sensor nodes taking into account the node capabilities.

Moreover, neighborhood programming abstractions [WM04, WSBC04] have been proposed, where network neighbors can eas-

ily share variables among each other. These abstractions pose an interesting opportunity for implementing our role assignment approach. In particular, each node in the network could set up a "sharing region" in order to exchange property values among nodes.

Inspired by cellular cooperation in biological organisms, Amorphous Computing [AAC+00] explores ways to program smart matter – very densely deployed collections of indistinguishable smart particles. In contrast, our approach is based on the observation that sensor nodes may significantly differ in their properties, may rely on a number of basic services (e.g., localization), and are less densely deployed. Also, we focus on the configuration of sensor networks, the actual "programming" (i.e., distributed data processing etc.) is not part of our work, although roles and other property values derived during role assignment may provide valuable input.

Our scheme for role assignment is somewhat similar to cellular automata [Wol94], where the state of a particle in a regular arrangement is completely defined by the previous values of a neighborhood of particles around it. Note that a classification of a subclass of cellular automata in [Wol94] indicates that a large group of automata converges to well-defined states. Major differences of our approach are that state updates are not synchronous, sensor nodes are not in a regular arrangement, and sensor nodes differ in their properties.

The role assignment solver complements the distributed role assignment algorithms by using integer linear programs (ILPs) to compute exact solutions to generic role assignment problems. ILPs have been applied to network configuration problems in various settings. In [BC02], the authors derive upper bounds on the lifetime of data-gathering networks by computing optimal configurations consisting of the roles *sensor*, *relay*, and *aggregator*. Moreover, integer program formulations have been used as a starting point for developing distributed approximation algorithms which solve problems that are related to our specifications, most prominently the minimum dominating set [KW03] and the facility location problem [MW05]. The latter approach [MW05] addresses an important subset of role assignment problems and will be discussed in detail when we address these problems in Chapter 3. Compared to these approaches, the role assignment solver provides a generic framework for translating *any* role assignment problem into an ILP.

2.3 Role Specifications

In this section we introduce the notation for *role specifications*. We first show how this approach can be used for a number of applications, later we will review the essential specification components in more detail.

2.3.1 Application Examples

Let us revise the examples sketched in the introduction into more formal specifications. Note that these role specifications will typically result in approximate solutions of the respective configuration problems.

Coverage. As mentioned earlier, nodes must be assigned `ON` and `OFF` roles. Requirements for the assignment of these roles are that the area of interest is covered by the sensors of `ON` nodes, and that `ON` nodes have sufficient remaining battery power. Assuming one is interested in coverage with temperature readings, one possible formulation could be:

```
1 ON :: {
2   temp-sensor == true &&
3   battery >= threshold &&
4   count(2 hops) {
5     role == ON &&
6     dist(super.pos, pos) <= sensing-range
7   } <= 1 }
8 OFF :: else
```

The rule in lines 1-7 specifies the conditions for a node to have `ON` status: it must have a temperature sensor and enough battery power (lines 2 and 3). As a third condition, we require that at most one other `ON` node should exist within this node's sensing range. This is specified by the `count` operator in line 4. It expects a hop-range as its first parameter and returns the number of nodes within this range for which the expression in curly braces evaluates to true. Here we request to evaluate nodes within 2 network hops. Note that the used property names (e.g., `role` in line 5, `pos` in line 6) in the nested expression refer to properties of the specified neighboring nodes. To access properties of the local node instead, the prefix `super` is used (e.g., `super.pos` in line 6). The `dist` operator used in line 6 returns the metric distance between two positions. In the example, it specifies that only nodes located within this node's sensing range should be counted.

In other settings it would be useful to retain state on network neighbors instead of just counting them. Clustering is such an example.

Clustering. A clustering approach needs to define assignment rules for clusterhead (`CH`), gateway (`GW`) and `SLAVE` roles. The assignment of these roles depends on a variety of parameters. Clusterheads should be more powerful devices (in terms of processing, memory, communication, and energy supply), because they act as hubs for many slaves. This may be easily formulated in terms of the property directory and is neglected here. For a role specification, consider the following basic scheme:

```
1 CH :: {
2   count(1 hop) {
3     role == CH
4   } == 0 }
5 GW :: {
6   clusterheads == retrieve(1 hop, 2) {
7     role == CH
8   } &&
9   count(2 hops) {
10    role == GW &&
11    clusterheads == super.clusterheads
12  } == 0 }
13 SLAVE :: else
```

A node that does not have any clusterhead among its neighbors declares itself clusterhead (`CH`, lines 1-4).

Nodes should be assigned the role gateway (`GW`) if they are neighbors to at least two clusterheads but are not aware of any other gateway nodes interconnecting the same two clusterheads.

To achieve this, we introduce the `retrieve` operator (line 6), which is similar to `count`, but returns a set of node identifiers instead of only counting the nodes. In this example, the retrieve operator is used to identify clusterheads in the 1-hop neighborhood of the node and to bind them to the local property `clusterheads` in line 6 (similar to binding of variables in declarative programming languages). Using the `clusterheads` property, we require in lines 9-12 that within 2 hops no other gateways should interconnect the same set of clusterheads.

The second parameter to `retrieve` in line 6 requests any *two* matching nodes. If not enough matching nodes exist, the `retrieve` expression evaluates to false. In this case, the `GW` role is not assigned, the property `clusterheads` remains undefined, and the evaluation of lines 9-12 can be omitted.

In-network Aggregation. Similar rules can be designed for an exemplary application requiring in-network aggregation.

```
1 AGG2 :: { analogous to AGG1 }
2 AGG1 :: {
3   count(2 hops) {
4     role == SOURCE &&
5     dist(pos, sink-pos) >
6       dist(super.pos, sink-pos)
7   } >= 2 &&
8   count(2 hops) {
9     role == AGG1
10  } == 0 }
11 SOURCE :: { temp-sensor == true }
```

In this example, sensor nodes equipped with temperature sensors act as data sources (line 11). A single sink node with known position *sink-pos* consumes aggregated data. Aggregator nodes (`AGG1`) should be placed in the close neighborhood of many sources (line 4) compared to which the aggregator is closer to the sink (lines 5-6) because data flows from sources to the sink. We furthermore require that no other nodes with role `AGG1` should exist within two hops.

Note that we used a second role `AGG2` for aggregators of higher level which aggregate information from nodes with role `AGG1` instead of sources. `AGG2` nodes should be similarly placed between the `AGG1` nodes and the sink and no other `AGG2` nodes should exist in their 2-hop neighborhood.

2.3.2 Syntax and Semantics

Let us review the specification techniques introduced in the examples. A role specification consists of a list of *roles* and associated conditions involving the values of local properties of a sensor node or the properties of well-defined sets of nodes in the neighborhood of the node. The conditions for a role k are determined by an associated role predicate c^k. We assume c^k has been preprocessed by the role compiler and rearranged into its disjunctive normal form:

$$c^k = (c^k_{11} \wedge \cdots \wedge c^k_{1n_1}) \vee (c^k_{21} \wedge \cdots \wedge c^k_{2n_2}) \vee \ldots \tag{2.1}$$

Three types of *atomic predicates* c^k_{ij} are supported:

Simple predicates are essentially Boolean expressions formulated in terms of properties and constants, possibly involving basic arithmetic operations.

Count predicates of the form

```
count(scope) { pred } rel const
```

can be used to count nodes that match a nested predicate **pred** within a given number of hops **scope** around the current node and compare the result to a constant expression **const** using a given relation **rel**.

Retrieve predicates are similar, these have the form

```
p == retrieve(scope,size) { pred }
```

and can be used to bind the identities of a set of nodes matching **pred** to a local property variable **p**. A parameter **size** specifies that at least **size** matching nodes must exist, otherwise the predicate evaluates to false. After evaluation, **p** contains the identifiers of the matching nodes and can be used as a local property.

Within count and retrieve operators, the nested predicate **pred** specifies the conditions under which a remote node is counted or retrieved, respectively. These conditions are arranged in a disjunctive normal form in which, essentially, only *simple predicates* are allowed. Because the properties used in **pred** generally reference property values of remote nodes, it is furthermore possible to pre-pend `super` to property names to reference properties of the current node instead.

As mentioned earlier, the property directory supports numeric and Boolean types, node positions, sets of node identifiers, and the enumeration `role`. When comparing node property values, *equality* is supported for all properties, while the usual ordering relations (such as $<$, $\leq$ etc.) are additionally available for all numeric properties.

Note that because retrieve predicates bind local properties which can be referenced by other *count* and *retrieve* statements, the former must be evaluated before predicates referencing the bound value. Moreover, the specification must not contain circular dependencies between any two retrieve statements that are part of the conditions of any role. This is checked by a compiler before sending the specification to the nodes. In Section 2.4, we describe distributed algorithms that can be used to implement the semantics described above.

The presented specification language obviously cannot capture all conceivable role assignment problems (see also Section 2.7 on this issue). However, from our experience it can be used to implement practical approximations of many configuration tasks. Moreover, our approach can

be extended in two ways to be more powerful: Firstly, custom predicates (such as the `dist` operator mentioned earlier) can be implemented by the programmer to support complex role assignment tasks. Secondly, applications may subscribe to certain role changes and changes of other properties. When notified of such a change, the application may perform any computations that cannot be expressed directly with the role specification language. In addition, the application may set values in the property directory, triggering the role assignment algorithm to revise role assignments to take into account the modified properties.

Generally, the role assignment abstraction enables the programmer to address configuration problems based on rather stable network properties (where there is a limit on the frequency of changes of properties on which the configuration decisions are based). In order to perform configuration, any algorithm that implements role assignment will have to reach consensus on the current state of the network within local network neighborhoods around each node.

Crucial for the locality of any role assignment algorithm is the given *scope* that is used in count and retrieve statements because it governs the degree of interdependence between nodes. We therefore define the *maximum scope* of a specification:

Maximum Scope. The *maximum scope* of a given specification is the highest hop-number used as a scope for count and retrieve statements.

Similarly we introduce a term for the set of nodes that can influence the role selection of a given node:

Critical Area. The *critical area* of a node u is the set of nodes v with

$$0 < d(u,v) \leq \text{maximum scope}$$

where $d(u,v)$ is the hop-length of the shortest path between u and v.

Next to the examples of Section 2.3.1, a range of basic building blocks for network configuration can be specified using the presented role specification language. For example, a tree rooted at the sink is easy to specify using adequately defined `CHILD` roles that set a node property `parent`. Moreover, it is possible to formulate NP-hard configuration problems using role assignment specifications, which implies that a role assignment cannot be computed within a guaranteed polynomial runtime. We will list such role specification examples in Section 2.9.

In the following, we describe a set of practical heuristics for role assignment which are efficient in the average case. The overhead of the presented algorithms will later be tested with a range of typical role specifications and network topologies in Section 2.6.

2.4 Distributed Role Assignment Algorithms

We motivated generic role assignment and sketched a possible distributed algorithm that could be used to implement role assignment in a position paper [RFMB04]. This algorithm is based on a fixpoint iteration, where each node would repeatedly fetch the current values of all relevant remote properties in order to evaluate the role predicates, eventually deciding on a (preliminary) role for itself. These evaluation cycles would have to be properly sequentialized among neighboring nodes in order to ensure consistent role assignments. Assuming that there is a fixpoint configuration, each node would end up with a role that does not change in subsequent evaluation cycles.

While this approach works in principle, it turns out that the overhead for locking and unlocking neighbors for sequentialization is prohibitively high. Therefore, we have examined more efficient algorithms that *proactively* distribute property values to neighbors. This is based on the observation that a node can decide which of its property values are relevant to what neighbors, because each node uses the same role specification.

This proactive approach [FR05] eliminates much of the overhead for "locking" nodes and concurrently introduces some redundancy that makes it more robust in the face of message loss. We present the basic algorithm below. Later in Section 2.4.7 we introduce two probabilistic initialization schemes that make it more efficient.

It must be emphasized that the algorithms are *heuristics* which may result in temporary inconsistencies due to the lack of sequentialization. We will employ randomized delays in order to avoid such inconsistencies. If a temporary inconsistency should occur, it will be removed in subsequent evaluations at one of the affected nodes.

The algorithms operate on a given role specification that needs to be distributed to the nodes. Role specifications are encoded into byte-optimized messages at the sink using a syntax tree that is constructed from the given specification. Note that we do not focus on the (reliable) distribution of the role specification. While we have implemented a flooding-based approach, several well-studied protocols for code distribution ex-

ist [LPCS04]. In some contexts, the specification may also be loaded into the node offline before deployment.

2.4.1 Overview

The basic algorithm is built around local *cache tables* maintained at each node, which contain a collection of (local and remote) properties that are relevant for role assignment. Eventually, the node will refer to its cache table to assign its own role, based on the information it has learned about its neighbors up to that time. We will refer to this as *local rule evaluation* as the node does not involve any additional remote data apart from its own cache.

In the upcoming subsections we will discuss how to *initialize* the node's cache table after the specification is received, how to *propagate* node properties that have changed, how to perform the above-mentioned *local rule evaluation*, how to adapt to *changing network properties*, and finally, how to detect *termination* of the algorithm.

2.4.2 Initialization

The initialization of the local cache table is performed upon receiving a new role specification. As all nodes share the same specification, they will essentially setup their tables in the same way.

In a first step, the node extracts the set of relevant properties from the specification. Further, the node uses the specification to infer the distance (in hops) over which each property needs to be propagated at most, which we will refer to as the property's *max* value. Such *max* values are computed from the scope of all count and retrieve operators within which the respective properties occur. Note that the node's role is also a property for which a *max* value can be computed.

If a property is only used in (local) simple predicates, its *max* value is 0 and the property will never be propagated to other nodes. Note that the information on property *max* values is constant for a given specification and their maximum corresponds to the specification's *maximum scope*. All these values are therefore pre-computed at the network base station and included in the specification message that is injected into the network.

Using the above information, a given node A initializes its cache table as shown in Figure 2.2. To illustrate the algorithm, we use a simplified coverage specification – as depicted – where we assume that the node's

sensing range is equal to its transmission range and we specify that no two neighbors are allowed to be ON concurrently.

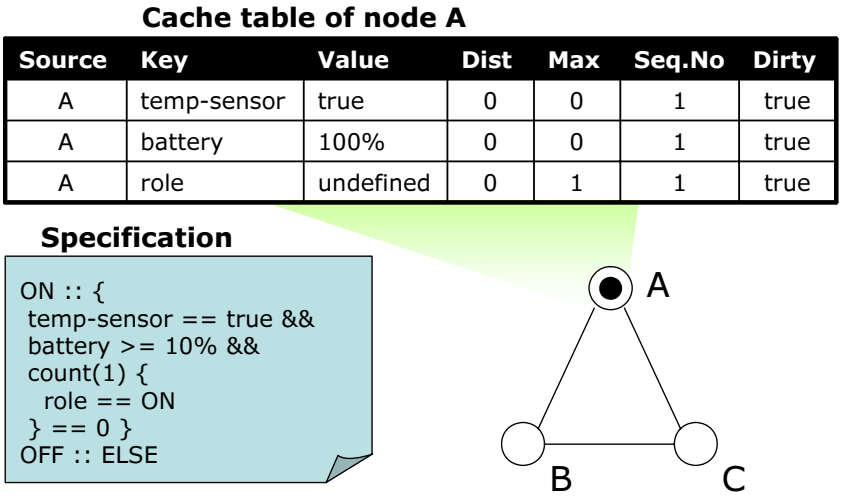


Figure 2.2: Node A after initialization

The table is initialized with a row for each property used in the specification. Each row contains a field denoting the source node (which is A in this case as information stems from A's local properties), the property key, and its current value. Furthermore, a *dist* field denotes the hop-distance between the source node and A. At initialization this value is always 0 because all information is local. The table also shows the *max* values described earlier which indicate how far the properties must be propagated.

Each row also contains a *sequence number*, which every node maintains for its local entries. It is initialized with 1 and increased every time a local property (with $dist = 0$) is updated. Finally, a *dirty* bit specifies whether this information was already propagated to neighbors (false) or not (true). Initially, this value is true.

The cache table – initialized with local properties as described here – contains redundant information that is already present in the property directory. When optimizing for memory, these entries could be generated just-in-time from the local property directory instead. Similarly, the *max* values could be stored only once for each property and *dist* values once for each node.

Once the cache table has been initialized, a node will schedule the execution of three procedures that we will describe in the following subsections (note that all random delays are uniformly distributed in the given intervals).

Property Propagation after a random delay $t_{\text{prop}} \sim (0, T_{\text{prop}})$. The delay helps aggregating newly arriving information into a single message and at the same time spreads traffic over a longer time interval. See Section 2.4.3 for details.

Local Rule Evaluation is scheduled after an initial delay t_{init}. This delay is chosen to allow for adequate property propagation from all nodes in the current node's *critical area*. It is computed from the maximum scope s using an additional random offset $t_{\text{eval}} \sim (0, T_{\text{eval}})$ to reduce the chance of simultaneous role evaluations:

$$t_{\text{init}} = sT_{\text{prop}} + t_{\text{eval}}$$

Failure Detection at regular intervals $T_{\text{heartbeat}}$.

2.4.3 Property Propagation

To transmit properties to its neighbors, a node creates an *update* message, essentially containing a list of cache table rows.

The message is composed from all rows with $dirty = true$ and $dist < max$. Entries with $dist = max$ have reached their maximal scope and need not be propagated any further (e.g., the local properties *battery* and *temp-sensor* in the coverage example). Essentially, the message contains a copy of these rows with the *dist* field increased by 1 and *max* and *dirty* fields left out (*dirty* is true anyway and *max* values can be derived from the specification). Furthermore, the node resets all dirty bits of its table to false. The resulting *update* message is broadcast to all neighbors. Note that property keys can be efficiently encoded as an integer index, because all nodes use the same specification.

The receivers of such *update* messages enter the contained information into their local tables. If entries of the incoming message and the local table refer to the same property of the same source node, the information with the larger sequence number is retained (note that sequence numbers are increased by the source node only, while other nodes forward them unmodified). If the information has taken a shorter path (source and sequence number are the same, but *dist* field is smaller) the *dist* field is set to the smaller value.

On the first incoming *update* message, receivers schedule

Property Propagation after a random delay $t_{\text{prop}} \sim (0, T_{\text{prop}})$ in order to forward new information with $dist < max$.

Local Rule Evaluation after a random delay $t_{\text{eval}} \sim (0, T_{\text{eval}})$. Generally, T_{eval} is chosen to be larger than T_{prop}, see Section 2.4.4 for details.

The property propagation delay t_{prop} fulfills two functions at different layers: Firstly, it is used to smooth out traffic bursts that would occur after a property change that is forwarded over multiple hops. When using a contention-based MAC layer, this would additionally reduce collisions, as transmit attempts are spread over a longer period of time. Secondly, it reduces the number of update messages by collecting many "dirty" table rows into a single message as shown in Figure 2.3.

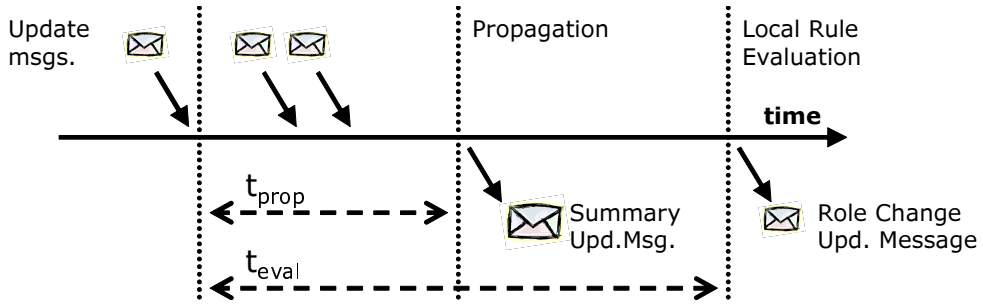


Figure 2.3: Property propagation and local rule evaluation

Our propagation procedure adds some redundancy when information is forwarded over multiple hops because one bit of information is typically forwarded over multiple paths. Yet this redundancy adds significantly to the robustness of the algorithm in face of packet loss. We will examine robustness in Section 2.6.

When the algorithm is used with low-duty-cycle MAC layers, one must ensure that the symmetry-breaking nature of the randomized timers is preserved. This can be achieved either by making active periods sufficiently long or by suspending algorithm timers in sleep periods (and thus stretch the algorithm execution time). What is essential is that the MAC layer does not synchronize property propagation to such an extent that it forces neighboring nodes to propagate past role changes simultaneously.

In our example from Figure 2.2, assume node A has recently updated its *role* property to `ON` and thus also increased the row's sequence number. In A's table, the only row with $dist < max$ and $dirty = true$ is the *role* property. A therefore broadcasts an update message to its neighbors containing its *role* property entry only. Assume that, similarly, node C has sent an *update* message, after it had picked the role `OFF`. Node B's cache table, after receiving update messages from A and C, is shown in

Figure 2.4. In the example, no information needs to be forwarded further: All rows either have $dist = max$ or $dirty = false$.

Cache table of node B

Src	Key	Value	Dist	Max	Seq.No	Dirty
B	temp-sensor	true	0	0	1	false
B	battery	100%	0	0	1	false
B	role	*undef.*	0	1	1	false
C	role	OFF	1	1	2	true
A	role	ON	1	1	2	true

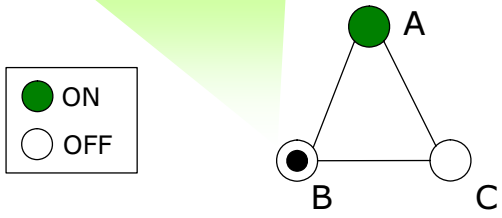


Figure 2.4: Sample cache table at node B

2.4.4 Local Rule Evaluation

As mentioned, local rule evaluation is either triggered by a new role specification, in this case delayed by t_{init}, or by the first update message following a previous local rule evaluation, in this case delayed by $t_{\text{eval}} \sim (0, T_{\text{eval}})$. The timer t_{eval} has two functions: Firstly, it helps avoid simultaneous role evaluations, for which we chose T_{eval} to be relatively large compared to T_{prop} (see Section 2.5 for timer settings). Secondly, it also helps avoid unnecessary transient roles by making a late but informed decision rather than performing many re-evaluations (e.g., after each incoming update message). A typical outcome where t_{eval} turns out larger than t_{prop} is shown in Figure 2.3.

On expiration of t_{eval} (or initially t_{init}), a node evaluates the role specification using its local cache table only. For this, it evaluates the role predicates of its specification sequentially and will assume the first role, for which the corresponding predicate matches. Each role predicate c^k is assumed to be in disjunctive normal form given in (2.1). Its *atomic predicates* c^k_{ij} can be evaluated sequentially.

Simple predicates are evaluated using property values from the local cache table.

Count predicates of the form

```
count(scope) { pred } rel const
```

consider cache table rows with $1 \leq dist \leq$ **scope**. The nested predicate **pred** is then applied to the source nodes of all these rows. The number of matching nodes is then compared to **const** using the given relation **rel**.

Retrieve predicates of the form

```
p == retrieve(scope,size) { pred }
```

are evaluated similarly. However, instead of only counting matching nodes, a set **S** of matching nodes is computed. Note that a retrieve statement c binds **p** only while evaluating its enclosing conjunction $c_x^k = (c_1 \wedge \cdots \wedge c_{n_x})$. If $|\mathbf{S}| <$ **size**, c_x^k evaluates to false. Otherwise, the remainder of c_x^k is evaluated for all $\mathbf{p} \subset \mathbf{S}$ with $|\mathbf{p}| =$ **size** until the enclosing predicate c_x^k becomes true. If no such **p** exists, the evaluation of c_x^k returns false.

If the node's role has changed, **property propagation** is triggered without delay.

In our example, node B performs local rule evaluation. B's cache table is shown in Figure 2.4. In the given example, B first checks the conditions for ON. Assuming that the node is equipped with a temperature sensor and a full battery, the first two conditions of the coverage specification evaluate to true. Evaluating the count statement requires counting table rows with property *role* and *dist = 1*. As the result of the count expression is 1, the predicate for role ON does not match. Node B continues by checking the conditions for OFF. As the "else" statement imposes no constraints, node B is assigned the role OFF.

2.4.5 Property and Network Dynamics

So far, we have described role assignment in a static network. In this section, we consider the effects of changes in the network on a computed configuration. For this, we distinguish three classes of changes.

The first class are *property changes*, e.g., the battery level has changed. The property directory is configured to notify the role assignment system of such changes. If a change occurred, the node updates the cache table and re-examines its chosen role via the **local rule evaluation** procedure.

Only if the changed property value affects other (local or remote) properties by changing the evaluation result of a predicate in the specification, an update message is composed and broadcast to all neighbors. Note that the number of update messages generated through property changes is limited by the minimal interval T_{min} between two *update* messages that are induced by property changes. The programmer may also specify a tolerance interval Δ_p for a property p, such that re-configuration is only triggered if p's value changes by more than Δ_p.

A second class of changes is when nodes *join* the network. For example, it is imaginable that additional nodes are deployed into a given area. When such a new node overhears any protocol message, it first requests a copy of the role specification from its network neighbors. After it received a reply, it uses the specification to initialize as described earlier and broadcast an update message to inform neighbors of its relevant properties.

The third class are *node failures*. To detect such failures, nodes send heartbeats every $T_{\text{heartbeat}}$. A heartbeat is essentially an *update* message, containing the key of the property that needs to be propagated farthest in the specification (e.g., the *role* property in our examples) with an empty property value. The sequence number is incremented at each heartbeat. At receiving nodes, the heartbeat message will update the sequence number field of the corresponding row in the cache table and set it to *dirty*. Forwarding nodes will thus include heartbeats in their *update* messages, provided the *maximum scope* has not been reached yet and that no other properties of the originating node were already forwarded within the node's *critical area* within the last $T_{\text{heartbeat}}$ seconds.

After sending (or suppressing) its own heartbeat, a node verifies whether all nodes contained in its cache table have sent information during the last $N_{\text{heartbeat}} \times T_{\text{heartbeat}}$ seconds. We use the factor $N_{\text{heartbeat}}$ to allow lost heartbeat messages before node failure is assumed and thus accommodate unstable links. $N_{heartbeat}$ is currently set to 3. Entries referring to nodes that have not sent any information by that time are deleted, followed by local rule evaluation and subsequent propagation if a role or property has changed.

2.4.6 Termination

Role assignment is continuously executed during the lifetime of a sensor network to adapt role assignments to changes in the network. Hence, role assignment itself does not terminate.

However, role assignment is typically executed in phases which start either after receiving a new role specification or on a property or neighborhood change (cf. above Section 2.4.5). In a role assignment phase, the above algorithm will perform a fixpoint iteration, in which a node typically iterates through a sequence of different roles.

If a fixpoint configuration exists, each node will eventually assume a stable role. Termination refers to detecting when such a stable role has been assumed. In our approach, we assume that a role is stable if it has not changed for $T_{\text{termination}}$ seconds. Note that applications are notified of such stable roles, only. Changes of local or neighborhood properties may later trigger re-assignment of roles. If such changes occur within a role assignment phase, these will be incorporated in the result of the current phase. However, changes must be infrequent enough to let such role assignment phases terminate. Therefore, properties that change very fast (e.g., a light sensor) are typically not useful as input for network configuration.

In our experiments, stable configurations are reached after only very few role-changes at each node (see Section 2.6 for details). However, there are role specifications where no such fixpoint configuration exists and no stable roles can be assumed, for example:

```
1 RED   :: { count(1) { role == GREEN } >  1 }
2 GREEN :: { count(1) { role == RED   } == 0 }
```

Here, a node requires the absence of `RED` neighbors to become `GREEN`, yet its neighbors become `RED` as a direct consequence of its own role having become `GREEN`.

We consider such specifications erroneous and provide three approaches to help a developer detect such faulty specifications. Firstly, our development environment provides a comprehensive simulation tool which can be used to evaluate role specifications in realistic network setups. Secondly, it includes a centralized solver component which allows the system to assess the *feasibility* of specifications. As we will discuss in detail in Section 2.8, the above specification is infeasible on any topology consisting of more than two connected nodes.

Last but not least, we use heuristics to detect potentially non-terminating specifications in an operative network. If a node goes through the same cycle of roles over and over again, non-termination is assumed. Another, simpler heuristic is to assume non-termination if a node exceeds a given number of role changes without reaching a stable phase.

2.4.7 Probabilistic Initialization

In the above, we have described the basic role assignment algorithm based on caching. In this algorithm, all nodes start with the initial role `undefined` upon which they start the role assignment iteration. In this section, we examine two approaches to initialize nodes more intelligently based on probabilistic estimates generated from the role specification. The above role assignment algorithm would then only *repair* inconsistencies of an initial configuration.

Dicing Roles

A first approach is to estimate the probability of each role based on a given specification and an estimate of the average network density. Assume the role specification contains specifications for roles $\{1,\ldots,q\}$. In the following, we will compute a set of probabilities $\{p_1,\ldots,p_q\}$ and, initially, let each node to take on role k with probability p_k. Note that we use these probabilities for initializing *every* node in the network, not incorporating any additional information about a given node or its neighborhood. Thus, $\{p_1,\ldots,p_q\}$ could even be pre-computed *offline* by a role compiler and disseminated to the nodes together with the role specification.

We will map the role specification to a system of q equations with q unknowns, namely the probabilities $p_1,\ldots,p_q$. We will now delineate how we translate different parts of the specification into this equation system.

For this transformation, let us assume the role specification contains role predicates of the general form of Equation (2.1):

$$c^k = (c^k_{11} \wedge \cdots \wedge c^k_{1n_1}) \vee (c^k_{21} \wedge \cdots \wedge c^k_{2n_2}) \vee \ldots$$

Let us assume that the values of c^k_{ij} are independent of each other. Let $P(c^k_{ij})$ denote the probability that c^k_{ij} is true. Assuming that probabilities $P(c^k_{ij})$ are known for all atomic predicates, we can derive the probability p_k of the role k:

$$p_k = P(c^k) = \sum_i \prod_j P(c^k_{ij}) \tag{2.2}$$

while we set the probability of a possible else role to

$$p_{\text{else}} = 1 - \sum_i p_i \tag{2.3}$$

We will now show how the probabilities $P(c^k_{ij})$ can be obtained for all types of *atomic predicates* c.

Let us first consider the case, where c is a *simple predicate*. If c is of the form `role==`r then $P(c) = p_r$. In all other cases, we require the programmer to explicitly specify $P(c)$. This is often possible, as we are only interested in the probabilities *at startup*. For example, the probability for the predicate `battery>10%` can be approximated with 1 at deployment time. Otherwise, an educated guess may be applied. The respective programmer-specified $P(c_{ij}^k)$ are distributed along with the specification.

For *count* and *retrieve* predicates, we assume that the average network density is known and that a function $E(h)$ for the expected number of nodes within h hops (the so-called h-neighborhood) can be estimated from deployment parameters[1].

Now consider a *count predicate* c of the form:

`count(`**`scope`**`) { `**`pred`**` } `**`rel lim`**

where the nested predicate **`pred`** is in disjunctive normal form and contains only *simple* sub-predicates c_{ij}^k. That is, we can compute $p_{\textbf{pred}} = P(\textbf{pred})$ according to Equation 2.2. Let us now consider the case where **`rel`** is "<=", the other cases can be solved in similar ways.

Estimating the number of nodes n to be expected within **`scope`** by $n = E(\textbf{scope})$, we can now formulate the probability that x out of n nodes match **`pred`** using the binomial distribution:

$$P(\textbf{x of n nodes match}) = \binom{n}{x} p_{\textbf{pred}}^x (1 - p_{\textbf{pred}})^{n-x} \tag{2.4}$$

Thus, the probability that less or equal **`lim`** nodes match **`pred`** (and thus the above count predicate c is true) corresponds to the sum the above probabilities for all $x \leq$ **`lim`** :

$$P(\texttt{count} = \texttt{true}) = \sum_{x=0}^{\textbf{lim}} \binom{n}{x} p_{\textbf{pred}}^x (1 - p_{\textbf{pred}})^{n-x} \tag{2.5}$$

Finally, a *retrieve predicate* c of the form

`p` `== retrieve(`**`scope,size`**`) { `**`pred`**` }`

requires that at least **`size`** nodes exist within **`scope`** that match the given nested predicate **`pred`**. For calculating role probabilities, we therefore consider an equivalent count statement:

`count(`**`scope`**`) { `**`pred`**` } >= `**`size`**

[1] In our implementation, we estimated $E(h)$ for unit disk graphs and random uniform node deployment on a plane. By fitting results of simulations we found that for any fixed node density $E(h) \sim h^2$.

So far, we have derived q equations with q unknowns by substituting $P(c_{ij}^k)$ in Equation (2.2) for $1 \leq k \leq q$. We use a fixpoint iteration to solve this equation system. Assume a set of probabilities for each role $p_k(t)$ are known at a given step t. Substituting these into the right side of equation (2.2), we can compute a set of new probabilities $\hat{p}_k(t+1)$. To avoid oscillations in this series, we add a memory term $p_k(t)$ that averages the old values into the newly chosen ones:

$$p_k(t+1) := \frac{1}{2}\hat{p}_k(t+1) + \frac{1}{2}p_k(t) \tag{2.6}$$

Note that at each step, we also normalize the $p_k(t+1)$ such that $\sum_k p_k(t+1) = 1$. We initialize this series with equal probabilities for each role $p_k(0) := 1/q$ and iterate until the series converges to a fixpoint. We would like to emphasize that this computation is done *offline* by the role compiler as all information needed is the specification and the mentioned estimation of the neighborhood size $E(h)$. The resulting probabilities are then flooded along with the specification upon which each node draws role k with probability p_k and then starts initialization as described in Section 2.4.2.

We will show in Section 2.6 how the probabilistic initialization can provide significant improvements over the baseline algorithm where all nodes start with the role `undefined`.

One Wave

In the above approach we approximate retrieve statements with count operators ignoring the fact that the local node sets bound by retrieve statements occur in other predicates of the specification. Hence, subpredicates are not independent of each other (as we assumed).

Furthermore, the result of a retrieve statement depends on the properties of remote nodes. However, the initialization of local properties (that are usually set by retrieve statements) cannot be performed probabilistically (at least it is highly unlikely that such initialization will be correct). Therefore, additional interaction to adequately initialize such local properties will be required.

In this regard, partial (deterministic) information is helpful and can be used to properly initialize the properties bound by retrieve statements, while other decisions can still be performed probabilistically. In this section we describe how we can use conditional probabilities to improve the stability of probabilistic decisions.

The basic idea for this approach is to leverage an existing network flood (e.g., for delivering the role specification) and execute the algorithm of Section 2.4 while forwarding this "propagation wave".

We generate such a propagation wave by scheduling the property propagation procedure of Section 2.4.3 differently: The sink is the only node that initially sends an *update* message (other nodes refrain from sending *update* messages before they received one). A node that received an update message, awaits a random propagation delay $t_{\text{waveprop}} \sim (0, T_{\text{waveprop}})$, then chooses its role (see below) and includes its own chosen role into the forwarded *update* message. Note that all information of these update messages can be piggybacked onto flooded role specification messages.

Let us consider the evaluation of the count predicate used as an example in the previous Section 2.4.7:

```
count(scope) { pred } <= lim
```

Figure 2.5 depicts the situation, where the propagation wave reaches a given node A. Let us assume that A expects n nodes in **scope** and m of these are behind the wave front (to the left of A). These have already chosen their roles and included all relevant information (including the selected role) in previous update messages. That is, A has received update messages from all nodes behind the wave (on the left). A will now decide probabilistically what it expects from the $n-m$ nodes on the right.

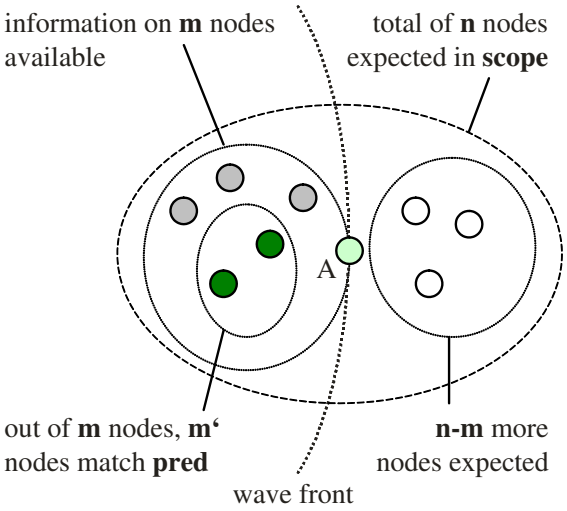


Figure 2.5: Propagation wave

For this, let us assume that out of the m known nodes on the left, m' match the nested predicate **pred**. We can now reformulate the probability that the count predicate is satisfied as the probability that **lim**$-m'$

"more" nodes match (out of the $n-m$ expected nodes on the right). The conditional probability that the count predicate c is true can be expressed as follows:

$$P'(c) = \sum_{x=0}^{\mathbf{lim}-\mathbf{m}'} \binom{\mathbf{n}-\mathbf{m}}{x} p_{\mathbf{pred}}^{x} (1-p_{\mathbf{pred}})^{\mathbf{n}-\mathbf{m}-x} \tag{2.7}$$

Note that count statements using other relations (greater or equal to a constant) can be treated analogously, and also the aspect that retrieve statements require a minimum number of matching nodes.

When evaluating a retrieve predicate, the node additionally binds the local properties to a set of nodes that are known to match the nested predicate, if such a set of the required size exists. If such a set exists, we consistently reflect the dependency between the local node and remote nodes (in our case the remote nodes behind/left of the wave).

For sub-predicates c in **`pred`** of the form `role==r` we still assign the corresponding pre-computed probability $P(c) = p_r$ of Section 2.4.7 as no additional information is available on the $n-m$ remote nodes that are in front of the wave.

This way, we derive q equations for the conditional probabilities p'_k, one for each role similar to Equation (2.2), but formulated in terms of the previously computed p_k.

$$p'_k = P'(c^k) = \sum_i \prod_j P'(c_{ij}^k) \tag{2.8}$$

What we changed is the way we obtain probabilities P' for the atomic predicates c_{ij}^k of the equation. On the nodes, we compute the values of p'_k in one step (without the need for fixpoint iteration) using the $\{p_k\}$ sent along with the specification and choose role k with the respective probability p'_k.

The advantage of the above algorithm is that role assignment can be performed almost entirely within *one* network-wide flood and that only few role assignments have to be "fixed" later on (using the baseline algorithm). Moreover, this approach is able to capture interdependencies between atomic predicates better. We will analyze the performance of this algorithm in Section 2.6.

