Department of Electrical Engineering and Information Technology
Distributed Multimodal Information Processing Group
Prof. Dr. Matthias Kranz

# Distributed Networks Using ROS – Cross-Network Middleware Communication using IPv6

## Verteilte Netze mit ROS – Netzübergreifende Middleware Kommunikation mit IPv6

**Tobias Schneider**
Diploma Thesis

| | |
|---|---|
| Author: | Tobias Schneider |
| Address: | |
| | |
| Matriculation Number: | |
| Professor: | Prof. Dr. Matthias Kranz |
| Advisor: | Dipl.-Ing. Luis Roalter |
| Begin: | 01.04.2012 |
| End: | 31.10.2012 |

Department of Electrical Engineering and Information Technology
Distributed Multimodal Information Processing Group
Prof. Dr. Matthias Kranz

# Declaration

I declare under penalty of perjury that I wrote this Diploma Thesis entitled

**Distributed Networks Using ROS – Cross-Network Middleware Communication using IPv6**

**Verteilte Netze mit ROS – Netzübergreifende Middleware Kommunikation mit IPv6**

by myself and that I used no other than the specified sources and tools.

Munich, November 21, 2012

Tobias Schneider

Tobias Schneider

# Kurzfassung

Aktuelle Verbesserungen in Rechenleistung, Energieaufnahme und Kosten von kleinen elektronischen Rechnern machen es möglich, mächtige und ausdauernde (drahtlose) Sensor- und Aktuatorsysteme zu kreieren. In Zukunft werden diese System ein Teil unseres täglichen Lebens werden, so wie das elektrische Licht zu Beginn des zwanzigsten Jahrhunderts ein Teil unseres Lebens wurde. Diese Netze aus aus Sensoren und Aktuatoren werden verwendet werden um intelligente Umgebungen zu schaffen, welche Daten verarbeiten, die von Sensoren gesammelt werden, um unser Leben in einer uns umgebebenden intelligenten Umgebung zu verbesseren.

Um diese Entwicklung zu unterstützen müssen neue Wege gefunden werden um riesige Mengen von Geräten in unsere aktuellen und zukünftigen Netzwerke zu integrieren. Das Internet ist eine Erfolgsgeschichte von offenden Standards und Modularität. Diese Konzepte können auch auf dieses aufstrebende Internet der Dinge angewendet werden. Das Internet vollzieht momentan einen Wandel zu einem neuen Internetprotokoll um selbst die größten Netzwerke zu unterstützen. Diese Technologie macht es möglich, die riesige Anzahl von zukünftigen Knoten in drahtlosen Sensornetzwerken, dem Internet der Dinge, Smartphones und der Dienstleistungsrobotik miteinander zu verbinden.

Um die Entwicklung von Anwendungen für solch massive Menge an Daten und Knoten zu unterstützen, müssen Abstraktionen gemacht werden. Durch die Bereitstellung einer Zwischenschicht zwischen den Anwendungen und den Geräten kann die Entwicklungszeit verkürzt und Mechanismen eingebaut werden, die die Privatsphäre von Menschen schützen, welche mit immer mehr Daten sammelnden Geräten leben werden.

Diese Arbeit verwendet das Robot Operating System, eine Zwischenschicht aus der Robotik, und führt Konzepte ein, um diese Zwischenschicht besser auf die Anforderungen von intelligenten Umgebungen anzupassen. Dies beinhaltet die Einführung des neuen IPv6 Internetprotokolls, Konzepten um die Privatsphäre der Nutzer zu wahren und der Bereitstellung einer skalierbaren Basis, um selbst die kleinsten Geräte aus drahtlosen Sensornetzwerken mit den rechenstarken Computern von Servicerobotern zu verbinden.

## Abstract

Recent advancements in the computing power, energy consumption and cost of small electronic computing devices make it possible to create powerful and long lasting (wireless) sensor and actuator systems. In the future, these systems will become part of our daily lives as electric light has become part of our live at the beginning of the 20th century. These networks consisting of sensors and actuators can be used to form Intelligent Environments, which will process data gathered by sensors to enhance our lives with services provided by an intelligent environment, surrounding us.

To support this development, new ways have to be found to integrate huge numbers of devices with our current and future networks. The Internet is a success story of open standards and modularity, these concepts can also be applied to this emerging Internet of Things. The Internet is currently performing a transition to a new Internet Protocol, to support even the largest networks. This technology makes it possible to bridge the huge number of future nodes in Wireless Sensor Networks, the Internet of Things, smartphones and service robotics together.

To support the development of applications for such a massive amount of nodes and data, abstractions have to be made. By providing a middleware between the applications and the devices, development can be sped up and mechanisms put in place, to preserve the privacy of people living with ever more devices gathering data about their lives.

This thesis takes the Robot Operating System, a middleware from the robotics community, and introduces concepts to make the middleware more suitable for Intelligent Environments. This includes the adoption of the new IPv6 Internet Protocol, concepts to secure the privacy of its users and providing a scalable base to interconnect even the smallest devices from wireless sensor networks with powerful computers like service robots.

# Contents

# Chapter 1.

# Introduction

Increasing computing power and decreasing energy consumption of integrated circuits has been a corner stone of the computer revolution at the end of the last century [1]. This trend is continuing, providing us with ever more efficient, smaller and cheaper sensors and communication devices [2]. This allows us to embed small computers into a variety of objects and environments, giving us the option to create a completely new and revolutionary way of interacting with computer systems. The line where computer interfaces end, and the physical world begins gets blurred [3]. Tangible User Interfaces [4] are created, where humans stop to use mouse, keyboard and screen to interact with the computer and instead use touchable and manipulable input-services, specific for the task they are trying to accomplish. When carefully designed, these systems seem to vanish and don't get recognized as computer systems any more. This was foreseen by Weiser in 1991 [5]. He called this new area of computing 'ubiquitous computing'. In ubiquitous computing the dedicated human computer interface disappears. It gets fully integrated into everyday objects and its users might not even realize anymore, that they are interacting with a computer. Just as we are not thinking about the electrical connections to the lamp anymore, when using a light switch.

This vision is complemented by the dawn of service robotics. Robotic systems are reaching a price and maturity level that makes them attractive as personal assistants in homes, public spaces and health care [6].

Ubiquitous computing has an impact on a personal as well as social dimension [7]. It can assist us in our own personal life by taking care of usually repetitive interaction with our current technology or by helping us remember important things. It can also assist us on a social level: Ubiquitous computing will be used in smart cities like Songdo, South Korea[1] in an effort to reduce the amount of used water by 30 percent and the amount of left over waste by 75 percent [7].

But the possibilities of ubiquitous computing come with a price. Through the increasing collection of data about our personal life and our societies, a completely new quality of surveillance is made possible. Even if the collected data is not used for any act of surveillance (by peers, the operator

[1] http://voices.mckinseyonsociety.com/at-home-in-the-aerotropolis/

of the system or even the state), the users of the system can not be certain of that fact. This alone can lead to a change in behavior of the users of the system. We, as the designers and implementers of this vision have to be careful not to put systems into effect that violate the individual right to privacy. Only through a very careful design and the trust of the users can ubiquitous computing be accepted by the public.

The next sections will cover a few of the very basic building blocks of ubiquitous computing.

## 1.1. The Internet of Things

The Internet of Things (IoT) is a term that has emerged over the last decade [8]. It describes the idea of connecting the physical world, made of things, with the global Internet, consisting of data and abstract concepts. Using standard Internet protocols, this makes it possible to achieve compatibility between different types of objects and technologies. These technologies can include barcodes, RFID tags, wireless networks and many other communication techniques.

While objects from the IoT can be used to sense environmental factors of our life and to control existing equipment, they can also be used to enhance exiting objects and sense properties of specific objects in our life. An example is the intelligent coffee mug [9]. The sensors attached to the mug can sense multiple properties of the mug like temperature or movement. Through these sensors it can also infer information about the user, handling the mug. By the means of standardization and the decreasing price of technology, these enhancements can become the norm for newly designed objects.