2.5 Development Environment

Generic role assignment is implemented within a development environment which allows programmers to test their specifications with different

kinds of network topologies before deploying them into operative networks. The development environment consists of a compiler for role specifications, a network setup and configuration component, a visualization tool for introspecting algorithm execution and results, and finally two back-ends that perform role assignment on the generated topology. The first back-end is a discrete event simulator that executes the presented distributed algorithms in a network environment using additive interference models and realistic wireless parameters. The second back-end is a solver-based analysis tool that can compute optimal role assignment solutions in a centralized manner.

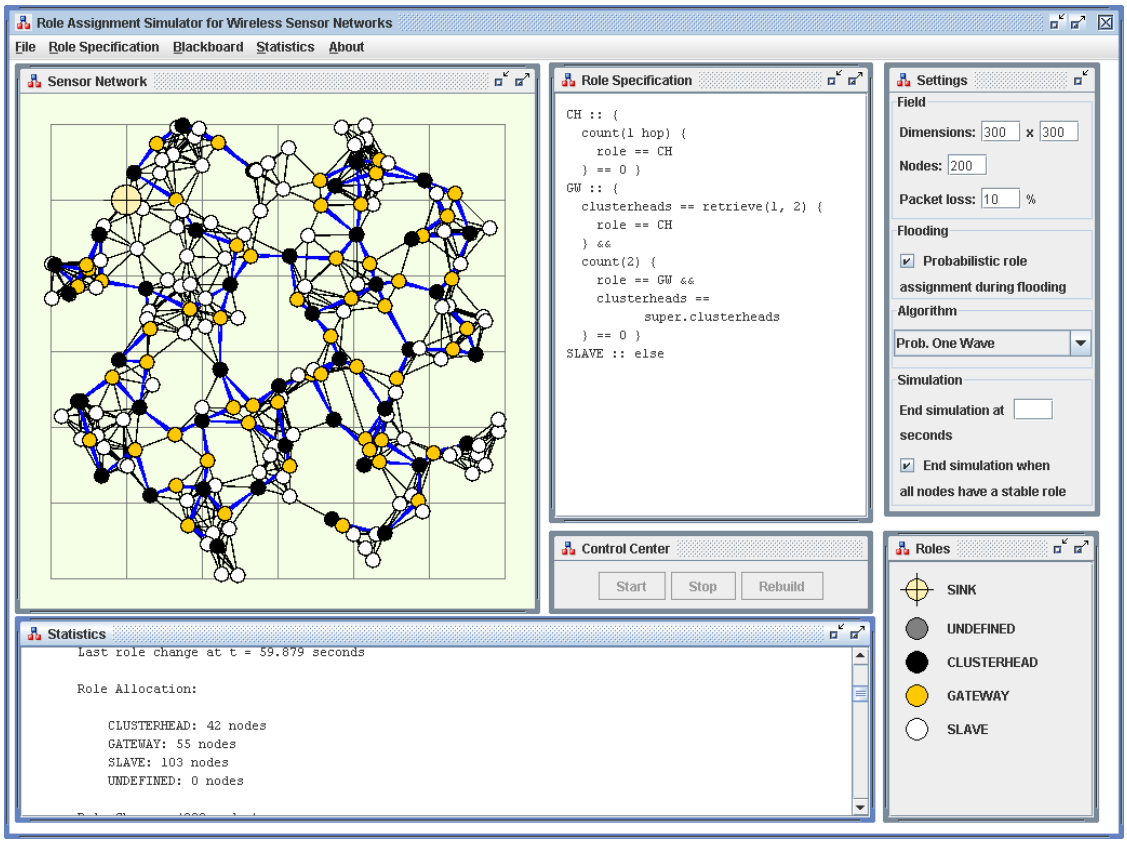


Figure 2.6: Role assignment simulation tool

The user interface is shown in Figure 2.6. It allows users to set up a network topology, define property directory contents (i.e., the actual node parameters) and select the algorithms for execution on the nodes. The role compiler then translates a given specification – in this case the clustering specification of Section 2.3.1 – into the corresponding syntax tree, performs necessary pre-processing and context checking (i.e., ensures that there are no circular dependencies and rearranges the role predicates into disjunctive normal form) and computes a number of additional parameters (such as role probabilities). Syntax tree and parameters are then encoded

into a role specification message, which is then interpreted on the simulated nodes.

Finally, the user can trigger delivery of the role specification message through a flood initiated at the network base station, upon which the network nodes initialize and execute the role assignment algorithm.

The visualization tool enables qualitative assessment of the algorithm execution and of role assignment results. Figure 2.6 shows exemplary results when using the *one-wave* variant of the cache table algorithm to implement the clustering specification introduced in Section 2.3.1. The highlighted edges are drawn between a gateway and the clusterheads it connects. Note that in the shown simulation, apart from messages lost due to collisions, an additional packet loss probability of 10% was enabled, which might have caused some – but only few – inconsistent role assignments. We will discuss quantitative results in Section 2.6.

We use the discrete event simulator JiST/SWANS [Bar04] as a simulation back-end. We adopted wireless transmission parameters from the CC1000 radio [CC04] that is in use on the BTnode [BTn06] platform, on Mica motes, and on many other platforms, please consider Figure 2.7(a) for details. The physical layer supports additive interference and two-ray fading. Using these parameters we obtain a maximum transmission range of about 33 m.

Parameter	**Value**	
bandwidth	38.4	kbit/s
transmit power	5	dBm
sensitivity	−96	dBm
receive threshold	−84	dBm
interference limit	−96	dBm
frequency	868	Mhz
antenna gain	0	dB
node height	5	cm
max. transmit range	33.5	m

(a) Wireless parameters

Parameter	**Value**
T_{prop}	3 s
T_{eval}	10 s
$T_{\text{heartbeat}}$	60 s
$N_{\text{heartbeat}}$	3
$T_{\text{termination}}$	60 s
T_{waveprop}	5 s
T_{min}	1 s

(b) Algorithm timers

Figure 2.7: Simulation parameters

We use a variant of the CSMA MAC described in [WC01] with timers and delays adapted for 38.4 kbit/s. Only the broadcast service is used (i.e., no channel reservations are performed). The MAC does not perform collision detection, or any other means to improve robustness. We deliberately chose such a simplistic MAC to study the robustness of our algorithm separately from "tricks" performed by more advanced MAC

layers. The obtained robustness results can be considered "worst case" and better values can be expected when using more sophisticated MAC layers (e.g., [PHC04, vDL03, YHE02]).

The second back-end for role assignment derives an integer linear program (ILP) formulation from a role specification and a given network topology. The ILP can provide insights into whether a specification is infeasible, that is, whether a valid assignment of roles to nodes exists at all. Moreover, it helps assess the quality of the distributed solutions by comparing them to solutions in which certain sets of roles are minimized or maximized (e.g., in the coverage example, the solver attempts to choose fewer ON nodes but still cover the respective area). Section 2.8 describes the solver back-end in detail.

2.6 Evaluation

An important requirement on programming abstractions is that the induced overhead should be proportional to the complexity of the specified problem. To gain an insight into whether our algorithms are adaptive in the above sense, we evaluated three specifications of increasing difficulty: The *coverage* example that we used to illustrate the *caching* algorithm in Section 2.4 can be considered a baseline case. Furthermore, we examine *aggregation* and *clustering* as defined by the specifications from Section 2.3.1. These both require propagation of additional properties over two hops. The *clustering* specification is especially challenging through use of the *retrieve* operator that makes role assignment decisions dependent on the *identities* of nodes in the neighborhood.

We will evaluate all three presented algorithms, the baseline cache table algorithm from Section 2.4 and its combination with the two probabilistic initialization schemes described in Section 2.4.7, to which we will refer as *cachetable*, *probabilistic* and *wave*, respectively. We set algorithm parameters according to Figure 2.7(b). To evaluate the protocols, we let the three algorithms assign roles to a previously uninitialized network and stop the simulation if all nodes were stable for $T_{\text{termination}}$ seconds (each node decides termination locally).

We performed experiments on random topologies that are connected with high probability. If there should be no multi-hop path from the sink to a node, this node simply does not participate in the experiment. In our evaluations, we place a variable number of nodes in a $300\,\text{m} \times 300\,\text{m}$ square field.

For measuring *overhead*, we consider the total number of messages that are sent by each node and – as messages can be of variable size – the total payload sent by each node in Section 2.6. Note that the number of messages includes (one) specification message per node, which is the initial flood.

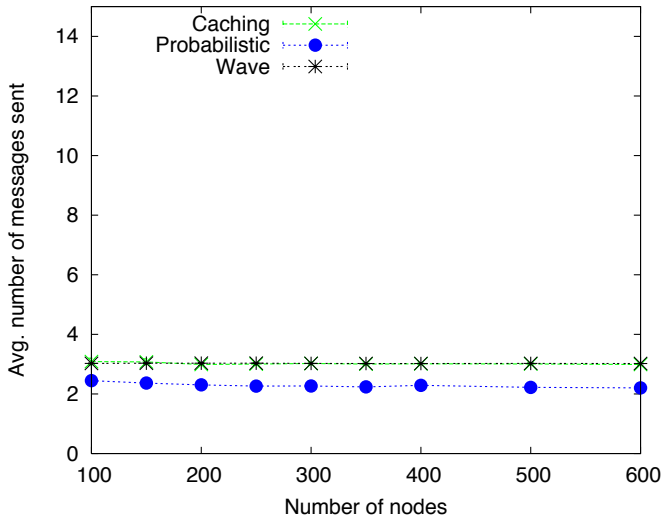
To quantify *correctness* of the outcome, we consider the (theoretical) network topology where two nodes have a direct network link if their distance is less than the maximum transmission range of 33.5m. An assigned role is considered correct if the respective role predicate matches for the set of neighbors obtained from the above theoretical topology.

Later on we will consider the *number of role changes* that occur after initialization to study *convergence* of our algorithms. For the *probabilistic* and *wave* algorithms, we do not count the initial role change that is induced by the probabilistic decision, as we are interested in the remaining inconsistency that has to be repaired. For each data point, we indicate 95% confidence levels obtained from repeated simulations on independently drawn random topologies.

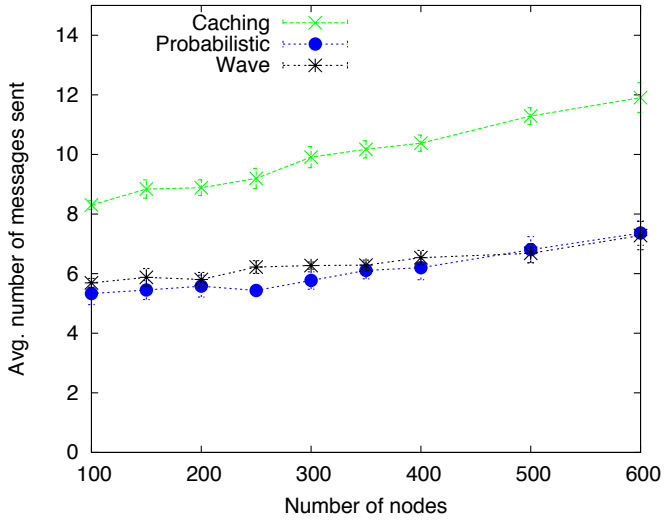
Overhead. We study the communication effort spent by the three presented algorithms. We vary the number of nodes in the confined area to see how increasing node density affects the performance of our algorithms. The average number of messages sent per node using each specification are shown in Figure 2.8.

The results of Figure 2.8(a) show that the simplified *coverage* specification can be implemented effortlessly by all three algorithms. The maximum of *three* messages includes the specification flood, the later propagation wave for the *wave* algorithm (we implemented propagation and role specification waves separately), and finally at most one more message that is used to check whether repairs are needed.

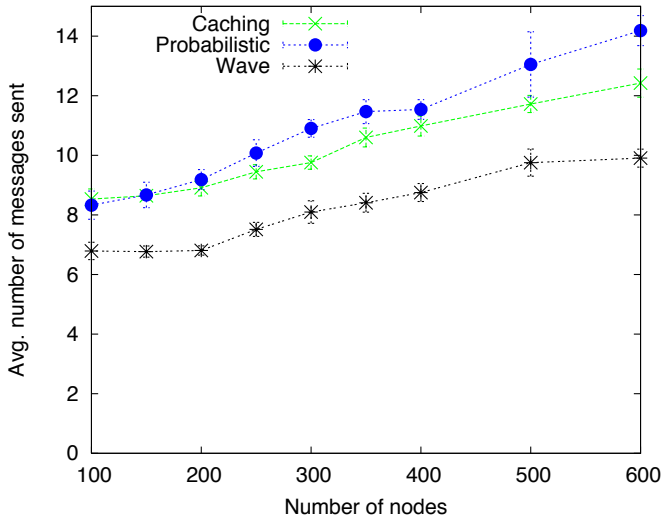
In the aggregation example of Figure 2.8(b), the *probabilistic* and *wave* variants outperform the baseline algorithm. Note that the *wave* algorithm does not perform better than *probabilistic*. This is due to the *aggregation* specification (cf. Section 2.3.1): The *wave* algorithm can improve performance only if it can exploit knowledge about nodes *behind* the wave front. For the first count statement of role `AGG1`, this would require that `SOURCE` nodes are behind the wave front. However, the specification further requires that `SOURCE` nodes are farther away from the sink than aggregators. This is unlikely to happen, because the wave propagates from



(a) Coverage



(b) Aggregation



(c) Clustering

Figure 2.8: Sent messages per node with increasing density

the sink outward. Therefore, no aggregator roles are assigned during the first wave.

For clustering in Figure 2.8(c), *probabilistic* performs slightly worse than *caching*. This can be explained by the fact that the choice of roles here depends very strongly on the *identities* of nodes in the neighborhood and not only on their roles. Hence it is very unlikely that a probabilistic decision "guesses" the right node identities, this has been one motivation for the design of the *wave* variant, which exhibits better performance. Nevertheless, it is notable that *probabilistic* still performs comparably to *caching*.

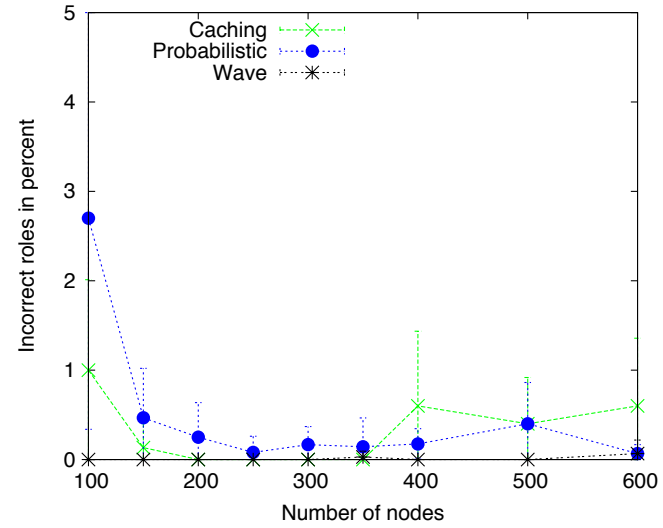
The relative performance of the algorithms with respect to payload size per node is similar. With the *wave* algorithm, the total payload size per node in the clustering example is 179 bytes for 200 nodes and 388 bytes for 600 nodes, resulting in average message sizes of 26 to 39 bytes, respectively. Results are similar in the *aggregation* case, while *coverage* only requires transmitting at most 40 bytes per node as a maximum over all algorithms and node densities. This includes quite high density values: 600 nodes in the given area yield an average node degree of around 20.

Correctness. We show the fraction of nodes with an *incorrect* role assignment in Figure 2.9. The baseline *coverage* case does not exhibit any significant incorrectness. This is due to the simplicity of the specification, where the count operator considers the 1-hop neighborhood only. Essentially, all nodes send one message to announce their role. If inconsistencies occur, these are repaired with a second message.

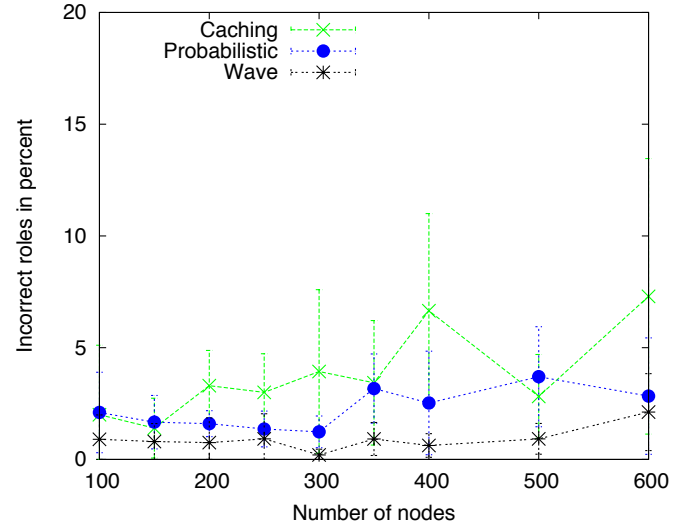
For *aggregation* and *clustering* specifications, the *caching* and *wave* algorithms perform best, while the *probabilistic* variant suffers from its deficiencies when used with retrieve operators. Note that even though we omitted any means for reliable message transfer, the wave algorithm achieves very low incorrectness numbers.

Note that shown incorrectness is due to unreliable message delivery only. If message delivery were reliable, improbable yet possible simultaneous role evaluations would not contribute to incorrectness, as these would be repaired in subsequent algorithm iterations. Nodes that end up with incorrect roles have not learned from each other's properties properly (i.e., at least one message must have been lost). A reliable MAC layer would incur zero *incorrectness* but (possibly) worse *convergence* results.

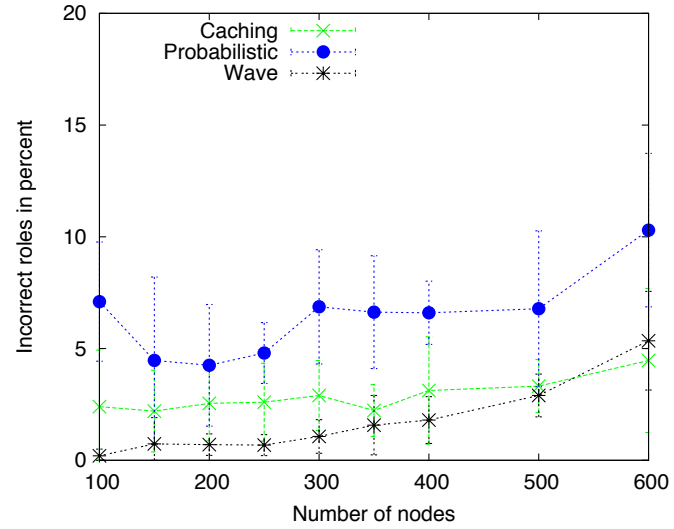
Robustness. To examine robustness in the face of message loss, we introduced an additional packet-loss probability. We measured again the ratio of incorrect role assignments to the total number of nodes for our



(a) Coverage



(b) Aggregation



(c) Clustering

Figure 2.9: Percentage of incorrect assignments from total nodes with increasing density

three examples. Results are shown in Figure 2.10. The x-axis denotes the probability used for dropping a message.

On this scale, the algorithms do not exhibit significant differences. Note that clustering is less robust than *coverage* and *aggregation*. This is due to the fact that the dependency of the predicates on neighbor nodes is much stronger in *clustering*: If, caused by a lost message from a clusterhead neighbor, a node inconsistently becomes clusterhead, many neighbors of the two will (incorrectly) become gateways, which they will have to correct later through additional communication effort. Therefore, errors tend to amplify. Nevertheless, with all three specifications, high packet loss still yields acceptable correctness levels.

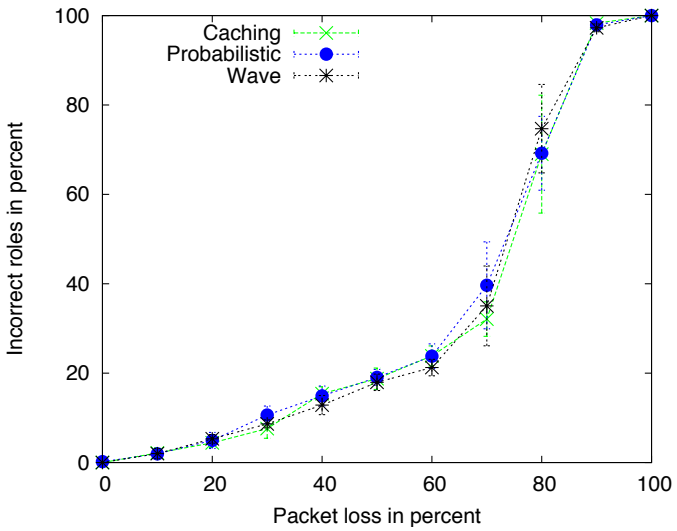
Note that the node neighborhood changes with message loss. Hence, the results depicted in Figure 2.10 can be interpreted as how well the algorithms can deal with dynamic neighborhoods. We plan further evaluation using bursts of lost messages that affect the perceived neighborhood more severely.

Convergence. In Figure 2.11, we quantify the number of role changes required after initialization until a node assumes a stable role. It shows the total number of role changes per node *after initialization*. Note that we do not count the probabilistic initialization for *probability* and *wave* as a role change as the additional overhead it incurs is negligible. The average number of role changes per node is less than 2 for all three specifications. That is, all communication effort is invested into property propagation, and almost no "undesired" repair iterations occur.

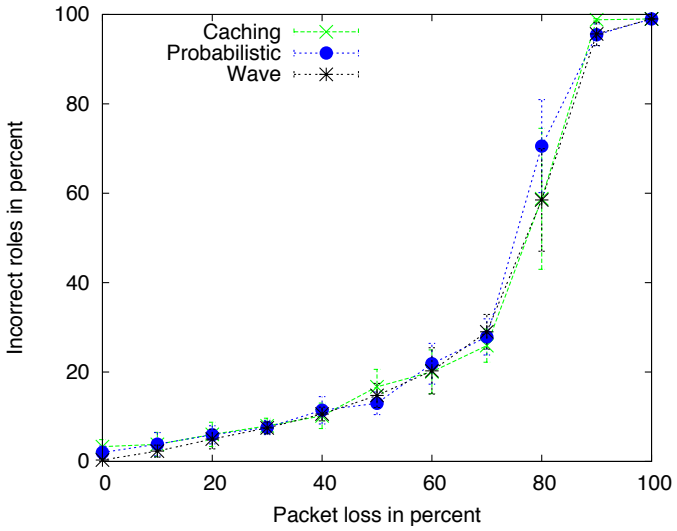
Note that the values shown for the *wave* algorithm are very low. After the initial wave, only a few repairs are needed. The number of repairs needed in the *coverage* case is effectively zero, while *clustering* and *aggregation* require at most one role change for every two nodes.

Scalability. A parameter that is crucial for the algorithm's efficiency is the *maximum scope* of the considered specification. To assess how the maximum scope affects overhead, we measured sent messages and payloads with the simplified *coverage* example – while varying the scope of its count operator from 1 to 4 in a setting with a constant number of 200 nodes.

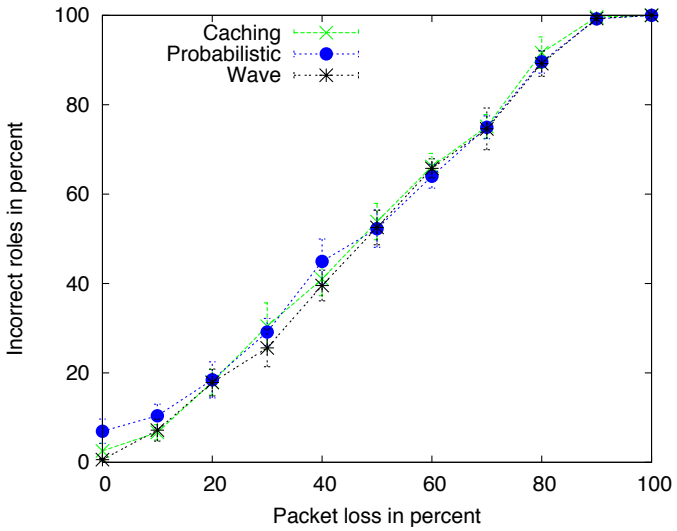
The results depicted in Figure 2.12 demonstrate that information is indeed combined into few update messages. With an average node degree of about 7 and a maximum scope of 4 hops, the employed number of messages still remains low. However, the total payload increases, e.g., from 21 bytes (for scope 1) to 400 bytes per node (scope 4) for the best



(a) Coverage

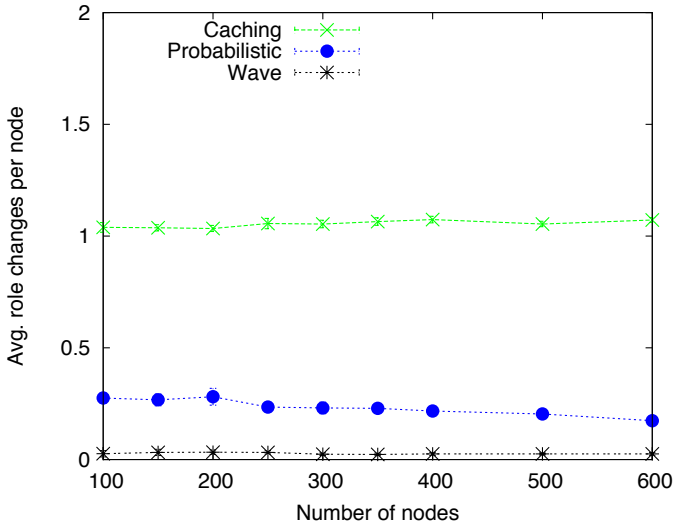


(b) Aggregation

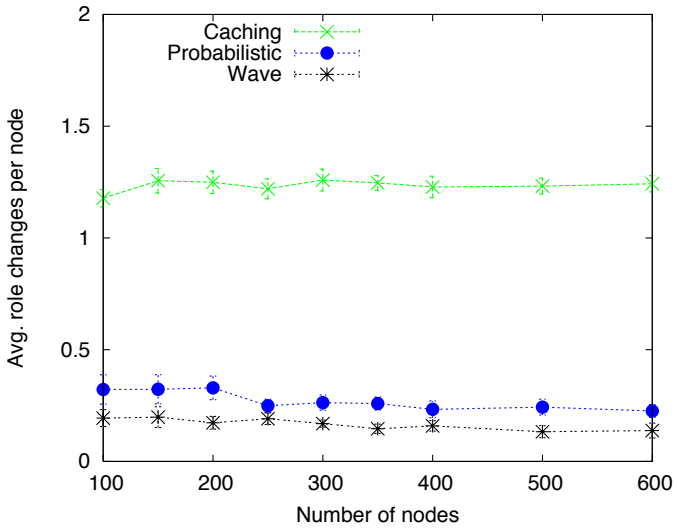


(c) Clustering

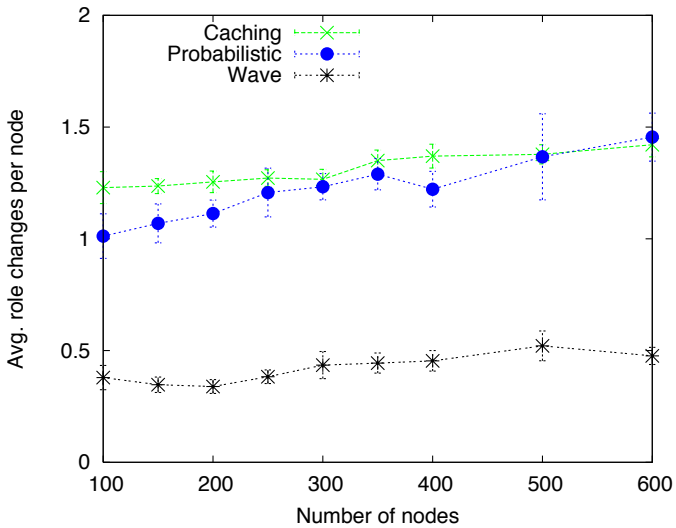
Figure 2.10: Robustness in face of dropped packets



(a) Coverage



(b) Aggregation



(c) Clustering

Figure 2.11: Total role changes per node with increasing density

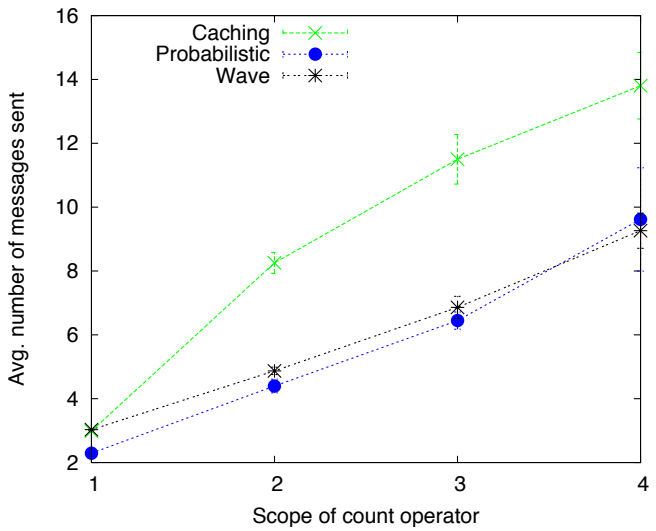


Figure 2.12: Coverage application with increasing scope

performing *probabilistic* algorithm. With a scope of 4 hops, the average message size is at most 48 bytes.

Summing up, the evaluation results indicate that the role initializations of *probabilistic* and *wave* can provide improvements over the baseline algorithm. However, we have also seen that there is no single best algorithm, although *wave* outperforms the other algorithms in most settings. With aggregation we found a case, where the *wave* algorithm is "tricked" by the specification. Apart from such special cases, a rule of thumb is that *wave* performs better than *probabilistic* for specifications that make use of `retrieve`. An interesting direction for future work would be the development of further heuristics to automatically derive from a given specification which algorithm can be expected to give the best results.

2.7 Qualitative Comparison

Throughout the chapter, we have used specific instances of coverage, clustering, and aggregation problems to illustrate and evaluate generic role assignment. In the literature, numerous specialized algorithms for solving various instances of these problems have been proposed. In this section, we discuss how these algorithms compare to our implementations that are based on generic role assignment.

For each of the problems, we selected a representative instance and algorithm from literature and discuss differences to our implementation with respect to semantics and efficiency.

Coverage. A common semantic for coverage is that all of a node's sensing range should be covered by other sensors before the node can turn off [TG03]. In our version of coverage we approximate this semantic by turning a node off if there are at least N (we use $N = 1$ in the example) other on nodes within sensing range. For a more direct implementation of the semantics in [TG03], a node would first `retrieve` all neighbors with overlapping sensing ranges (i.e., which are at most twice the sensing range away) and then use a custom predicate to decide whether all of the node's sensing range is covered by these neighbors. This specification would require the role assignment algorithm to first propagate *position* and *role* properties across the required number of hops and then perform local rule evaluation after random delays. Chosen roles would be immediately propagated to affected neighbors. Our algorithm would (quite closely) match the implementation of [TG03] in the variant where sensing ranges are assumed equal at all nodes. □

Clustering. Our version of clustering was inspired by (but is not identical to) passive clustering (PC) [KG02]. With PC, a node declares itself clusterhead if none of its neighbors have done so before. A node becomes a gateway if $\alpha CH + \beta > GW$ where CH and GW are the numbers of clusterhead and gateway neighbors, respectively, and $\alpha, \beta > 0$ are algorithm parameters that control the desired number of gateways in the system. This approach could be directly implemented by generic role assignment with an overhead comparable to the implementation in [KG02], where only the `role` property is shared among neighbors.

With PC, protocol information is piggybacked on existing network traffic. In contrast, generic role assignment uses a separate protocol, resulting in some traffic overhead when compared to PC. An interesting direction for future work would be *passive* generic role assignment, where protocol information would be piggybacked as with PC. □

Aggregation. A common optimization criterion for aggregator placement is to minimize the total network traffic, which is identical to solving the Steiner-tree problem. Various heuristics are used in the literature to approximate this NP-hard problem. One commonly used heuristic is called *center at nearest source* [IEGH02], where among a number of neighboring source nodes, the one closest to the sink is selected to act as an aggregator. This approach can be implemented with generic role assignment, where a node would use `count` to decide whether there is another source in the neighborhood that is closer to the sink. The scope of `count` (i.e., size of the considered neighborhood) would be tuned to achieve a reason-

able trade-off between algorithm overhead and optimality of aggregator selection.

If geographic positions are used, the implementation of *closer* is straightforward by propagating the position property of a node within the scope. In the original algorithm in [IEGH02], this is done by a network-wide flood. If hop-distance to the sink is used instead of geographic positions, an additional role specification would be needed to obtain the hop-distance of each node from the sink. For this, each node would have a property `dist` that specifies its distance from the sink. A node would then `retrieve` among its neighbors the node with the smallest `dist` value and set its `dist` property to the retrieved value plus one. This would require a simple extension of the `retrieve` operator to sort the retrieved nodes according to some user-specified criterion (e.g., minimum of a property value). Note that other, custom-implemented, components that determine the `dist` property could be integrated with role assignment via the property directory. □

Apart from the above specific examples, we can make some more general observations about the efficiency of generic role assignment when compared to specialized algorithms. With generic role assignment, the value of a property is *always* propagated to *all* nodes in the scope of this property. In some cases, however, it may not be necessary to propagate certain property values. In other cases, certain nodes may not need to propagate a property at all or only to certain nodes within a scope. As part of future work we will investigate whether such optimizations can also be supported by generic role assignment.

2.8 Role Assignment Solver

In the previous sections, we have mentioned a couple of open issues with the described generic role assignment system. The first of these issues is *termination*. For specifications, for which no assignment of roles to nodes exists that satisfies all rules, our distributed algorithms may not converge. Therefore, mechanisms are needed that assist developers with detecting such faulty specifications. The second open issue is that many practical role assignment problems include some global *optimization criteria*, such as the minimization of the number of `ON` nodes in the above coverage problem. Such problems cannot be expressed with the role specification syntax from Section 2.3.2.

The role assignment solver provides an analysis tool for approaching these issues. In addition to the syntax specified in Section 2.3.2, the solver component allows developers to specify global optimization criteria of the form `min` **`role`** or `max` **`role`**, where **`role`** can be any of the roles that have been defined, for example:

```
min ON
```

Multiple such specifications may be issued alongside a role assignment specification to demand minimization or maximization of the respective set of roles. The complete role specification and a given network topology are then used as input to the role assignment solver. Given such input, the solver will compute a role assignment taking into account the specified optimization criteria, or state *infeasible* if no feasible role assignment exists.

The remainder of this section is structured in two parts. First, in Section 2.8.1, we provide a mapping of the role specification language to Integer Linear Programs (ILP). This mapping, performed within the development environment described in Section 2.5, returns a problem formulation that can be used as an input to the CPLEX commercial solver. Second, in Section 2.8.2, we use the resulting ILPs and CPLEX to re-examine some of the role assignment specifications presented in Section 2.3. In particular, we show how infeasible (non-terminating) specifications can be detected, and quantify improvements that can be achieved by using global optimization criteria.

2.8.1 Integer Program Mapping

An instance of a role assignment problem consists of a role specification (a set R of role predicates according to the syntax described in Section 2.3), all property values of all nodes that are referenced by the role specification, and a sensor network graph $G = (V, E)$ with $n = |V|$ participating nodes.

G is used to define the elements of the h-hop neighborhood matrices $\mathbf{A}^{(h)}$ as follows:

$$A_{ij}^{(h)} = \begin{cases} 1 & \exists \text{ path from node } i \text{ to node } j \\ & \text{with length } \leq h \\ 0 & \text{otherwise} \end{cases}$$

The mapper generates a total of $h = 1 \ldots S$ such matrices, where S denotes the maximum scope that occurs in all *count* and *retrieve* statements

of a specification. Note that these matrices can be readily computed from the graph G's adjacency matrix $\mathbf{A}^{(1)}$. The above notation $A_{ij}^{(h)}$ will be used to formulate the ILP constraints of *count* and *retrieve* operators in Section *Mapping Predicates* below.

Role Assignment Variables

We first introduce the variables we use to encode the solution of a generic role assignment problem. Unless otherwise noted, all variables are binary with values $\in \{0,1\}$.

The first set of binary variables x_{ik} is used to encode whether a node i satisfies the predicate c^k of a given role k:

$$x_{ik} = \begin{cases} 1 & \text{if node } i \text{ satisfies } c^k \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

By c^k we refer to the Boolean predicate describing the conditions for assuming role k as discussed in Section 2.3.2. We will ensure that (2.9) holds by translating c^k into a set of equivalent constraints on x_{ik} below.

Furthermore, we require that at least one role predicate (from the set of defined roles R) must match for every node, otherwise, the role assignment is not feasible. This gives rise to the constraint

$$\sum_{k \in R} x_{ik} \geq 1 \quad \forall \text{ nodes } i \,. \tag{2.10}$$

Note that, in order to avoid infeasibility, the programmer can always specify an `else` role q which does not imply any constraints as it does not have any sub-predicates (see Section 2.3.2). Therefore, the predicate of an `else` role would always be satisfied, and, according to the above definition (2.9), a solution should always set x_{iq} to 1.

This points to a different aspect of the variables x_{ik}: The properties and network neighborhood of a given node i could satisfy the conditions for more than one role, as we do not require the programmer to specify disjunct conditions in the predicates c^k. Therefore, more than one c^k can be satisfied, requiring – because of (2.9) – that more than one x_{ik} is set to 1 for a given node i. Thus, the variables x_{ik} cannot directly be used as an output variables of the role assignment solver.

Instead, the output variables should follow the semantics of the distributed algorithms of Section 2.4, namely, roles defined earlier in the

specification should be given priority over later ones if multiple role predicates match. These semantics imply that the *first* matching role with $x_{ik} = 1$ is assigned to a node i. This is implemented using an additional set of binary result variables y_{ik} which we require to be 1 *only* if the conditions for roles $1, ..., k-1$ (i.e., roles specified *prior* to k) are not satisfied. Here we assume that k denotes role k's position in the list of roles specified by the programmer (we can achieve this by simply re-numbering the roles). The following constraint formulates the necessity that y_{ik} must equal 1 (i.e., the role must be assigned) if the given role predicate is true ($x_{ik} = 1$) and at the same time all predicates stated earlier in the role list are false ($\sum_{l=1}^{k-1} x_{il} = 0$):

$$x_{ik} - \sum_{l=1}^{k-1} x_{il} \leq y_{ik} \quad \forall k, i \tag{2.11}$$

Vice versa, in an additional constraint (2.12), we require that role k can only be assigned if the corresponding predicate is true ($x_{ik} = 1$). Further, (2.13) requires that exactly one role from the role set R should be assigned.

$$y_{ik} \leq x_{ik} \quad \forall k, i \tag{2.12}$$

$$\sum_{k \in R} y_{ik} = 1 \quad \forall i \tag{2.13}$$

Finally, we introduce binary variables for node sets **p** that are bound to the results of retrieve predicates of the form

p == retrieve(**scope,size**) { **pred** } .