The usage of standard Internet protocols makes it possible to combine different systems and products. The online platform Cosm[2] provides an example, how this interoperability can be exploited. The website provides users with the service of collecting and visualizing data from their sensors. It can be used with a variety of different sensors and relies mainly on the use of the HTTP protocol to transport the sensor data. It can be configured to pull the data actively from the sensors or get them pushed when the sensor decides to send an update. The great variety of supported hardware and direct access to the sensors has been made possible by the direct connection of the sensors to the Internet. Similarly, in [10] a system is show which closes the loop between different devices of the Internet of Things via the web based micro blogging service Twitter. It allows users to observe the interactions of the different parts of the system and follow their state.

[2]Cosm, `https://www.cosm.com`

## 1.2. Wireless Sensor Networks

Wireless Sensor Network are formed by devices connected via radio links. These devices are usually called nodes. This most of them are interconnected to each other and even need each other to communicate to the rest of the Internet. They usually form a so-called mesh-network which relies on the ability of all nodes to receive and send data from and to other nodes. A node is usually comprised of a low-power processing unit, a low-power and low-data-rate radio interface and a small battery. Conserving energy and radio bandwidth is of central importance for this kind of devices. Some are even powered by harvesting energy from their surroundings (i.e. light, temperature, vibration, movement). These nodes have even stricter requirements on the energy consumption [11].

Modern Wireless Sensor Networks are based on standardized protocols. Most notably the Internet Protocol(IP). It forms the basis of virtually all communication over the Internet today. By adopting the Internet Protocol, the Wireless Sensor Networks can profit from a variety of already developed solutions for the Internet and reach a lot of existing infrastructure. Standard protocols make it easier to access nodes from existing networks and interact with it using common tools or even a web browser. Current operating systems for Wireless Sensor Networks include Contiki [12] and TinyOS [13].

## 1.3. Intelligent Environments

Intelligent Environments are ordinary environments which are equipped with computational devices to enhance the experience of the persons inside the Intelligent Environment. Sensors collect data about the user and the environment itself and combine it to provide intelligent services to the user. This can be achieved by combining the Internet of Things together with a powerful software architecture that can process and interpret the data coming from the sensors in a smart way. The goal of an Intelligent Environment is to vanish into the background, so the user can act as if it wasn't there. This makes it necessary to develop highly independent and self adjusting algorithms and applications. They need to be able to cope with an ever changing set of available actuator, sensors and other resources. An example for this kind of environment is provided in [14]. Here the environment starts to learn, what the users inside it, want to achieve and helps them by using its resources. It utilizes autonomous software programs (Intelligent Agents) which are designed to be able to learn the user's behaviors and intents. They then use this knowledge together with the infrastructure, provided by the intelligent environment, to support the user with its actions.

The Intelligent Environment is made up by many sensors, actuators, personal devices and other computing and networking infrastructure. This also includes devices to display information like projectors or electronic door signs, as well as interaction devices. A special role has to be granted to devices that travel through the smart space. These nomadic devices can reveal a great deal of information about current state of the smart space, but also pose a special case that has to be considered when creating applications for a smart space.

Nowadays smartphones are becoming a personal device of most users. They can create an interface to the smart space when ever the normal interaction of the user with the objects is not sufficient. They can also provide the system with accurate location information regarding the user who owns the smartphone.

Sensor data in Intelligent Environments can come from physical sensors attached to nodes of the IoT or other devices. But it can also be provided by so-called 'virtual sensors' which are sensors that are implemented in software. These can include the arrival of mail or the connection status of a network adapter. Data can also come from 'logical sensors'. These sensors process data from physical and/or virtual sensors and provide the result as a new sensor value. This concept makes it easier to create services in the Intelligent Environment, that need the fusion of data from different sensors. It decouples the generation of sensor data from the reasoning about the data and provides a uniform programming interface to different kinds of data sources.

## 1.4. Middleware

To manage the vast amount of devices and the services they provide, a well structured system has to be put in place. Such a management system is generally called the middleware. It provides an abstraction layer between the devices providing the services and other (software) components which want to use these services. It provides unified naming and semantics of all connected devices. This allows components to be reused in different situations. It might also provide more sophisticated functions like compound services or a security concept to prevent misuse of the system. Apart from these functional aspects, a middleware provides tools to manage, monitor and administrate the complex system of interconnected devices.

The term middleware originated in the 1980s, when computer systems used to process business applications started getting more complex. There are different approaches to implement a middleware. Some of them try to be very general while others try to solve specific problems within their domain.

Many types of middleware have been proposed to be used in Intelligent Environments. Some of them are of a general architecture and some of them have been specifically developed for Intelligent Environments. So far no single middleware has achieved widespread use in the academic community. This is partly caused by either the lack of special functionality for Intelligent Environments, the sheer amount of constant work that is needed to support a middleware over time, or the general acceptance of yet another middleware on the market. Gaia [15], MundoCore

[16], MiddleWhere [17] and UBIWARE [18] are examples for middlewares, designed for Intelligent Environments or the Internet of Things.

## 1.5. Robotics Middleware and Immobots

During my previous studies at the Institute for Cognitive Systems at Technische Universität München, the choice of the middleware was also an important topic. Some of the researchers chose to not use a middleware at all, the rest had to choose from only a couple of choices. During the last few years, one of these middlewares became the dominant middleware in robotics: the Robot Operating System (ROS). Since then, many robotic researchers have adopted ROS as the underlying middleware for their projects. The very modular construction of ROS lead to a very active community which is busy developing ever new software modules for ROS. Driver modules exist for almost every commercially available robotics hardware platform.

In 1996 Williams and Nayak introduced the the notion of Immobots [19]. An Immobot is a robot that has characteristics of a normal robot like sensor richness and autonomy, but it is not able to change its own position. It is therefore an immobile robot. Intelligent Environments meet these requirements as well. The concept of Immobots makes it possible to exploit the maturity and community approval of a robotics middleware in the field of Intelligent Environments.

This approach has already been described by Roalter et al. in [20]. They are describing how ROS can be used in Intelligent Environments.

The results of their evaluation shows, that the use of ROS can reduce the time needed to set up an environment and the time needed to conduct experiments. This means that more time is available to actually implement and test algorithms for Intelligent Environments.

Although the results seem promising, ROS has some specific properties that limit it in the field of Intelligent Environments. The major shortcomings that this thesis will cover are:

1. Lack of support for the latest Internet Protocol standard IPv6.
2. The focus on a single coordinator for all connected nodes and services.
3. The general assumption that devices are connected via high bandwidth network links.
4. Lack of a system or standardized rules to organize the internal namespaces.
5. No concepts are in place to secure the transferred data.

These factors limit the usability of ROS for real world applications in the filed of Intelligent Environments. This thesis will introduce a scalable multimaster support for ROS, better support for small devices from the Internet of Things and a concept to make ROS more secure and preserve the privacy of its users.

## 1.6. Overview Over The Next Chapters

- Chapter 2 gives an overview over current technologies used in Intelligent Environments. It details basic design principles and introduces a selected number of current middlewares.
- Chapter 3 contains the design decisions made to improve the ROS for usage in Intelligent Environments.
- Chapter 4 covers the actual implementation of the concepts using ROS.
- In Chapter 5 the results of this work are evaluated using a real ROS network.
- Chapter 6 concludes the thesis and details further steps which have to be taken to make ROS a successful middleware for Intelligent Environments.

# Chapter 2.

# State of the Art/Related Work

This chapter gives an overview over current technologies used in Intelligent Environments. It covers the basic principles on which current middleware implementations are based on. Furthermore a selected number of middleware implementations is introduced. The chapter also gives an introduction into current networking protocols used by these middlewares. It argues why no current middleware has yet reached the acceptance as a de facto standard in the community, introduces the robotics middleware ROS in more detail and shows the benefits of using it.

## 2.1. Basic Design Principles for Middlewares

Various middlewares have been proposed, designed and implemented. There are major differences between these middlewares, though most of them follow similar basic design guidelines. Most middlewares can be sorted into the following categories, regarding their basic concept:

1. Publish/Subscribe oriented design
2. Service oriented design
3. Database oriented design
4. Schizophrenic design

The following sections explain the different design principles:

### 2.1.1. Publish/Subscribe

This principle is centered around the so-called publish/subscribe design pattern. It assumes, that Intelligent Environments are data driven. Data is derived and made available by so-called publishers. If another part of the system wants to get access to this data, it has to subscribe itself to the data stream that is published by this publisher. This provides a very loose coupling between the different parts of the system. As long as the publisher and the subscriber can agree on a common data format, no other common basis between the two has to exist, to allow them to communicate. The data stream carries discrete messages, covering information about a specific topic, between the publisher and the subscriber. Therefore, data streams are organized as a set of topics, to which a subscriber can announce its interest.

There can be multiple subscribers, subscribing to a single publisher and multiple publishers being subscribed by a single subscriber. The different publishers do not have to be aware of each other, as well as the different subscribers do not have to be aware of each other.

To give the subscribers access to the publishers data streams, a database (registry) needs to be created, which contains all information necessary, to locate data streams of interest for the different subscribers. There are different options to implement such a registry:

#### Central Registry Server

A central node is chosen to act as the central registry server. It is contacted by every publisher and stores the information about the publisher in an internal database. Subscribers have to contact the registry every time, they want to retrieve information about a data stream.

#### Distributed Registry Server

In this scenario, the registry is composed by a multiple set of servers, and every server holds a part of the registry. When a subscriber asks one of the registry servers for information, the registry server might already have all the needed data. Otherwise the registry server contacts other servers and queries them for the information.

#### Distributed Registry Database

In this design the database is distributed over an arbitrary set of nodes in the network. There is no special coordinator node and there are also no special servers, holding a specific part of the registry. Instead the registry is distributed in even parts over the whole network using Distributed Hash Tables (DHT [21]).

After an application has found a data stream of interest, it must subscribe itself to events, coming from that stream. There are three basic principles, how this can be implemented:

#### Direct Connection to the Publishers

As presented in Figure 2.1, in this case the subscribing node directly contacts every publisher, which is publishing a specific topic and maintains a separate subscription to each of them. It must

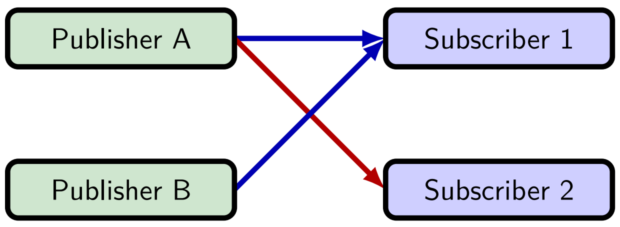


Figure 2.1.: Two subscribers subscribing to two different topics. Publisher A is publishing both topics, while Publisher B publishes only one.

also monitor the system for publishers that start or stop to publish this topic. This system has the advantage that each connection between a publisher and a subscriber is independent of other connections. This leads to a system which is stable against network failures or crashed nodes. If the bandwidth of the underlying network is large enough, it also provides a low latency between the publishers and the nodes. The major disadvantage is the redundant transmission of data over the same network links when more than one subscriber is connected via the same link to the same publisher. This can also lead to network congestion if the network link does not have the necessary bandwidth.

### Indirect Connection Via a Central Broker

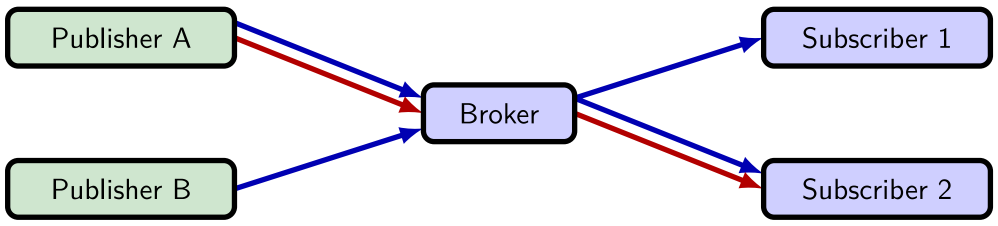


Figure 2.2.: The broker in between acts as a switchboard for all communication.

In this scenario data does not flow directly from the publishers to the subscribers. Instead the data streams of all publishers are concentrated in a single node as shown in Figure 2.2. This node is called a broker. Subscribers only need to contact the broker. They also do not have to keep track of the state of the publishers, as the broker will keep connections to every publisher that has at least one matching subscriber. The broker can offer additional services to publishers and subscribers. It can serve as a central point of trust, to which different nodes can connect and exchange data, without needing to exchange encryption keys. It can also enforce policies, which regulate data sharing between different parts of the system. The system still suffers from data transmitted redundantly over the same network links and introduces a single point of failure. When the broker is not able to forward messages anymore, no subscriber is anymore able to receive messages. The concentration of all data exchange on a single system can also lead to severe latency, since the system has to deal with many requests at once.

#### Indirect Connection Via a Network of Brokers

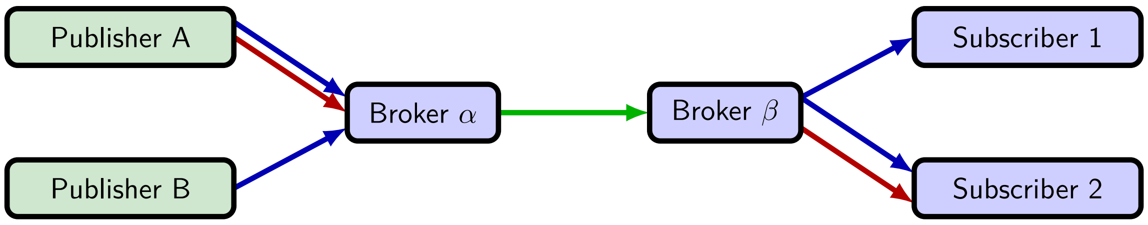


Figure 2.3.: A network of brokers forwards messages to different parts in the system.

In this scenario the the central broker is replaced by a distributed set of brokers. These brokers maintain connections between each other to route messages between them. Figure 2.3 provides an example with two brokers forwarding messages between each other. If a subscriber is interested in a specific topic, it can contact any broker and request this topic. The broker network is now responsible to route messages for this topic to this broker. The broker then sends the messages to the subscriber.

The solutions involving one or more brokers, enables more possibilities to embed logic and other arbitrary functionality inside the brokers. Examples are filtering or transformation of topics, without duplicating this effort in every single subscriber.

### 2.1.2. Service Oriented Approach

The Service Oriented Approach (SOA) is focused around the concept of invoking methods in remote nodes. Every node, which is participating in the system, offers an interface to the other nodes, to call procedures which are implemented on this node. The procedures can the be called with a Remote Procedure Call(RPC) protocol. There exists a whole set of mature RPC protocols like CORBA, SOAP or XML-RPC, designed for business applications. Recently a new set of protocols has been devised to specially fit the needs for smaller foot print devices used in the Internet of Things like DPWS [22] or TinySOA [23].

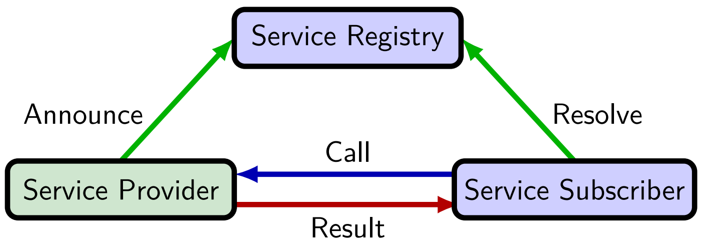


Figure 2.4.: Example for a Service Oriented Approach: One node calls the service of another node and receives the result.

Usually only a specific set of procedures are made available to other nodes, to ensure the stable operation of the node. There are many security considerations to be made, as the execution of procedures might cause unwanted side-effects inside the nodes.

The Service Oriented Approach also needs some kind of registry, where nodes can announce which procedures are offered to other nodes. The principles that were developed for publish/subscribe systems to announce topics can also be applied to this problem. In Figure 2.4 the service provider announces a service, which the subscriber can resolve and call.

When designed properly, the Service Oriented Approach can also provide a loose coupling between the different components. A few important design criteria to incorporate the Service Oriented Approach with Wireless Sensor Networks are stated in [24] and [25].

A disadvantage of the Service Oriented Approach is the coupling of returned data to the original request. This prevents the data from being directly reused or cached for other applications, which might be interested in the data. Each application has to make a dedicated call to the service, to get the desired data.

### 2.1.3. Database Oriented Design

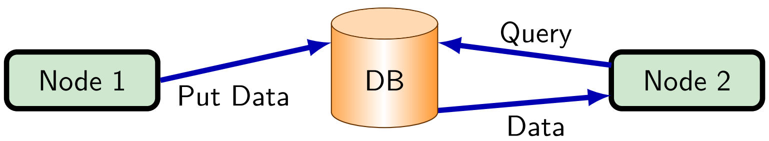


Figure 2.5.: The database stores the data from the sensors. Other nodes need to query the database for new data.