For each such set **p**, we include an additional set of variables $q_i^p(j)$ with $j \in V$ with the following interpretation:

$$q_i^p(j) = \begin{cases} 1 & \text{if the set } p \text{ at node } i \text{ contains node } j \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

To preserve a readable notation, we will sometimes omit the index p and just write $q_i(j)$ implicitly referring to the respective set p.

Objective Function

We can now reformulate the optimization criteria of the form max **role** or min **role** into an objective function in terms of the variables y_{ik}. Assume that out of a set of roles R stated in a given role assignment specifi-

cation, the user would like to minimize the set of roles $m \subset R$ and maximize the set of roles $M \subset R$ in the network. The corresponding objective function is

$$\text{minimize} \sum_{i \in V} \left(\sum_{k \in m} y_{ik} - \sum_{k \in M} y_{ik} \right).$$

Note that one may easily re-state this objective function to formulate more complex optimization criteria in terms of the variables y_{ik}. For example, one may express the desire to maintain certain ratios between `ON` and `OFF` nodes (in the coverage example), or assign weights to the importance of minimizing or maximizing certain roles.

Mapping Predicates

We will in the following show the mapping of role predicates c^k to respective ILP constraints on the predicate variables x_{ik}. Note that the constraints will depend on i because the network neighborhood is a function of the node identity i.

Consider a role predicate given in its disjunctive normal form as described in Section 2.3.2

$$c^k = \underbrace{(c^k_{11} \wedge \cdots \wedge c^k_{1n_1})}_{a^k_1} \vee \underbrace{(c^k_{21} \wedge \cdots \wedge c^k_{2n_2})}_{a^k_2} \vee \cdots . \tag{2.15}$$

As a first step, we translate the Boolean operations to ILP constraints by means of a (standard) ILP modeling technique. For this purpose, we use additional indicator variables $u_i(c)$ which (similarly to x_{ik}) indicate whether an atomic predicate c occurring anywhere in (2.15) is satisfied at a given node i.

Disjunctions and conjunctions can be expressed in terms of ILP constraints as follows. Assume a conjunctive term of the form ($c = c_1 \wedge \cdots \wedge c_p$), consisting of p terms, and let $u_i(c_1) \ldots u_i(c_p)$ be the respective indicator variables for a given node i. We will use an additional variable $u_i(c)$ to indicate whether the whole conjunction c is true. We therefore require (for every node i) that $\sum u_i(c_q) \geq p$ if and only if $u_i(c) = 1$. The

constraints modeling necessity and sufficiency are:

$$\left.\begin{array}{rcl} \sum_{q=1}^{p} u_i(c_q) & \leq & u_i(c) + p - 1 \\ \sum_{q=1}^{p} u_i(c_q) & \geq & p \times u_i(c) \end{array}\right\} \forall i \tag{2.16}$$

Similarly, for an analogous disjunction of the form $a = a_1 \vee \cdots \vee a_p$, we require that at least one of the indicator variables $u_i(a_q)$ of a node i is 1, thus $\sum u_i(a_q) \geq 1$ if and only if $u_i(a) = 1$, for all nodes i. The constraints modeling necessity and sufficiency are:

$$\left.\begin{array}{rcl} \sum_{q=1}^{p} u_i(a_q) & \leq & p \times u_i(a) \\ \sum_{q=1}^{p} u_i(a_q) & \geq & u_i(a) \end{array}\right\} \forall i \tag{2.17}$$

We will use (2.16) and (2.17) on several occasions when we need to model conjunctions or alternatives. Once conjunctions and alternatives can be modeled, the remaining task is to map the atomic predicates c_{qr}^k of (2.15) to constraints on their respective indicator variables $u_i(c_{qr}^k)$. This mapping is implemented in various ways depending on the type of the atomic predicate.

Simple Predicates. We begin with simple predicates c that are *local* in that they refer only to the properties of a given single node. These are formulated in terms of property values of the node – which are, essentially, known constants – and can therefore be evaluated before generating the ILP. Hence, the respective indicator variable $u_i(c)$ can simply be replaced by either 0 or 1.

Simple predicates that are *nested* in *count* or *retrieve* statements are special cases that will be considered under *nested predicates* below.

Count Predicates. Consider a count predicate c of the form:

`count(`**`scope`**`) { `**`pred`**` } <= `**`lim`**

In the following we formulate equivalence between $u_i(c) = 1$ and c. Let $u_j^{\texttt{(pred)}}$ be the variable indicating whether **`pred`** is true at a node j. We constrain $u_i(c)$ as follows to formulate the necessity that $(u_i(c) = 1) \rightarrow c$,

$$\sum_{j \neq i} A_{ij}^{\texttt{(scope)}} u_j^{\texttt{(pred)}} \leq (1 - u_i(c))M + \texttt{lim} \tag{2.18}$$

where M is a constant value greater than the total number of nodes n. Note that the above reduces to either (2.19) or (2.20) if the indicator $u_i(c)$ is 1 or 0, respectively:

$$\sum_{j \neq i} A_{ij}^{(\texttt{scope})} u_j^{(\texttt{pred})} \leq \texttt{lim} \tag{2.19}$$

$$\sum_{j \neq i} A_{ij}^{(\texttt{scope})} u_j^{(\texttt{pred})} \leq M + \texttt{lim} \tag{2.20}$$

The former case (2.19) exactly formulates the semantics of the above count predicate c, namely that the number of nodes j within **scope** that match **pred** should be less or equal to **lim**. In the latter case (2.20), the constraint is nullified by M, as $M > n$ and the left-hand sum will never be larger than the number of nodes.

The above equations (2.19) and (2.20) formulate the necessity that $u_i(c) \rightarrow c$. In a second step, we formulate the sufficiency $c \rightarrow u_i(c)$, equivalent to $u_i(c) \vee \neg c$, using the constraint

$$\sum_{j \neq i} A_{ij}^{(\texttt{scope})} u_j^{(\texttt{pred})} \geq (\texttt{lim} + 1)(1 - u_i(c)) \tag{2.21}$$

which requires $\neg c$ if $u_i(c) = 0$ while imposing only a trivial constraint if $u_i(c) = 1$. In summary, the constraints (2.18) and (2.21) model a count operator that uses the relation $\leq$. Note that count operators using other relations can be treated analogously.

Retrieve Predicates. A retrieve predicate c of the form

```
p == retrieve(scope,size) { pred }
```

evaluates to true if all of the following three requirements are met.

First, at least **size** nodes must exist within **scope** that match the given nested predicate **pred**. We model this requirement as a count statement of the form

```
count(scope) { pred } >= size
```

Second, we must ensure that every retrieved node j that is in the set **p** of node i really is in **scope** and also matches the predicate **pred**

$$q_i^{\texttt{p}}(j) \leq A_{ij}^{(\texttt{scope})} u_i^{(\texttt{pred})}. \tag{2.22}$$

Third, we require that the number of elements in the set **p** is **size**

$$\sum_{j=1}^{n} q_i^{(\mathbf{p})}(j) = \mathbf{size}. \tag{2.23}$$

Finally, we can formulate that $u(c) = 1$ if and only if all of the above requirements hold at the same time using the approach for implementing conjunctions given in (2.16).

Nested Predicates. In the following consider predicates that occur nested in *count* or *retrieve* statements with a given **scope**.

For a nested predicate c that refers to the *role* of the node, i.e., `role==`k, no separate indicator variable $u_j(c)$ is needed. We can instead re-use the existing result variable y_{jk} which indicates whether the role of a node j is k.

A special case are simple predicates c that check equality between two properties that represent node sets such as

clusterheads == **super.clusterheads**

in the clustering example.

To model set equality, we introduce additional indicator variables Q_{ij} that are set to 1 if and only if the set **clusterheads** of node i is equal to the set **super.clusterheads** at node j. The interesting case is when the two sets are bound by retrieve predicates (as in the above clustering example), and thus constitute variables of the algorithm. In these cases, the parameter **size** of the respective retrieve predicates indicates the size of the set (we assume the sets are of equal size, otherwise the compiler can already set $u(c)$ to 0). In all other cases Q_{ij} can be evaluated before generating the ILP and simply noted as 1 or 0, respectively.

In the following, we describe the constraints on Q_{ij}. These are formulated in terms of the variables $q_i(k)$ representing the set of node i, where $q_i(k) = 1$ if and only if the element $k \in \{1,...,n\}$ is contained in the set at node i, as defined in (2.14). Likewise, respective variables $q_j(k)$ for node j are used. We first compute a set of n helper variables $q_{ij}(k)$ for which we require that $q_{ij}(k) = q_i(k) \wedge q_j(k)$ using equation (2.16). Figure 2.13 shows an example with two sets of size 2 in which black circles indicate that the respective $q_*(k) = 1$.

Using $q_{ij}(k)$, we can formulate that variables Q_{ij} should be 1 if the intersection of the sets at node i and j contains at least **size** elements (i.e., if the sets are equal):

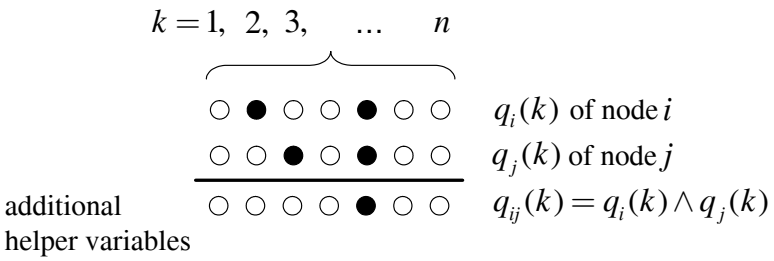


Figure 2.13: Set equality

$$\sum_{k=1}^{n} q_{ij}(k) \leq Q_{ij} - 1 + \texttt{size} \tag{2.24}$$

Otherwise, Q_{ij} should be 0:

$$\sum_{k=1}^{n} q_{ij}(k) \geq \texttt{size} \times Q_{ij} \tag{2.25}$$

Finally, we can replace $u_j(c)$ that occur nested within a `count` or `retrieve` statement of a given node i by the corresponding Q_{ij}.

Complexity

Depending on the complexity of the role specifications, the resulting ILPs are solvable in reasonable time for up to 1000 nodes using the CPLEX [CPX] commercial solver. Some specifications, however, can result in rather large ILPs with many indicator variables and constraints. This is particularly the case with specifications that contain *retrieve* predicates. Below we therefore analyze how the above mapping could be implemented using a much lower number of variables – an improvement which we have so far omitted for ease of exposition. The number of variables can be used as an indicator for the compute time of a generated ILP.

In the above, we used a total number of $n \times m$ variables to encode the basic role assignment decision, where m denotes the number of specified roles and n denotes the number of nodes in the network. Further, we added another set of n variables for each node, to encode local properties containing node sets that are bound by retrieve statements, resulting in another $n^2 \times b$ variables, where b represents the total number of different node sets used in the role specification.

Moreover, in the above mapping of role predicates, we introduced a number of additional indicator variables. A total of $n \times a$ variables are used to encode atomic predicates at each node, where a is the number of

atomic predicates in the specification. The most complex atomic predicate – set equality – required additional n indicator variables for each *pair* of nodes, summing up to n^3 variables.

However, one can reduce the number of variables by exploiting the locality (i.e., limited scope of count and retrieve predicates) inherent in the role specification language. This is comparable to the way the distributed algorithms of Section 2.4 exploit the locality stated in the specification in that their overhead is strictly dependent on the size of the scopes used in *count* and *retrieve* predicates.

Similarly, the number of variables needed for the ILP depends on the scope sizes used in the role specification. Consider a retrieve statement binding a local property **p**. Due to the limited scope, rather than n variables, each node i will only need k_i variables to encode **p**, where k_i corresponds to the number of nodes in the *scope*-hop neighborhood of i as illustrated in Figure 2.14. Moreover, the set of nodes that needs to be encoded in the helper variables q_{ij} is even smaller, as $q_{ij}(p)$ must only be provided for nodes p which are located in the intersection of the scopes of nodes i and j.

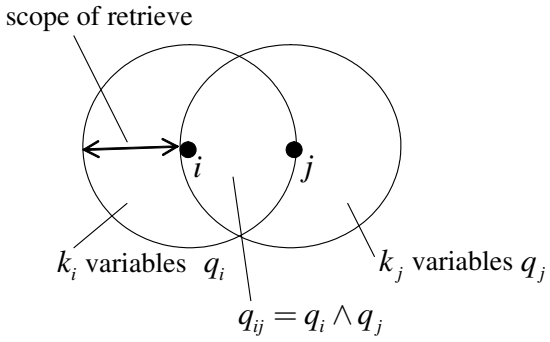


Figure 2.14: The number of variables required for modeling set equality depends on the specified scopes

With these improvements, we obtain a much smaller number of variables and coefficients. For example, for a 100 node network with nodes being placed randomly in a 300 m by 300 m area, the ILP implementing the clustering specification from Section 2.3.1 contains only 2428 variables (rather than 100^3) and is solvable within seconds using CPLEX [CPX].

2.8.2 Evaluation

We extended the existing role assignment development environment described in Section 2.5 with the capability of generating an ILP representing the execution of a given specification on a given topology. The generated ILP is formulated using the ZIMPL [Koc05] modeling language, which we use as a front end for the CPLEX solver. Finally, the development environment visualizes the ILP's results on the respective network topology.

Using the above ILP mapping of role specifications, we can re-examine the role assignment specifications regarding their *optimality*, *feasibility* and *termination*, which were open problems so far.

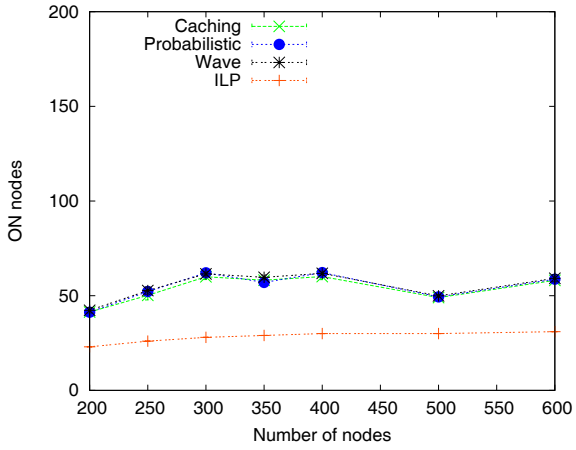
One new aspect of the generated ILP is that it enables the programmer to express a desired *optimality* goal, i.e., to minimize or maximize the number of nodes that are assigned a certain role. While the existing specifications were not devised with a possible optimization in mind – rather, these were designed as input for distributed algorithms which assign roles in a greedy fashion –, it is nevertheless interesting how the *optimal* results compare to the ones found using the distributed algorithms of Section 2.4.

As illustrated in Figure 2.15, we examined the minimum number of `ON` nodes required to ensure coverage based on the *coverage* specification that was used in the evaluation of our distributed algorithms in Section 2.6. As in earlier evaluations, nodes with a communication range of around 33 m were randomly distributed in a square area of 300 m by 300 m.

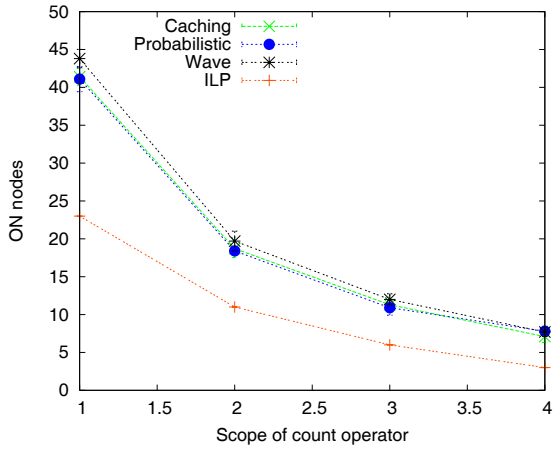
Previously, we studied the same specification with several distributed algorithms (*caching*, *probabilistic* and *wave*) in terms of their efficiency regarding communication traffic, but did not employ any strategy to minimize the number of `ON` nodes. It is therefore not surprising that the ILP approach can yield solutions with about half as many `ON` nodes as the distributed algorithms.

As mentioned, the distributed algorithms assign roles in a greedy fashion and do not attempt any optimization. If developers are interested to improve the quality of the results obtained by the distributed algorithms, they may add additional conditions to the role specifications (using the original specification techniques of Section 2.3). In this regard, the optimal solution obtained via the ILP indicates the room for potential improvement using more refined role specifications.

Moreover, the role assignment solver can provide interesting insights into undesirable effects of a role specification. While the clustering spec-



(a) Increasing node density



(b) Increasing scope

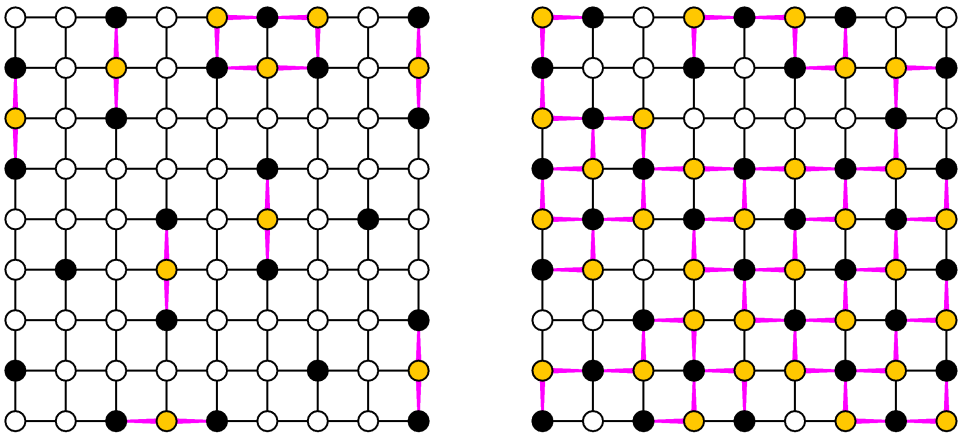
Figure 2.15: Number of ON nodes in simplified coverage example

ification (re-stated in Figure 2.16 for convenience), when interpreted by the distributed algorithms, typically results in a connected backbone network of clusterheads and gateways (Figure 2.17(b)), minimizing both the `CH` and `GW` roles results in a set of isolated clusters (Figure 2.17(a)), which are usually not desirable in practical applications. The optimized assignment places clusterheads exactly three hops away from each other, thus ensuring that each slave is a neighbor of at least one clusterhead, while at the same time avoiding assignment of the gateway role (as hardly any node has more than one clusterhead neighbor). This hints at a weakness of the specification, namely that it does not enforce that clusterhead and gateway nodes should provide a *connected* backbone.

For the case that clusterhead and gateway roles are maximized, an exemplary solution is shown in Figure 2.17(c). Here, clusterhead nodes are exactly two hops away and arranged in an ordered fashion that allows for many gateway nodes. Essentially, Figures 2.17(a) and 2.17(c) show

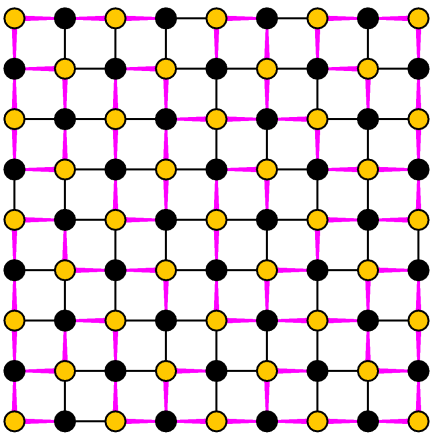
```
1  CH :: {
2    count(1 hop) {
3      role == CH
4    } == 0 }
5  GW :: {
6    clusterheads == retrieve(1 hop, 2) {
7      role == CH
8    } &&
9    count(2 hops) {
10     role == GW &&
11     clusterheads == super.clusterheads
12   } == 0 }
13 SLAVE :: else
```

Figure 2.16: Original clustering specification



(a) Minimizing

(b) Greedy



(c) Maximizing

Figure 2.17: Clustering results on a 9x9 grid, minimizing vs. maximizing clusterheads and gateways (clusterheads are black, gateways are yellow, slaves are white, and edges between gateways and their clusterheads are emphasized)

two extremal outcomes that may occur with the current clustering specification – hinting the developer that there is a potential problem with this specification.

The clustering specification performs more accurately with distributed algorithms that greedily assign roles as seen in Figure 2.17(b) – as it was originally designed as an input to these algorithms. With their semantics, the clustering specification results in a connected backbone in most cases. However, there is no guarantee that the backbone is always connected. We therefore discuss a set of specification heuristics that overcome this weakness in Section 2.9 below.

Enabling the programmer to formulate an additional constraint, namely that a set of roles should be *connected*, would be an interesting extension for the solver component. Such a feature would complement related work [MLM+05] which estimates whether a set of roles would form a connected backbone in the average case.

Finally, the ILP translation helps to better understand erroneous specifications that do not *terminate* when using the current distributed algorithms. Consider the mentioned `RED-GREEN` example for which the distributed algorithms of Section 2.4 would change back and forth between the roles `RED` and `GREEN`:

```
1 RED :: { count(1) { role == GREEN } >= 1 }
2 GREEN :: { count(1) { role == RED } <= 0 }
```

Using the ILP mapping of the above specification we could show that the specification is in fact *infeasible* – that is, no solution of the generated ILP exists – on a range of different network topologies.

Apart from evaluating feasibility on a given network graph, we can also reformulate the ILP to check whether, given a certain number of nodes, there exists *any* network graph for which the problem has a solution. By doing so for the above specification, we obtained graphs consisting of single isolated `GREEN` nodes as the only feasible combination of network topology and role specification. When adding the (practical) requirement that every node should have at least one neighbor, no solution can be found, indicating that the specification is infeasible on *any* reasonable network topology.

For illustration of this result, consider the simplest graph with two nodes u, v and an edge (u,v) as shown in Figure 2.18. Looking carefully, we can see that, out of the 4 assignments that are possible in this graph, none is feasible. While the last example (both nodes green) seems feasible at first,

Figure 2.18: RED / GREEN example: None of the four possible role assignments satisfies the role specification

it is not: As both nodes also match the predicate for RED, the assignment does not comply with the requirement that the *first* matching role should be assigned.

This example suggests that there is a close relationship between specifications that do not terminate when executed by the distributed algorithms and specifications that result in infeasible ILPs. What is certain is that infeasible specifications cannot provide any correct result with the distributed algorithms, and should therefore be discarded before distributing them to the network.

Unfortunately, it is uncertain whether a complementary claim also holds, namely, that *one* feasible solution implies fast convergence of the distributed algorithms. Nevertheless, there is an argument supporting this claim. First, note that the convergence speed is likely to be dependent on the *number* of feasible assignments in the network. If the number of feasible assignments is very small, the distributed algorithms are unlikely to find a correct solution. Instead, they will either end up in a partly incomplete configuration (where some nodes have taken on feasible roles and later on some nodes are assigned the role *undefined* as, based on the specification, no feasible roles for them remain), or in an endless iteration (as in the RED-GREEN example). If the number of feasible solutions is large, however, the algorithm is likely to converge to a feasible configuration fast.

In this context, we argue that a feasible specification will usually have many feasible solutions, not just a few. This has to do with the symmetries of the system, which are caused by the fact that the *same* specification is distributed to all nodes: If for a given specification at least *one* feasible assignment exists on each of a range of different network topologies, these assignments can be shifted, rotated, tiled, or combined into a plethora of feasible solutions. This argument is consistent with the results of our evaluation of the distributed algorithms (Section 2.6), in which the convergence period always turned out very short.

In particular, note how the most useful role assignment specifications (such as the coverage, clustering, and aggregation specifications we evaluated in Section 2.6) focus on choosing *special* nodes that perform a specific task (such as the *on*, *clusterhead*, or *aggregator* nodes, respectively). These specifications allow for a particularly high number of feasible solutions, because it does not matter which node from a set of network neighbors is elected *special*. Once an arbitrary initial choice of special nodes has been made, a feasible solution, in which the special nodes' neighbors are assigned *slave* or *gateway* roles, can be found easily.

2.9 Additional Role Specifications and Language Extensions

The previous sections have described and evaluated a comprehensive system for assigning roles to sensor nodes. Based on the role assignment solver of the previous Section 2.8, a deficiency of the example specification we used for clustering has become apparent, namely, that it does not guarantee that clusterhead nodes are connected.

Before concluding this chapter, we therefore briefly discuss a few examples of additional role assignment specifications. These should underline the generality of the system and provide important building blocks from which developers can derive their own specifications consisting of combinations and variations of the specifications provided here.

Further, we discuss a set of refinements of these specifications, which we use to point out valuable extensions to the role assignment language. These extensions would be easy to embed into the distributed role assignment algorithm and at the same time would make it more flexible and generic.

Data Gathering Tree. We have mentioned that generic role assignment can be used to construct trees. Consider the following simple specification:

```
1 CHILD   :: {
2   parent == retrieve(1 hop, 1) {
3     role == SINK ||
4     role == CHILD
5   }
6 DISCONNECTED :: else
```

The specification requires that `CHILD` nodes are either neighbors of the network's sink or of other `CHILD` nodes. Because all nodes start in an

undefined role, the role `CHILD` implies that the node is truly connected to the sink. Because every node except the sink picks one parent, the above specification implements a tree that grows from the sink. The local property `parent` is used to retain a node's parent in the tree and to make it available to other applications by means of the node's property directory.

Note that after constructing the initial configuration, after a failure of a parent node it may happen that `CHILD` nodes re-connect to other `CHILD` nodes forming a cycle which is not connected to the sink. This is a common problem in tree construction protocols and can be addressed by adding a local `counter` property that stores a node's hop distance to the sink. Every node would pick a parent whose counter is smaller than its own and, further, set its own counter to the one of its parent plus one. Such details could be implemented using the syntax we introduce in this section. Note that we omit these details, when we implement trees in the specifications below.

Connected Clustering. The clustering specification, which we introduced in Section 2.3.1, can be enhanced by constructing a tree in a similar manner.

We mentioned that the original specification does not guarantee that a *connected* communication backbone of clusterheads and gateways is constructed. Instead, it relies on heuristic constraints on the gateway role that shall provide a connected topology in most cases and yet keep the total number of gateway nodes low.

By a slight adaptation of the specification, however, a connected topology can be obtained. Consider the enhanced clustering specification shown in Figure 2.19, which only adds the lines 5–8 to the original clustering specification introduced in Section 2.3.1:

The additional requirement for the assignment of the clusterhead (`CH`) role (lines 5–8) is derived from the specification of the `CHILD` role above. A node is allowed to become a clusterhead *only* if a "parent" clusterhead (or the sink) is present within two hops of the local node. This way, based on the provided specification for gateways (`GW`), later on at least one `GW` node will be elected connecting the new clusterhead to its parent.

Clustered Tree. The above combination of the tree and clustering specifications hint at a refined version of this specification. For example, the specification of the gateway role `GW` still focuses on connecting *arbitrary* pairs of clusterheads. If a developer, instead, prefers to construct a clustered tree rooted at the sink, and does not desire any additional inter-clusterhead gateways, the specification shown in Figure 2.20 can be used.

```
CH   :: {
  count(1 hop) {
    role == CH
  } == 0 &&
  count(2 hop) {
    role == SINK ||
    role == CH
  } >= 1  }
GW :: {
  clusterheads == retrieve(1 hop, 2) {
    role == CH
  } &&
  count(2 hops) {
    role == GW &&
    clusterheads == super.clusterheads
  } == 0  }
SLAVE :: else
```

Figure 2.19: Connected clustering specification

```
CH   :: {
  count(1 hop) {
    role == CH
  } == 0 &&
  parent = retrieve(2 hop, 1) {
    role == SINK ||
    role == CH }
GW :: {
  clusterheads == retrieve(1 hop, 2) {
    role == CH
  } &&
  count(2 hops) {
    role == GW &&
    clusterheads == super.clusterheads
  } == 0 &&
  (clusterheads(1) == clusterheads(2).parent ||
   clusterheads(2) == clusterheads(1).parent)  }
SLAVE :: else
```

Figure 2.20: Clustered tree specification

Compared with Figure 2.19, line 5 now binds the `parent` clusterhead in the tree using `retrieve`. Moreover, an additional requirement has been added to the gateway role `GW` in lines 16–17. This statement requires that among the two `clusterheads` connected by a potential gateway, the *first* should be the parent of the *second* or vice versa. This way, once a tree of clusterhead roles has been established – in which clusterheads are always two hops apart – `GW` nodes will simply connect the original tree constructed using the `CH` role. In this specification, more nodes can be assigned the energy efficient `SLAVE` role, as much fewer nodes function as gateways.

The statements in lines 16–17 use an extended syntax to access the members of the set `clusterheads` in an ordered manner. This syntax would be mapped onto set operations performed by an enhanced property directory. In order to support the notation used in lines 16–17, the property directory must store node sets in some arbitrary order and provide for access to individual elements of these sets. While such features are not present in our current implementation, we expect that an existing (and more powerful) property sharing component can be used for this purpose, especially because various literature has argued for adequate node-level components that are conceptually similar to our property directory [KHHK04, LMMR05]. Vice versa, many other systems (other than role assignment) could benefit from a more powerful implementation of the property directory and, in particular, from data structures that facilitate dealing with nodes, sets of nodes, and properties of these nodes.

In the example of lines 16–17, any deterministic ordering of the nodes in the `clusterheads` set would suffice (e.g., based on node identifiers). If the developer could additionally specify which node property should be used to impose an order on the nodes in the set, even more powerful specifications become possible. In the remaining examples, we will use the syntax

`nodeset(`**`index,prop`**`)`

where the above expression references the node at position **`index`** in `nodeset` when the elements of `nodeset` are ordered by ascending values of the node property **`prop`**. When we used the expression `clusterheads(1)` in the specification of the clustered tree above, in the default case, node identifiers were used to induce an order on the set `clusterheads` – spelled out, the expression would state `clusterheads(1,id)`, where `id` represents a property in the nodes' property directory which contains the node's identifier.

Best Clusterhead. Using such expressions referring to members of node sets, many simple yet effective applications can be formulated. Consider the following alternative clustering specification:

```
1 CH   :: {
2   count(1 hop) {
3     role == CH
4   } == 0 &&
5   connectedClients == count(1 hop) {
6     role == SLAVE &&
7     chId == super.id
8   }
9 SLAVE   :: {
10  clusterheads == retrieve(1 hop) {
11    role == CH
12  } &&
13  chId == clusterheads(1, connectedClients).id }
```

In the above specification, we augmented the clusterhead role with a counter `connectedClients` for counting slave nodes connected to the current clusterhead (line 5). The respective `count` expression counts `SLAVE` nodes whose associated clusterhead – which is stored by `SLAVE` nodes in their local property `chId` – matches the identifier of the current clusterhead (`super.id`) in line 7.

Using the `connectedClients` property, we can specify that `SLAVE` nodes should select the clusterhead node which has the least number of slaves connected to it. This is done in line 13. From the set `clusterheads`, the identifier of the *first* node is selected – while the order of clusterheads is imposed by their property `connectedClients`.

We would like to note that the above specification is unstable, because only very slight changes in the number of connected clients could cause slaves to switch their clusterhead (and again cause the number of connected clients to change). However, this instability occurs only when two clusterheads have near equal numbers of connected clients and can therefore be approached the same way as with other properties that undergo many small changes (cf. Section 2.4.5): The programmer may specify a tolerance threshold $\Delta(\mathit{connectedClients})$, which would force the algorithm to abstain from re-configuration if *connectedClients* changes by less than the specified Δ.

Coloring. A common way to reduce interference among wireless transmitters is to assign different channels to neighboring nodes. Such channel assignments, usually modeled by vertex coloring problems [KV02],

are particularly useful if the used sensor node hardware includes multiple transceivers [BKM+04] or if node transceivers support multiple communication channels [Tmo05], as the assignment of different channels to neighboring nodes can effectively improve network throughput. A specification for selecting a low-interference channel[2] is shown below:

```
1 C1   :: {
2   count(1 hop) {
3     role == C1
4   } == 0 }
5 C2   :: {
6   count(1 hop) {
7     role == C2
8   } == 0 }
9 C3   :: {
10  count(1 hop) {
11    role == C3
12  } == 0 }
```

The above specification simply assigns the roles `C1` to `C3`. According to the semantics of the distributed algorithms presented earlier in this chapter, the role `UNDEFINED` will be assigned if no interference free channel (`C1`-`C3`) should be available because all channels are allocated to neighboring nodes.

This semantic detail constitutes an important difference to full-fledged coloring algorithms: Given the above specification, our distributed algorithms will not always find an assignment of channels to network nodes, even if such an assignment (a 3-coloring) exists. Instead, the algorithm will quickly iterate to *some* solution in which nodes will stay in `UNDEFINED` if previous role assignments in the network neighborhood happen to have "consumed" the three available channels.

Alternatively, instead of just choosing `UNDEFINED`, an algorithm could initiate re-configuration at neighboring nodes, which would eventually find a feasible 3-coloring solution if one exists. In our initial design, we decided against enforcing such "strong" semantics because these come at a very high cost: A role assignment problem which has only few (or only one) feasible solution would be caught up in harmful iterations through many role changes. These strong semantics are implemented only in the centralized solver component, which is more powerful. Using the above specification as an input, the centralized solver can compute a three-coloring if one exists or otherwise return *infeasible*.

[2] In this example, we assume that a main communication channel performing role assignment and routing has been selected beforehand and remains unchanged by this specification.

In contrast, having nodes locally revert to UNDEFINED if no other role is feasible, removes a large amount of the interdependence between neighboring nodes and allows the algorithms to converge quickly. Moreover, these "weaker" semantics are quite useful. For example, in sparse networks in which many feasible role assignments according to the above specification exist, the distributed algorithm will very quickly converge to one of them – or to a solution in which only very few nodes remain with UNDEFINED. Moreover, the application developer may provide custom code for nodes in the UNDEFINED role, i.e., which lets the node sleep until a channel becomes available.

We close this section with an example specification of the above problem for a larger number of channels:

```
1 CHANNEL-ASSIGNED   :: {
2   neighbors == retrieve(1 hop) {
3     role == CHANNEL-ASSIGNED
4   } &&
5   channel ==
6     draw-from {1..MAX-CHANNEL} \ neighbors.channel
7 }
8 UNDEFINED :: else
```

The above specification demonstrates a few syntactic features, which can be used to render the existing role assignment system more flexible. These features, as above, can be mapped onto local (possibly custom-implemented) library functions or to more powerful operators provided by the local node's runtime. In this example, the function `draw-from(`**`S`**`)` chooses a random element from a given set **S**, or evaluates to false if $\mathbf{S} = \emptyset$.

The operation `{1..`n`}` constructs a set consisting of the values 1 through n. If a local property **P** is of type node set, the expression **P.x** can be used to generate a set of all different node properties **x** that occurred at nodes in the set **P**. Note that the expression `neighbors.channel` is well-defined, as the check for the CHANNEL-ASSIGNED role (line 3), based on short-circuit semantics, ensures that the property `channel` is set properly at all nodes in `neighbors`.

The symbol '\' refers to set subtraction. In summary, the expression in lines 5–6 chooses a random channel from the channels not used by nodes in the one-hop neighborhood of the current node.

In the above, we have listed a set of specification examples that should underline the initial claim that role assignment is a generic concept that can be useful for a wide range of applications. In some of these specifications, we introduced slight extensions to the specification syntax, which

we deem worthwhile implementing in future work, as we expect them to provide for a new set of important applications. All envisioned extensions involve only local operations that could be supported by a more powerful local runtime environment on the network nodes, while the basic role assignment protocol (based on repeated property propagation and subsequent rule evaluation based on a nodes local knowledge) could still be used to support these specifications as well.

2.10 Summary

In this chapter, we have investigated a novel programming abstraction called generic role assignment, which automatically assigns roles to sensor nodes based on their properties and the properties of their network neighborhood.

We have presented a distributed algorithm for role assignment and two variations that perform probabilistic initialization. Through an extensive quantitative evaluation, we have shown that role assignment is not only a powerful tool, but can also be implemented in an efficient and robust way. Further, we described an extension to our role assignment system that maps a given role assignment instance – consisting of a network topology, a set of node properties, and a role assignment specification – to an integer linear program formulation. This extension provides developers with a valuable tool for analyzing the feasibility, optimality, and termination of role assignment specifications.

A particularly noteworthy application of generic role assignment is rapid prototyping for sensor networks. Currently, the deployment of sensor networks often involves a trial-and-error phase, where algorithms and protocols are tested in different configurations in real-world settings (e.g., [WAJR+05]). With generic role assignment, these different configurations could be easily generated and changed. While all components under test could be loaded onto the nodes before deployment, these could be started and stopped through the assignment of certain roles and initialize their respective configuration parameters from the property directory.

As we have argued, one desired property of role-based network configuration is that it obeys certain global optimization criteria, for example, minimizing the number of required *on* nodes while maintaining coverage. While such optimization can be performed using the provided solver component, we have not provided such a feature in our distributed algorithms.

The distributed generic role assignment approach instead focuses on ease of use: It has been designed to allow developers to rapidly prototype configuration *heuristics* whose quality is sufficient. For example, the abstraction can well express various clustering heuristics. This includes the clustering specifications we presented in this chapter as well as the connected dominating set heuristics used by passive clustering [KG02], or a maximum independent set (implemented by our system if one specifies that *on* nodes are not allowed to have *on* neighbors). A maximal independent set can be used to approximate the minimum dominating set problem as (in unit disk graphs) it is only a constant factor worse than the optimal configuration [Mos07].

Nevertheless, a generic algorithm that computes optimal solutions for a wide-range role-based configuration problems is an intruiging goal. Based on the variety of applications which can be expressed using the role specification language, devising an algorithm that is applicable to all conceivable specifications remains a challenge. As it turns out, however, such an algorithm can be provided for a key subclass of role-based configuration problems, as we describe in the next chapter.