In a database oriented design all data from sensors is collected in a database. Other nodes can query this database using query languages like SQL [25]. This makes it possible for other nodes to formulate complex queries without the need to process all data involved. Although this creates applications with a small foot print on the nodes, this approach has limited interactivity and concentrates the complete management in a singe database, creating a single point of failure.

### 2.1.4. Schizophrenic Design

A schizophrenic design takes into consideration, that network protocols constantly evolve and new ones are designed [26]. There are also different protocols, which will always coexist next to each other, performing essentially the same functionality. This can be result of market forces or the need to stay compatible to older designs. It is therefore beneficial to design the middleware in a way, that it can support different protocols on top of its core logic. The goal is to make it easy to add new protocols as they become relevant to the system. The different protocols might have different views on the underlying Intelligent Environment and the middleware might have to emulate certain aspect of it, to satisfy the needs of some protocols.

## 2.2. Security and Privacy in Intelligent Environments

Apart from implementing the functional requirements for Intelligent Environments, security and privacy are probably the most important non-functional aspect of a middleware [27]. If one of these aspects is neglected, the survival chances of the middleware in a real world deployment are very low.

The two aspects are tightly connected to each other, but a secure system is by no account also a system, that ensures the privacy of its users.

A secure system must meet the following requirements:

- Authentication of its users.
- Authorization of requests.
- No leak of data to unauthorized users.
- No acceptance of data from unauthorized users.

While a system respecting the privacy of its users must meet the following criteria:

- Clear communication to the users, which data is collected.
- Clear communication to the users, who has access to data and why.
- The possibility of an Opt-in or Opt-out into different parts of this data collection.
- A way to ensure that the choices of the users are respected.

### 2.2.1. Security Considerations

The deep integration of an Intelligent Environment into the daily lives of its users requires that it has to withstand malicious intents from third parties. Although this seems obvious, in reality strict security policies are usually one of the last things considered, when designing or implementing a system. This results from the large complexity which is added by the security layers and which in turn hampers developing and especially debugging a system. Even in large industrial situations where expensive equipment and business revenue is at stake, security considerations seem to always be an afterthought. This has recently been publicly demonstrated by the attacks carried out by the Stuxnet worm. It specialized on attacking industrial control systems deployed all over the world. Even though these systems were controlling important infrastructure, they had almost no protection against attacks at all [28].

### Security and Embedded Devices

Implementing a secure system for Intelligent Environments is challenging, as many different types of devices have to talk to each other. Many of these might have very limited computing power, e.g. nodes in Wireless Sensor Networks. This opens up two basic options: The first options is to implement different security layers, each covering a certain branch of the system, and provide adapters between these layers. The second option is implementing a unified security system with end-to-end encryption, authentication and authorization. This is especially interesting when incorporating the Internet of Things where generally, direct connections between the communication partners are expected.

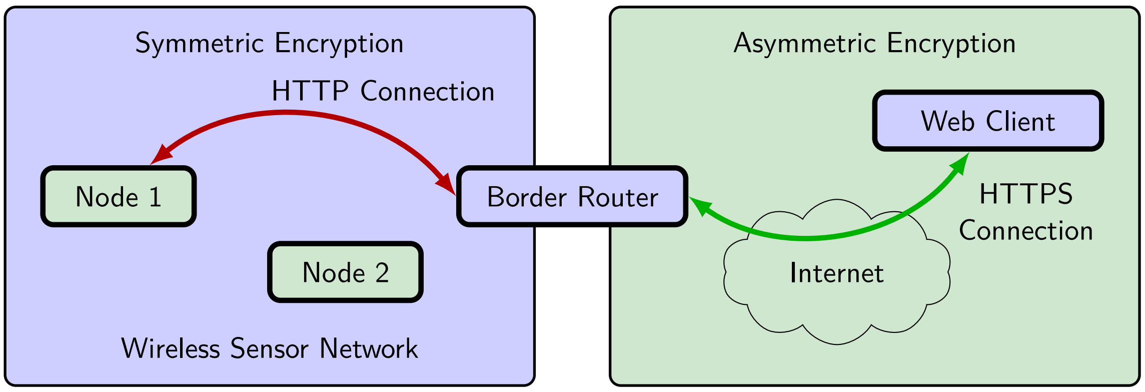


Figure 2.6.: The border router accepts asymmetrically encrypted HTTPS connections, decrypts them and forwards them to the nodes in the Wireless Sensor Network. Node 2 can also decrypt the connection.

An example using Wireless Sensor Networks shows, how a layered architecture can be implemented is presented in Figure 2.6.

In this example the nodes of the Wireless Sensor Network are connecting to the Internet through a border-router. The nodes are heavily constrained in terms of available memory and processing power. The border router on the other hand has a decent amount of memory and processing power. As it is also the gateway, through which the nodes connect to the Internet. It can act as a protocol translator between the Internet and the nodes. Wireless Sensor Networks usually use a symmetric encryption with a shared key between the nodes and the border router. Internet protocols on the other hand often use asymmetric cryptography, as it provides much more flexibility in its usage. It also needs much more computing power to be executed. To bridge between these two standards, the border router can terminate requests to the nodes locally, handle the asymmetric encryption, encrypt the request again for the node, receive the response from the node and send the result back over the Internet. Likewise, the same principle can be used when the node sends a request to the Internet.

It has to be noted, that in this scenario all nodes must trust each other, since every node can decrypt the data of every other node. Thus, every node of the Wireless Sensor System needs to be trusted with this data. When a device gets compromised or is not trusted anymore, a way has to be provided to revoke the current key and distribute a new key to the still trusted devices.

Although this system does fulfill the task of encrypting the data from the nodes for the web, it does not take into consideration where to handle the authentication and authorization of the request.

This leads to the question on how to choose a concept, which can handle the authorization and authentication for a system in Intelligent Environments. [29] provides and overview over different technologies used to create security aware middlewares. Many different middlewares are examined, but none is able to cover every aspect to create a secure system. The authors note that this indeed is a complex task to achieve with the constrained devices, used in the Internet of Things.

### 2.2.2. Privacy Considerations

The problem of privacy in Intelligent Environments leads to two main considerations:

First, the actual implementation of the criteria introduced in section 2.2. Yamada and Kamioka [30] analyze different categories of systems implementing these criteria. Jian et al. [31] separate privacy aware systems into three categories:

1. Prevention: The reduction of exchange and storage of privacy related data.
2. Avoidance: Minimizing the risks and maximizing the benefits of the transfer and storage of data. This is often combined with getting the consent from the (human) users.
3. Detection: Detection of misuse of private data, so the violators can be prosecuted.

Second, the task of reassuring the the users, that these criteria are actually enforced. The users usually can not get access to the inner workings of the system or do not have the needed expertise to understand all of its operations. This can lead to the suspicion that the system might actually not implement sufficient privacy protection. As public inner knowledge of the system can contradict the requirements for security and privacy, a trusted third party can be used. This third party must have the trust of the users as well as the system operator. When a sufficient number of these third parties exist, no concentration of possible access to the complete private data of many users starts to form.

## 2.3. Selection of Different Middlewares for Intelligent Environments

This section introduces a few different middlewares that have been proposed to be used with Intelligent Environments and that make use of standard protocols or have already been accepted by specific communities as a new standard.

### 2.3.1. SensorsMW

In [25] the SensorsMW middleware is presented. SensorsMW provides a service oriented approach for integrating Wireless Sensor Networks into the Internet. The proposed middleware uses standard web technologies to implement the service oriented approach.

The authors argue that it is not feasible to implement complete support for a service oriented approach on the nodes directly. Instead they propose to use a powerful gateway that abstracts all complexities from the Wireless Sensor Network.

Although this leads to a smaller foot print on the wireless nodes, it places assumptions on how the wireless nodes will communicate with other devices on the Internet. This hinders the adoption of new protocols based on new and emerging Internet standards. It basically generates a lock-in of the wireless network to the currently used protocols and standards.