3 Distributed Facility Location

In an important subclass of role-based configuration problems, a set of network nodes must be chosen to provide a service to their network neighborhood. Many application examples we have discussed in the previous chapter belong to this class, such as *clustering* [BMP04], where cluster heads serve as communication hubs for nearby nodes, or *aggregator placement* [MFHH02], where some nodes collect and aggregate sensor data from nearby sensor nodes. Recently, tiered sensor networks [GGJ+06] have been proposed, consisting of resource-poor sensor nodes in the first tier and powerful hub nodes in the second tier. In these networks, every sensor node is assigned to and controlled by a hub node. Note that in all of the above examples, configuration consists in electing some nodes as *servers* while the remaining *client* nodes are assigned to a server.

While, next to generic role assignment, many specialized proposals exist for finding such network configurations, these approaches often do not pay attention to optimizing the *overall cost* of these configurations, which consists of two components: on the one hand, the costs of operating the servers (e.g., representing the servers' increased communication load as these forward traffic for many clients), and, on the other hand, the costs of communication between clients and their server. In wireless networks, the latter cost can be dependent on the physical distance between a client and its server (as a longer wireless link requires higher transmit power and thus increased energy consumption), on the number of hops in a multi-hop network graph, or on interference and network congestion. In all cases, lowering communication costs by means of additional server nodes may prove beneficial.

The goal of this chapter is to provide a *generic* and *practical* mechanism for finding cost-optimized solutions to the above self-configuration problems. Our approach is based on the observation that the above optimization problem can be modeled as an (uncapacitated) *facility location problem* [Vyg05, Vaz03]. There, we are given a set F of *facilities*, a set C of *clients* (also known as cities or customers), a cost f_i for open-

ing a facility $i \in F$ and connection costs c_{ij} for connecting client j to facility i. The objective is to *open* a subset of facilities in F and connect each client to an open facility such that the sum of connection and opening costs is minimized. Open facilities represent server nodes which can be used to provide a particular service to network neighbors (just like `clusterhead` nodes in the previous chapter). The remaining client nodes correspond to nodes assigned the role `slave`.

Although the facility location problem has been studied extensively in the past, no *practical* solutions exist that would be suitable for multi-hop sensor networks. While distributed algorithms for facility location exist, they are either not generally applicable [GLS06], require a certain (albeit small) amount of global knowledge [MW05], require impractical communication models [MW05, CEPS05], or (based on the provided approximation factor [CEPS05]) might not improve over the heuristic clustering specifications we provided in the previous chapter – or over clustering heuristics from related work.

In this chapter, we therefore contribute a local facility location algorithm that lends itself well for implementation in multi-hop sensor networks, and provides an approximation factor of 1.61 for metric instances (cf. Section 3.1). By means of an experimental study, we show that, for typical problem instances derived from sensor network configuration problems, the algorithm provides near-optimal solutions and terminates after few communication rounds.

While the above view adopts a static graph model of sensor networks, practical sensor networks are rather dynamic: Nodes may fail and the quality of wireless links fluctuates over time. To make our algorithm applicable to such realistic settings, we propose a set of rules to repair a sensor network configuration in case of node failures, additions, and link quality changes. Further, we evaluate our algorithms and the proposed dynamic adaptation rules using link quality traces obtained from an operative sensor network deployment.

Compared with the previous chapter, the provided facility location algorithms export a narrower configuration interface to application developers. Instead of role specifications, developers are expected to provide a set of opening and connection cost parameters or, alternatively, a custom-implemented function which allows nodes to compute the respective cost parameters locally. In return, the underlying implementation can provide intriguing performance guarantees, which constitute the main motivation of this chapter.

3.1 Model and Applications

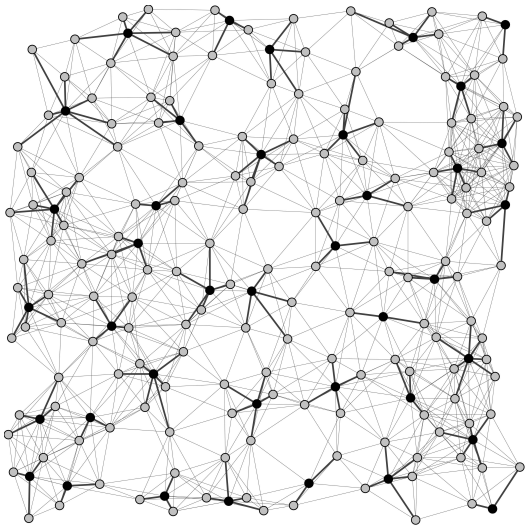
The facility location problem can be used to model configuration decisions in multi-hop networks in a variety of ways. As before, we model the multi-hop network subject to configuration as a graph $G = (V,E)$. Typically, a network node takes on the role of a client and that of a potential facility at the same time, that is, $F = C = V$. If a node $i \in V$ is selected as an open facility, it will take on the role of a *server*. The remaining slave nodes will be associated to a certain server node in their network neighborhood.

In some cases, only a subset of nodes have the necessary capabilities (e.g., remaining energy, available sensors, communication bandwidth, or processing power) that make them eligible as a facility, which results in $F \subseteq C = V$. The role assignment framework presented in the previous Chapter 2 may be used to compute the set of potential facility nodes F. The algorithms of this chapter can then be used to open a subset from the nodes F which serves the network nodes C efficiently.

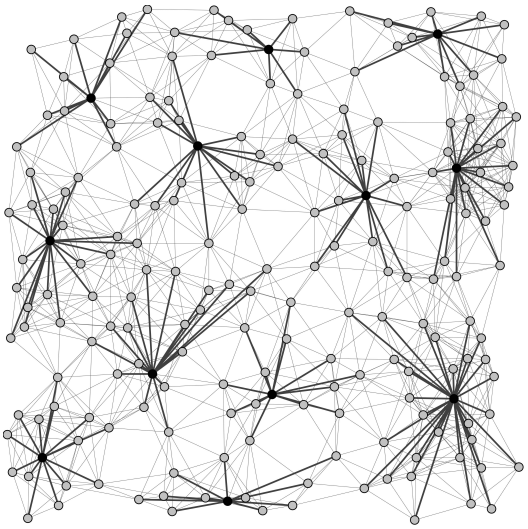
The respective settings of opening costs f_i and connection costs c_{ij} can implement various applications. In most cases, connection costs c_{ij} of an existing a network link $(i, j) \in E$ represent a link quality metric that can be determined locally at the nodes. In the application examples of this chapter, we assume that the nodes can control the power they use for sending. Therefore, we set connection costs c_{ij} as proportional to $g(i, j)^2$ where $g(i, j)$ denotes the distance in m between i and j – implementing the energy-level required to send messages from j to i.

For pairs of nodes (i, j) which cannot directly communicate in the underlying network graph, i.e., $(i, j) \notin E$, different settings of c_{ij} allow to specify desired configurations. Setting $c_{ij} = \infty$ when $(i, j) \notin E$ would require that every client has a facility as a direct neighbor. We will denote this problem formulation as *one-hop*. An example configuration using such parameters is shown in Figure 3.1(a).

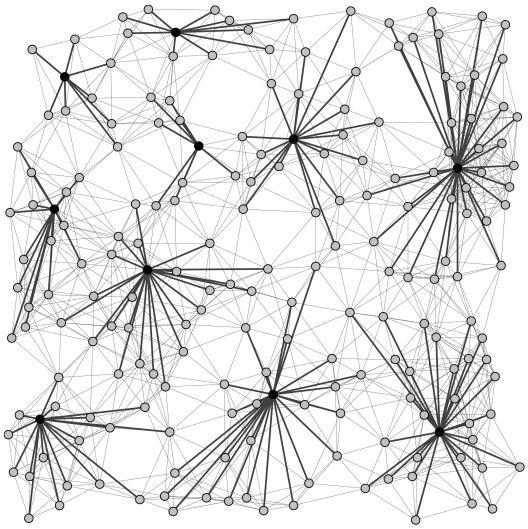
Alternatively, the network developer may be interested in allowing nodes to connect to facilities which are an arbitrary number of hops away – as long as the communication costs adequately represent the energy required to communicate between i and j. To implement such configurations, c_{ij} can be set to the shortest network path between i and j. The required shortest-paths computation will be achieved using a local flood around a given node i. We denote this problem as *multi-hop* and show an example configuration in Figure 3.1(b).



(a) *One-hop* example, $f_i = 1$



(b) *Multi-hop* example, $f_i = 5$

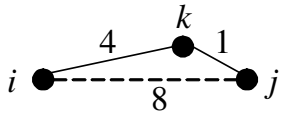


(c) *Multi-hop* example, $f_i = 2 \times D(\text{sink}, i)$

Figure 3.1: Effects of different opening cost parameters; $D(sink, i)$ denotes the shortest-path distance to the sink, which is located in the upper left corner

Depending on the particular setting of c_{ij} one may obtain instances of the facility location problem which are not *metric*. The definition of a *metric* instance is that connection costs obey the triangle inequality, which requires that for any three nodes i, j, k the direct path is shorter than a detour ($c_{ij} \leq c_{ik} + c_{kj}$).

If the input to a facility location algorithm is non-metric, the problem is particularly hard to solve (see Section 3.2 below). Particularly, the *one-hop* setting results in a non-metric instance because of two reasons: The first is that, for two non-neighboring nodes, the one-hop path is modeled by infinite costs, while a multi-hop path exists. The second is that setting connection costs proportional to the squared geographic distance between two nodes results in non-metric instances, for example:



In turn, the *multi-hop* problem formulation, in which c_{ij} are computed on the fly to correspond to the shortest path between i and j, represents a *metric* instance of the facility location problem.

Finally, the opening costs provide developers with an additional lever to express properties of the configuration they desire. For example, the opening costs f_i of a node i can represent the communication effort involved if this node were to become a service provider for its network neighborhood. In a network in which open facilities represent clusterheads which must keep their receivers on at all times to receive traffic from associated clients, f_i can quantify the energy required for operating the node's transceiver. This is usually a multiple of c_{ij}, the effort involved in a client node's communication with its clusterhead in a given epoch.

Moreover, opening costs f_i may take into account the effort involved if clusterheads were to forward data to the network sink. In Figure 3.1(c), we show an example of a *multi-hop* configuration in which opening costs f_i are proportional to the node i's network distance to the sink. Such parameterization is useful if one assumes that clusterheads must perform repeated communication with the sink which, based on aggregation, does not increase significantly with the number of connected clients. In this setting, it is efficient to let facilities that are far from the sink connect more clients at higher connection costs, as shown in the results of Figure 3.1(c).

In the course of this chapter, we will use the three parameterizations sketched in Figure 3.1 to evaluate a set of distributed facility location al-

gorithms which can be applied to the above *one-hop* and *multi-hop* problem settings.

3.2 Related Work

An ample amount of literature exists on *centralized* approximation algorithms for the NP-hard facility location problem [Vyg05]. Such algorithms are not applicable as collecting a global view of the network topology at a single point (e.g., at the network basestation) represents a prohibitive communication overhead.

For *non-metric* instances of the facility location problem, even approximations are hard to come by: As the set cover problem can be reduced to (non-metric) facility location, the best achievable approximation ratio (even with a centralized algorithm) is logarithmic[1] in the number of nodes [Fei98]. A classic and simple algorithm [Hoc82] already comes close to this lower bound. Distributed approximations are rare: [MW05] solve non-metric facility location even in a constant number of communication rounds. However, the algorithm requires that a coefficient ρ, which is computed from a global view of the problem instance, is distributed to all nodes before algorithm execution – which prevents it from being used "as-is" in practice. Moreover, the algorithm requires global communication among all relevant clients and facilities and therefore can only efficiently be used in the *one-hop* setting where such communication can be implemented efficiently by wireless broadcast. Finally, the best approximation factor it can obtain, which is independent of the problem instance, is on the order of $O(\log(m+n)\log(mn))$ where m and n denote the number of facilities and clients, respectively.

For *metric* instances of the facility location problem, much better approximation factors $\in O(1)$ can be achieved. While it has been shown [GK99] that a polynomial-time algorithm cannot obtain an approximation ratio better than 1.463, a centralized algorithm [MYZ02] already provides a solution that is at most a factor of 1.52 away from the optimum. For the metric case, a distributed algorithm has been mentioned in [CEPS05] which solves a constrained version of the problem in which facilities and clients may be at most 3 hops away. It provides a $3+\varepsilon$ approximation factor derived from a parallelized execution of a respective centralized algorithm [JV99] and is formulated in terms of

[1] This holds unless every problem in NP can be solved in $O(n^{O(\log\log n)})$ time.

a synchronous message passing model. The same paper [CEPS05] includes additional versions, which restrict the facility location problem in one way or another. Only recently, a constrained version of the facility location problem (in which all opening costs are equal) has been addressed in a distributed manner [GLS06]. This algorithm also requires a synchronous message passing model and global communication among all nodes in each round. Finally, a distributed algorithm based on hill-climbing [KSW05] addresses a version of the problem in which exactly k facilities are opened. In this approach, the worst-case time complexity and the obtained approximation factor are not discussed explicitly.

In this chapter, we develop a distributed version of a centralized algorithm [JMM+03] which provides an 1.61 approximation factor with metric instances. Compared to related work, our work improves on the approximation factor achievable in a distributed manner. Moreover, we provide adaptations for executing this algorithm in multi-hop networks for which, to our knowledge, no local algorithm with guaranteed worst-case approximation factor exists. Finally, compared to [MW05, CEPS05, GLS06], our algorithms do not require a synchronous message passing model. Instead, they perform synchronization among network neighbors implicitly as nodes wait for incoming messages.

The remainder of this chapter is structured as follows. In Section 3.3, we briefly summarize the centralized algorithms [JMM+03] which provide the foundation for our work. We then describe their distributed reformulation in two steps. The first version, given in Section 3.4, still requires global communication, namely, that all clients communicate with all relevant facilities in each step, and is therefore only applicable to the *one-hop* setting, where this can be efficiently implemented as a wireless broadcast. In the second step, we use this algorithm as a subroutine in the algorithms of Section 3.5, which distribute messages only to a local neighborhood around the sending node and may therefore be used in multi-hop networks. Finally, we provide experimental results in Section 3.6 and a summary of the chapter in Section 3.7.

3.3 Centralized Algorithms

Jain et al. [JMM+03] devised two centralized approximation algorithms for the facility location problem. Both use the notion of a *star* (i, B) consisting of a facility i and an arbitrary choice of clients $B \subseteq C$ (in clustering terminology, a star corresponds to a cluster leader and a set of associated

slave nodes). The first algorithm from [JMM+03] is given in Algorithm 1. In its core step (line 1.3), the algorithm selects the star (i,B) with best (lowest) cost efficiency. The cost efficiency of a star is defined as

$$c(i,B) = \left(f_i + \sum c_{ij}\right) / |B| \tag{3.1}$$

and represents the average cost per client which this star adds to the total cost.

Algorithm 1: Centralized 1.861-approximation algorithm [JMM+03]

1.1 set $U = C$
1.2 **while** $U \neq \emptyset$ **do**
1.3 find most cost-efficient star (i,B) with $B \subseteq U$
1.4 open facility i (if not already open)
1.5 set $\sigma(j) = i$ for all $j \in B$
1.6 set $U = U \setminus B$
1.7 set $f_i = 0$

Therefore, in each step, the algorithm selects the most cost-efficient star (i,B), opens the respective facility i, connects all clients $j \in B$ to i (sets $\sigma(\mathrm{j}) = i$), and from this point on disregards all (now connected) clients in B. The algorithm terminates once all clients are connected.

Note that in spite of there being exponentially many sets $B \subseteq U$, the most efficient star can be found in polynomial time: For each facility i, clients j can be sorted by ascending connection cost to i. Any most cost-efficient star spanning some $k = |B|$ clients will consist of the first k clients with lowest connection costs – all other subsets of k clients can be disregarded as these cannot be more efficient. Hence, at most $|C|$ different sets must be considered.

When a facility i is opened, its opening cost f_i is set to zero. This allows facility i to be chosen again to connect additional clients in later iterations, based on a cost-efficiency that disregards i's opening costs f_i – as the facility i has already been opened before in order to serve other clients. For metric instances, Algorithm 1 provides a 1.861 approximation factor. Note that line 1.7 constitutes the only difference to a classic algorithm [Hoc82], whose approximation factor for metric instances is much worse. An even better approximation factor of 1.61 can be obtained when changing the above algorithm to additionally take into account the benefit of opening a facility i for clients that are already connected to some other facility. This involves two changes.

First, this requires that a revised cost-efficiency definition is used in line 1.3. We let $B(i)$ denote the set of clients j which are already connected to some facility $\sigma(j)$ and would benefit if i would be opened as their connection cost to i would be lower than their current connection cost $c_{\sigma(j)j}$, i.e.,

$$B(i) = \left\{ j \in C \text{ with } \sigma(j) \neq \textit{none} \text{ and } c_{ij} < c_{\sigma(j)j} \right\}. \tag{3.2}$$

The cost efficiency of a star (i,B) can now be restated as

$$c(i,B) = \left(f_i + \sum_{j \in B} c_{ij} - \sum_{j \in B(i)} \left(c_{\sigma(j)j} - c_{ij} \right) \right) / |B|. \tag{3.3}$$

A second analogous change is made to line 1.5. In addition to the clients which are part of the most-efficient star (i,B), all already-connected clients $B(i)$ which benefit from switching are connected to i. For this, line 1.5 becomes

$$\text{set } \sigma(j) = i \text{ for all } j \in B \cup B(i).$$

The authors prove in [JMM+03] that this change improves the approximation factor to 1.61 for metric instances. In the following, we will present a distributed version of this 1.61-algorithm. In the discussed distributed adaptations, we will always use the revised cost-efficiency definition given in equation (3.3).

3.4 One-hop Approximation

Consider the distributed algorithms given in Algorithm 2 (for facilities) and 3 (for clients). We will show below that they perform the exact same steps as the centralized Algorithm 1. While these algorithms require that each client communicates with each facility and vice versa, the algorithms can be also applied "locally" such that each node communicates only with its network neighbors. This way, they can be used to compute a solution to the *one-hop* version of the facility location problem, for example, to compute an energy-efficient clustering that takes the costs of individual links into account. Unfortunately, this constrained problem version results in a non-metric instance (see Section 3.1) and thus the approximation guarantee of 1.61 cannot be preserved. However, in the next section, we will use these algorithms as a subroutine to obtain an algorithm that maintains the approximation factor of 1.61 for multi-hop sensor networks. Moreover,

we will show that Algorithms 2 and 3 compute good solutions even when they are executed on non-metric instances in our experimental results of Section 3.6.

We assume that after an initial neighbor discovery phase, each client j knows the set of neighboring facilities, which it stores in the local variable F_j, and also the connection costs c_{ij} to facilities $i \in F_j$. Vice versa, each facility i knows the set of neighboring clients C_i and c_{ij} of all $i \in C_i$. In the following we will simply write C and F, as the respective indices i and j can be deduced from the context.

Algorithm 2: Distributed formulation of Algorithm 1 for Facility i

```
2.1  set U = C
2.2  repeat
2.3      find most cost-efficient star (i,B) with B ⊆ U
2.4      send c(i,B) to all j ∈ U
2.5      receive "connect-requests" from set B* ⊆ U
2.6      if B* = B then
2.7          open facility i (if not already open)
2.8          send "open" to all j ∈ F
2.9          set U = U \ B
2.10         set f_i = 0
2.11     receive σ(j) ≠ none from set C_a
2.12     set U = U \ C_a
2.13 until U = ∅
```

Algorithm 3: Distributed formulation of Algorithm 1 for a Client j

```
3.1  repeat
3.2      receive c(i,B) from all i ∈ F
3.3      i* = argmin_{i∈F} c(i,B) // use node ids to break ties among equal c(i,B)
3.4      send "connect-request" to i*
3.5      if received "open" from i* then
3.6          set σ(j) = i*
3.7          send σ(j) to all i ∈ F
3.8  until connected
3.9  on "open" from i with c_ij < c_{σ(j)j}
3.10     set σ(j) = i
3.11     send σ(j) to all i ∈ F
```

As in Algorithm 1, this time each facility i maintains a set U of unconnected clients which is initially equal to C (line 2.1). Facilities start a round by finding the most cost-efficient star (i,B) with respect to U and sending the respective cost efficiency $c(i,B)$ to all clients in B (lines 2.3-2.4). In turn, the clients can expect to receive cost-efficiency numbers

$c(i,B)$ from all facilities $i \in F$ (line 3.2). In order to connect the most cost-efficient star among the many existing ones, clients reply to the facility i^* that has sent the lowest $c(i^*,B)$ with a "connect request" (line 3.4). In turn, facilities collect a set of clients B^* which have sent these "connect requests" (line 2.5). Intuitively, a facility should only be opened if $B = B^*$, that is, if it has connect requests from all clients B in its most efficient star (line 2.6). This is necessary, as it could happen that some clients in B have decided to connect to a different facility than i as this facility spans a more cost efficient star. So, if all clients in B are ready to connect, facility i opens, notifies all clients in B about this, removes the connected clients B from U, and sets its opening costs to 0 (lines 2.7-2.10) as in the centralized algorithm.

If a client j receives such an "open" message from the same facility i^* which it had previously selected as the most cost efficient, it can connect to i^* (lines 3.5-3.6). Further, in line 3.7, client j notifies all facilities that it is now connected to i^*, which update their sets of unconnected clients U in lines 2.11-2.12.

Once connected, clients simply switch the facility they are connected to in case a closer facility becomes available (lines 3.9-3.10). This feature enables the 1.61 approximation factor. Note that whenever a client changes its facility $\sigma(j)$, it informs all facilities about this (lines 3.7 and 3.11). All these $\sigma(j)$ messages include the associated connection costs $c_{\sigma(j)j}$ and will be received in line 2.11 of the facility algorithm. By the next iteration, facilities will have received $\sigma(j)$ and $c_{\sigma(j)j}$ from all relevant clients, and will therefore be able to correctly compute the most cost-efficient star (line 2.3) according to Eq.(3.3).

Discussion. In the following, we argue that the distributed and the centralized versions are equivalent. For this, we denote one execution of the inner loops at Algorithms 3 and 4 as a round. Note that the distributed version opens some stars out-of-order, that is, earlier than the centralized version. The following lemma states that these stars are disjoint from any star that might follow and has lower cost-efficiency.

Lemma 3.4.1 *Let U^k be the set of uncovered clients prior to the beginning of round k. If a client j is part of a star (i,B) opened by the distributed algorithm in round k, then there is no star (i',B') considering $B' \subseteq U^k$ with $j \in B'$ and $c(i',B') < c(i,B)$.*

Proof Assume the contrary, namely that a star (i',B') exists with $c(i',B') < c(i,B)$ and say j is a client in $B' \cap B$. Note that $B' \subseteq U^k$, and

therefore i' will choose some star (i', B'') with cost-efficiency $c(i', B'') \leq c(i', B')$ in line 2.3. However, as (i, B) is opened in round k, client j has sent its connect request to i and not to i', which implies $c(i', B') \geq c(i, B)$ and contradicts the assumption.

Given the above, we can show that the stars opened by the distributed algorithm can be re-ordered to correspond to the execution of the centralized algorithm.

Theorem 3.4.2 *The distributed and centralized versions are equivalent.*

Proof We sequentialize the distributed algorithm as follows: In the sequentialized version we open only one star (the globally most cost-efficient star) per round. Further, we postpone opening a star (i, B) which has been opened in parallel by the distributed algorithm to a later round prior to which all stars (i', B') with $c(i', B') < c(i, B)$ have been processed. Let (i', B') denote one such star. Because of Lemma 3.4.1, $B' \cap B = \emptyset$, and therefore opening (i', B') ahead of time does not remove any client in B from U and therefore does not interfere with opening (i, B). Similarly, postponing any (i, B) will not allow that a more cost-efficient star including elements of B is formed earlier – again by Lemma 3.4.1. Postponing (i, B) can further influence (raise) the cost-efficiency of the stars (i', B') as it changes the set $B(i)$ for these facilities and thus may change the order in which these are processed. However, as by Lemma 3.4.1 all these stars are mutually disjoint, the order in which they are opened does not affect total costs. Finally, all stars opened in parallel are disjoint and re-ordering them does not change algorithm execution.

Therefore, the sequentialized version opens the same stars as the distributed algorithm. Moreover, as the sequentialized version opens the most cost-efficient star in every round, it implements the execution of the centralized algorithm.

Algorithms 2 and 3 therefore provide an approximation factor of 1.61. While the quality of the obtained solutions is intruiging, the worst-case number of rounds required by Algorithms 2 and 3 is linear in the number of nodes, because there can be a unique point of activity around the globally most cost-efficient facility i^* in each round: Consider for instance a chain of m facilities located on a line, where each pair of facilities is interconnected by at least one client, and assume that facilities in the chain have monotonously decreasing cost efficiencies. Each client situated between two facilities will send a "connect-request" to only one of them (the

more cost efficient), thus the second cannot open. In this example, only the facility at the end of the chain can be opened in one round. Similarly, once at least one facility is open, it could happen that in each round only one client connects to this facility. The worst-case runtime is therefore $O(n)$, in which n is the number of network nodes.

The linear number of rounds required by every node in the worst-case would constitute a high overhead in large networks. However, a worst-case configuration on a larger scale is highly improbable. We will evaluate the number of rounds required on typical instances derived from sensor networks in Section 3.6.

As we mentioned, the above algorithm only achieves an 1.61 approximation factor with *metric* instances. As the metric property implies that a graph must be complete, executing the above algorithms on metric instances implies global communication between all clients and facilities. Global communication can only be implemented efficiently in scenarios where all nodes are in wireless range of each other.

In our evaluation of Section 3.6, however, we will show that the above algorithms perform well even with non-metric *one-hop* instances. Moreover, below we will use the above algorithms as subroutines in an adapted "local" version that functions properly in multi-hop networks.

3.5 Multi-hop Approximation

The described algorithm can be changed to work in multi-hop settings using only a slight adaptation. As it turns out, if connection costs represent shortest paths between network nodes, the communication performed by the algorithms can be restricted to small network neighborhoods. Specifically, if one is interested in determining whether a facility i has a cost-efficiency of less than a certain threshold s, it is sufficient to consider only clients j that are reachable by i over a path with costs of at most s, i.e., clients j with $c_{ij} \leq s$. To see this, consider the definition of a facility's cost-efficiency and assume that some star's cost efficiency $c(i,B) \leq s$. One can always obtain an even smaller cost-efficiency once one removes the clients $j \in B'$ which have $c_{ij} > s$, that is, $c(i,B \setminus B') < c(i,B)$.

Similarly, for any facility i, the clients j with $c_{\sigma(j)j} > c_{ij}$ will not benefit from switching to i. Consequently, these clients will not occur in the set $B(i)$ of Eq. (3.3). Therefore, it is sufficient that clients j which become connected to $\sigma(j)$ distribute $\sigma(j)$ only to facilities i with cost $c_{ij} < c_{\sigma(j)j}$.

Algorithm 4: Multi-Hop Adaptation of Algorithm 3 for a Client j

```
4.1  set s = 1, set σ(j) = none
4.2  repeat
4.3      set s = s × a
4.4      send "start(s)" to all i ∈ F_s
4.5      if no "begin(s)" received then continue
4.6      repeat
4.7          receive c(i,B) from all facilities F_s
4.8          set F_a = {i ∈ F_s with c(i,B) ≤ s}
4.9          if F_a ≠ ∅ then
4.10             i* = argmin_{i∈F_a} c(i,B) // use node ids to break ties
4.11             send "connect-request" to i*
4.12             if received "open(s)" from i* then
4.13                 set σ(j) = i*
4.14                 send σ(j) to all i ∈ F_s
4.15     until connected or F_a = ∅
4.16 until connected
4.17 on "open(s*)" from i with c_ij < c_{σ(j)j}
4.18     set σ(j) = i
4.19     send σ(j) to all i ∈ F_{s*}
```

In an outer loop added around Algorithms 2 and 3, we therefore exponentially increase the communication scope s, that is, the maximum distance over which messages are forwarded. Specifically, given a certain scope s, a message is only flooded within a localized neighborhood $N_s(i)$ around the sending node i, where $N_s(i) := \{j \in V \text{ with } c_{ij} \leq s\}$. Note that if the direct link (i, j) is not present in the network graph, c_{ij} representing the shortest path from j to i can be determined on the fly while flooding a message within $N_s(j)$. Nodes simply stop forwarding a message if it has covered a distance larger than s or if it has already been received over a shorter path.

The updated versions are given in Algorithm 4 (clients) and Algorithm 5 (facilities). In the following, we will respectively use C_s and F_s to refer to client and facility nodes within scope s of the current node.

In the outer loop, the considered scope s is raised exponentially (lines 4.3 and 5.3) by a constant a. To initialize an outer round, clients, which have not yet been connected, send a "start" message containing their current scope s to all facilities in scope (line 4.4). In turn, facilities wait for at least one such "start" message for a certain time (line 5.4) upon which they reply "begin(s)". The waiting period must be long enough to allow relevant clients to send the respective start messages and finish earlier rounds. If no "start" messages were received, facilities simply advance to the next outer round (line 5.4) to wait for "start" messages from

Algorithm 5: Multi-Hop Adaptation of Algorithm 2 for Facility i

```
5.1  set s = 1
5.2  repeat
5.3      set s = s × a
5.4      if "start(s)" received then send "begin(s)" to all j ∈ C_s else continue
5.5      query σ(j) from all j ∈ C_s
5.6      set U_s = {j ∈ C_s with σ(j) = none}
5.7      repeat
5.8          find most cost-efficient star (i,B) with B ⊆ U_s
5.9          send c(i,B) to all j ∈ U_s
5.10         if c(i,B) ≤ s then
5.11             receive "connect-requests" from set B* ⊆ U_s.
5.12             if B* = B then
5.13                 open facility i (if not already open)
5.14                 send "open(s)" to all j ∈ C
5.15                 set U_s = U_s \ B, set f_i = 0
5.16             receive σ(j) ≠ none from some clients B' ⊆ U_s
5.17             set U_s = U_s \ B'
5.18     until U_s = ∅ or c(i,B) > s
5.19 until s > s_max
```

a larger scope. Clients, analogously, wait and then skip the current round if no neighboring facility has sent "begin".

A start message sent by a client j thus triggers execution of one outer round at all the facilities in scope F_s. Facilities then query all clients in scope for their status $\sigma(j)$ in line 5.5 and compute the set of yet unconnected clients U_s. This query-reply cycle allows the facility to wait for all relevant clients to catch up to the current scope s. Clients reply to this query once they have reached scope s – note that we have omitted the respective code in the client algorithm. Similarly clients can wait for facilities lagging behind in line 4.7 where they expect to receive a message from all facilities in scope.

After this initialization, facilities execute Algorithm 2 in an inner loop (lines 5.7-5.18) and clients react accordingly (lines 4.6-4.15) implementing Algorithm 3. Compared to Algorithms 2 and 3, the termination conditions of the inner loops must be changed to allow clients and facilities to proceed to a larger scope in a properly synchronized manner. As with the 1-hop version, clients terminate their inner loop once they are connected (line 4.15) and facilities once no active clients remain in scope (line 5.18). In addition, within an inner-loop with scope s, the algorithm should only consider stars (i,B) with cost-efficiency $c(i,B) < s$. Therefore, facilities only proceed with the current inner loop as long as they are efficient enough for this scope (lines 5.10 and 5.18) while in turn clients

only proceed with their inner loop as long as there is a facility in scope that is efficient enough to connect them (lines 4.8,4.9 and 4.15).

Finally, once a client has been connected (4.17-4.19), it acts analogously to Algorithm 3: It simply changes its facility if this is beneficial and notifies all relevant facilities about it. Here the client can synchronize to the scope s^* of the sending facility as it is included in the received "open" message to ensure that all relevant facilities are informed. Note that the messages sent in line 4.19 are also received by facilities still performing their inner loop in line 5.16.

Discussion. The algorithms presented in this section enhance Algorithms 2 and 3 by making them "local", meaning that clients do not need to communicate with all relevant facilities but only with the ones in a confined neighborhood. This allows to perform shortest-paths computations in these confined neighborhoods which, in turn, give rise to metric instances and preserve the approximation factor of Algorithm 1.

An additional outer loop provides for both, an adequate expansion of the involved communication scope and for sufficient synchronization of the nodes in scope without depending on a synchronous communication model. Note that the outer loop increases the number of rounds in the worst case only slightly: As in Algorithms 2 and 3, every single node may constitute a single point of activity only *once* during algorithm execution. The added runtime is based on the total number of steps of the outer loop which amounts to $\log_a s_{\max}$ where $s_{\max}$ is the efficiency of the least-efficient star chosen by some node in line 5.8 and a denotes the factor by which scopes are increased in each outer round. The worst-case number of communication rounds is therefore in $O(n + \log s_{\max})$.

In this term, we treat a local broadcast as a single operation and neglect the time for multi-hop message propagation. In the worst case, however, broadcast scopes may become as large as the network diameter. Moreover, there is a certain latency involved in the synchronization performed in lines 4.5 and 5.4 (parts of the network might have to wait for other parts to complete earlier outer rounds). Such multi-hop propagation and synchronization efforts are not considered in the runtime of related work [MW05, GLS06] designed for synchronous (and global) communication models. If taken into account, this would require multiplying the network's hop-diameter to the worst-case runtime.

Such high runtimes do not materizalize in practical *multi-hop* instances, as we will show in Section 3.6. In particular, the runtime does not in-

crease with the network size and the involved communication scopes (determined by $s_{\max}$) are fairly small compared to the network diameter[2].

Dynamic Re-configuration. In real-world deployments of sensor networks, link qualities change over time and nodes may fail. To accommodate for *major* changes in the network topology, the algorithms are re-executed at regular intervals. As such re-starts involve relatively high overhead, these are performed only infrequently (e.g., once a day). In between such re-starts, a client j combines periodic re-evaluations of link costs c_{ij} (within a local scope of size $c_{\sigma(j)j}$) with a liveness check on the facility $\sigma(j)$. In both cases, if $\sigma(j)$ has failed or a closer open facility has been found, client j re-connects to the closest open facility. In the next section, we will show that such adaptations suffice to maintain a close-to-optimal configuration over longer periods of time.

3.6 Evaluation

In the following we present results from two distinct sets of experiments. The first, detailed in Section 3.6.1, is based on simulations which test the scalability of the proposed algorithms. The second, detailed in Section 3.6.2, tests the applicability of the proposed algorithms to operational networks with dynamic links.

3.6.1 Scalability

In the experiments based on simulations, we uniformly deployed a variable number of nodes onto a 300 m by 300 m area. The network graph has an edge $(i, j) \in E$ if nodes i and j are less than 30 m apart. Assuming that nodes can control their transmit power, for $(i, j) \in E$ we set the connection costs c_{ij} to $g(i, j)^2$, in which $g(i, j)$ denotes the distance in meters between i and j, and normalize them such that $c_{ij} \in [0, 1]$.

Scenarios. To test our algorithms with a range of applications, we examined three different parameterizations of the problem, of which qualitative results were shown earlier in Figure 3.1. In the first, shown in Figure 3.1(a), we set opening costs $f_i = 1$ and additionally require that clients and facilities must be neighbors. We tested the one-hop Algorithms 2 and 3 on such instances.

[2] In fact, $s_{\max}$ depends on the application and is proportional to the hardness of the specified configuration problem: If the desired configuration is inherently non-local, for example, if high opening cost settings require the algorithms to choose only one facility that serves the whole network, $s_{\max}$ will be high.

Further, we tested the multi-hop Algorithms 4 and 5 in two different settings, which were also already introduced in Figures 3.1(b) and 3.1(c), respectively. In the first, we set $f_i = 5$ to denote that a high effort is required to operate a cluster leader. In the second scenario, we assumed that cluster leaders must send much data to the network base station and therefore their operation costs increase with their network distance to the sink (yielding smaller stars close to the sink and larger ones further away).

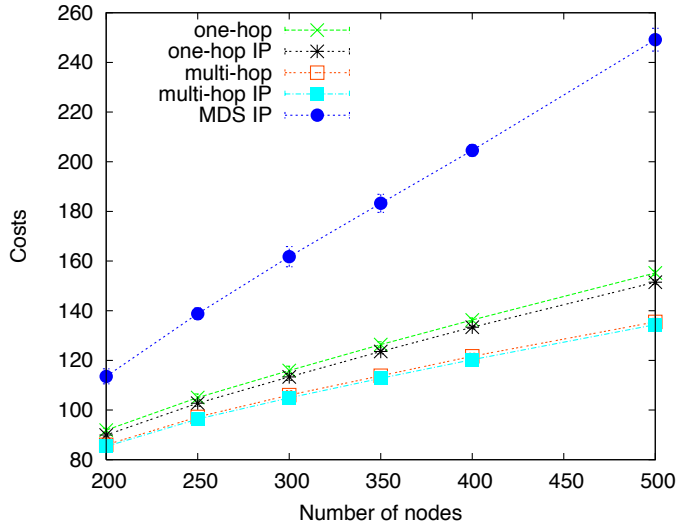
One-Hop Clusters. In the *one-hop* setting (Figure 3.1(a)), we evaluated the costs of configurations produced by different algorithms while varying the number of nodes in the simulation area (that is, the node density). The results are given in Figure 3.2(a) which shows the costs obtained with the following five methods.

One-hop denotes the simple one-hop algorithms of Section 3.4. Respectively, *one-hop IP* refers to the optimal configuration of the constrained case which requires clients to connect to facilities which are direct network neighbors. Further, *multi-hop* denotes the multi-hop algorithm described in Section 3.5, which has a 1.61 approximation guarantee. Here, clients may connect to facilities which are an arbitrary number of hops away. Respectively, *multi-hop IP* computes the optimal solution to the facility location problem, in which facilities and clients may be multiple hops apart and the instance is made metric by a centralized shortest-paths computation. Finally, *MDS IP* denotes the optimal solution to the minimum dominating set problem, in which dominator nodes represent open facilities and slave nodes are clients that connect to the closest dominator node. The costs are computed using the original (non-metric) instance.