### 2.3.2. DPWS

Moritz et al. [32] propose a middleware based on the Device Profile for Web-services (DPWS) [22]. DPWS is a standard that has been developed to lower the footprint of the Web Services (WS) standard defined by the W3C. It includes specifications for executing remote procedure calls, security, authorization and service discovery. Its advantages are the usage of standard Internet protocols and the option to have decentralized service discovery. It does not make use of a centralized gateway. Every node is running a complete independent software stack, supporting DPWS.

### 2.3.3. MQTT

Message Queue Telemetry Transport (MQTT[1]) is a middleware developed by IBM, which is targeted at the Internet of Things. It follows the publish/subscribe concept with broker nodes which relay messages through the network. MQTT-S [33] is an extension to MQTT, which is

[1]http://mqtt.org

targeted specifically at Wireless Sensor Networks and their lossy links. While MQTT-S is still a very young protocol, MQTT is very mature and in active usage in real world applications.

Both protocols do not include a security or privacy concept and also do not directly cover the discovery of nodes or services in the network.

## 2.4. Acceptance of Current Middlewares by the Community

Although many more systems like CARMEN [34], Gaia [15], MiddleWhere [17], Context Toolkit [35] or UBIWARE [18] have been proposed, designed and implemented, no middleware has yet reached the status of a de-facto standard [20, 24, 26, 36–39]. Most research projects have a limited time frame in which the research is conducted and the final results are presented. After this period no more research is invested into the system. The code base does not get updated anymore and does not stay compatible with the fast changing landscape of operating systems and hardware platforms. Keeping a complex middleware setup up to date with the latest trends in computing requires the efforts of a community that has to form around the middleware. Furthermore the end of the research project does not induce confidence, to use the system in other permanent installations.

Although the limited time frame of active development makes not implications about the quality of these middlewares, most of the also share the lack of having a proper simulation. The ability to simulate a system, without actually having to deploy it, enables a wider audience to develop, test and evaluate the system.

## 2.5. The Robot Operating System

The Robot Operating System (ROS,[40]) is a middleware specifically designed to ease the development of applications for robots. During the last years it has gained a substantial amount of attention in the academic robotics community. It is focused on the publisher/subscriber concept, but also supports a service oriented design.

### 2.5.1. Nodes and Topics

Inside ROS, components are called nodes. Each node can publish multiple topics and subscribe to multiple topics. A topic consists of its name and a message type. The message type defines what kind of information is carried by a specific topic. A node is a piece of software that can contain a mixture of publishers, subscribers and service providers. Multiple nodes can run simultaneously

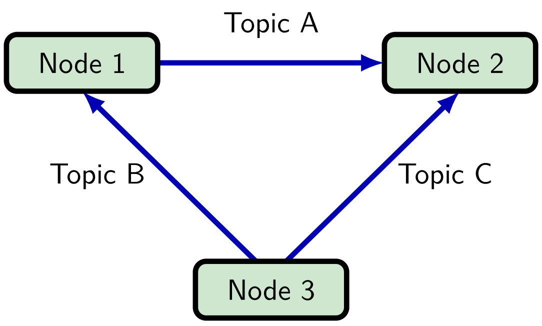


Figure 2.7.: A ROS setup with multiple nodes publishing and subscribing to different topics. A ROS node can publish and subscribe multiple topics at the same time.

on a single machine and open connections to other nodes via network connections. Figure 2.7 shows an example configuration with nodes publishing and subscribing to different topics.

### 2.5.2. Message Types

```
Header header
uint32 height
uint32 width
uint8[] data
```

Figure 2.8.: Example declaration of a message type that can carry an image taken by a camera.

The message type defines exactly what kind of data is transported by a specific topic. A message type is defined by a message declaration. Message declarations are textual descriptions of the content of a message. Figure 2.8 shows an example of a message declaration. This description is compiled into source code for the different programming languages which ROS supports. To prevent conflicts, each message type is protected with a checksum. When opening a connection, the publisher and the subscriber check that their checksums of the message type match. If this is not the case, the two nodes can not exchange messages. Both, the publisher and the subscriber, need to have local copies of the message type available. When the node starts up, the known message types are loaded. This makes it necessary to distribute the message types each node is using, before the node can be launched.

ROS already defines a set of built-in message types that can be used by all nodes without the need to transfer message declarations to each node.

### 2.5.3. Services

ROS also supports communication via services. In contrast to the communication using topics which is completely asynchronous, services provide synchronous communication.

A service is provided by one node and called by another node. The calling node provides data to be processed and the providing node returns the result of the service. A service consists of its name and a pair of message types.

### 2.5.4. The Master Node

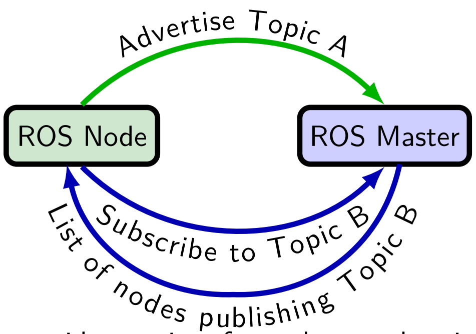


Figure 2.9.: The ROS master provides services for nodes to advertise topics and find other nodes publishing topics.

A central element of a ROS environment is the master node. The task of this special node is to hold the central registry of all active topics and services.

The master node exposes its interface via the XML-RPC standard. This standard uses XML transferred over a HTTP connection. When a client node opens a connection to the master node, the client node sends the name of the called service, together with the input data as an XML document, to the master node. The master node then sends the result back over the same connection, again encapsulated as XML. Figure 2.11 shows an example communication between the master node and another node, subscribing to a topic. After the call has been executed, the connection can either be closed or kept alive by the client node. It is also possible to make calls to multiple procedures in a single transaction. The use of the web standards HTTP and XML allows the interface to be used by many programming languages with relative ease.

Nodes use XML-RPC calls of the master node to:

- register a new publisher.
- get a list of published topics.
- get a list of current publishers of a specific topic.
- register a new service.

- get a list of services.
- get more information about a specific service.

Figure 2.9 shows the flow of information between a node and the master, when publishing or subscribing to a topic.

The master node presents the only way a node can get the information needed to contact another node. If the master node fails or is not reachable, no new connections can be established.

Figure 2.11 also shows the verbosity of the XML-RPC interface. To subscribe to a single topic, with two publishers already publishing to it, over 700 Bytes of XML documents are transferred. Although it is possible to use this protocol with Wireless Sensor Networks, it carries a significant overhead.

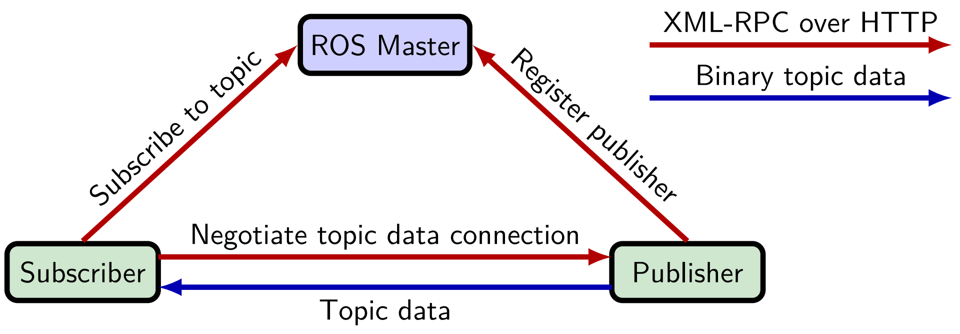


Figure 2.10.: The nodes exchange topic data directly, without intervention of the master.

The publish/subscribe system of ROS does not include brokers. After querying the master node, the nodes open direct connections between each other. This includes an additional XML-RPC server at the publisher, that is used to negotiate how the actual topic data will be transferred. For this an additional persistent connection is opened, which transfers the topic data with a more efficient binary protocol. Figure 2.10 shows the different connections that exist between the different parts of a ROS network.

### 2.5.5. Namespaces

To support a hierarchical arrangement of nodes, topics and services, ROS supports namespaces. The namespaces are managed by the master node. There are three separate namespaces for nodes, topics and services. Each namespace consists of the root node '/' and a hierarchical organization of identifiers and separators ('/') below it. Figure 2.12 shows an example topic namespace for the topics of an example system. In this example the temperature of room1 could be accessed by subscribing to the topic '/building1/room1/sensors/temperature'.