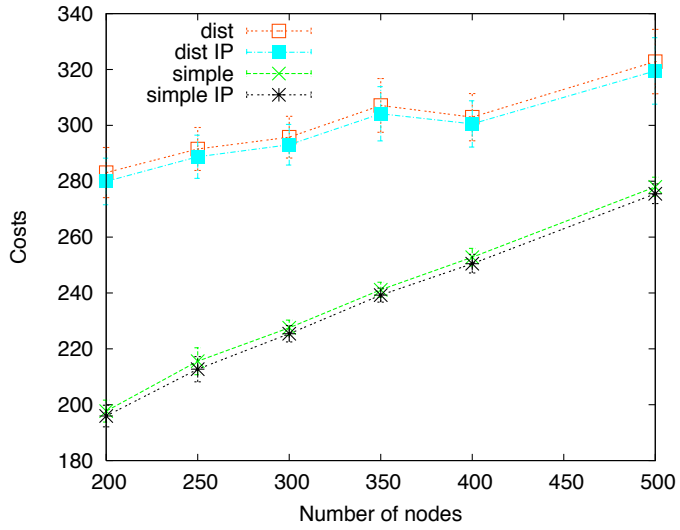
The costs of a minimum dominating set (*MDS IP*) which suffer from expensive long links mark one end of the optimization spectrum. Here we argued that facility location can provide a more energy efficient configuration as it takes costs of the links between clients and servers into account. On the other hand, the optimal facility-location based configuration (*multi-hop IP*) marks the other end of the spectrum as it represents a lower bound for the employed approximation algorithms.

The *one-hop* algorithm performs well and is even close to the respective optimal configuration *one-hop IP*, although it operates on non-metric instances and thus without a guaranteed approximation factor.

Note that in this particular setting, the constrained versions, which require facilities and clients to be direct neighbors (*one-hop* and the optimal *one-hop IP*), are not far away from the *multi-hop* results and the optimum of the unconstrained case (*multi-hop IP*). This is due to the low opening



(a) One-hop



(b) Multi-hop

Figure 3.2: Performance of one-hop and multi-hop algorithms

costs we used, which are set to $f_i = 1$ for all facilities. With larger opening costs, multi-hop solutions would benefit more from larger stars.

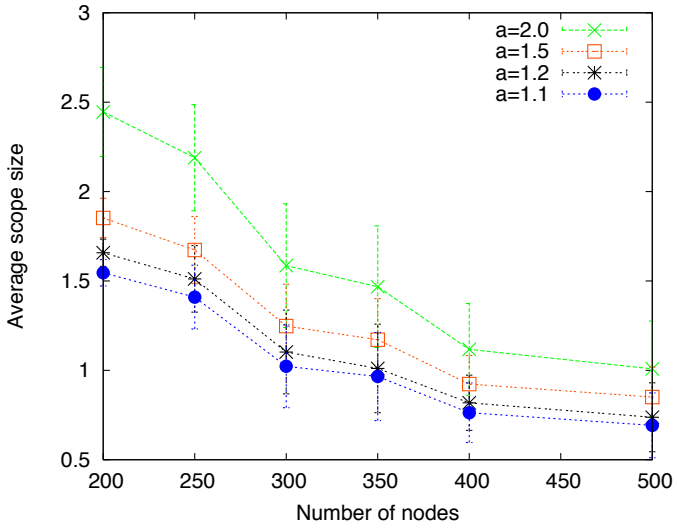
Multi-Hop Clusters. In the experiments shown in Figure 3.2(b), we additionally evaluate the quality of the solutions obtained by the multi-hop algorithm with the two different opening cost settings shown in Figures 3.1(b) and 3.1(c). In the first (denoted as *simple*) we set opening costs to a constant $f_i = 5$ which corresponds to configurations as shown in Figure 3.1(b). In the second, denoted as *dist*, we apply the heuristic shown in Figure 3.1(c), where the opening costs correspond to twice the costs of the shortest path to the sink. In both cases, the results of the distributed implementation are very close to the achievable optimum computed by CPLEX on the same instance.

Runtime and Overhead. In the experiments shown in Figure 3.2(b), the scope s started out with 0.2 and a was set to 2, thus doubling the scope in each outer round. Note, however, that these two parameters do not influence the quality of the obtained solution. Rather, they determine the trade-off achieved between the runtime of the algorithms and the scope within which messages are sent. On the one hand, the smaller a is set, the less likely it is that scopes are increased by too much (in vain). On the other hand, lower values of a increase the number of outer rounds[3].

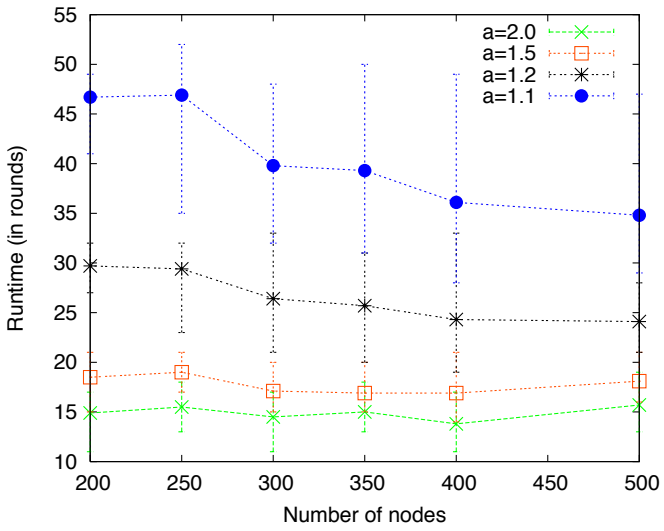
Figure 3.3 demonstrates this trade-off as observed in the simulation run corresponding to Figure 3.2(b). In Figure 3.3(a) we show the average scope with which messages were sent during algorithm execution, given different settings of a (the scope s always starts at 0.2). The lower we set a, the better the results as the scope is increased by smaller amounts. Note that in general, the effort involved in the execution of our algorithm is proportional to the "locality" implied by the problem instance: On the one hand, if opening costs are high (here $f_i = 5$), a facility will generally connect clients in a larger neighborhood (as seen in Figure 3.1(b)). On the other hand, the experienced scopes are even much lower with small opening costs (e.g., for $f_i = 1$, not shown).

In contrast, in Figure 3.3(b), we show the runtime in rounds (one round corresponds to one iteration of the inner loop) of the multi-hop algorithm on the same instances. Note that, while previously the error bars indicated confidence intervals of 95%, we use them in Figure 3.3(b) to mark the maximum and minimum values that occurred in 10 random instances (as

[3]Interstingly, our experiments show that the average number of communication rounds, whose worst case amounts to $O(n + \log_a s_{\max})$, is in fact dominated by the latter term $\log_a s_{\max}$. The case where single nodes slow down algorithm execution (giving rise to the first term n) is highly unlikely.



(a) Scope



(b) Rounds

Figure 3.3: Average scope size vs. total runtime (in rounds). In Figure 3.3(b) the error bars denote the maximum and the minimum that occurred.

we are particularly interested in the maximum value). The results show that – while in theory the worst case runtime can be large – in typical instances based on multi-hop networks the runtime is sufficiently small and does not even grow with the number of nodes. Moreover, based on the trade-off between runtime and scope size, the runtime improves with higher a values. Finally, the scope size decreases with increasing network density. This is due to the fact that, given certain opening costs, the algorithms will connect stars of around the same size (namely, facilities are opened once enough clients are connected to pay for opening them). Therefore, smaller stars are opened in denser networks and the cumulated communication overhead stays the same.

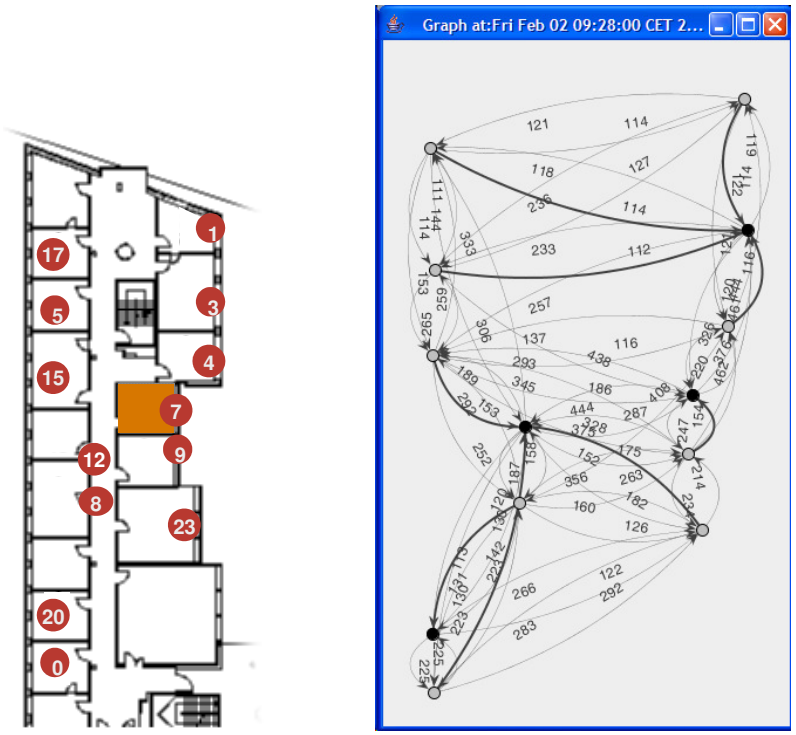
3.6.2 Network Dynamics

One open question is whether such, albeit close-to-optimal solutions, can provide a benefit for real-world deployments in which the network topology changes over time. To obtain realistic link qualities, we extended a testbed of 13 TMote Sky modules that gather temperature, humidity, and light measurements from our office premises to record network topology information as well. In addition to its sensor measurements, every 5 seconds, a node reports the set of nodes from which an application-layer message has been received since the last update.

Such topology information received from each node i allows to compute a (packet-level) link quality estimate $e_{ij}(t)$ for each network link directed from j to i [WTC03]. The estimate $e_{ij}(t)$ is based on the packet success rate $r_{ij} = \frac{\text{packets received in T}}{\text{packets expected in T}}$ which is smoothened using an exponentially weighted moving average such that $e_{ij}(t) = \alpha r_{ij}(t) + (1-\alpha)e_{ij}(t-1)$. In our experiments, we set α=0.6 according to [WTC03] and T to 300 s. We transform the quality estimates $e_{ij} \in [0,1]$ into link cost estimates by setting $c_{ij} = 1 + 10(1-e_{ij})$ if $e_{ij} > 0.5$ and $c_{ij} = \infty$, otherwise. Further, we set opening costs to constant $f_i = 2$.

To give the reader an impression of the examined networks, Figure 3.4 shows our sensor node deployment, the resulting network topology, and a configuration computed by the multi-hop algorithms.

Given the link costs $\{c_{ij}(t_0)\}$ observed at a certain time of the experiment t_0, we let the presented multi-hop algorithms compute a configuration (a set of open facilities and assigned clients), whose costs $C(t_0,t)$ vary with t as link qualities change over time. Once a configuration has



(a) Deployment plan

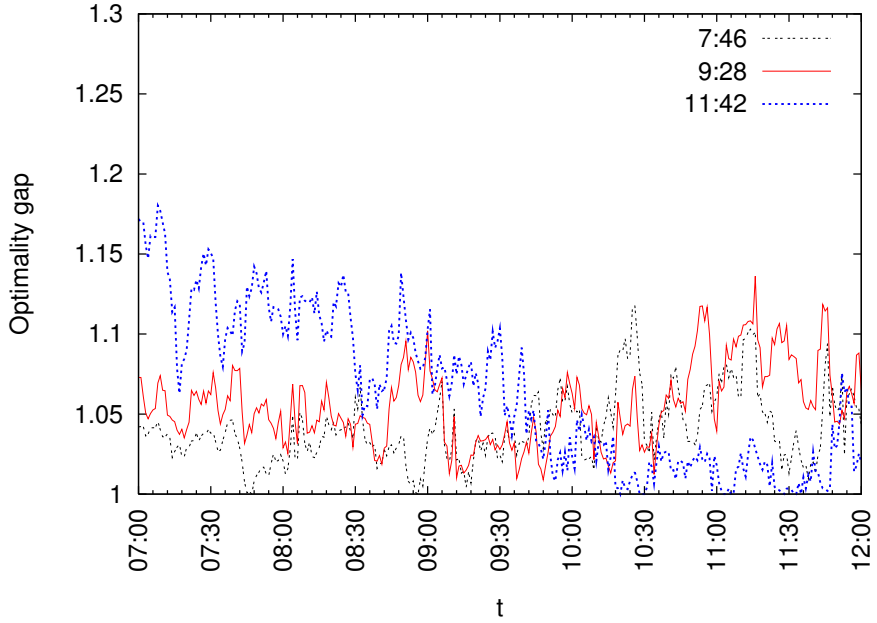
(b) Network topology at 9:28 showing $c_{ij} \times 100$ and computed configuration

Figure 3.4: Deployment plan, network topology, and computed configuration

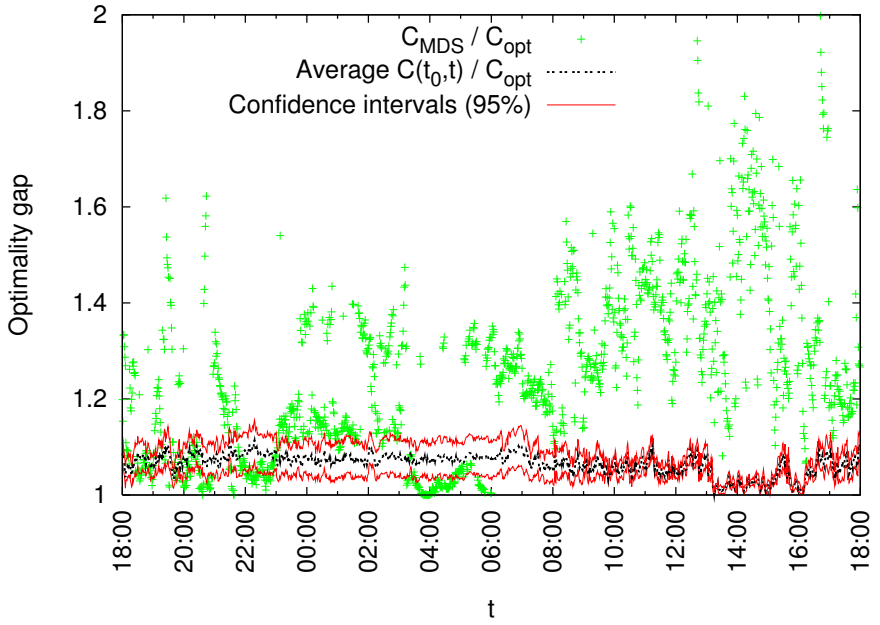
been computed, only small dynamic adaptations (detailed in Section 3.5) are performed.

In Figure 3.5(a), we show the ratio between $C(t_0,t)$ and the costs of an optimal configuration C_{opt} computed by CPLEX – for configurations computed at three arbitrarily chosen instants of time t_0. Observe how at $t = t_0$, e.g. at 7:46 or at 11:42, the respective optimality gap is close to 1. As expected, however, this is not always the case. For example the configuration obtained at t_0=9:28 is not optimal even at this time.

In Figure 3.5(a), one can observe how the time t_0 at which the initial configuration is computed influences the respective outcome of $C(t_0,t)$. To obtain more general results, t_0 is randomly drawn from the total 24 hour interval corresponding to available topology data and used to compute the respective curve $C(t_0,t)$ in 20 repeated simulation runs. The ratio of the *average* $C(t_0,t)$ to the costs of the optimal configuration is shown Figure 3.5(b). In addition, Figure 3.5(b) shows the costs C_{MDS} of a minimum dominating set computed by CPLEX for each instant of experiment time. The latter costs can be used as an assessment of whether a much faster MDS approximation, which can be re-executed frequently, could out-perform a facility location algorithm executed more rarely. As said earlier, however, MDS-based configurations require slaves to use expen-



(a) $C(t_0,t)/C_{\text{opt}}$ for different t_0 during 5 hours



(b) Average $C(t_0,t)/C_{\text{opt}}$ vs. MDS during 24 hours

Figure 3.5: Solutions' optimality over time

sive links (with poor link quality estimates) to communicate with their cluster leader. Such "bad" links are often the most volatile and cause the costs of an MDS-based configuration to diverge significantly from an optimal configuration. While this is not always the case (Figure 3.5(b) has portions in which MDS is close-to-optimal), one can observe that facility-location based configurations, which focus on high-quality links, are robust with respect to varying link qualities. The observed gap to an optimal configuration remains small – in the observed 24 hours it stayed below 10% at all times.

3.7 Summary

In this chapter, we motivated the use of facility location algorithms to address configuration tasks in multi-hop networks as they can flexibly implement many sensor-network configuration problems, such as an energy-efficient clustering, a clustering in which cluster leaders can connect nodes through multiple hops, or a configuration in which cluster leaders are chosen based on their distance to the sink. We claim that many more such applications of the problem can be found.

Further, we have shown that algorithms which are very good in theory (with an approximation factor of 1.61 while even a centralized polynomial algorithm cannot be better than 1.463) can be feasibly transformed for distributed execution. The transformations we described resulted in (to our knowledge) the first facility location algorithm which can be efficiently executed in multi-hop networks.

In the experimental evaluation, we were able to show that although our algorithm exhibits a high worst-case runtime, in typical sensor-network instances it terminates in only few communication rounds. Moreover, by analyzing the scopes within which messages were forwarded during algorithm execution, we showed that the devised algorithm, although equivalent to its centralized ancestor, requires only very local communication. Further, we showed that the distributed algorithm always performs close to the optimal solution, a quality which it inherits from the centralized version [JMM$^+$03].

4 Query Scoping

In the previous two chapters, we provided role-based configuration approaches for sensor networks using multi-hop communication. However, such support for role-based configuration is required in other network models as well. In this chapter, we consider a configuration algorithm specifically designed for a large scale sensing system in which every sensor node is reachable by the network base station in one network hop. Specifically, we focus on a sensing infrastructure based on mobile phones and on a specific application which allows users to find lost or misplaced items using their phone.

As in Chapter 3, we are considering a subclass of the generic role assignment problems introduced in Chapter 2. However, the discussed application, as well as the involved network model and the addressed configuration problems, differ significantly from the previous Chapters 2 and 3. We will therefore start this chapter by taking a detour describing the application scenarios supported by our approach and the architecture of the underlying wide-area sensing system.

Specifically, this system allows users to keep track of real-world objects, e.g., their personal belongings, using a mobile phone and to use the phone to locate lost or misplaced items. The phones carried by users constitute the sensing infrastructure of this system, each phone being integrated with a short-range sensor for detecting objects in its proximity. The application, detailed in Section 4.1, includes different usage scenarios from storing the context in which an object has left the sensing range of the local device to allowing object owners to search for objects using the sensing infrastructure provided by the cellular network and the mobile phones of other system participants.

The considered application gives rise to a configuration approach that is organized differently than in the two previous chapters – exploring the design space of wireless sensor network configuration in yet another direction. The developed algorithm addresses a configuration problem that is specific to the underlying network model: In a large scale system in which, potentially, every mobile phone connected to a cellular network

can be used as a sensor, one can neither distribute a search query to all available sensors nor aggregate all sensor readings (e.g., *object X seen by sensor A*) in a database for later query.

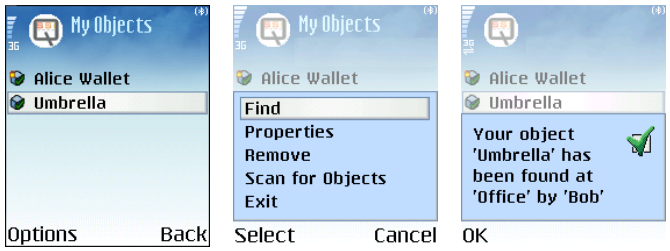
Instead, everytime a user issues a query to find an object, the configuration algorithm must assign the roles *sensing* and *idle* to the nodes of the network, such that the *sensing* nodes are likely to obtain the desired information, while *idle* nodes can keep their sensors off and, effectively, may even refrain from any communication with the network basestation. the main contribution of this chapter is therefore a flexible query scoping algorithm that selects nodes based on their likelihood to locate an object of interest to the user.

As the algorithm is based on a wide range of real-world data stored by various software components of the presented object search system, its evaluation is not straightforward. Therefore, the evaluation section focuses on verifying the practicability of the mobile-phone-based system itself, that is, the coverage it provides and the performance it exhibits when trying to find an object. To demonstrate the query scoping algorithm itself, we discuss how it can be applied to a set of application scenarios and present qualitative results for the involved configuration problems.

The chapter is structured as follows. We first describe the object localization application in detail in Section 4.1. After reviewing related work in Section 4.2, we overview the system's architecture in Section 4.3 to motivate why various relevant history data is available at different components distributed in the system. This data will then be used by the query scoping system, which we describe in Section 4.4. With a focus on the practicability of the object localization application, we discuss the privacy implications of the developed system in Section 4.5 and evaluate the sensing performance obtained from sensors carried by users in a comprehensive study in Section 4.6. We close the chapter with a summary in Section 4.7.

4.1 The Object Localization Application

Our application requires that *object sensors*, which are able to detect the nearby presence of an electronically tagged item, can be integrated with mobile phones. Various technologies could be employed for this purpose. For example, passive RFID tags are expected be attached to many consumer products in the near future as they may realize significant cost savings in stock and supply chain management. In particular, passive



(a) (b) (c)

Figure 4.1: User issues an object search query

UHF RFID technology or active tags with a small autonomous power source [RF 06] can provide reading ranges of a couple of meters even with small reader modules. If improved variants of today's handheld RFID readers were integrated into mobile phones, a ubiquitous system could be deployed within a few years using the short innovation cycle established through mobile phone sales. In addition to RFID, other upcoming radio communication technologies, some even compatible with the phone's Bluetooth capability, could be used to identify objects in the phone's proximity in a similar way. If small, inexpensive Bluetooth-discoverable tags [Wib06] can be built, a ubiquitous object sensing infrastructure is already in place today.

Each tagging technology defines a certain trade-off between tag costs, achievable identification range, and costs of reader hardware. Irrespective of the employed technology, we assume for our scenario that suitable object sensors can be integrated into mobile phones, as it is already the case today with Bluetooth or NFC. While we evaluate the benefit of an increased sensing range later in this chapter, the final choice of the technology is based on costs versus range trade-offs which remain to be explored in a concrete product's business plan. Our system prototype currently requires tagging objects with battery-powered tags (BTnodes [BTn06]) and uses the phones' built-in Bluetooth discovery for object sensing.

Our object localization application involves two prominent use cases. In the *remember* use case, users can task their mobile device to store the context in which an object leaves its local sensing range. This includes a trace of the user's location before and after the loss event and other people present or other personal objects carried along at the time the object was left behind. As there will be numerous managed objects which users leave behind on a regular basis (for example when leaving their home), users will find it unpleasant to receive notifications each time objects go out of range. Instead, the relevant data is silently stored and can be used at a

later time to help the user recall the circumstances of the loss or to hint at the location of a lost object.

In the *find* use case, the user can query the system for an item from the list of objects which have previously been associated to the user (Figure 4.1). The system will then forward the query to a set of object sensors which, based on user preferences and system settings, are presumed good candidates to find the object. Object search strategies can be based on various heuristics, such as querying sensors near the location where the object was last in range of the user's device (determined by the *remember* use case), or querying object sensing devices which have been previously associated with the user (for example, users may install stationary object sensing devices at certain prominent locations such as their workplace or their holiday home). Once a remote sensor has located the object, the user will receive a notification containing the object's location as shown in Fig. 4.1(c).

Taken together, these use cases benefit from the coverage provided by the mobile device both in time and in space: The *remember* use case benefits from the mobile phone's ability to cover the user's daily life over longer periods of time to record a "hint" on where a lost object might be located. Complementing this functionality, the *find* use case uses this hint as a starting point for performing a wide area search that aggregates sensor data from a larger number of system participants during short periods of time.

Further, our system supports additional use cases related to object monitoring and sensing. We omit them here as they are outside the focus of this chapter. For details on the complete application, we refer to the original publications on the object localization prototype and the underlying system architecture [FRB+07, FBRK07, FBMK07].

4.2 Related Work

Various research papers argue for the relevance of locating everyday objects, monitoring the presence of items, or avoiding their loss. Many such systems [WFGH99, DKB03, YSM05] suggest a pre-installed object sensing infrastructure, which is costly to deploy and to maintain. Instead, we use mobile phones implementing a people-centric sensing infrastructure. In the Smart Watch prototype [BBH+04], RFID readers transmit their current readings to the passing user's personal device. The user is then notified if objects are missing compared to readings which were col-

lected earlier. The *delegate* use case was first mentioned in another prototype [SHK+05]. Both systems focus on reminding users before a loss takes place. As mentioned, we assume that tagged objects will often be intentionally left behind and therefore avoid immediate notification. Instead, we study how the location of an item can be determined on a user's request.

Note that an object search system could also be implemented by proactively sending all sensor readings to a centralized service which would then be queried when an object needs to be located. Such a system would face the scalability challenge of a global data collection system, such as IrisNet [GKK+03] or Hourglass [PLS+06]. In contrast, we use a reactive, that is, query-based approach, as the number of sensor readings (e.g., *object X seen by object sensor A*) is expected to be much larger than the number of queries. Moreover, our system avoids aggregation of all readings in a centralized database as this would have severe implications for the user's personal privacy. In particular, it is incompatible with a privacy enhancing feature of our system: Certain objects, which have previously been associated to their owner, can only be detected by a sensor after this sensor has received an explicit query for the given object (see Section 4.5 for details).

Some related work has similarly advocated the use of the existing infrastructure of the mobile network: The authors of [TP05] focus on a middleware architecture motivated by a health-care application. Similarly, recent work [BEH+06, EAL+06] has motivated people-centric sensing with a focus on applications and their architectural requirements. In this context, we also want to mention Nokia's recent SensorPlanet [Sen07] initiative, which focuses on building a mobile-phone-based platform for research on large-scale wireless sensor networks. Compared to these approaches, we address two issues which we consider central to large scale sensing applications: *Query scoping* (determining which sensors should be queried from a large and homogeneous sensor array) and the properties of the wide-area *sensor coverage* obtained by user-carried sensors.

The query scoping algorithm, which we will describe further down, is based on a classic approach known as *uniform cost search* [RN95]. Its innovation does not consist in the search technique itself, but in the means we provide for its parameterization. In particular, our query scoping approach satisfies two requirements: On the one hand, it allows to use *any* data stored by application services to parameterize search and, on

the other hand, it efficiently accesses data sources which are distributed throughout the system.

As query scoping is based on history data stored by the application, the system is only indirectly related to systems concerned with *scoping* in multi-hop sensor networks [SFCB04, MP06, WSBC04]. These systems focus on letting individual nodes decide whether they are part of a certain scope. These decisions are made by sensor nodes themselves based on certain criteria which each node can evaluate locally.

In contrast, our query scoping system performs an *a priori* selection of nodes. This way, the nodes which are not in scope remain completely unscathed from a user query and from the scoping algorithm, that is, these are not required to perform any evaluation (e.g., on their scope membership) nor any communication with the server. This is required because the scope changes with every query, and because the scale of the system makes frequent communication with all nodes infeasible.

Therefore, the query scoping algorithm is executed in a centralized manner on a server in the back-end infrastructure. Considering the underlying *one-hop* network model, executing a centralized algorithm involves less overhead than in multi-hop models. For example, in our model, there is no non-trivial topology information which must be gathered at the sink. Instead, as we will argue in the remainder of the chapter, there is enough application-specific information available on the server side of the infrastructure, such that query scoping can be performed *offline*, that is, before communicating with the network.

Approaches addressing the *sensor selection* problem, such as [BKG06, IB05], have a goal that is similar to ours, namely, to select the set of sensors that best fulfills the application's sensing requirements. These systems assume that the application's requirements are formulated in terms of a utility function, which assigns a certain utility value to the concurrent measurements of any given subset of sensors. The sensor selection problem involves finding a subset of nodes that maximizes utility while obeying certain constraints on the total sensing costs (or, vice versa, minimizing sensing costs while maintaining a certain utility).

In comparison, we provide a complementary approach focusing on practical means for *initializing* the utility function itself – based on distributed stores of history data available in the system. While, once the utility function is initialized, we do also apply a set of practical sensor selection heuristics, these cannot integrate aspects from [BKG06, IB05] in a straightforward manner. This is due to the different sensing model of

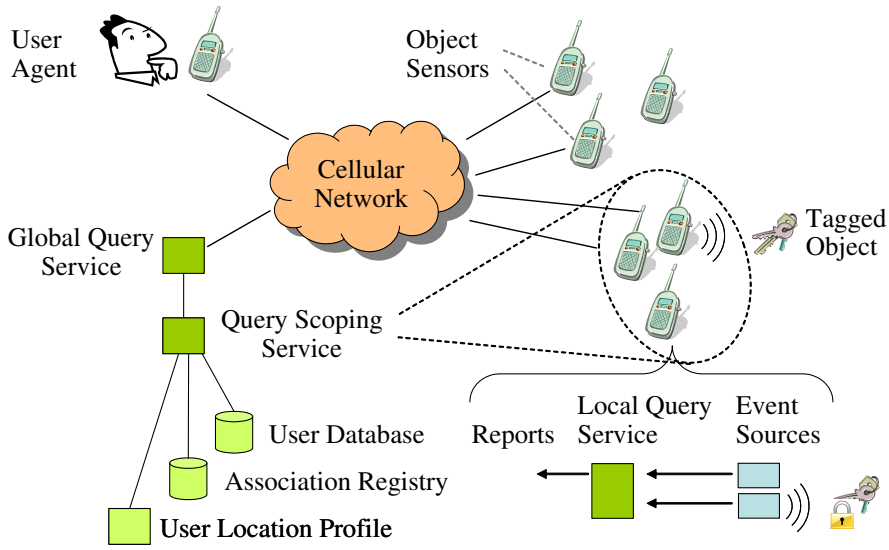


Figure 4.2: System architecture

our application, which involves intrinsically mobile sensor nodes. Nevertheless, we expect that our sensor selection heuristics can be optimized further. We provide a few improvement guidelines based on our evaluation in Section 4.6.

4.3 System Architecture

Figure 4.2 shows an overview of the system architecture. As mentioned above, mobile phones are used to link sensing functionality to users and the back-end infrastructure. Sensing functionality on mobile phones includes sensing the presence of tagged items, the phone's location, and other information relevant for remembering the context of an object's loss, as detailed in Section 4.3.1. Furthermore, our architecture involves *application specific* services, such as association of objects and their owners, a user database, and profiling services that can be used to deduct heuristics for computing the scope of an object search query. These services will be discussed in Section 4.3.2. All of the above are integrated by *query services* which support the implementation of the application's use cases. These include the *local query service*, used to set up individual mobile phones, the *global query service*, used to route queries on the global scale, and the *query scoping service*, which supports the latter in determining suitable receivers of a query (that are likely to produce a result) based on all kinds of history data stored by the application. These three-fold query services will be detailed in Section 4.4.

4.3.1 Sensing Functionality

Apart from object sensors, which we assume to be integrated into each mobile phone, we suppose that mobile devices have access to a variety of additional sensor information, for example, to a sensor determining the location of the user carrying the device.

Event Sources. All sensor information is encapsulated in application-specific software components which we name *event sources*. An event source generates events of a pre-determined type, either in a periodic manner or when a certain condition is met. Our system uses event sources that generate an event whenever a tagged object comes *in range* or goes *out of range* of the device or when the device's *location* changes. It also uses an event source for *persons* who are within short wireless range of the user device.

In our prototype implementation, the *location* event source generates an event every time the mobile phone's cell identifier changes. Object tagging is implemented using BTnodes [BTn06], tiny devices equipped with a Bluetooth radio, and object sensing is realized using Bluetooth discovery. The *in range* and *out of range* event sources, respectively, generate an event every time a new object is discovered by Bluetooth or a previously seen object has been missing in several rounds of discovery. Similarly, the *person* event source generates periodic events which contain a list of nearby users based on the Bluetooth names of their phones.

Sensor Authentication. Both, object tags and devices representing persons, can restrict their visibility and allow only authenticated sensors to detect their presence. Such authentication is based on a shared secret which is exchanged between the actor initiating sensing and the sensed entity at an earlier time (the association service described below is used for this purpose). After the initial key exchange, protected entities can be sensed only by sending an authenticated message. We discuss details of this concept which can be implemented on tags of a small form factor [EHJ04] in Section 4.5.

Our implementation of sensor authentication is prototypical. Protected persons and objects are emulated by turning off visibility to Bluetooth discovery, while the Bluetooth MAC address takes on the role of a shared secret: Sensing polls (for persons or objects) consist of a connection attempt at the correct address.

4.3.2 Application Specific Services

In our architecture, three system services have been designed to take on specific functionality mentioned in the object localization application.

Association. The association service serves three main purposes. First, it keeps track of associations between users and objects (Figure 4.1(a) shows the list of Alice's associated objects). While objects can be detected by any object sensor in their initial state, after association, they are detectable only within queries initiated by the object owner's device. Second, user to object sensor association allows users to maintain a set of object sensors that are particularly relevant for them (e.g., object sensors which have been installed by the user at home or at work; Bob's mobile device has previously been associated with Alice in the scenario depicted in Figure 4.1). Third, user to user association enables group access rights to certain objects, but is also used as a basis for disclosing the identity of associated users to each other (in Figure 4.1(c), Alice is shown the identity of Bob based on a previous association between them, otherwise the notification would only contain the location of the object). Similarly, users may choose to be visible only to *persons* event sources set up by associated users.

Location Profile. This optional service performs statistics on the locations (such as the physical location or the observed network cells) in which users spend most of their time. This makes it possible to implement a search strategy (i.e., a query scoping strategy) which gives preference to sensors at these locations, assuming that they represent places that contain personal belongings, such as a user's work place or home.

Our prototype includes a rudimentary adaptation of a network-cell-based profiling system [LRT04] augmented with functionality for naming certain network cells which are particularly relevant (such as 'Office' in Figure 4.1(c)).

User Database. We assume that the mobile network operator offers users a database service in which application data such as previous reports of certain objects can be stored. The service can be used, for example, to record a log of objects reported by an object sensor for later query. Similarly, it can be employed to store the details on the circumstances of an out-of-range event as required by the *remember* use case.

4.4 Query Services

The *query services* form the integrating central point of the system, wiring the distributed components for the required application task. The back-end infrastructure hosts the *global query service*, which provides support for query dissemination, cost control, and validity management for user queries. Moreover, each mobile device executes a *local query service* dealing with handling data requests on the level of the local mobile phone.

The *query scoping service* supports the global query service by determining which sensors should be involved in a wide area query, integrating information known at various application-specific services. For example, a query can be disseminated to a subset of associated users (based on information stored in the association registry), to locations that are particularly relevant for the user (based on location profile data), or near locations where the object was observed in the past (based on the user's database).

These query services are supported by a set of middleware services which are not detailed here. For instance, *storage services* are available both on the mobile device and in the back-end infrastructure – and are also used to implement the user database and the association service. Their implementations including transparent data marshaling and database interaction functionality for resource-constrained devices are described in [BL06]. Further, means for local event forwarding, wide area point to point messaging, and OSGi-based component execution platforms for both, the mobile device and the back-end infrastructure, have been implemented in our prototype. For details on these services, we refer to [FRB+07].

4.4.1 Query Service Interface

The query interface specifies the receivers of the query (its *scope*), the receivers of produced reports (*sinks*), and, further, various limits on the involved query effort. The *scope* can either consist of a single mobile device (in the *remember* scenario the targeted device is simply the user's phone) or a custom-implemented *scope provider* component (as in the find use case). Scope providers, required to return a list of object sensors sorted by relevance, may be used to pass on a variety of search strategies to the global query service: For example, an association-based scope provider may simply return a list of previously associated object sensors, a location-profile-based scope provider would return sensors at locations

where the user spends much time; the query scoping algorithm which we present in Section 4.4.2 also represents a scope provider component that can combine various search strategies. The reports generated by the query, in turn, will be delivered to the specified *sinks*. Various system services may be wired as sinks: In the *remember* use case, reports will be delivered to the local database on the user's phone, in the *find* use case, reports cause a direct notification on the user interface.

Moreover, different effort limits can be issued with the query: $c_{\max}$ limits the monetary costs billed to the user, $q_{\max}$ the number of object sensors which are involved in the query, $e_{\max}$ the total number of reports generated by the query, and, finally, $t_{\max}$ the total time the query shall be active. The limits $q_{\max}$ and $t_{\max}$ are particularly relevant for the *find* example. Here, the global query service will distribute the query to the first $q_{\max}$ sensors returned by the scope provider. The term $q_{\max}$ is equivalent to the number of messages sent by the global query service in the dissemination phase and can be used to limit the communication costs. The parameter $t_{\max}$ may be used to specify the time that is usually required for an adequate search strategy to find an object and thus limits the search effort of inadequate strategies. Moreover, it allows for an iteration of different search strategies of increasing costs, where the next strategy is initiated after an unsuccessful timeout of the previous. We will discuss suitable numbers for $t_{\max}$ in Section 4.6.

To obtain suitable search strategies for wide-area queries of the *find* scenario is a challenge. Various heuristics can be used to distribute a query to a relevant subset of object sensors. For example, one may distribute it to sensors near the location where the object was last observed. Similarly, all conceivable heuristics will be based on some kind of history data available in the system.

To elaborate on this, we show a simple data model of our application in Figure 4.3: *Objects* are associated with *users* (object owners) by the association service and also with *locations* (e.g., cells) where an object has been observed in the past. Users may choose to record a history of their location on their mobile device (*user history*) or to enable the *location profile* service, which computes locations that a user visits frequently. In this simple model, locations are related to other locations via the *neighborhood* relation. Moreover, users can be associated with *object sensors* they often use (e.g., which they have installed in their office or car) and with other users who are family, friends, or colleagues. Finally, the mobile network operator keeps a database (*object sensor registry*) which stores

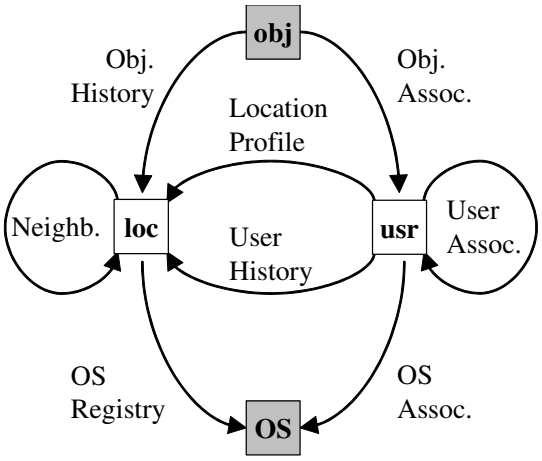


Figure 4.3: Example Data model

the current location (e.g., the current network cell) of object sensors (including certain mobile phones which can be used as object sensors as well as stationary sensors).

In order to simplify its description, we omitted some details in the data model (most prominently a more refined location model). However, it is sufficient to demonstrate that many conceivable search heuristics correspond to paths in Figure 4.3 from an entity of type *object* at the top to entities of type *object sensor*, which should be queried for the object. For example, we can query object sensors which:

I) Are near the location where the object was last seen.
II) Are near locations recently visited by the user.
III) Are near locations where the user spends a large amount of her/his time.
IV) Are associated with the object owner (as in Figure 4.1).
V) Match the above strategies III and IV for a different (associated) user, such as a family member, or even for a friend of a friend, etc.

While, intuitively, none of these heuristics can guarantee success, they all incorporate sensible assumptions on where users keep personal belongings and where these are generally left. Note how each heuristics represents a path in the data model of Figure 4.3: Heuristics I corresponds to the path (*obj-loc-object sensor*) on the left, while heuristics V corresponds to the path *obj-usr-usr-loc-object sensor*.

Based on these considerations, a data model is a suitable means to express the real-world links between various types of data stored by the system, and can be used to generate a variety of search strategies – including all of the above. In particular, the application programmer may assign

weights to each edge in the model, representing the application programmer's estimate whether exploring entities according to the relation type r will be useful in the search, on a scale from 1 (very useful) to ∞ (not useful). For example, to implement heuristics I, the *object history*, *neighborhood*, and *object sensor registry* relation types would have weight 1, and all others weight ∞.

4.4.2 Query Scoping Algorithm

Parameterized with a weighted *data model*, a *source entity* o (i.e., the sought object) and a *destination type* (i.e., object sensors), the scoping algorithm will "unfold" the data model into a search tree that contains entities stored by system services which are somehow related to the source entity o. The algorithm traverses the entities in the tree in order of decreasing (estimated) relatedness, essentially implementing a variant of the *uniform cost search* [RN95].

More concretely, the algorithm works as follows: It maintains a set of entities visited V and a result set Q. Each entity $t \in V$ will be assigned a *relevance* measure $c(t)$, denoting how related t is to the source entity o given the data currently known to the system. Initially (line 6.1), $V = \{o\}$ and $c(o) = 0$, moreover $Q = \emptyset$.

Algorithm 6: Scoping algorithm based on uniform cost search.

Input: Data model $M = (E, R)$ with entity types E and relation types R, weights $w(r)$ for relation types $r \in R$, source entity o (of type $\in E$), destination type $d \in E$, entity limit $q_{\max}$.

Output: Result set Q consisting of entities of type d

6.1 set $V = \{o\}$, $c(o) = 0$, $Q = \emptyset$
6.2 **while** $|Q| \leq q_{\max}$ **do**
6.3 pick relation (u, v) between entity $u \in V$ and $v \notin V$ with smallest $c(u) + c(u, v)$
6.4 or exit loop if no such relation (u, v) exists
6.5 add v to V, set $c(v) = c(u) + c(u, v)$
6.6 **if** v of type d **then** add v to Q
6.7 **return** Q

Based on the data referenced by the given data model, the algorithm in its core step (line 6.3) considers relations $\{(u, v)\}$ between entities $u \in V$ and $v \notin V$. If such relations exist, the algorithm picks the relation (u, v) with the smallest $c(u) + c(u, v)$.

By defining the cost function $c(u, v)$, developers can parameterize the execution of the query scoping algorithm. Typically, $c(u, v)$ is a function formulated in terms of two estimates: The first are weights $w(r)$ provided

by developers to assess whether the relation type r which stores the relation (u,v) is assumed useful in the search. Moreover, implementatons of $c(u,v)$ commonly take into account a second estimate, referred to as $g(u,v)$, by which the service that stores (u,v) may provide a relative order among different entities v related to u. In particular, $g(u,v)$ estimates the relevance of v *given* that u is relevant for the search. For example, the *location profile* service could estimate $g(user,loc)$ for different locations using statistics on the amount of time the *user* spends in them. For the *object history* relation, it is intuitive that the *latest* location where the object was observed is the most relevant. We will provide concrete examples for $c(u,v)$ in Section 4.4.3.

In the remainder of the loop, the algorithm adds v to the set of visited entities V, updates the relevance estimate $c(v)$ of v (line 6.4), and adds v to the result set if it is of type *destination type* (line 6.5). The algorithm repeats these steps until up to *entity limit* related entities have been found, and then returns Q (if the chosen destination type is *object sensor*, the entity limit corresponds to the limit on queried sensors $q_{\max}$).

A requirement for the edge cost function $c(u,v)$ is that it shall return values on the same scale ranging from 1 (for very related) to ∞ (for unrelated). If this is the case, the shortest path sums $c(t)$ maintain the same semantics. Based on the costs $c(u,v)$, the algorithm explores the most relevant relations first, and the result list Q contains entities t of type *destination type* in order of increasing $c(t)$, that is, in order of decreasing relatedness to the start entity o.

While based on classic search methods, the presented approach lends itself well to distributed data sources: Each system service that implements a relation type accessed by the algorithm is required to provide a single interface method *next(u)*, which allows to iterate through the entities v related to u in the order of increasing $c(u,v)$. Note that both the runtime and the space complexity of the algorithm depend on the data referenced in the given data model.

The algorithm implements a variant of *uniform cost search* [RN95] based on the edge costs $c(u,v)$. The result set Q contains entities t of type *destination type* in order of increasing cost $c(t)$, that is, in order of decreasing relatedness to the start entity o. Compared to uniform cost search, our approach does not expand all children of a node u at a time as there are potentially too many, due to several relations relevant for u and a potentially huge list of related entities provided by each one. Instead, at each step, we only add a single child entity v to the search tree.

4.4.3 Example Parameterizations

We aim to demonstrate the flexibility of the above algorithm for implementing a wide-range of search strategies using a set of application examples. These examples implement variations of the proposed heuristics I (to search where an object has been reported in the past) and therefore involve the relation types *object history*, *neighborhood*, and *object sensor registry* which are relevant for these heuristics.

Application Data

First, we briefly describe the data stored by each of these relation types. To illustrate scoping results, we assumed a location model based on cell identifiers. This model is based on a very rudimentary localization technology, which is also used in our prototype system: Object sensors (that is, the respective mobile phones) "measure" their location by observing the network cell they are associated with at a given point in time.

We will therefore use cell-based implementations of the *object history*, *neighborhood*, and *object sensor registry* relations. The *object history* simply returns a set of cells where the object was observed in the past. Given a certain cell, the *object sensor registry* will return object sensors that (at the time of the query) are associated to that cell. As we aim to demonstrate scoping with a large network, the implementation of the object sensor registry is based on our simulator software (detailed later on in Section 4.6), which associates object sensors to the cell with the strongest signal at the sensor's physical location. Such signal strengths are computed using realistic antenna locations and orientations [EGT05]. Finally, the *neighborhood* relation associates a given cell to cells which, based on their signal propagation, cover an area adjacent to the given cell. In a future system, mobile network operators are expected to be able to compute the cell neighborhood relation in a similar way, for example, based on the cell handovers observed in the network.

Search Tree Example

For an initial illustration of the algorithm's semantics, consider the example instant of algorithm execution shown in Figure 4.4. In order to implement the proposed heuristics I and search based on previous locations of an object, the application developer has disabled all relation types (by

setting $c(u,v) = \infty$) except for the relevant relation types *object history*, *neighborhood*, and *object sensor registry*.

Each enabled relation type can be used to generate child entities using the $next(u)$ method, which the algorithm invokes at the service implementing the respective relation type. Starting with the sought object o, only the *object history* relation type is enabled, for which a call to $next(o)$ returns the location where the object has been seen last, in this example in *cell1*. Given a cell entity, *object sensor registry* may generate child entities representing object sensors in that cell, while *neighborhood* generates child nodes which are other, neighboring, cells. The other relations will never contribute any child entities because they have infinite edge costs.

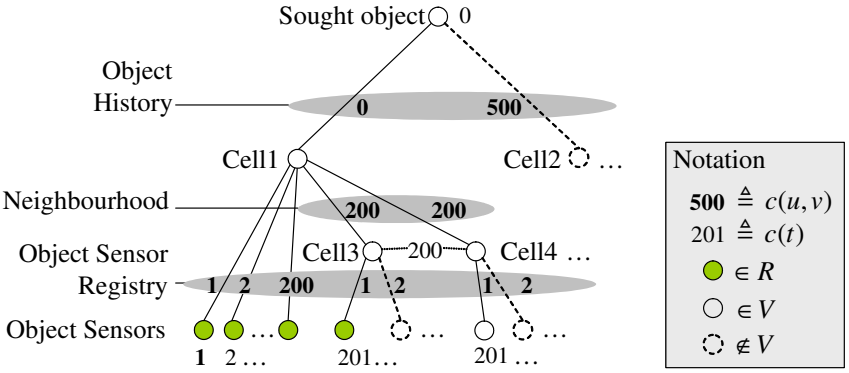


Figure 4.4: Example search tree generated by three relation types

As the algorithm's basic step will always add the edge leading to the smallest $c(t)$, one may express preference for one relation type over another by means of the edge cost function $c(u,v)$. In the shown execution, because $c(cell1,*)$ is set to 200 for the *neighborhood* relation, the algorithm explores (and adds to the scope) 200 object sensors in *cell1* first, and later explores object sensors in cell1's neighboring cells *cell3* and *cell4*.

Edge Cost Functions

In the following, we will give parameterizations for the edge-cost functions $c(u,v)$ which are able to implement a range of different search strategies. Although arbitrarily complex implementations of the cost function $c(u,v)$ could be provided by the system designer, in the examples below, we will use a simplified form of $c(u,v)$ which already suffices to formulate a range of examples. In our examples, $c(u,v)$ has the form

$$c(u,v) = a(r) + b(r) \times g(u,v) \tag{4.1}$$

in which $a(r)$ and $b(r)$ are parameters defining the relevance of the relation type r. Moreover, in this example implementation of $c(u,v)$, $g(u,v)$ corresponds to the index of v in the list of entities related to u – sorted by decreasing relevance. For example, the *object history* relation would set $g(obj, cell1) = 1$ if *cell1* were the location where *obj* was last seen, $g(obj, cell2) = 2$ for a second, less relevant, location *cell2* where the object was seen earlier. Note that $g(u,v)$ could implement much more fine-grained estimates of the importance of different entities v that are related to an entity u. Nevertheless, this simple implementation of $g(u,v)$ (returning v's place in the list of related entities) already allows for a range of heterogeneous search strategies, as we will show below.

As the algorithm implements uniform cost search based on $c(u,v)$, the cost parameters $a(r)$ and $b(r)$ have the following semantics:

$a(r)$ Defines how the *first* entity v contributed by r will be prioritized compared to other parts of the search tree. Once the algorithm has explored u, it will grow other parts of the search tree by $a(r)$ before it adds v. The higher $a(r)$ in $c(u,v)$, the less important is r compared to other relations contributing children to u and to other parts of the tree.

$b(r)$ Defines how an *additional* entity w contributed by r will be prioritized compared to other parts of the search tree. Such an entity w will be added to the tree only after all other parts of the tree have grown by $b(r)$. The higher $b(r)$, the less important are additional entities contributed by r.

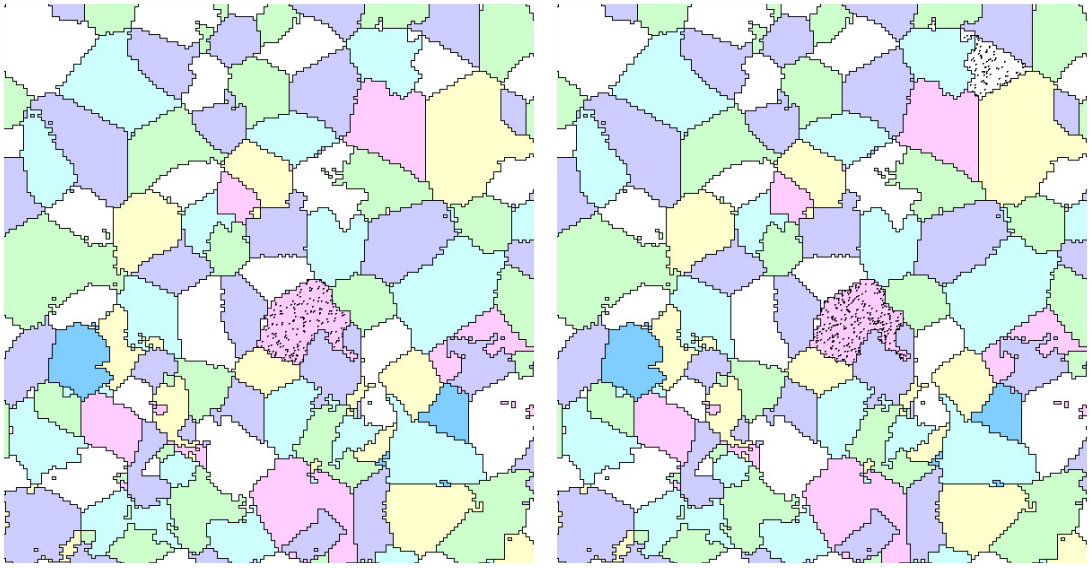
Searching based on the Object's History

The two weight parameters $a(r)$ and $b(r)$ can be illustrated using the example of Figure 4.5(a), which implements a variation of search strategy I: On the one hand, as the strategy involves searching at locations where the object was observed in the past, the *object history* relation is relevant in the search. Therefore, the algorithm should add the *first* related entity immediately, and $a(\textit{object history}) = 0$. On the other hand, considering *additional* locations further back in time is considered less useful in this example, therefore, $b(\textit{object history})$ is high in order to let $c(u,v)$ for such entities grow fast.

More specifically, based on the algorithm's uniform cost property, the ratio between $b(\textit{object history}) = 500$ and $b(\textit{object sensor registry}) = 1$

relation type r	$a(r)$	$b(r)$
object history	0	500
neighborhood	∞	∞
object sensor registry	0	1

(a) Weight parameters for searching at locations where an object has been observed in the past. For other r, $a(r) = b(r) = \infty$.



(b) Scope size $q_{\max} = 200$ (c) Scope size $q_{\max} = 400$

Figure 4.5: Example strategy and scoping results with different limits on the number of queried sensors $q_{\max}$. Sensors in scope are marked with black dots.

determines how the algorithm will access the two relation types: It will explore 500 object sensors at already visited locations (if these exist) using the *object sensor registry* before examining additional locations using the *object history* relation and adding sensors there to the scope.

Once a set of locations is included in the scope, the parameters of the *neighborhood* relation determine how fast neighboring locations should be explored. In this example, only the very cell where the object was last seen should be included in the search and, therefore, the neighborhood relation is disabled by setting $a(r)$ and $b(r)$ – thus, $c(u,v)$ – to ∞. In contrast, the algorithm should immediately start exploring object sensors in every identified cell, which is why $a(\textit{object sensor registry}) = 0$.

An example result – showing the sensors in scope given these parameters – is presented in Figures 4.5(b) and 4.5(c). For illustration, the figures include the simulated cell boundaries (lines where the signals from two neighboring cells have equal strength). We assume that the object was last reported in *cell1*, which is located in the center of the simulation area, and, earlier, in *cell2*, which is located to the north-east of the center.

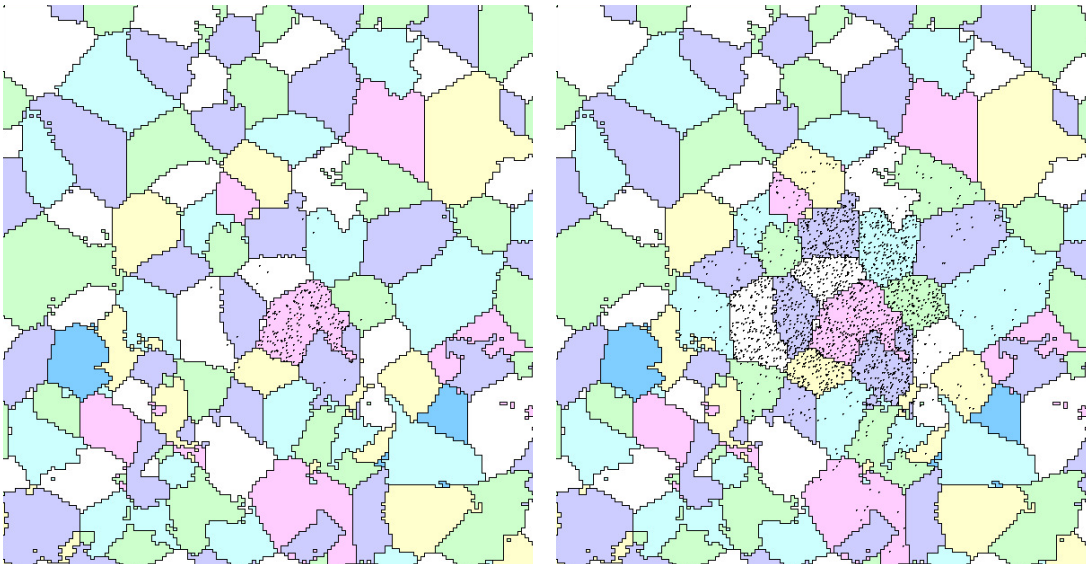
Because $b(\textit{object sensor registry})$ is low compared to $b(\textit{object history})$, additional sensors in *cell1* are added to the scope before *cell2* is considered (Figure 4.5(b)). If one increases $q_{\max}$ (the limit on the number of sensors) to have the algorithm return a larger scope, it will first add 500 sensors in *cell1* to the scope, and then additional sensors from *cell2* (Figure 4.5(c)).

Searching Neighborhoods

The above example can be altered in various ways. To have the algorithm also include neighboring cells in the search, e.g., to allow for the case that the locations returned by *object history* are imprecise, the parameterization of the *neighborhood* relation can be changed as shown in Figure 4.6(a). Setting $a(\textit{neighborhood}) = 200$ causes the algorithm to begin adding sensors from neighboring cells to the scope once about 200 sensors associated with already known cells have been added by means of the *object sensor registry* relation.

relation type r	$a(r)$	$b(r)$
object history	0	500
neighborhood	**200**	**0**
object sensor registry	0	1

(a) Weight parameters for searching where the object has been observed in the past and at neighboring locations as well. For other r, $a(r) = b(r) = \infty$.



(b) Scope size $q_{\max} = 220$ (c) Scope size $q_{\max} = 2000$

Figure 4.6: A search strategy based on the location of the last report and its neighborhood

Note that Figure 4.4 (introduced above) corresponds to such a parameterization (Figure 4.6(a)) and (roughly) to the results shown in Figure 4.6(b). It shows the instant of execution after the algorithm has added all object sensors with $c(t) \leq 200$ to the result set Q, and is about to add object sensors from neighboring cells in a uniform manner (in Figure 4.4 the next basic step will add a sensor in *cell4*). Figure 4.6(c) demonstrates the effect of an expanding scope size $q_{\max}$. The figure shows how, based on the uniform cost search semantics, cells within *two* hops are explored. This is based on the implications of the fraction $\frac{b(object\ history)}{a(neighborhood)} > 2$: The algorithm can begin expanding neighbors twice before an additional location from the object history relation would be added to the scope.

Because $b(neighborhood) = 0$, the order $g(cell1, v)$ in which neighbors of *cell1* are returned by the neighborhood relation does not have an impact on the execution of the algorithm: All neighboring cells are considered equally important and therefore child nodes (sensors in these cells) are added to the result set in equal proportions.

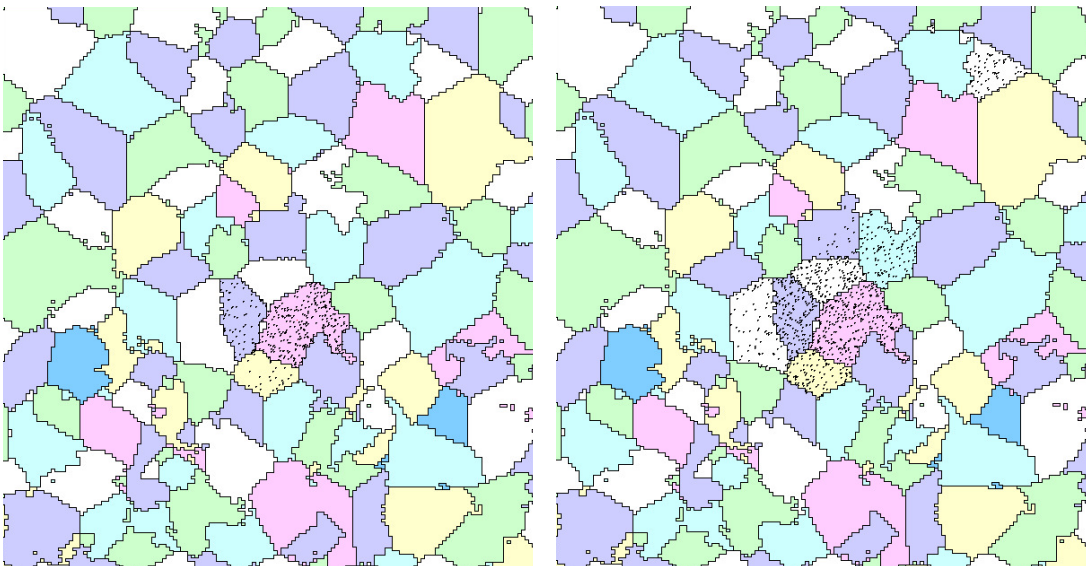
Ordering Neighbors

If an order of neighboring cells should be relevant, e.g., if based on information stored in the system some cells can be assumed "closer" to *cell1* than others, the developer may assign more significance to the order $g(u,v)$ returned by the *neighborhood* relation using a higher $b(neighborhood)$ parameter as shown in Figure 4.7(a).

Based on the uniform cost search property, 50 object sensors from *object sensor registry* will be added to the result set before the *next* cell is retrieved from the *neighborhood* relation. This is reflected in the respective scoping results shown Figure 4.7(b) where the scope is focused around the original cell and two (assumedly most relevant) neighbors. Expanding the scope to more sensors adds more neighboring cells, and, once the scope is large enough, the algorithm starts adding sensors at the second location returned by the *object history* relation (Figure 4.7(c)). Note that cell neighborhood is defined as cells sharing a common border and, due to non contiguous cell coverage, some cells are considered neighbors although this is not apparent in the figures. Further, note that the order $g(u,v)$ among neighboring cells is chosen randomly in this example while, in a future system, neighboring cells could be prioritized by various profiling services, e.g., based on data from the location profile [LRT04]: If users tend to switch between neighboring cells frequently, these cells are

relation type r	$a(r)$	$b(r)$
object history	0	500
neighborhood	200	**50**
object sensor registry	0	1

(a) Variation of the parameters to take into account the order in which cells are returned by the neighborhood relation; for all other relation types r, $a(r) = b(r) = \infty$



(b) $q_{\max}$=400 (c) $q_{\max}$=1000

Figure 4.7: Search strategy taking the order of neighboring locations into account

likely to cover the same physical location and, therefore, are both likely to be useful in the search.

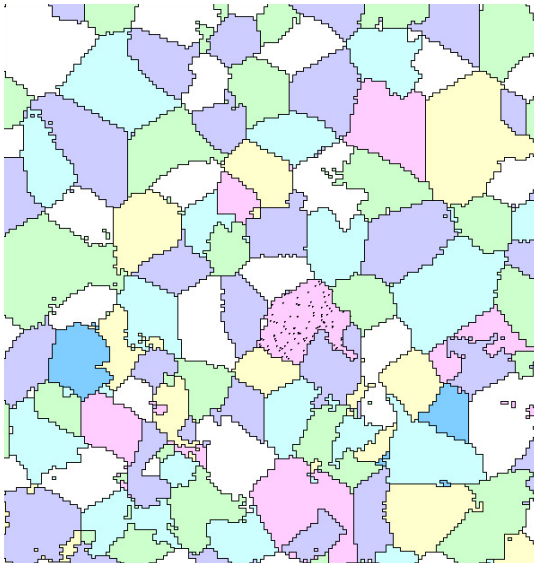
A Combined Search Strategy

We close with a last example, which demonstrates how the algorithm may implement combined search strategies that are based on multiple paths in the data model. In the previous parameterizations (e.g., in Figure 4.6(a)), the neighborhood relation was prioritized compared to the *object history* relation type). To state that the search should explore previous locations of the object as well, one could decrease $b(\textit{object history})$, and therefore implement a mixed strategy based on data on an object's previous locations and on the neighborhood of these locations.

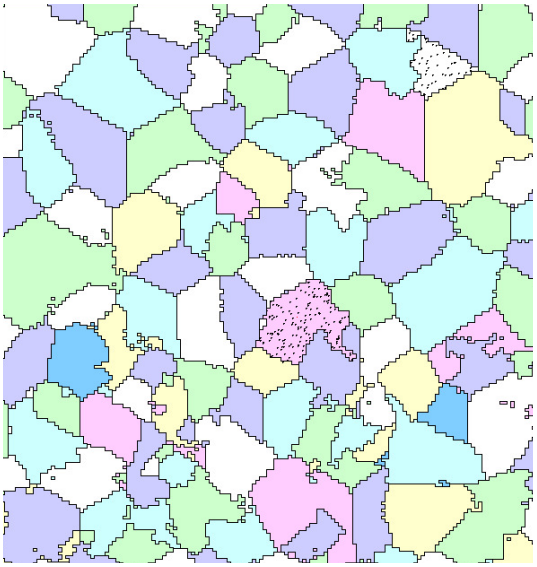
Such a combined strategy is shown in Figure 4.8(a). Setting $b(\textit{object history}) = 100$ demands that the search should consider an *additional* location where the object was reported, once 100 sensors have been added at previous locations. Moreover, the ratio $\frac{a(\textit{neighborhood})}{b(\textit{object history})}$ implies that a second history location is considered in the scope, before the

relation type r	$a(r)$	$b(r)$
object history	0	**100**
neighborhood	200	**0**
object sensor registry	0	1

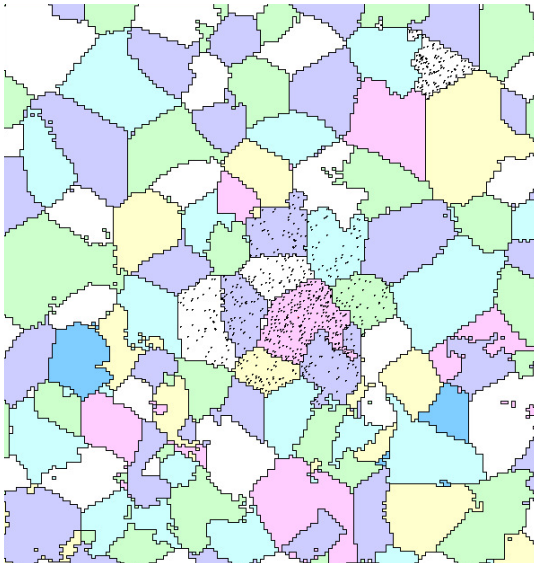
(a) Example weight parameters for a mixed strategy based on object history and neighborhood; for other r, $a(r) = b(r) = \infty$



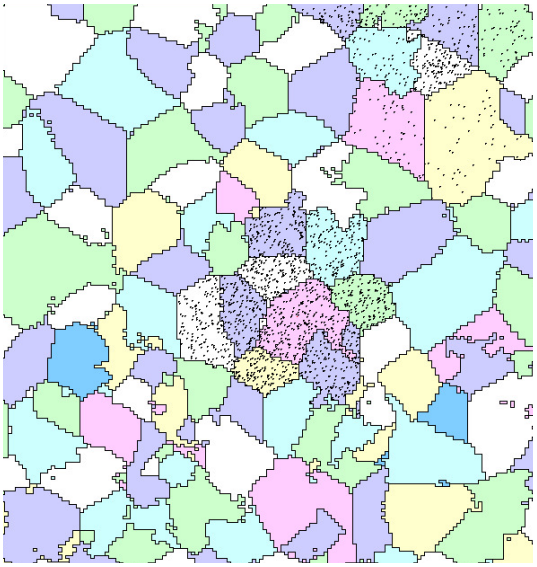
(b) $q_{\max}$=100



(c) $q_{\max}$=200



(d) $q_{\max}$=800



(e) $q_{\max}$=2000

Figure 4.8: A combined search strategy (the search scope is expanded in four steps)

neighborhood relation will start contributing neighboring cells and add sensors near these locations to the resulting scope Q.

We show a set of scopes computed according to Figure 4.8(a) in Figures 4.8(b)-4.8(e). The results highlight how the algorithm adaptively integrates different search strategies while the user expands the size of the query scope $q_{\max}$. When the scope size is low (in Figure 4.8(b)) the strategy with highest preference (to include sensors at the location where the object was reported last) is considered. If the system cannot find the object in this small scope, the user might choose to increase $q_{\max}$. This will produce an enlarged scope (Figure 4.8(c)) that includes sensors at a second location while still prioritizing the first. Once even more sensors are permitted, and because no additional sensors are available apart from the most probable locations (*cell1* and *cell2*), the scope expands by means of the neighborhood relation only (Figures 4.8(d) and 4.8(e)).

4.4.4 Discussion

In comparison with previous chapters, application requirements have driven the design of a very specific role assignment algorithm which assigns the role *sensing* to nodes inside the computed scope.

One the one hand, requirements included allowing users to formulate (and even automatically generate) a wide range of strategies for selecting sensing nodes. To achieve this, the algorithm was required to be able to take into account *arbitrary* information available at system services. This has led to an interface, described above, which is based on data models of the application domain – annotated with weight parameters that define the relevance of relations in the model.

On the other hand, efficient communication between different sources of data used by the query scoping algorithm is required, because the relations driving the algorithm's execution are stored at distributed application services. Therefore, the interface between the algorithm and data sources has been streamlined to a simple $next(u)$ method, which is called only once for every related entity.

These requirements have been met well. Some open issues, however, remain.

Semantics

The first issue is related to the *semantics* of a given set of parameterized functions $\{c(u,v)\}$. While these functions provide a flexible configura-

tion interface, the effects of a given function definition $c(u,v)$ are not a priori well understood by developers. Instead, the exported semantics are closely bound to the algorithm's execution. In contrast, the Boolean predicates used in the generic role assignment specifications described in Chapter 2 are a much more intuitive concept. Similarly, the implications of the parameters issued to the facility location algorithms of Chapter 3 can be considered well-understood concepts: They represent an estimate of the cost of either operating a hub or communicating with a hub in a given epoch of network operation.

In comparison, the understanding of the provided levers to the above query scoping algorithm may be less intuitive. This seems to be a feature as well as a drawback. Using the functions $c(u,v)$, the developers are provided a very fine-grained control over the algorithm's execution. In turn, for specialized search strategies, the developers must perform an important translation between the search strategy they have in mind and the respective design of the functions $c(u,v)$.

Note, however, that this is mostly the case if *multiple* paths through the data model are explored in combined search strategies. If a single path is followed instead, as is mostly the case in the heuristics I-V described in Section 4.4.1 above, the parameterization is much simpler, mostly just enabling or disabling certain relationship types.

Moreover, we argue that even for more involved strategies, an application developer is provided a valuable configuration tool that combines distributed data sources quickly and effectively. When combining different paths through the data model, care must be taken to make the relevance estimates $g(u,v)$ (issued by different relations) comparable – by means of weights attached to the respective cost functions.

Finally, while the above algorithm is flexible and requires only a limited amount of communication between distributed data sources, compared to other search algorithms, the described approach involves an additional approximation heuristic: During search, it considers only the *shortest* path to a given entity, and disregards all longer paths that may be discovered later on during algorithm execution. This is a viable approach and also crucial for enabling efficient exploration of the distributed data sources. However, it neglects the fact that a large number of ignored long paths could imply that an entity t is more relevant than it is apparent in the length of the shortest path $c(t)$.

More sophisticated algorithms (such as PageRank [BP98]) let the node weight $c(t)$ represent a weighted sum of *all* paths to t, which is computed

by solving a system of linear equations. Such algorithms require that all paths are explored before computing the rank (in terms of relevance) of an individual sensor. In contrast, our application requires that sensitive data stores (e.g., association or the location profile) can be stored on user's devices to protect the user's privacy (cf. Section 4.5). Therefore, the exploration of all relevant paths at the same time would not be possible – unless one gathers all data in a centralized database. We perceive such centralized data collection as a significant disadvantage based on its privacy implications, in particular, given the large amount of personal information our system may collect on its users' everyday lives.

Space Complexity

A second issue of the presented algorithm is that it inherits the exponential space complexity present in any breadth-first search. This does not represent a significant disadvantage in our application domain, however: Firstly, the costs of an object search billed to the user will most likely be proportional to the number of queried object sensors (due to communication costs). As only a constant (user-defined) budget is granted, at most a *constant* number of object sensor entities will be in the search tree (as any visited object sensor will be queried during search). Secondly, the number of intermediate entities (like locations) in the tree can also be limited by raising the weight parameters of individual relations, which will re-configure the algorithm to explore "deeper" paths earlier and reach object sensor entities faster. In the example execution shown in Figure 4.8(a), setting $a(\mathit{neighborhood})$ higher would force the algorithm to explore many object sensors at already known locations before including additional locations in the search.

4.4.5 Global Query Service

In the following, we will discuss how the presented algorithm is employed by the global query service to implement the search strategies I-V in practice. For this, note that search strategies I-III employ the *object sensor registry* relation, which associates a set of locations L to a set of object sensors near them. Due to user mobility, however, the object sensors near the set L will change with time. Therefore query scoping is decoupled from the actual movements of users and their object sensors. That is, when heuristics I-III are implemented, *location* will be the *destination type* parameter passed to the search algorithm. The returned set of loca-

tions L, for example a set of cells, is then passed on to the global query service, which will distribute the query at these locations. The global query service is then concerned with maintaining a computed query scope over time while observing the cost control parameters ($q_{\max}$, $t_{\max}$, $e_{\max}$, and $c_{\max}$) specified at the query interface.

If search strategy IV is chosen, a set of object sensors is determined by the scoping algorithm. Here, a query will be distributed to the first $q_{\max}$ sensors returned by the scoping algorithm and be active for at most $t_{\max}$ time.

If search strategies I-III are chosen, the scoping algorithm will not directly return a set of object sensors, but a set of locations, as mentioned above. In the basic location model we employ, these can either be a set of cells (the most basic localization already available on the phone) or a set of geographic points (if phone localization is more precise) together with an associated measurement error. Because the set of object sensors associated with these locations may change over time, our system installs (or un-installs) a query at sensors which come close to (or, respectively, depart from) these locations. Whether a sensor s is close to the returned locations is defined by the implementation of a predicate f (which maps s to either *true* or *false*).

Depending on the way *locations* are modeled, we use two different implementations of $f(s)$. Given a set of cells C, $f(s)$ will be true if the mobile phone (with its object sensor s) is currently served by any of the cells in C. Note that this information is already available at the mobile network operator, that is, it can be accessed on the server side of our infrastructure without incurring additional communication costs.

In case the mobile devices are equipped with more accurate positioning means, the locations returned by query scoping will instead be a set of geographic points P. Here, $f(s)$ will be true if the current position measured by a mobile phone's object sensor s is within a certain range r away from the points P. This range r will depend on the error incurred at the positioning sensor when the points in P were measured (we will discuss a concrete implementation in Section 4.6). Note that such additional positioning information will only improve the efficiency of query dissemination if positioning information of all object sensors is already known at a database on the server. Otherwise, it would be inefficient to propagate all object sensor positions to the server before query dissemination, and therefore a different approach is chosen: The query is distributed to object sensors in a set of cells C which "cover" the whole area surrounding the

points P (the actual object sensor will be turned on only later, once the predicate $f(s)$ evaluates to true). Note that the total number of distributed queries is now the same as if locations were a set of cells C.

When installing queries for such location-based strategies I-III, the total number of object sensors at which a query will be installed (q_{total}) is made up of two parts, $q_{\text{total}} = q_{\text{init}} + q_{\text{mob}}$. Here, q_{init} denotes the number of users queried initially at the time the query is issued – chosen out of the initial query scope $S_{\text{init}} = \{s | f(s) = \text{true}\}$. In addition to q_{init}, the query will be installed at a second set of sensors q_{mob} for which $f(s)$ becomes true while the query is active.

After a query installation at an object sensor s, object sensing will be performed continuously until $t_{\max}$ expires. The mobile device associated with s un-installs the query autonomously either when $f(s)$ becomes false or when $t_{\max}$ expires.

A query is declared successful, if some object sensor s reports having found o at time t_{reply} with $t_{\text{reply}} \leq t_{\max}$. The current position of s represents the location at which the object was found and will be included in the reply issued to the user. A query is terminated without success, once the query timeout $t_{\max}$ is reached.

4.5 Privacy Considerations

The query service, presented in the previous sections, makes use of a wide variety of personal user data. In the following, we describe a few privacy enhancing features of our system.

Most prominently, tagged objects and persons can be protected from being sensed by unauthorized users as proposed in [EHJ04]. This protection is based on a shared secret x, which is known both at an authorized user device U and at the sensed entity (e.g., at a tagged object o). With query initiation, U may issue a zero-knowledge authentication message (ZAM), say m, in which the shared secret x is made oblivious by means of a random session key r and a current time stamp d. The ZAM is formatted such that the receiving tagged object o can recover r as it knows x and also receives an authentication proof that U truly knows x[1]. Based on this feature, if authentication fails, o simply does not reply. Moreover, o will only reply to the first reception of m. Therefore, an adversary A cannot easily make use of forwarded queries to locate objects which have been lost by other users in public spaces. To be effective, A must not only have been forwarded the find query containing m but also be the first to

sense o using *m*. To achieve this, *A* needs to control an infrastructure of object sensors that finds objects faster than the infrastructure provided by regular system users – in most cases an unrealistic assumption.

As noted, a ZAM represents a permit for sensing *once*. For use cases in which sensing of an object or person *o* must be performed *multiple times* within a given query (e.g., *delegate* or *gate*), there are three options: The first is to let an authorized user device *U* re-issue a proper ZAM every time, which requires repeated end-to-end communication between *o* and *U*. The second is to entrust the shared secret to the global query service (run by the mobile network operator on the back-end server) which may then issue authentication messages on its own. The third is to propagate the shared secret even further to the remote sensor *R* (such as the object sensor employed in the *gate* scenario) for the time of query execution. The latter two options lower the communication overhead (the overhead of the third option is lowest) but require that the user is confident that the entrusted entity (e.g., the remote sensor *R*) does not leak the shared secret during the time of the query. After query termination, the shared secret *x* may be changed – such a change could be propagated to multiple users *U* previously associated with *o* using the same authentication protocol.

A second privacy enhancing feature of our system is the component deployment infrastructure, which allows to re-configure the execution platform of services containing sensitive information within the application. That is, the association registry or the location profile, which should enhance object search functionality, could be executed on the user's mobile device instead of the server back-end, giving users full physical control over their data. Similarly, deployment of the location profile service could be completely omitted – based on user preferences.