ROS uses namespaces to prevent nodes with the same name or topic with the same name from affecting each other. The namespaces are used to categorize the names of nodes and topics, so that the same names only appear in different branches of the namespace. In the example there are two topics for the temperature. Since they reside in different parts of the namespace, they do not collide. ROS does not specify any rules which define how the namespaces should be organized. This task is part of the design of a particular system and the different applications have to agree on common layouts for the namespaces.

### 2.5.6. The Parameter Server

The parameter server stores key/value pairs of data inside a hierarchical structure similar to the namespaces. Nodes can globally create keys on the parameter server and read data from the parameter server. It enables the nodes to share mostly static data between them. It is also used to configure the behavior of a node based on parameters inside the parameter server.

The parameter server is part of the ROS master node and shares its XML-RPC interface.

## 2.6. ROS for Intelligent Environments

It seems not advisable to implement a new middleware, but to use a middleware which has a solid user base, is actively supported and already has features which make it an interesting choice for Intelligent Environments. Roalter et al. have already shown that ROS can serve as a middleware in a deployed example scenario [41].

ROS is already in active use by the robotics community. This leads to a strong synergy with Intelligent Environments, as it allows the integration of robots into the setting of an Intelligent Environment and even handle the Intelligent Environment like another robot inside the system.

ROS already has major support for the simulation of a ROS based system. It supports the simulation of the complex systems, including 3D visualization and physics simulation. The modular approach of ROS makes it easy to mix simulated nodes with real nodes. This makes it possible to develop new sensors and actors in a realistic scenario, without having to setup a complete physical environment for them.

ROS also allows the collection of data during runtime and playback of this data at a later time. This helps when developing algorithms for Intelligent Environments and testing them against the same input data.

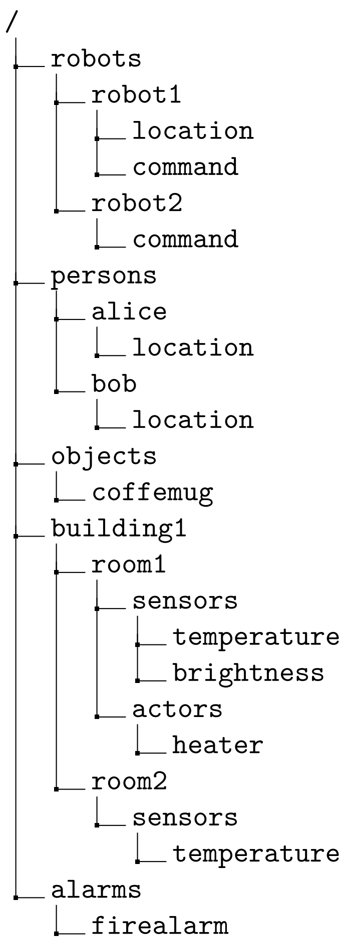
```
<?xml version='1.0'?>
<methodCall>
<methodName>registerSubscriber</methodName>
<params>
<param>
<value><string>/subscriber1</string></value>
</param>
<param>
<value><string>/topicA</string></value>
</param>
<param>
<value><string>std_msgs/String</string></value>
</param>
<param>
<value><string>http://subscriber1:50931/</string></value>
</param>
</params>
</methodCall>
```

(a) Request sent to the master node to subscribe to a topic.

```
<?xml version='1.0'?>
<methodResponse>
<params>
<param>
<value><array><data>
<value><int>1</int></value>
<value><string>Subscribed to [/topicA]</string></value>
<value><array><data>
<value><string>http://publisher1:56070/</string></value>
<value><string>http://publisher2:57452/</string></value>
</data></array></value>
</data></array></value>
</param>
</params>
</methodResponse>
```

(b) Reply of the master node with a list of nodes providing the topic.

Figure 2.11.: Example XML-RPC communication with the master



```
/
└─ robots
│  ├─ robot1
│  │  ├─ location
│  │  └─ command
│  └─ robot2
│     └─ command
├─ persons
│  ├─ alice
│  │  └─ location
│  └─ bob
│     └─ location
├─ objects
│  └─ coffemug
├─ building1
│  ├─ room1
│  │  ├─ sensors
│  │  │  ├─ temperature
│  │  │  └─ brightness
│  │  └─ actors
│  │     └─ heater
│  └─ room2
│     └─ sensors
│        └─ temperature
└─ alarms
   └─ firealarm
```

Figure 2.12.: Example ROS topic namespace.

## 2.7. General Introduction to Networking

### 2.7.1. The Internet Protocol

The Internet Protocol (IP) is the basic protocol underlying almost all of today's communications. The protocol is based on packet transmission. A packet contains a certain amount of data as well as the destination and origin of the packet. The network which transports the packet is responsible to transmit it from the origin to the destination.

There is no guarantee that a packet will be delivered in a specified time or even at all. Higher level protocols have to be used to ensure delivery.

#### IPv4

There are different versions of the Internet Protocol. Currently IPv4 [42] and IPv6 [43] are the only two versions in use. IPv4 has been in use since 1981, when the Internet was still an academic and military experiment. The most important aspect of the IPv4 protocol is the size of its address fields. Every address field has a size of 4 octets (32 Bits). This allows the IPv4 protocol to address about 4 billion different destinations. During the last decade the pool of free IPv4 addresses has shrunken rapidly. During the writing of this thesis, the last remaining IPv4 addresses available in Europe are being assigned to registrants.

#### IPv6

The newer IPv6 protocol has been designed to combat the inevitable exhaustion of the IPv4 address space. It also reduces the work needed to route one packet from its origin to its destination. To achieve this, the size of the address fields in an IPv6 packet has been increased to 16 octets (128 Bits). This increases the number of available addresses by a factor of $8 * 10^{28}$. While this number seems to be unreasonably large, the increased number of bits available has additional benefits. They allow additional information to be coded inside the address of a specific host inside the network (see Figure 2.13). For example this information can contain the location or function of the host inside the network. Especially the location of the host inside the network allows Internet routers to more efficiently route packets from one location to another, as they only have to look at the bits indicating the approximate location of a host.

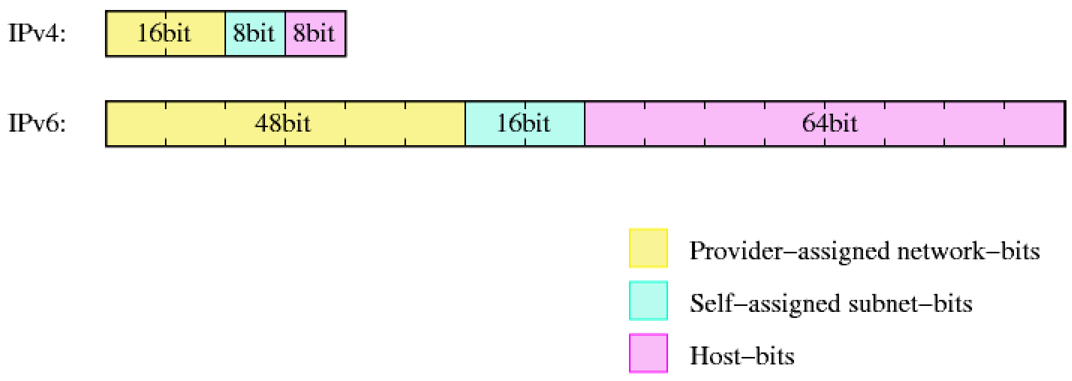


Figure 2.13.: The address format of IPv6 allows to encode 64 Bits of information in the host bits. (Source: http://www.netbsd.org)

### 2.7.2. TCP and UDP

The Internet Protocol makes no assumption about the data carried by it and also provides no guarantee that the packet will actually be delivered. This makes higher level protocols necessary that can provide applications with additional services.

The simplest and also one of the most used packet types is the UDP packet. It provides the application with an indication about the contents of a specific packet by associating it with a port number. The port number describes how a specific packet should be processed in the destination host.

The TCP protocol is more complex and guarantees the in-order delivery of the data contained in TCP packets. TCP also uses port numbers and additionally carries information about the relative location of the packet inside a stream of packets. It provides automatic mechanisms to recover, when IP packet get lost and can not be delivered (packet loss). This leads to a certain overhead when communication over TCP and also requires the two hosts to be able to reach each other, even if only one host wants to send data.

### 2.7.3. Multicasts

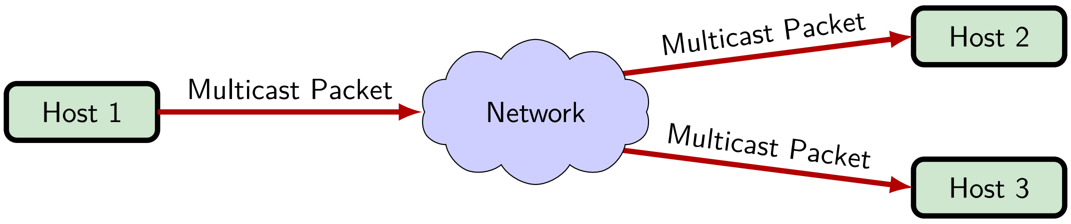


Figure 2.14.: A multicast packet is forwarded by the network to every node that has joined the associated multicast group.

Multicasts form a special class of UDP packets. A multicast is used for one to many communication. One sender sends out a packet and multiple other hosts are reached by the message. A host wishing to receive multicasts has to join a multicast group. A multicast group is defined by its multicast address. There are some preallocated multicasts addresses[2] and also multicast ranges that can be used freely inside the local network. After a host has declared its interest for a specific multicast group, the network will try to send all multicasts packets that belong to that group to the host. As shown in Figure 2.14, the network is responsible to to this in a way that does not consume unneeded resources. The sender of the multicast packet does not know which hosts or how many hosts are receiving its packet.

Multicasts are effective if the sender of a message does not know which host is expecting a message or if the same message has to be sent to multiple hosts. This has drawbacks, when the sender needs to know it the packet actually reached its destination, since there is no feedback when a packet gets dropped in the network.

### 2.7.4. Routing in IP Networks

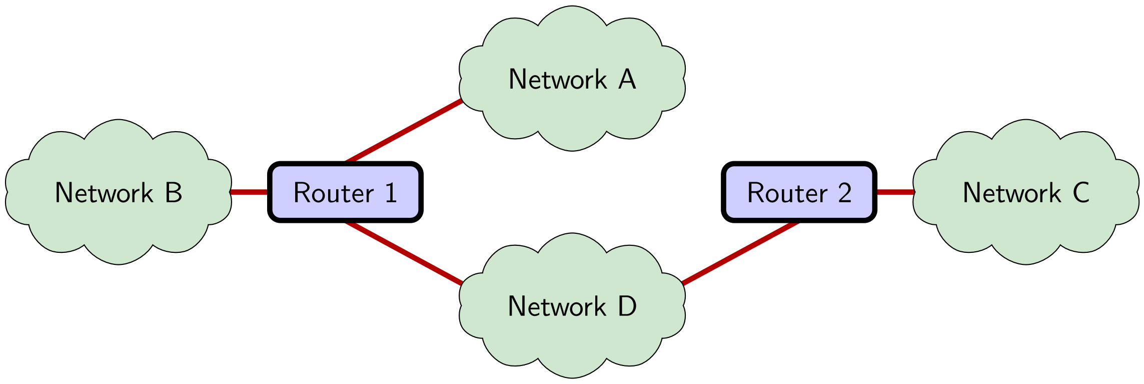


Figure 2.15.: Routers connect multiple networks with each other.

Inside a local network, all devices can reach each other directly. This is possible, as the network equipment is still able to keep a map of the complete network in its memory. When expanding the network, this does not work anymore and dedicated devices are needed to connect different networks together, to form a bigger network. These devices are called routers. Routers connect two local networks with each other. They concentrate the data transfer between the two networks. Figure 2.15 shows an example with four networks connected by two routers. The Internet is formed by many such local networks, which are connected via routers to form a network of networks.

An important aspect of this configuration is that often it is not possible to exchange multicast packets between different local networks which are connected with routers. This is to prevent

[2]IANA Internet Protocol Version 6 Multicast Addresses, `http://www.iana.org/assignments/ipv6-multicast-addresses`

excessive data transfer between the the networks, which could be caused by multicasts, and the lack of support for multicasts in many smaller routers.

### 2.7.5. DNS

To make it easier for humans to memorize the address of a host, the Domain Name System (DNS) provides a way to translate human readable names into IP addresses. This also makes it easier to change the underlying IP address of a device, without having to change references pointing to it.

DNS uses a hierarchical network of DNS servers to store DNS records. These records can hold IP addresses as well as other additional information about a host. Each DNS server is responsible for a specific part of the DNS namespace. If it does not have the needed information for a request, it can contact other DNS servers to get the needed information. This keeps the system decentralized and able to hold billions of entries.

Although the DNS system is mostly used to contact remote hosts on the Internet and is configured mostly in a static way, it can also serve inside a dynamic local network. In this setup a local DNS server is setup in the local network. Every time a new device is connected to the network, the device sends its desired hostname to the network. The DNS server uses this information together with the IP address of the device to include the name in the local directory.

# Chapter 3.

# Concepts to Improve ROS for Intelligent Environments

Although ROS already has many advantages like the support for publish/subscribe, services, simulation and the use of standard protocols, it still has some major shortcomings when put to use in a system for Intelligent Environments.

This includes the dependence on a single master node, the direct connections of the nodes wasting bandwidth, no regulations for the namespaces, no support for the new IPv6 protocol and no security considerations.

This chapter will detail these problems in more detail and offer ways to improve the situation for ROS in the field of Intelligent Environments.

## 3.1. Shortcomings of ROS

ROS may be a good solution for robotic environments, with a limited number of robots which are connected by high bandwidth LAN connections. It shows major limitations when used inside a larger IE spanning over large areas and consisting of lots of nodes with limited bandwidth and lossy connections.

This is mainly caused by the architecture of ROS consisting of a single master and the nodes which are talking directly with each other. Furthermore, the single master leads to a single point of failure: When the master stops working, no new connection can be established by any node inside the system. It can also lead to a severe load to the network connection of the master node. If this connection has a non negligible latency from one part of the system, this will be observable by sluggish behavior of this part of the system.

The direct connections between publishers and subscribers put a large network load on the network connection of the publishers. They have to send the same message to each of their subscribers.

The network connection of publishers with a limited bandwidth to the rest of the system can be overloaded if too many subscribers try to get messages from it.

There are no specific rules, where to put a new node or topic into the namespaces. It is up to the implementation of the node or its configuration, to specify a preselected and fixed path inside the namespace. This can lead to a complex and incomprehensible namespace, which limits its usefulness and can lead to collisions if no special care is taken.

ROS itself is currently fully based on the IPv4 protocol. The IPv4 Internet protocol has been in service since 1981 [42]. As the pool of free IPv4 addresses nears depletion [44], it makes no sense to base new technologies for the Internet of Things on this protocol. With the much larger scope of the Internet of Things, much more address are going to be needed to uniquely identify each device. Therefore, most of the current devices, which can be counted to the IoT, base their communication on the IPv6 protocol.

## 3.2. Support for IPv6

As mentioned before, the Internet of Things depends heavily on the next generation Internet Protocol IPv6. This makes it hard to integrate a network of IPv6 capable sensor nodes into a ROS environment, as ROS has no support for the IPv6 protocol.

It is therefore advisable to add IPv6 support to ROS. ROS applications are built using ROS libraries, which abstract the actual network interaction from the applications. It is only necessary to add IPv6 support to these libraries. Even the ROS master node is using these libraries, so there is no special handling required for the master node.

### 3.2.1. Hostnames

Fortunately, the protocols used by ROS do not directly use IP addresses. They rely completely on the ability of the machine they are running on, to be able to resolve a hostname into an IP address that can be used by the libraries.

This makes it possible to add IPv6 support to ROS without modifications to either the applications built on top of the libraries or the underlying protocols. The only requirement on the actual network, in which IPv6 shall be used, is that it is possible for every host to resolve the name of any other potential host into an IPv6 address.

This can be either achieved by adding every possible hostname and its IPv6 address into the local configuration files of the host running the application, or by setting up a working DNS environment. Hosts entering the network need to contact some server inside the network to get their entry into the DNS system.

This is usually performed using the DHCP protocol. It is often used together with IPv4, since it centralizes the assignment of IP addresses to hostnames. This makes it possible to assign every device the same IP (if there are no collisions due to the available amount of IPv4 addresses), to prevent misconfiguration and to keep a list of known hosts.