Some potential privacy threats, however, remain. One threat is shared by any system providing wide-area object localization, namely, that an adversary may attach a properly associated object to some person he or she wishes to track. Avoiding such threats is hard as they are related to the core functionality of the envisioned system. Possible approaches would be to limit the frequency of user queries, or make repeated consecutive queries for an object expensive. Vice versa, adversaries may leave an object at a known location, and then check whether associated users arrive at this location. Here, the required transparency can be obtained by mak-

[1] A ZAM has three parts: $m = (d; r \text{ xor } \text{hash}(d \text{ xor } x); \text{hash}(r \text{ xor } x))$. Using d and part two, o can compute r. Using r, o can check part three, which confirms that the sender truly knew the secret x. For details we refer to the original paper [EHJ04].

ing association services symmetric, such that an object sensor carrier is allowed to see which associated users sent queries to his or her device – while un-associated users are kept oblivious to each other by the mobile network operator.

Several extensions of these concepts are conceivable. For example, users can be offered a feature to protect private spaces [YSM05], such as their home, from queries issued by unauthorized users. While dependent on accurate and verifiable location sensors for user devices, such protection of spaces could be enforced by the mobile network operator, which exerts the role of a gatekeeper in both directions, when forwarding user queries and when forwarding replies. This is similar to the way, in the presented system, the network operator will ensure that a user's identity is only disclosed to associated users.

4.6 Evaluation

So far, we have described a bundle of generic query services that support our object localization application with query dissemination. These involved a query scoping algorithm which is able to implement a range of different search strategies.

The evaluation of the devised query scoping algorithm, however, is far from straightforward. To begin with, an evaluation must involve a large set of sensors in order to determine whether the scoping algorithm is both effective and efficient. This requires either an extensive user study, or experiments performed using simulations. If an extensive user study is not feasible based on time and budget constraints, the remaining option for large-scale experiments are simulations.

Unfortunately, evaluation by simulation requires the provision of two models. The first is a simulation model of the *real* circumstances of objects that are lost or misplaced, for example, of a user that has left an object at a friend's house. Such a model would already be hard to justify. Moreover, a second model is also required. This second model is implied in the used search strategy, which essentially models the *assumed* circumstances of an object's loss (for example, a strategy matching the circumstances of the loss would search using social relationships and the owner's location trace). If the assumptions made in the search strategy match the reality of the simulation model, the search will find the object. Otherwise it will not.

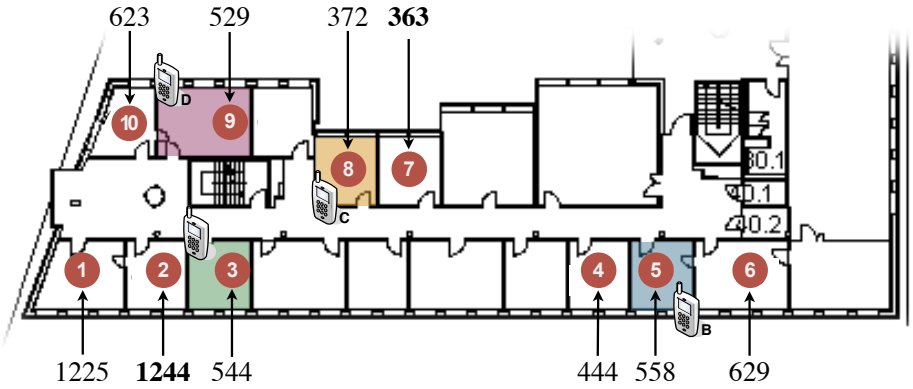


Figure 4.9: Experiment setup: Average reply times (in seconds) for the 10 tagged objects in different rooms

Based on these considerations, in the remainder of this chapter, we avoid generating full models of the system's future use. Instead, in the evaluations of this section, we assume that the employed search strategies are in fact *correct*. Such evaluation allows to assess the performance of a user-carried infrastructure for finding an object, given that the user will at some point come up with a correct search strategy.

In the following, we therefore provide an evaluation of the performance of our system using two strategies mentioned in the course of the chapter – using a small real-world experiment and a set of simulations.

4.6.1 Real-world Experiment

We first come back to the scenario from the introduction of this chapter (Figure 4.1), where the user is at home and tries to verify the whereabouts of a given object which was left at the office. The mobile phones of the user's officemates (e.g., Bob) are registered with the association service and thus are considered relevant object sensors.

Our experiment was performed with four users working on the same floor. The users were given mobile phones running the object search prototype already tasked to perform continuous object sensing for all objects (using repeated Bluetooth discovery) and to report their findings in regular intervals to the back-end database. Similarly, 10 BTnodes [BTn06] representing tagged objects were distributed in various rooms of the same floor. Figure 4.9 shows the experiment's setup (tagged objects are shown as numbered circles while the offices of the four participating users are shaded).

Note that while Bluetooth may be too expensive and battery-intensive to be used as an object tagging technology in a practical system, it neverthe-

less allows to test whether, given a better-suited technology with similar radio range, the mobility of a few office colleagues suffices to detect a given object in reasonable time.

During four consecutive days, all sensor readings entered the database as *(user, time, obj_id)* tuples. We considered only core office hours, that is, readings reported after the fourth user arrived in the morning and before the first left at night. This resulted in 30 hours of data. Based on these data, the reply time of a search query for a given object o issued at an arbitrary time t_q to the four office colleagues can be computed as $t_r = t_{\mathrm{DB}} - t_q$, where t_{DB} is the time of the next database entry on the object o.

Based on this consideration, we computed the average reply time for each object, assuming that queries for this object were distributed uniformly over the experiment time. In order to save messaging costs, user devices cached seen objects and only re-reported them to the database 10 minutes after their last report on the same object. This way, even if an object sensor has seen the object continuously, the resulting reports will yield an average query reply time of 5 minutes instead of zero.

For each object, Figure 4.9 shows the average reply time in seconds. Intuitively, we expect to obtain low values for objects with a participating user in the same room (objects 3, 5, 8, and 9). Further, note that the best results were obtained for objects close to the printer and the coffee machine (objects 7 and 8), while objects in rooms that were not visited by the participants during the experiment yield worse results.

We show a cumulative density curve of the observed reply times for object 2 (with worst results), object 7 (with best results), and the average over all objects in Figure 4.12(a) further down. In all cases, reasonable success rates (about 80%) could be obtained with a maximum query time $t_{\max}$ of 30 minutes.

4.6.2 Simulation Model

In our experiments described above, we focused on a small and confined search area and query scope. In the remaining evaluations, we use simulations to investigate the characteristics of an object search system operating in the wide-area with a larger user base – to provide design guidelines for a future realistic system.

As we mentioned, adequate models of a large-scale execution environment are difficult to obtain, as these must consider many aspects of daily life. To provide an accurate basis for system design, models must include

the number of participating users, the frequency at which these users lose or search for certain objects, particular scenarios where objects are lost, and the number of tagged objects owned by each user. Intuitively, such a model contains many parameters which cannot be influenced by the system designer. We refer to such parameters as *environmental parameters*. Our approach to these parameters is to investigate a significant portion of this rather vast parameter space.

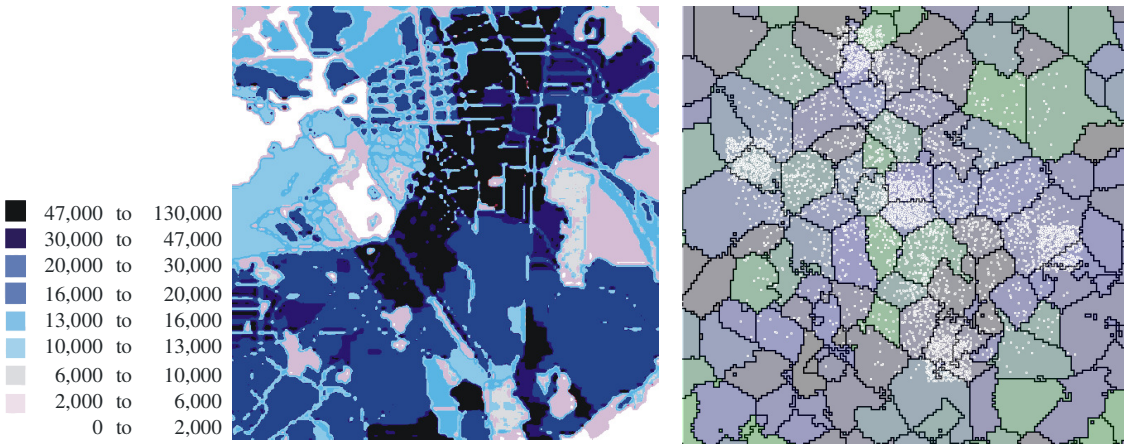
In contrast, there are some *design parameters* which determine the system's performance and can more or less directly be set and varied by the system developer. These include the size of the search scope (the number of users who participate in searching for an object), the sensing range of an object sensor (which can be influenced by employing more expensive tag and object sensing hardware), or the timeout used for queries. For these *design parameters* we aim to find the most appropriate values, i.e., the parameter settings which can implement object search with the least communication overhead for a given success rate.

Scenario. In the evaluated scenario, a user misplaces an object o and later issues a search query to the global query service. We assume that at the time the object left the range of the object sensor, the user's mobile device recorded its location p. This location p is used as a hint in the search (implementing heuristics I presented in Section 4.4.2). We evaluate two versions, a *cell-based* version in which p is a cell, and a *position-based* version in which p is a geographic point measured with a certain error. In both versions, query scoping is performed as described in Section 4.4.5.

Metrics. The main metric we observe is the *success rate* of our system. This rate corresponds to the fraction of queries for which a notification from some object sensor is received within the query timeout $t_{\max}$. Further, we examine the overhead for query distribution q_{total} including the part q_{mob} which is caused by user mobility.

In our simulations, we do not examine object sensing costs explicitly, as we expect wide-area query dissemination to dominate the total cost due to the object sensor's shorter wireless range and potential energy efficient implementations of object sensing (e.g., it is usually sufficient to briefly activate an object sensor every time it is moved).

Environment Model. We assume that the object is left in a densely populated urban environment. In this setting, we study how an object can be found by users who move according to pedestrian mobility models (see details on the mobility models below) in a square area of $1\,\text{km}^2$. The



(a) Population distribution per km^2 during the day of a 10 km^2 square area of downtown Lisbon [FCX$^+$03]

(b) Simulation on the same area: Cell coverage model [EGT05] and users (dots) distributed according to the campus mobility model

Figure 4.10: Environment models

choice of the user density u_d is derived from the total daytime population as estimated by the Momentum project [FCX$^+$03] (a downtown Lisbon example which we cite from [FCX$^+$03] is shown in Figure 4.10(a)). For urban environments, the authors estimate the fraction of mobile-phone users with "pedestrian" mobility patterns as around 50%-70% [FCX$^+$03, p. 37] from the total. The total includes other users who are assumed to be stationary or moving differently, e.g., with higher speeds on streets; we omit these users in our simulations. Moreover, as we are only interested in users associated with *a single* mobile provider, we choose more pessimistic values for the user density u_d: We vary u_d from 100-2000 users/km^2, values which represent only a small fraction of the estimated daytime population shown in Figure 4.10(a). Our default participant density of u_d=500 users/km^2, for example, corresponds to only *one* pedestrian user per 2000 m^2 of office space.

As mentioned, in some settings we rely on cell identifiers for positioning. To study such scenarios, we use actual position and orientation data from UMTS antennas together with a detailed model of land use types (e.g., buildings, highways, open, water) provided by the Momentum project to compute the strongest-signal cell for each point of the simulation area [EGT05, Mom06]. In Figure 4.10(b), we show an example of a resulting cell-coverage map computed for a UMTS network of downtown Lisbon. For cell-based scenarios the simulation area is enlarged to 10 km^2 to avoid border effects. While we are aware that in reality several cells may be observed at a given location at different points in time, we assume for our study that the object can be found in the cell where

it was last seen. Results for scenarios in which several cells need to be searched to cover a certain location could be extrapolated from the results we provide.

Mobility Models. Generally, a tagged object will not move once its owner left it somewhere. In turn, the users' mobility model is a crucial aspect in our evaluation, as it determines the coverage obtained by object sensors carried by users. Therefore, the user densities we assume in our simulations are rather small compared to actual densities observed in urban environments or on office floors. In this regard, the fraction of simulated users represents the subset of all users who move according to the model we simulate. Additional users, e.g., sitting in their offices or moving differently, would then only improve results.

In the most basic setting, we use a random waypoint mobility model parameterized for pedestrian users. Users pick a random destination and start moving towards it with a speed drawn uniformly from the interval (2,4) km/h. (The average speed of 3 km/h is chosen according to the ETSI guidelines [ETS98].) As our simulation area can be fairly large, we choose trip destinations within 200 m from the user's current position.

We also use a second mobility model which was derived from WLAN traces observed on the Dartmouth campus [KKK06]. The model includes *hotspot* regions which represent central points of the campus (for example, a hotel, a library, or a cafeteria), which tend to contain many users and also represent popular destinations chosen by the campus population. In our adaptation, we use five hotspot regions: one in the middle and four shifted to each side of our simulation area. Each hotspot region's size is one hundredth of the simulation area. Half of the trips of a given user are made inside the current hotspot and half are directed to another arbitrary hotspot on the campus. The remaining (non-hotspot) area is called the *cold* region. In our simulation, users never choose a destination in the cold region but only travel through it. As hotspot regions have a higher density, their positions and sizes are apparent in Figure 4.10(b), which shows a total density of 500 users/km^2 in a 10 km^2 area of downtown Lisbon.

The chosen trips include 2 to 5 waypoints. The speed and pause times follow log-normal distributions parameterized according to [KKK06, table 3]. The pause time distribution has a mean of 0.71 hours with a high standard deviation of several hours, as [KKK06] found that users tend to stay in a hotspots for longer periods.

To avoid an initial transient period, we used initializations of user trips according to the perfect simulation method [LV05]. In the campus mobility model, however, some distributions had to be estimated, and thus a transient period of 1000 s remains. Object search queries are issued only after this period.

Sensor Model. A mobile device operates its object sensor continuously as long as a query is installed and running. In the default case, we assume that the object sensor has a sensing range of 5 m, that is, the object sensor sends a notification once the user carrying it comes within 5 m of the sought object.

Moreover, in some simulations we assume that the user has a position sensor available (e.g, GPS). To model the sensor's localization error, the position returned by the sensor is drawn uniformly from a disk centered around the actual position of the user. We refer to the radius of this disk as the *positioning error* e_p used in the simulation (e_p is set to 100 m if nothing else is stated). Note that with this error distribution, the density of observing an actual error, say e, is proportional to the circumference of a circle with radius e, and therefore the mean error is $(1/\sqrt{2})e_p$. This positioning error occurs not only when the owner's mobile device records the position p where it has last seen an object, but also when distributing a query to object sensors near p, as these object sensors' positions are measured with an independent positioning error.

Alternatively, we also model a scenario in which the positioning sensor simply returns the UMTS cell to which the mobile phone is currently connected. This is a worst case scenario as most cell-based localization approaches combine signal strength information from multiple nearby cells together with antenna positions and so-called time advancement values to obtain more accurate localization results.

4.6.3 Simulation Results

Using the simple scenario and the environment models described above, we aim to investigate several aspects of a future object sensing system. Foremost, given some scope, we want to confirm whether it is possible to find objects with reasonable success rates and small-enough overhead. Further, we aim to investigate how cell-based scopes compare to position-based scopes and to a random query dissemination strategy which queries a certain fraction of all users. Moreover, we aim to gain insights into the

sensitivity of the system's performance with regard to parameters such as user mobility, object sensing range, or chosen query timeouts.

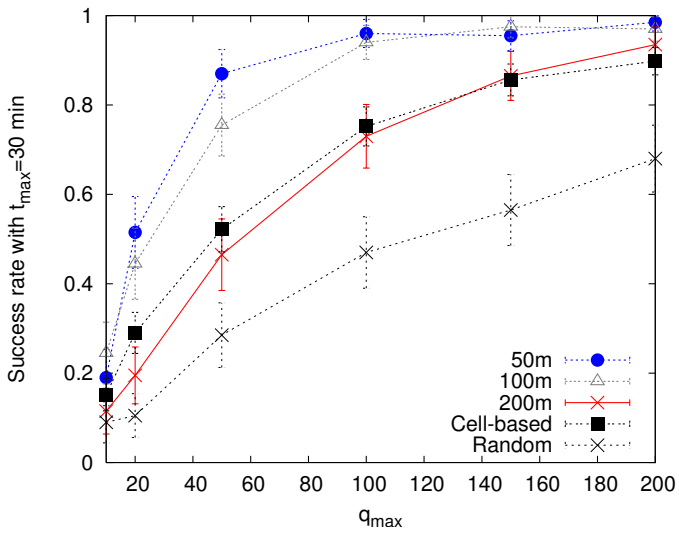
Success Rate

In the first set of simulation runs, we investigated the query success rate observed with *position-based* scoping and *cell-based* scoping. These scopes are implemented according to Section 4.4.5, based on the location p where the object was last in range of the user's device. In the position-based version, the scope S_{init} consists of sensors within a disk around p of radius $r = s_r + e_p$ where s_r denotes the range of the user's object sensor and e_p the maximum positioning error. In both versions, if $|S_{\text{init}}| > q_{\max}$, then $q_{\max}$ sensors are randomly chosen from S_{init}.

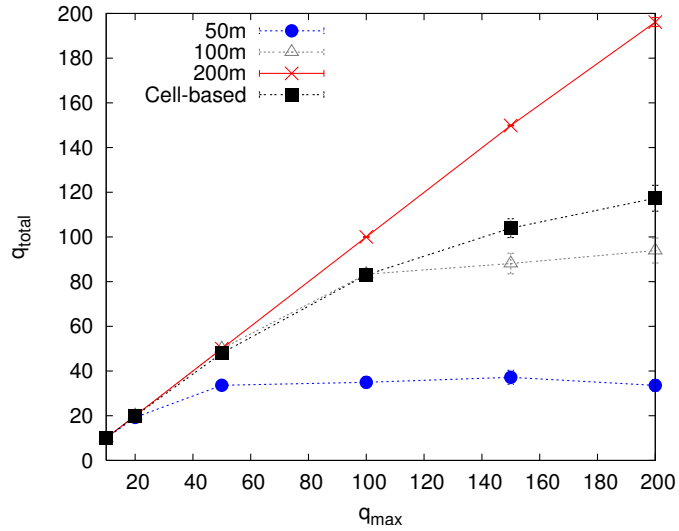
Figure 4.11(a) shows the fraction of successful queries (upon which the queried sensors have located the object within 30 minutes) when the user-imposed limit $q_{\max}$ on the number of queried sensors is varied. Five different graphs show the results obtained with different positioning errors e_p (from e_p=50 m to e_p=200 m), cell-based scoping, and a random strategy where we distribute the query to a fraction of $q_{\max}/500$ of all users. For all graphs, the obtained success rate can be increased by raising $q_{\max}$ and reaches acceptable levels with $q_{\max}$=200.

The communication effort involved in the same runs, determined by the number of sensors q_{total} which were actually queried, is shown in Figure 4.11(b). As, by the definition of our protocol, the search area becomes larger with an increased positioning error, the required effort increases as well. Similarly, searching the coverage area of the cell where the object was left requires sending more messages before obtaining reasonable success rates. Note, however, how in Figure 4.11(b) the number of queried sensors q_{total} at some point stops growing with the user-imposed limit $q_{\max}$. This is because with small enough scopes the object is found before the limit $q_{\max}$ is reached. Observe also, how the performance of cell-based scoping is comparable to a 200 m positioning error and even outperforms the latter in terms of the communication effort q_{total}. This is because with position-based scoping and large positioning errors, many ineffective queries are sent to mobile devices which erroneously measured a position which was close to the position hint p.

Finally, as Figure 4.11(a) shows, any scoping performs better than a random strategy. Even if 40% of all users are queried (i.e, $q_{\max} = 200$), the success rate of the random strategy is still only around 60%. Needless



(a) Success rate



(b) Sent messages

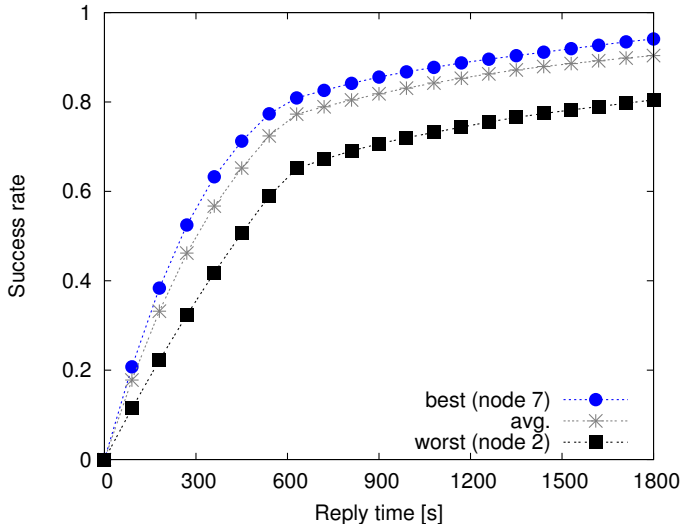
Figure 4.11: Success rate and overhead with different positioning technologies

to say, its communication effort is worst as it is proportional to the total number of users (not shown).

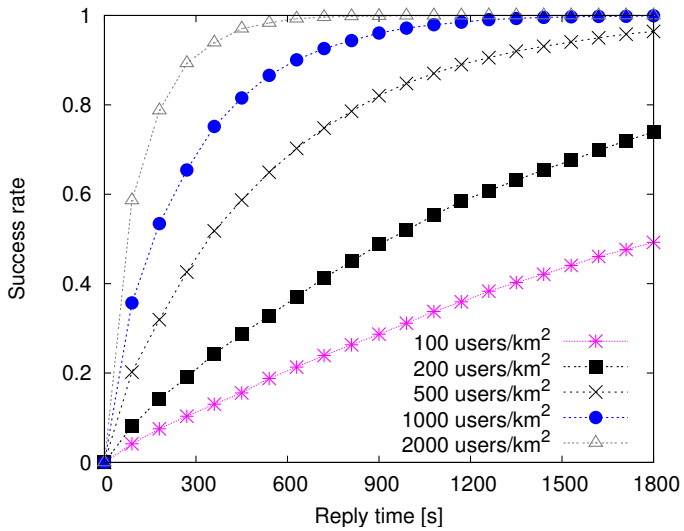
Timeout, Sensing Range, and Different Mobility Models

Apart from scoping, several other parameters may significantly influence the performance of the system.

The first is the timeout used for queries. Here, the question is whether a more adequate choice of the timeout (previously set to $t_{\max}$=30 min) waiting for successful replies can be made. Note that choosing an adequate timeout is particularly relevant when *object sensing itself* is considered a significant cost. Especially because in reality the object might be out-



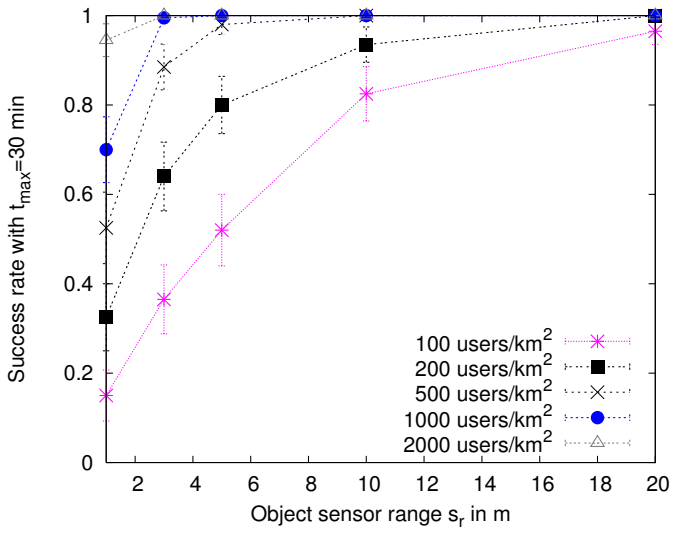
(a) Real-world measurements



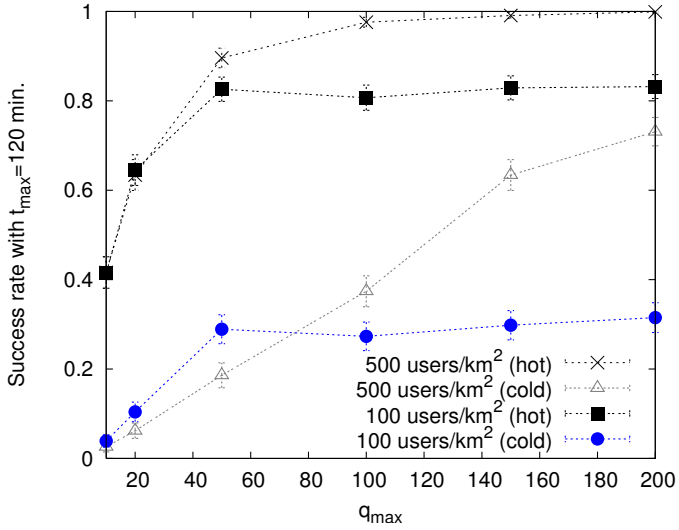
(b) Random waypoint simulations

Figure 4.12: Cumulative density functions of reply times

side the chosen scope, it is important not to sense in vain for too long, but at the same time to issue a confident "not-found" reply. Further, the system performance is expected to vary with the user density. We show the interplay of these two parameters with position-based scoping in Figure 4.12(b). Each graph represents the cumulative density function of the reply time obtained after 5000 repeated simulation runs (each data point represents the fraction of requests answered within the given timeout) in which no limit on the communication effort $q_{\max}$ was set. As expected, the likeliness of finding the object increases with a longer timeout, but for high user-densities short timeouts (5 to 10 minutes) are already sufficient. Moreover, high success rates can be obtained with $t_{\max}$=30 min even with



(a) Sensing range



(b) Campus mobility

Figure 4.13: Varying sensing range and mobility

user densities as low as 500 users/km^2. For lower densities longer timeouts must be used.

Observe that the graphs of Figure 4.12(a) measured in our office floor experiments (see Section 4.6.1), in which the actual user density was larger than 4000 users/km^2, are comparable to user densities of 500 to 200 users/km^2 in Figure 4.12(b). This is compatible with our earlier conjecture that the random waypoint simulation only models the "pedestrian" fraction out of the total users, and confirms that the approach to look at user densities which are smaller than in reality is valid.

A second important parameter, which is expected to have a large impact on the performance, is the sensing range of the employed object sensors. In the runs shown in Figure 4.13(a), we demonstrate the impact of

the sensing range on the success rate of position-based scoping. With 2000 users/km^2, even a sensing range of 1 m yields acceptable results. As expected, however, the sensing range has a high impact. When designing a practical system that shall be robust to small user densities, it seems worthwhile to invest in object sensing technology with a higher range.

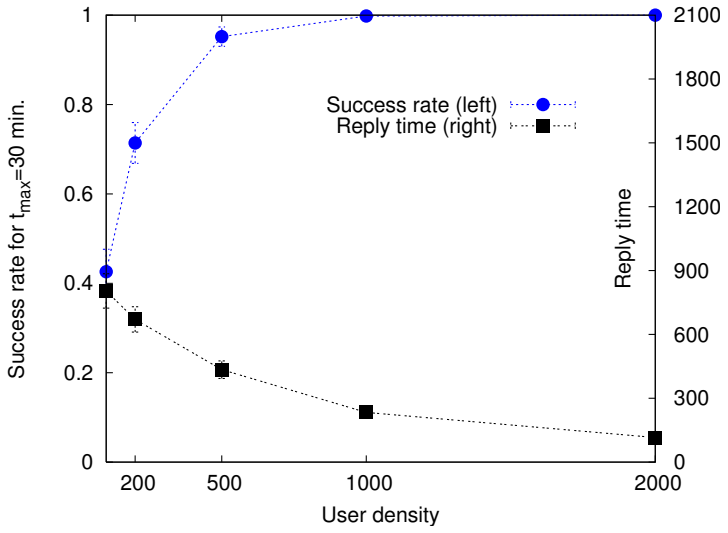
Finally, a third important parameter is the mobility of the system's participants. Here it is unclear whether the random waypoint model used is perhaps too optimistic. Figure 4.13(b) shows the success rate observed with the campus mobility model when raising the message limit $q_{\max}$. We show four graphs for the cases in which the object was left in a hotspot or in the cold region with two different user densities. Because user pause times in this model are quite long, we extended the query timeout $t_{\max}$ to 2 hours. Note, however, that the total number of queries remains limited to $q_{\max}$ and therefore the results remain comparable to earlier simulation runs shown in Figure 4.11(a). Here, for very small user densities, the success rate cannot be improved by raising $q_{\max}$ as the timeout remains the dominating constraint. For 500 users/km^2, however, the object can often be found with at most 200 messages even if it lies in the cold region.

Increasing the User Density

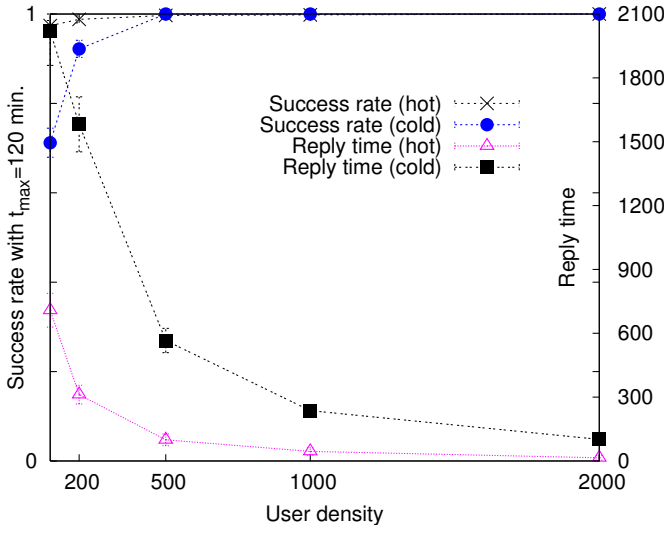
Additional insights can be gained when the user density is varied. Such experiments were performed with cell-based scoping and are shown in Figures 4.14 and 4.15. We show the success rate and the query reply time while varying the user density in Figure 4.14(a), and analogously the results for the campus mobility model in Figure 4.14(b). The corresponding overheads are shown in Figure 4.15(a) and 4.15(b), respectively. Note that for these runs no limit $q_{\max}$ is set.

Both overhead figures show the total overhead q_{total} and the overhead due to user mobility q_{mob} included in the total. Observe that q_{mob} does not increase with higher user densities. A reason for this is that query reply time decreases with increased user density and therefore compensates for the expected increase in the mobility-based overhead. Quite differently, q_{init} (equal to $q_{\text{total}} - q_{\text{mob}}$) increases proportionally to the user density as the number of queries is not limited by a certain $q_{\max}$.

The main result here is that once the success rate is good, an increased number of messages is "wasted" towards lowering the reply time. In other words: waiting for users to move is more efficient than simply querying more users. As a consequence, if a higher reply time were acceptable,

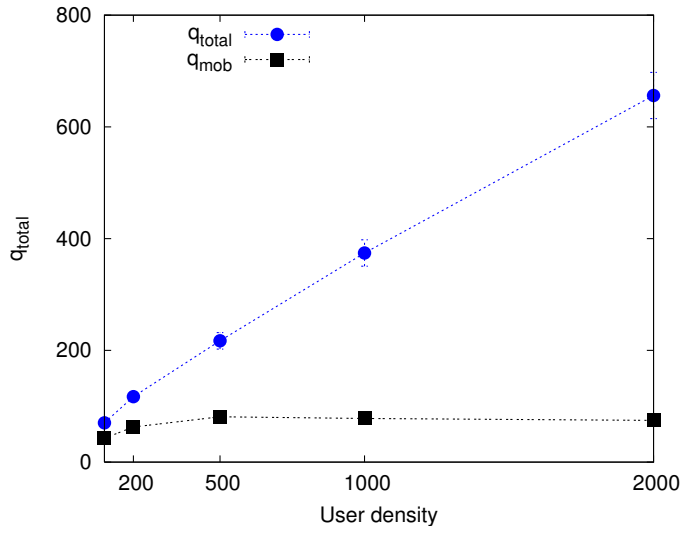


(a) Performance (random waypoint)

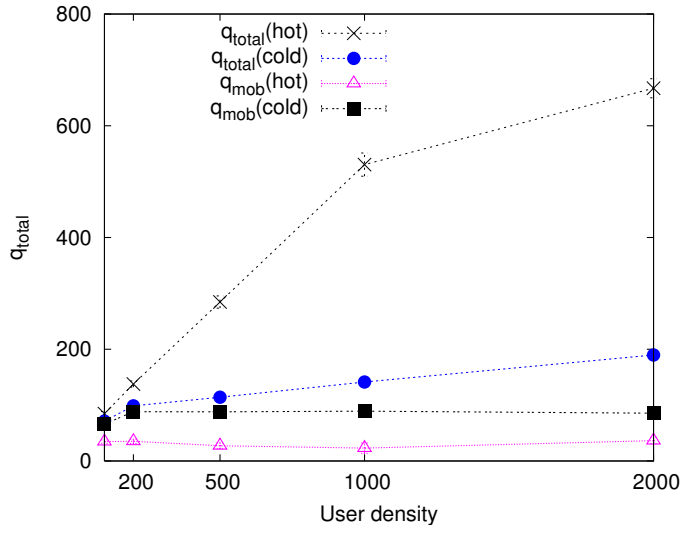


(b) Performance (campus)

Figure 4.14: Cell-based scoping with varying user density



(a) Overhead (random waypoint)



(b) Overhead (campus)

Figure 4.15: Overhead of the experiments shown in Figure 4.14

then the protocol can do with much less queries by choosing a smaller initial number of queries q_{init}.

Summing up, the above results are encouraging. In all our experiments, we could observe high rate of successful queries, that is, of objects being found. While the time until a reply can be obtained varies with user mobility and density, our conjecture – that most of the time an object will be found eventually – was confirmed. Moreover, we could show that even in settings with high positioning errors. or which rely solely on the observed cell identifier for localization, the total overhead for distributing an object search query remains acceptably low.

Finally, in certain circumstances the system may even work reasonably with very low participant densities which represent a hundredth of the expected daytime population in an urban area. Based on these results, we assume that a similarly designed object localization system can be implemented in practice.

4.7 Summary

In this chapter, we motivated and designed a comprehensive system for managing and finding everyday items using mobile phones. We discussed the architecture, design, and expected performance of this system, together with a flexible query scoping algorithm that generates object search heuristics from application data.

Our system makes use of an unconventional approach, as it relies on the participants' mobility in order to cover an essential portion of the search space. We therefore spent significant effort on modeling and testing the circumstances in which such an object search system would be used in the large.

A prominent challenge of the presented object localization system is to select a set of sensors that are likely to cover the phenomenon of interest, as a query cannot be distributed to all available sensors based on the scale of the system. This challenge was approached by a query scoping algorithm that exports a generic interface for its configuration. Based on the observation that every conceivable query scoping strategy would involve exploring history data stored by the system, the algorithm takes an annotated data model for an input, by which developers may parameterize algorithm to implement arbitrary query scoping heuristics.

While different in its motivation and the underlying network model, the approach repeats a pattern which re-alligns it with the previous chapters.

It uses a role-based configuration algorithm which exports a generic interface for setting up its behavior.

5 Conclusion

In the preceeding chapters, we have examined three rather heterogeneous approaches to wireless sensor network configuration. In this final chapter, we summarize the presented contributions and discuss their limitations as well as possible extensions.

5.1 Contributions

Each of the presented approaches allows programmers to express desired configuration behaviour in a generic way. We summarize the taken approaches in the following three sections.

5.1.1 Generic Role Assignment

Generic role assignment is a programmable service which supports developers with network configuration tasks. Based on properties of the nodes and their network neighborhood, developers can specify conditions for assigning roles to sensor nodes. We argued that many network configuration problems can be considered instances of generic role assignment as they can be expressed in terms of concise role assignment conditions.

Next to being expressive, the system is also efficient. In our evaluation, we showed that the communication costs incurred by generic role assignment are proportional to the "hardness" of the specified problem. Moreover, we discussed how the overhead involved in implementing role assignment problems is often comparable to specialized implementations of these problems.

To help developers assess the quality of role assignment conditions (so-called role specifications), we provided a verification tool which can be used to perform offline analysis of a given role assignment problem. In particular, this tool can be used to detect erroneous specifications which will not generate any useful configuration when deployed to a network in operation.

5.1.2 Facility Location

The second contribution zooms in on a large subclass of role assignment problems which is concerned with selecting nodes that provide a service to client nodes in their network neighborhood.

This subclass of problems can be modeled by the facility location problem. We therefore designed a distributed approximation algorithm for the facility location problem which can be practically implemented on wireless sensor nodes. Based on a set of parameters determining the cost involved in operating server nodes and the cost of communication between clients and their servers, the provided algorithm is able to find an optimal trade-off between these two kinds of costs involved in network operation.

Using experiments, we could show that the presented algorithm finds near-optimal solutions while depending only on local communication constrained to small network neighborhoods. Moreover, we found that a possible drawback of this algorithm, namely a high worst-case runtime, does rarely materialize in practice. In our experiments, the algorithm's runtime is small and constant, that is, independent of the network size.

Apart from being close to optimal in the average case, the computed solutions are guaranteed to be within a factor of 1.61 from the optimum, even in the worst-case. By integrating very good theoretical results (not even a centralized polynomial algorithm can provide an approximation factor better than 1.463) into a practicable local algorithm, the presented work can be considered to build a bridge between theory and practice.

5.1.3 Query Scoping

Our third contribution focused on a different subclass of configuration problems, which involves selecting the subset of sensor nodes which is most suitable for addressing a given sensing task.

The motivating application requires embedding sensors for detecting nearby objects into users' mobile phones and makes use of the resulting sensing infrastructure to locate lost or misplaced objects. In order to be efficient, this system selects an appropriate subset of all sensors for handling a given user query on a misplaced object. In particular, the sensor selection is performed offline at the network base station avoiding communicating with all sensors for each query.

The query scoping system has been custom tailored to its motivating application: It selects sensors based on their utility in locating everyday items which are of interest to users. In this context, we argued that the

system may flexibly generate a large range of sensor selection heuristics, based on arbitrary data that is stored by the application (such as previous reports on an object).

In our evaluation, we compared the performance of query scoping to random sensor selection strategies. Further, we evaluated the practicability of the mobile-phone-based object localization application itself. In particular, we examined various trade-offs involved in the design of user-centric sensing systems, and showed that adding sensors to only a small fraction of the participants' mobile phones already suffices for implementing the object localization application.

5.2 Limitations

As all three services summarized above provide generic interfaces for their parameterization, a set of common limitations can be associated to each of them. The first is related to the *expressiveness* of the service interface. For example, each service has been custom-tailored to a certain class of problems and does not allow for the implementation of functionality beyond this class. The second is related to the *efficiency* of the service implementation when compared with custom-built specialized solutions of similar problems. Last but not least, even if a service interface provides a high level of abstraction, developers must still perform a translation between the desired application logic and the provided service interface, to which we refer to as the interface's *semantic gap*.

In the following three sections, we discuss the limitations of each contribution according to the above categories, as well as specific limitations which can be associated to each contribution.

5.2.1 Generic Role Assignment

Clearly, generic role assignment cannot be used to formulate every conceivable role assignment problem. Instead, its *expressiveness* is limited by design. For example, generic role assignment is specialized to symmetric problems, where each node follows the same set of rules. Moreover, the system focuses on configuration tasks whose role assignment conditions are formulated in terms of rather stable network properties. While node properties may change over time, generic role assignment is only useful if (on average) one role assignment iteration can conclude and the

network is allowed to benefit from the computed configuration before the next property changes necessitating re-configuration take place.

Moreover, generic role assignment is limited to configurations in which greedy assignment of roles according to the specification makes sense, that is, which do not require to observe certain global optimization criteria. For example, while it has been possible to guarantee near-optimal solutions for the configuration tasks that can be modeled by the facility location problem, such optimality guarantees are hard to achieve for all problems specifiable by the generic role assignment system.

Further, we mentioned that the *efficiency* of the system is comparable to specialized implementations in many cases. However, one drawback is that nodes propagate properties occurring in the role specification to all nodes within a local network neighborhood. Specialized implementations, instead, may limit property propagation to nodes which benefit from the information.

Finally, generic role assignment's *semantic gap* is hard to assess adequately and only tentative statements can be made. Boolean role assignment conditions are established concepts, which can readily capture the developer's logic (consider the clustering example, where cluster leaders are connected by gateways which function as bridges between them). What might diverge from developers' intentions are the effects caused by unforeseen interdependencies between conditions for assigning different roles, which may in some cases result in non-terminating role assignment iterations. The role assignment solver, providing offline specification analysis, was built to overcome this concern.

A specific limitation of generic role assignment is that the speed of its convergence to a stable set of roles is an open issue – although this drawback is not apparent in our experiments. A formal framework that categorizes classes of role-assignment problems based on their convergence properties (as has been done in [Wol94] for a synchronous execution model and a constrained set of assignment rules) would be a valuable extension of our work. Because of the generality of the discussed role specifications, however, this remains a challenging open problem.

5.2.2 Distributed Facility Location

Analogous limitations exist for the provided facility location algorithms. Intuitively, their *expresiveness* is limited to configuration tasks that can be modeled by the facility location problem. In particular, we considered the

uncapacitated version of the problem, which does not set a limit on the number of clients per server node. Such limits, however, can be useful in many applications. A possible extension of our algorithm therefore is to address the *capacitated* facility location problem, as we describe in Section 5.3.2.

Moreover, our algorithms may suffer from drawbacks in *efficiency* when applied to problems which are "easier" than facility location. For example, the algorithms can be parameterized to compute a minimum dominating set. However, specialized approaches which are based on assigning roles probabilistically [Lub85] are able to provide low approximation factors involving fewer communication rounds.

Finally, the *semantic gap* of the facility location algorithms can be considered low, as the connection and opening cost parameters may well represent the energy spent by each of the two roles (client and server) within a given application.

A specific limitation of the described facility location algorithms is their high worst-case runtime. Attempts to speed up facility location algorithms, however, should not compromise the (average-case) optimality of the obtained solution, as this would diminish the practical benefit of the approach.

5.2.3 Query Scoping

Our query scoping approach provides a framework for determining the utility of sensors for answering user queries. While limited to the domain of sensor selection, the *expresiveness* of this framework is relatively high. Developers may issue arbitrary data models annotated with custom-implemented functions defining the relevance of the relations contained in the model.

Further, the *efficiency* of the approach has not been compared with specialized implementations. This is because highly customized search strategies are not considered useful in the context of the object localization application.

Finally, we mentioned above that the *semantic gap* can be viewed as relatively high compared to the previous two contributions. This is based on the fact that the semantics of a given set of weight functions (assigned to relations of the data model) become only apparent when considered together with the breadth-first search algorithm used for query scoping. Re-formulating the semantics of these functions (and of the relevance pa-

rameters assigned to edges of the search graph) to denote probabilistic relevance estimates (corresponding to the reciprocal of the currently used cost-based estimates), could overcome this drawback.

Moreover, because the taken approach is application specific, it might occur that a given selection problem can be captured more effectively by a different specification technique (other than by annotated data models), for example, by directly defining a utility function that returns the relevance of a sensor with respect to a given sensing task.

5.3 Outlook

There are a number of ways in which each of our contributions could be improved. In the following three sections, we discuss potential areas of future work. Further, in Section 5.3.4 we provide an overview of the core functionality addressed by this thesis, in order to provide guidelines for the design of a future, more comprehensive, network configuration service.

5.3.1 Generic Role Assignment

Generic role assignment could be extended in various ways. The first is to allow nodes to take on multiple roles instead of just one. Implementing this extension would require changing the system's semantics (currently assigning the first role of the specification whose assignment conditions match) to instead assign *all* roles whose conditions match. These changes would be particularly beneficial for applications, in which nodes perform different tasks concurrently. Note that such changed semantics could easily be added to the existing system: The node protocol would assign all matching roles; the centralized solver would simply omit some constraints.

A similar extension consists in maintaining the current role assignment semantics, but allow programmers to issue multiple role specifications which are all evaluated by the system at the same time. Given a set of specifications, the system would assign exactly one role from each of them, where each specification allows programmers to control a parallel set of node tasks. For example, nodes could adapt the behaviour of their routing components according to described clustering specification, while, at the same time, the mentioned coverage specification determines when to power up, or respectively shut down, the node's sensing function-

ality. Note that this extension can easily be implemented by re-executing the existing local rule evaluation engine a couple of times for each specified group of roles.

A third extension involves providing a more powerful local runtime environment on the nodes. We have mentioned how additional operations over local node properties (such a set intersection, substraction, or even custom-implemented library functions) enable a range of more advanced role specifications.

Finally, one could provide heuristics that attempt to realize certain global properties in the computed configurations. Most importantly, one could optimize the use of certain roles (as already supported by the centralized solver). Moreover, developers could be offered a feature, which (if enabled for a certain subset of roles) attempts to obtain configurations in which the specified subsets of roles form a connected backbone.

5.3.2 Distributed Facility Location

The presented work on facility location could be continued on several paths. The algorithms we described could be made faster, possibly employing a technique inspired by [MW05], in which stars are connected "fractionally" in small parallel steps and the obtained fractional solution is rounded later.

Moreover, in wireless multi-hop networks two "harder" versions of the facility location problem have particular applicability, for which, to our knowledge, no distributed algorithms exist at all.

The first is *capacitated* facility location, which allows developers to impose a limit c on the clients which can be connected to one server node. This (harder) problem definition may require a significantly revised algorithm if a low approximation factor is desired. However, a slight revision of the existing algorithms, in which nodes consider only the subset of stars containing less than c clients, could already provide a useful approach.

The second is *robust* facility location, which requires every client to be redundantly connected by k facilities. Here, again, our algorithms could already provide useful heuristics if client nodes are allowed to connect to more than one server (be part of k stars simultaneously) during algorithm execution.

Finally, there are other promising applications of distributed facility location, which can be implemented by different parameterizations of the provided facility location algorithms. For example, one could construct

whole data aggregation trees, by re-executing the described algorithms several times. In particular, one could parameterize consecutive executions of the algorithms to select aggregators of different levels (that is, to connect stars consisting of stars computed in the previous round) in each execution, and thus construct an efficient communication tree.

5.3.3 Query Scoping

The provided query scoping approach is application specific. One could provide an application-independent query scoping system by having nodes perform statistics on their output over time. Once these statistics are communicated to the network basestation, the basestation may direct a query for a certain event of interest to sensors which, based on previous statistics, are likely to observe such events.

5.3.4 A Comprehensive Configuration Framework

Apart from extensions to each individual service, an interesting path of future work is to provide a comprehensive comfiguration service that addresses functionality from all three contributions within a common framework. In this section, we are therefore interested in highlighting common concepts which re-appear in all our contributions, and can thus be expected to be part of any future service that is concerned with network configuration. In particular, we assume that a more advanced configuration service can approach all these concepts in a unified manner.

An aspect common to all our contributions on role-based configuration is that they implicitly provide functionality for *grouping* nodes. For example, generic role assignment forms groups in two distinct ways: On the one hand, nodes taking on the same role can be considered part of the same group. On the other hand, roles which require that a given set of nodes must be present in the local network neighborhood (for example an *aggregator* node which depends on a set of *data sources*) implicitly form a group consisting of the data sources and the chosen aggregator node. Facility location similarly associates nodes into clusters consisting of a server and a set of clients. Query scoping groups nodes based on their likelihood to sense certain information.

Based on these observations, and the fact that similar functionality is involved in many related services [WSBC04, WM04, MP06, SFCB04, HKS+04] and programming frameworks [GGG05, NW04, BPRL05], we

expect such composition and grouping of nodes to be part of any programming framework for wireless sensor networks. Therefore, in order to provide guidelines for the design of a future configuration system, it seems worthwhile to zoom in further on the different aspects involved in functionality for grouping nodes. Specifically, we found three such aspects.

Composition. The first aspect involves combining services or data (such as streams of some data type) from different nodes into more high-level services. For example, a cluster leader in a network performing fire detection could use its associated nodes to provide high-level reports on the current status of a large portion of the observation area. A different example is smart farming where two sources of humidity and temperature sensors from the same area (possibly not collocated on the same node) must be combined to evaluate the likeliness of infection with certain fungi [Ins]. Apart from composing streams of data, similarly, other services provided by network nodes (such as localization or time synchronization) could be composed into a more high level service. In summary, composition focuses on *connecting* services or data from different nodes into a more high-level bundle.

Exclusion. The second aspect involves granting nodes exclusive access to certain sensing or processing resources, or ensuring that certain tasks are processed by a limited number of nodes (from a network neighborhood) only. Coverage problems are based on exclusion, as they prevent more than one node from sensing the same area. In clustering approaches, typically a single cluster leader makes exclusive use of the data provided by associated slave nodes. The main purpose of exclusion aspects in wireless sensor networks is energy efficiency. It attempts to avoid redundant data gathering or computation within certain areas of the network, when the task can be performed sufficiently well by a single node or a small subset of nodes. Applications that require a group of nodes to reach consensus on a certain value often elect one group leader to choose the value for them, which is also a special case of exclusion functionality.

Selection. The third aspect involves selecting the most useful subset of nodes based on the required task. Selection functionality can be straightforward: The task may involve collecting data from a certain area (in which case sensors from this area are selected while the rest of the network remains inactive) or only from a given subject such as a certain building or bridge (in which only sensors located near the subject are rel-

evant for the sensing task). Alternatively, more involved methods for selecting nodes can be provided, for example, the sensor ranking performed by our query scoping approach. Note that selection can be performed *offline* at the network base station, as done by our query scoping system, as well as *online* inside of the network, where the nodes themselves compute whether they should be selected (as could be implemented by generic role assignment).

Note the orthogonality of the three aspects above. *Composition* wires the different sensors together into compound entities. *Exclusion* avoids that certain data or tasks are processed redundantly by multiple entities. *Selection* chooses the most useful entities for a given sensing task.

The expressiveness of generic role assignment can be explained by the fact that it integrates all three aspects. It implements *composition* by allowing to specify conditions requiring that nodes with certain properties (or roles) should be present in the network neighborhood. It also addresses *exclusion* as it allows to require that no other nodes with certain properties should be present. Finally, the role abstraction can be used as a means for *online selection* of nodes by enabling developers to include selection conditions in terms of local node node properties, such as a node's location, its current sensor readings, or its battery status, into role assignment conditions.

Facility location achieves specific intertwined *composition* and *exclusion* functionality involved in the clustering problem. While it composes groups of nodes into stars, at the same time it ensures that the star's clients are exclusively connected by the facility rooting the star (that is, that stars do not overlap and therefore information from clients is not processed redundantly). Finally, query scoping expands on the orthogonal selection problem, an aspect that is not present in the facility location system.

A future configuration service would need to integrate *composition*, *exclusion* and *selection* into a common programming framework. While generic role assignment has performed such integration, a future system might benefit from providing programmers with more explicit control over the above aspects and the interaction between them.

A promising candidate to address *composition* are graphs which state the desired composed structure (such as task graphs in ATaG [BPRL05]). Constraining these graphs to have the form of trees would allow for more efficient implementation of the configuration runtime and still capture most functionality of the *composition* aspect. Note that facility location addresses composition for graphs which have the form of stars.

Extensions regarding *exclusion* would allow developers to define additional entities (such as services, node groups or geographic areas) and constraints on the access to them by other entities. Note that – just like the accessed entities – the accessing entities do not need to be nodes.

Additional support for *selection* might include allowing developers to plug in functions that compute a ranking of nodes, either *online* based on certain local properties, such as the number of clients connected to a clusterhead or the nodes' battery level, or *offline* based on statistics on local properties which have been propagated to the network base station. Automated mechanisms for efficient online selection according to a given ranking function, and for the propagation of statistics to the network base station in order to perform offline selection based on the same function later on, may be another interesting direction to pursue in a future system.

5.4 Concluding Remarks

In this thesis, we approached role-based configuration problems from three distinct points of view which were founded on different application domains. Due to the wide range of applications which require role-based configuration, we believe that we touched upon core challenges which will manifest themselves in every network configuration problem. By facing these challenges in a variety of ways, we provide a more comprehensive understanding of potential solutions – by which we hope to facilitate the design of future systems concerned with the configuration of wireless sensor networks.

Bibliography

[AAC+00] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Jr. Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous Computing. *Communications of the ACM*, 43(5):74–82, March 2000.

[ABC+04] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. A. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, March 2004.

[ASSC02] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38(4):393–422, March 2002.

[Bar04] Rimon Barr. *An efficient, unifying approach to simulation using virtual machines*. PhD thesis, Cornell University, May 2004.

[BB03] Boris Jan Bonfils and Philippe Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks (IPSN'03)*, pages 47–62, April 2003.

[BBH+04] Gaetano Borriello, Waylon Brunette, Matthew Hall, Carl Hartung, and Cameron Tangney. Reminding about tagged objects using passive RFIDs. In *Proceedings of the 6th International Conference on Ubiquitous Computing (UbiComp'04)*, Nottingham, England, September 2004.

[BC02] Manish Bhardwaj and Anantha P. Chandrakasan. Bounding the lifetime of sensor networks via optimal role assignments. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'02)*, New York, NY, USA, 2002.

[BEH+06] J. Burke, D. Estrin, M. Hansen, A. Parker, N. Ramanathan, S. Reddy, and M. B. Srivastava. Participatory sensing. In *SENSYS'06 Workshop on World-Sensor-Web: Mobile Device Centric Sensor Networks and Applications (WSW'06)*, Boulder, CO, USA, November 2006.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, Hong Kong, China, January 2001.

[BIK+04] Cristian Borcea, Chalermek Intanagonwiwat, Porlin Kang, Ulrich Kremer, and Liviu Iftode. Spatial programming using smart messages: Design and implementation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 690–699, Tokyo, Japan, March 2004. IEEE Computer Society.

[BKG06] Fang Bian, David Kempe, and Ramesh Govindan. Utility based sensor selection. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks (IPSN'06)*, pages 11–18, Nashville, Tennessee, USA, April 2006. ACM Press.

[BKM+04] Jan Beutel, Oliver Kasten, Friedemann Mattern, Kay Römer, Lothar Thiele, and Frank Siegemund. Prototyping Sensor Network Applications with BTnodes. In *Proceedings of the 1st European Workshop on Wireless Sensor Networks (EWSN'04)*, Berlin, Germany, January 2004.

[BL06] Philipp Bolliger and Marc Langheinrich. Distributed persistence for limited devices. In *System Support for Ubiquitous Computing Workshop (UbiSys'06) at UbiComp'06*, Orange County, CA, USA, September 2006.

[BLM05] Philippe Bonnet, Martin Leopold, and Jan Madsen. The Hogthrob Probject. `http://www.hogthrob.dk/`, 2005.

[BMP04] Stefano Basagni, Michele Mastrogiovanni, and Chiara Petrioli. A performance comparison of protocols for clustering and backbone formation in large scale ad hoc networks. In *Proceedings of the 1st IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS'04)*, 2004.

[Bok81] Shahid H. Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Trans. on Software Engineering*, 7(6):583–589, November 1981.

[Bok87] Shahid H. Bokhari. *Assignment problems in parallel and distributed computing*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International World Wide Web Conference (WWW'98)*, pages 107–117, Brisbane, Australia, April 1998. Elsevier Science Publishers B. V.

[BPRL05] Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The Abstract Task Graph: a methodology for architecture-independent programming of networked sensor systems. In *Proceedings of the 2005 workshop on end-to-end, sense-and-respond systems, applications and services (EESR'05)*, pages 19–24, Seattle, WA, USA, 2005. USENIX Association.

[BS03] Athanassios Boulis and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MOBISYS'03)*, San Francisco, CA, USA, May 2003.

[BTn06] BTnodes. `www.btnode.ethz.ch`, 2006.

[CC04] SmartRF CC1000 Datasheet (rev. 2.2). Chipcon AS, April 2004. www.chipcon.com/files/CC1000_Data_Sheet_2_2.pdf.

[CEPS05] Fabian Chudak, Thomas Erlebach, Alessandro Panconesi, and Mauro Sozio. Primal-dual distributed algorithms for covering and facility location problems. Unpublished Manuscript, 2005.

[CGG+05] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. Mobile data collection in sensor networks: The TINYLIME Middleware. *Elsevier Pervasive and Mobile Computing Journal*, 4(1):446–469, December 2005.

[CHP+04] P. Corke, S. Hrabar, R. Peterson, D. Rus, S. Saripalli, and G. Sukhatme. Autonomous deployment and repair of a sensor network using an unmanned aerial vehicle. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA'04)*, pages 3602–3608, May 2004.

[Cle96] Scott H. Clearwater, editor. *Market-based control: a paradigm for distributed resource allocation*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[CPX] CPLEX ILP solver. www.ilog.com/products/cplex/.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th IEEE International Conference on Local Computer Networks (LCN'04)*, Washington, DC, USA, 2004.

[DKB03] Christian Decker, Uwe Kubach, and Michael Beigl. Revealing the retail black box by interaction sensing. In *3rd International Workshop on Smart Appliances and Wearable Computing (SAWC'03) at ICDCS'03*, Providence, RI, USA, May 2003.

[EAL+06] Shane B. Eisenman, Gahng-Seop Ahn, Nicholas D. Lane, Emiliano Miluzzo, Ronald A. Peterson, and Andrew T. Campbell. MetroSense project: People-centric sensing at scale. In *SENSYS'06 Workshop on World-Sensor-Web:*

Mobile Device Centric Sensor Networks and Applications (WSW'06), Boulder, CO, USA, November 2006.

[EGT05] Andreas Eisenblätter, Hans-Florian Geerdes, and Ulrich Türke. Public UMTS radio network evaluation and planning scenarios. *International Journal on Mobile Network Design and Innovation*, 1(1):40–53, 2005.

[EHJ04] Stephan J. Engberg, Morten B. Harning, and Christian D. Jensen. Zero-knowledge device authentication: Privacy & security enhanced RFID preserving business value and consumer convenience. In *Proceedings of the 2nd Annual Conference on Privacy, Security and Trust (PST'04)*, October 2004.

[ETS98] ETSI. Selection procedures for the choice of radio transmission technologies of the UMTS. Technical Report 3.2.0, European Telecommunications Standards Institute, April 1998.

[FBMK07] Christian Frank, Philipp Bolliger, Friedemann Mattern, and Wolfgang Kellerer. The Sensor Internet at work: Locating objects using mobile phones. Submitted for publication, 2007.

[FBRK07] Christian Frank, Philipp Bolliger, Christof Roduner, and Wolfgang Kellerer. Objects calling home: Locating objects using mobile phones. In *Proceedings of the 5th International Conference on Pervasive Computing (Pervasive'07)*, Toronto, ON, Canada, May 2007.

[FCX+03] Lucio Ferreira, Luis M. Correia, David Xavier, Allen Vasconcelos, and Erik Fledderus. Deliverable d1.4: Final report on traffic estimation and services characterisation. Technical Report IST-2000-28088, Momentum Project, 2003.

[Fei98] Uriel Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45(4), 1998.

[FR05] Christian Frank and Kay Römer. Algorithms for generic role assignment in wireless sensor networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SENSYS'05)*, San Diego, CA, USA, November 2005.

[FR06] Christian Frank and Kay Römer. Solving generic role assignment exactly. In *Proceedings of the 14th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'06)*, Island of Rhodes, Greece, April 2006.

[FR07] Christian Frank and Kay Römer. Distributed facility location algorithms for flexible configuration of wireless sensor networks. In *Proceedings of the 3rd International Conference on Distributed Computing in Sensor Systems (DCOSS'07)*, Santa Fe, NM, USA, June 2007.

[Fra07] Christian Frank. Facility location. In Dorothea Wagner and Roger Wattenhofer, editors, *Algorithms for Sensor and Ad Hoc Networks*. Springer-Verlag, 2007.

[FRB+07] Christian Frank, Christof Roduner, Philipp Bolliger, Chie Noda, and Wolfgang Kellerer. A service architecture for monitoring physical objects using mobile phones. In *Proceedings of the 7th International Workshop on Applications and Services in Wireless Networks (ASWN'07)*, Santander, Spain, May 2007.

[FRNK06] Christian Frank, Christof Roduner, Chie Noda, and Wolfgang Kellerer. Query scoping for the Sensor Internet. In *Proceedings of the IEEE International Conference on Pervasive Services (ICPS'06)*, Lyon, France, June 2006.

[GEG+03] Benjamin Greenstein, Deborah Estrin, Ramesh Govindan, Sylvia Ratnasamy, and Scott Shenker. DIFS: a distributed index for features in sensor networks. In *Proceedings of the 1st IEEE International Workshop on Sensor Network Protocols and Applications (SNPA'03)*, pages 163–173, Anchorage, AK, USA, May 2003.

[GGG05] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using Kairos. In *Proceedings of the 1st International Conference on Distributed Computing in Sensor Systems (DCOSS'05)*, Marina del Rey, CA, USA, June 2005.

[GGJ+06] Omprakash Gnawali, Ben Greenstein, Ki-Young Jang, August Joki, Jeongyeup Paek, Marcos Vieira, Deborah Estrin,

Ramesh Govindan, and Eddie Kohler. The TENET architecture for tiered sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SENSYS'06)*, Boulder, CO, USA, November 2006.

[GK99] Sudipto Guha and Samir Khuller. Greedy strikes back: Improved facility location algorithms. *Journal of Algorithms*, 31:228–248, 1999.

[GKK+03] Phillip B. Gibbons, Brad Karp, Yan Ke, Suman Nath, and Srinivasan Seshan. IrisNet: An architecture for a worldwide sensor web. *IEEE Pervasive Computing*, 2(4), 2003.

[GLS06] Joachim Gehweiler, Christiane Lammersen, and Christian Sohler. A distributed $O(1)$-approximation algorithm for the uniform facility location problem. In *Proceedings of the 18th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'06)*, Cambridge, MA, USA, 2006.

[GZDG05] Himanshu Gupta, Zongheng Zhou, Samir R. Das, and Quinyi Gu. Connected sensor cover: Self-organization of sensor networks for efficient query execution. *ACM/IEEE Transactions on Networking*, 2005.

[HCB00] Wendi R. Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS'00)*, 2000.

[HKS+04] Tian He, Suda Krishnamurthy, John A. Stankovic, Tarek F. Abdelzaher, Liqian Luo, Radu Stoleru, Ting Yan, Lin Gu, Jonathan Hui, and Bruce Krogh. An energy-efficient surveillance system using wireless sensor networks. In *In Proceedings of Second International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2004.

[HKS+05] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani B. Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MOBISYS'05)*, pages 163–176, Seattle, WA, USA, June 2005.

[HMCP04] Wendi R. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo. Middleware to support sensor network applications. *IEEE Network*, pages 6–14, January 2004.

[Hoc82] Dorit S. Hochbaum. Heuristics for the fixed cost median problem. *Mathematical Programming*, 22(1):148–162, 1982.

[IB05] Volkan Isler and Ruzena Bajcsy. The sensor selection problem for bounded uncertainty sensing models. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN'05)*, Los Angeles, CA, USA, April 2005.

[IEGH02] Chalermek Intanagonwiwat, Deborah Estrin, Ramesh Govindan, and John Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2002. IEEE Computer Society.

[IGE00] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MOBICOM'00)*, Boston, MA, USA, August 2000. ACM Press.

[Ins] Pessl Instruments. Potato and onion disease models. `www.metos.at/diseasemodels/Potato.pdf`.

[JMM+03] Kamal Jain, Mohammad Mahdian, Evangelos Markakis, Amin Saberi, and Vijai V. Vazirani. Greedy facility location algorithms analyzed using dual fitting with factor-revealing LP. *Journal of the ACM*, 50:795–824, November 2003.

[JOW+02] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. *ACM SIGPLAN Notices*, 37(10):96–107, 2002.

[JV99] Kamal Jain and Vijai V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pages 2–13, October 1999.

[KG02] Taek Jin Kwon and Mario Gerla. Efficient flooding with passive clustering (PC) in ad hoc networks. *Computer Communication Review*, 32(1):44–56, January 2002.

[KHHK04] Andreas Köpke, Vlado Handziski, Jan-Hinrich Hauer, and Holger Karl. Structuring the information flow in component-based protocol implementations for wireless sensor nodes. In *Proceedings of the Work-in-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN'04)*, pages 41–45, Berlin, Germany, January 2004.

[KKK06] Minkyong Kim, David Kotz, and Songkuk Kim. Extracting a mobility model from real user traces. In *Proceedings of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'06)*, Barcelona, Spain, April 2006.

[Koc05] Thorsten Koch. ZIMPL *(Zuse Institute Mathematical Programming Language) User Guide*, February 2005.

[KR05] Oliver Kasten and Kay Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN'05)*, pages 45–52, Los Angeles, USA, April 2005.

[KSG03] Manish Kochhal, Loren Schwiebert, and Sandeep Gupta. Role-based hierarchical self organization for wireless ad hoc sensor networks. In *Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'03)*, San Diego, CA, USA, 2003.

[KSW05] Denis Krivitski, Assaf Schuster, and Ran Wolff. A local facility location algorithm for sensor networks. In *Proceedings of the 1st International Conference on Distributed Computing in Sensor Systems (DCOSS'05)*, Marina del Rey, CA, USA, June 2005.

[KV02] Bernhard Korte and Jens Vygen. *Colouring*, chapter 16.2, pages 367–373. Springer-Verlag, 2nd edition, 2002.

[KW03] Fabian Kuhn and Roger Wattenhofer. Constant-time distributed dominating set approximation. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC'03)*, July 2003.

[KW05] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. Wiley, May 2005.

[LAHS06] Liqian Luo, Tarek F. Abdelzaher, Tian He, and John A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems*, 5(3):543–576, 2006.

[LC02] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. *Operating Systems Review*, 36(5):85 – 95, December 2002.

[LMFJ+04] Konrad Lorincz, David J. Malan, Thaddeus R.F. Fulford-Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoffrey Mainland, and Matt Welsh. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing*, 3(4):16–23, October 2004.

[LMMR05] Andreas Lachenmann, Pedro José Marrón, Daniel Minder, and Kurt Rothermel. An analysis of cross-layer interactions in sensor network applications. In *Proceedings of the 2nd International Conference on Intelligent Sensors, Sensor Networks & Information Processing (ISSNIP'05)*, pages 121–126, December 2005.

[LPCS04] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the First Symposium on Networked System Design and Implementation (NSDI'04)*, San Francisco, CA, USA, March 2004.

[LRT04] Kari Laasonen, Mika Raento, and Hannu Toivonen. Adaptive on-device location recognition. In *Proceedings of the*

2nd International Conference on Pervasive Computing (Pervasive'04), Vienna, Austria, April 2004.

[LRW+05] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin Gun Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MOBISYS'05)*, Seattle, WA, USA, June 2005.

[LSS03] Shuoqi Li, Sang H. Son, and John A. Stankovic. Event detection services using data service middleware in distributed sensor networks. In *Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks (IPSN'03)*, volume 2634 of *Lecture Notes in Computer Science*, pages 502 – 517, Palo Alto, CA, USA, April 2003. Springer-Verlag.

[LSZM04] Ting Liu, Christopher M. Sadler, Pei Zhang, and Margaret Martonosi. Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MOBISYS'04)*, Boston, MA, USA, June 2004. ACM Press.

[Lub85] Michael Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC'85)*, pages 1–10, New York, NY, USA, May 1985. ACM Press.

[LV05] Jean-Yves Le Boudec and Milan Vojnović. Perfect simulation and stationarity of a class of mobility models. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*, Miami, FL, USA, March 2005.

[Lyn81] Nancy Lynch. Upper bounds for static resource allocation in a distributed system. *Journal of Computer and System Sciences*, 23(2):254–278, 1981.

[MFHH02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation Service for

Ad-Hoc Sensor Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, USA, December 2002.

[MFHH03] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 22th International Conference on Management of Data / Principles of Database Systems (SIGMOD/PODS'03)*, pages 491–502, San Diego, CA, USA, June 2003.

[MFHH05] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.

[MKPS01] Seapahn Meguerdichian, Farinaz Koushanfar, Miodrag Potkonjak, and Mani B. Srivastava. Coverage problems in wireless ad-hoc sensor networks. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'01)*, volume 3, pages 1380–1387, 2001.

[MLM+05] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN'05)*, January 2005.

[Mom06] Momentum. Models and simulation for network planning and control of UMTS. `momentum.zib.de/data.php`, May 2006.

[Mos07] Thomas Moscibroda. Clustering. In Dorothea Wagner and Roger Wattenhofer, editors, *Algorithms for Sensor and Ad Hoc Networks*. Springer-Verlag, 2007.

[MP06] Luca Mottola and Gian Pietro Picco. Logical Neighborhoods: A programming abstraction for wireless sensor networks. In *Proceedings of the 2nd International Conference on Distributed Computing in Sensor Systems (DCOSS'06)*, pages 150–167, San Francisco, CA, USA, June 2006.

[MPS+02] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, USA, September 2002.

[MPW05] Geoffrey Mainland, David C. Parkes, and Matt Welsh. Decentralized, adaptive resource allocation for sensor networks. In *Proceedings of the Second Symposium on Networked System Design and Implementation (NSDI'05)*, Boston, MA, USA, May 2005.

[MW05] Thomas Moscibroda and Roger Wattenhofer. Facility location: Distributed approximation. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing (PODC'05)*, pages 108–117, 2005.

[MYZ02] Mohammad Mahdian, Yinyu Ye, and Jiawei Zhang. Improved approximation algorithms for metric facility location problems. In *Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX'02)*, Rome, Italy, September 2002.

[NAW05] Ryan Newton, Arvind, and Matt Welsh. Building up to macroprogramming: an intermediate language for sensor networks. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN'05)*, Los Angeles, CA, USA, 2005.

[NMW07] Ryan Newton, Greg Morrisett, and Matt Welsh. The Regiment macroprogramming system. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN'07)*, pages 489–498, Cambridge, MA, USA, 2007.

[NW04] Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In *Proceeedings of the 1st International Workshop on Data Management for Sensor Networks (DMSN'04)*, pages 78–87, Toronto, ON, Canada, 2004.

[PHC04] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *Pro-*

ceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SENSYS'04), pages 95–107, Baltimore, MD, USA, 2004.

[PLS+06] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, Atlanta, GA, USA, April 2006.

[Ram99] Subramanian Ramanathan. A unified framework and algorithm for channel assignment in wireless networks. *Wireless Networks*, 5(2):81–94, 1999.

[RF 06] RF Code, Inc. Mantis™ active RFID tags 433 MHz data sheet. `www.rfcode.com/data_sheets/433_mantis_tags.pdf`, 2006.

[RFMB04] Kay Römer, Christian Frank, Pedro José Marrón, and Christian Becker. Generic role assignment for wireless sensor networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A modern approach*, chapter 3, pages 75–76. Prentice Hall, 1995.

[SAGP00] Katayoun Sohrabi, Vishal Ailawadhi, Jay Gao, and Gregory J. Pottie. Protocols for self organization of a wireless sensor network. *Personal Communication Magazine*, 7:16–27, 2000.

[San05] Paolo Santi. *Topology Control in Wireless Ad Hoc and Sensor Networks*. Wiley, 2005.

[Sen07] SensorPlanet. `www.sensorplanet.org`, 2007.

[SFCB04] Jan Steffan, Ludger Fiege, Mariano Cilia, and Alejandro Buchmann. Scoping in wireless sensor networks: a position paper. In *Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing*, pages 167–171, Toronto, Ontario, Canada, 2004. ACM Press.

[SHK+05] Hirobumi Shimizu, Osamu Hanzawa, Kenichiro Kanehana, Hiroki Saito, Niwat Thepvilojanapong, Kaoru Sezaki, and Yoshito Tobe. Association management between everyday objects and personal devices for passengers in urban areas. Demonstration Abstract in Adjunct Proceedings of Pervasive'05, Munich, Germany, May 2005.

[SK00] Lakshminarayanan Subramanian and Randy H. Katz. An architecture for building self-configurable systems. In *Proceedings of the 1st ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC'01)*, Boston, MA, USA, August 2000.

[SP01] Sasa Slijepcevic and Miodrag Potkonjak. Power efficient organization of wireless sensor networks. In *Proceedings of the IEEE International Conference on Communications (ICC'01)*, Helsinki, Finland, June 2001.

[Sto77] Harold S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):83–93, Jan 1977.

[TG03] Di Tian and Nicolas D. Georganas. A node scheduling scheme for energy conservation in large wireless sensor networks. *Wireless Communications and Mobile Computing*, 3(2):271–290, 2003.

[Tmo05] Tmote Sky datasheet 1.0.2. `www.moteiv.com`, December 2005.

[TOS] TinyOS. webs.cs.berkeley.edu/tos/.

[TP05] Dirk Trossen and Dana Pavel. Building a ubiquitous platform for remote sensing using smartphones. In *Proceedings of the 2nd Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services (MobiQuitous'05)*, pages 485–489, July 2005.

[TPS+05] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. In *Proceedings of the 3rd Interna-*

tional Conference on Embedded Networked Sensor Systems (SENSYS'05), San Diego, CA, USA, November 2005.

[TWS06] Kirsten Terfloth, Georg Wittenburg, and Jochen H. Schiller. FACTS - a rule-based middleware architecture for wireless sensor networks. In *Proceedings of the 1st International Conference on Communication Systems Software and Middleware (COMSWARE'06)*, New Delhi, India, 2006.

[UWMG05] Andreas Ulbrich, Torben Weis, Gero Mühl, and Kurt Geihs. Application Development for Actuator and Sensor Networks. In *GI Workshop on Sensor Networks*, ETH Zurich, Switzerland, March 2005.

[Vaz03] Vijai V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2003.

[vDL03] Tijs van Dam and Koen Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SENSYS'03)*, Los Angeles, CA, USA, November 2003.

[Vyg05] Jens Vygen. Approximation algorithms for facility location problems. Technical Report 05950-OR, Research Institute for Discrete Mathematics, University of Bonn, 2005.

[WAJR+05] Geoffrey Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN'05)*, Istanbul, Turkey, January 2005.

[WC01] Alec Woo and David Culler. A transmission control scheme for media access in sensor networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MOBICOM'01)*, pages 221–235, Rome, Italy, 2001.

[WFGH99] Roy Want, Kenneth P. Fishkin, Anuj Gujar, and Beverly L. Harrison. Bridging Physical and Virtual Worlds with Electronic Tags. In *Proceedings of the ACM SIGCHI Conference*

on Human Factors in Computing Systems (CHI'99), pages 370–377, Pittsburgh, PA, USA, May 1999.

[Wib06] Wibree Technology. `www.wibree.com`, 2006.

[WM04] Matt Welsh and Geoffrey Mainland. Programming sensor networks using abstract regions. In *Proceedings of the First Symposium on Networked System Design and Implementation (NSDI'04)*, San Francisco, CA, USA, March 2004.

[Wol94] Stephen Wolfram. *Cellular Automata and Complexity*. Addison-Wesley, 1994.

[WSBC04] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: A neighborhood abstraction for sensor networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MOBISYS'04)*, Boston, MA, USA, June 2004.

[WTC03] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SENSYS'03)*, Los Angeles, CA, USA, November 2003.

[WZ04] Roger Wattenhofer and Aaron Zollinger. XTC: A practical topology control algorithm for ad-hoc networks. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, NM, USA, April 2004.

[XHE01] Ya Xu, John Heidemann, and Deborah Estrin. Geography-informed energy conservation for ad-hoc routing. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MOBICOM'01)*, Rome, Italy, July 2001.

[XRC+04] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A wireless sensor network for structural monitoring. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SENSYS'04)*, Baltimore, MD, USA, November 2004.

[YHE02] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'02)*, New York, NY, USA, June 2002.

[YSM05] Kok Kiong Yap, Vikram Srinivasan, and Mehul Motani. MAX: Human-centric search of the physical world. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SENSYS'05)*, San Diego, CA, USA, November 2005.

Curriculum Vitae

Christian Frank

Personal Data

Date of Birth	November 3rd, 1978
Birthplace	Timişioara, Romania
Citizenship	Germany

Education

1989 – 1998	*Canisius-Kolleg (Gymnasium)*, Grammar School, Berlin, Germany
June 12th, 1998	Abitur
1998 – 1999	Study of Computer Science at the *Humboldt Universität*, Berlin, Germany
1999 – 2000	Study of Business Administration at the *European Business School*, Oestrich-Winkel, Germany
2000 – 2003	Study of Computer Science at the *Technische Universität Berlin*, Germany
August 15th, 2003	Master of Science (Diplom) in Computer Science
2004 – 2007	Ph.D. Student at the Department of Computer Science, *ETH Zurich*, Switzerland