With IPv6 the DHCP mechanism has been widely superseded by the use of Stateless Auto Configuration. This technique makes it possible to give every host always the same IP, while removing the need for a DHCP server, which needs to keep track of every host [45].

Stateless Auto Configuration does not use the name of a specific hosts, but its EUI-48 or EUI-64 global identifier. The name of the host is not transmitted to the network.

Although Stateless Auto Configuration is very attractive for wireless sensor networks, it is not suitable to be used with ROS and IPv6. It would make it necessary to manually enter every node at the local DNS server. It is therefore necessary to use the DHCPv6 protocol in networks, which are to use IPv6 together with ROS.

### 3.2.2. Dual-Stack Operation

#### On the Client Side

When addressing a host by its hostname, it is not clear if a connection should be made using IPv4 or IPv6. The hostname does not encode this information. Only a lookup of the hostname reveals if the host has an IPv4 address, an IPv6 address or, possibly, both.

This leads the establishing side of the connection with the choice of either trying to connect via IPv4 or IPv6. Yet, the existence of an IPv4 and an IPv6 address does not make it clear, whether the host is also listening for connections on either of these address. It is the responsibility of the client, to try and make a connection to one or both of these options. This is obviously limited by the availability of either IPv4 or IPv6 to the client.

When the client prefers one address type, it can try to establish the connection first using this type. If the connection attempt fails it can fall back to other type. The time needed to detect a a failed attempt to make a connection can be very short or range in the tens of seconds, depending on the configuration of the remote host.

If the client has no preference it can decide to make a connection to both addresses at the same time. The first connection which gets a positive reply from the remote host is then used to establish the connection.

This is in general only possible when using TCP to initiate the connection. As only TCP guarantees that a connection is created before actual data is transferred. If the first technique is used, the remote host should reply with an ICMP packet, stating that the port is closed, if there is no application listening on that address. But often these packets get filtered in firewalls or they are dropped on the way to the sender. When such a packet does not reach the sender, the sender has no way of knowing that the packet did not reach the remote host.

If the second technique is used with UDP, it is possible that the remote host receives the packet twice. Depending on the protocol used, it might not be able to recognize this situation and act in an unintended way. This can be overcome by designing the protocol in a way, that rejects duplicated packets, even if they are received from the same host via different IP versions.

#### On the Server Side

Most modern Operating Systems give developers the option to simultaneously listen on both, IPv4 and IPv6 addresses. This makes it possible to write applications that are agnostic to the way that is used to connect to them. This mode of operation is called dual-stack mode. When a connection using IPv4 is made in this mode, the IPv4 address of the client is mapped to a special IPv6 address, which represents the IPv4 address (IPv4-mapped IPv6 addresses, [46]). The application can use this address like a regular IPv6 address. The underlying operating system has to translate the mapped address back to IPv4, when data is sent back to the client.

This approach is now being considered as dangerous and there is dispute between different Operating System manufactures, if this behavior should be enabled by default or not. The problem arises, when the IPv4-mapped IPv6 address is transmitted to another application. Although the address appears to be a reachable IPv6 address, only the original application is in a position to map it back to an IPv4 address [47].

## 3.3. Multimaster Support

The ROS master node forms the central node of a ROS system. It contains the registry with the data of all other nodes in the system. In its current version, the ROS master also implements the ROS parameter server. This makes the ROS master node a single point of failure. If the master node is not reachable anymore by other nodes, no new connections can be made and the contents of the parameter server are not available to the nodes anymore.

There are multiple reasons, that can lead to such a situation:

- The machine, which runs the master node, is shut down.

- The master node application has crashed.
- The network connection to the master node breaks.
- The master node is attacked using a Denial-of-Service attack

Additionally in this topology, a single machine must handle all connection requests of all the nodes in the system. This can lead to a high computing and network load for a single machine inside the network, which can lead to unexpected delays in its operation as well as high network loads on the network links, connecting the machine to the rest of the network.

A common solution to this problem is to distribute the task of such a service over multiple machines, which are scattered over the network. Using this technique, a single point of failure is prevented and local connections to the distributed masters lead to lower loads on the network. Figure 3.1 shows how two ROS master nodes keep each other synchronized, which allows the nodes attached to one master node, to reach the nodes attached to the other master node.

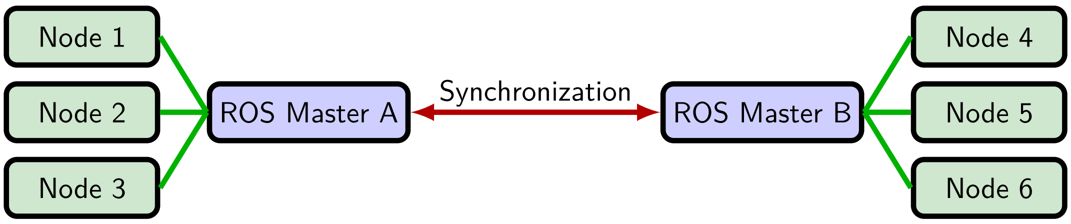


Figure 3.1.: Multiple master nodes share the task of managing the nodes.

### 3.3.1. Existing Systems

There have been some attempts to provide multimaster solutions for ROS. The most notable ones include the foreign relay concept and the multimaster implementation of the Fraunhofer Institute for Communication, Information Processing and Ergonomics (FKIE).

#### foreign_relay Node

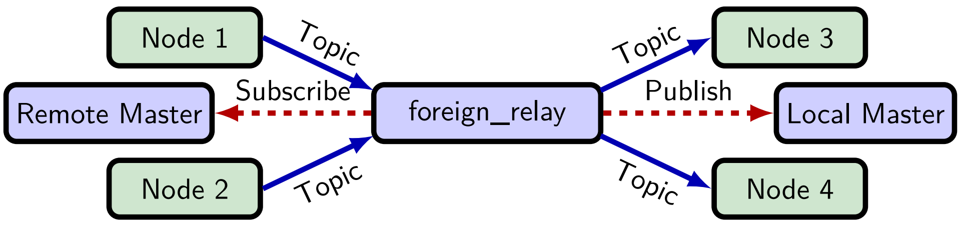


Figure 3.2.: The foreign_relay forwards a single topic from one ROS system to another ROS system.

The foreign_relay node is a node inside the multimaster_experimental package of ROS. It can subscribe to a single topic on a remote master and publish the data at the local master. Figure 3.2 shows the flow of information between two ROS systems, when using the foreign_relay.

The advantage of this solution is that the foreign_relay subscribes only once to the specified remote topic and distributes the data of the topic locally to multiple subscribers. This keeps the used bandwidth at the remote nodes minimal. A major disadvantage is the static configuration and the restriction to only relay a single topic.

### Multimaster Package

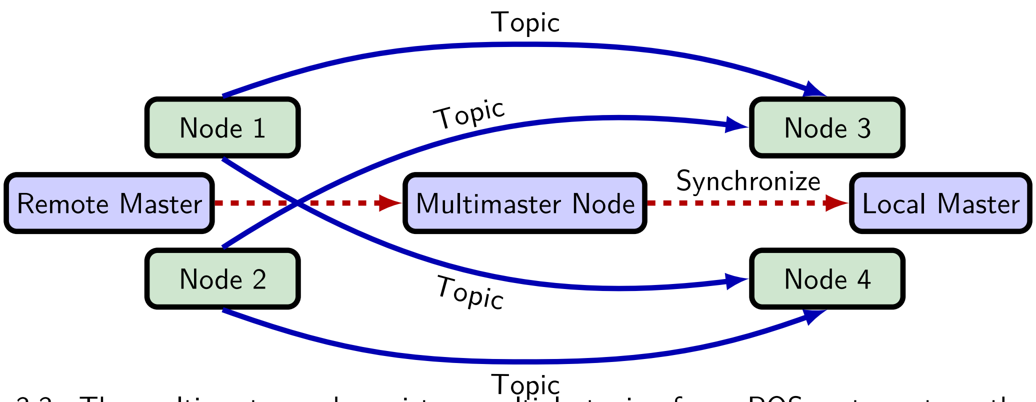


Figure 3.3.: The multimaster node registers multiple topic of one ROS system at another ROS system.

The multimaster package is similar to the foreign_relay node. It can register a set of local topics and services at a remote server and register another set of remote topics and services at the local server. As shown in Figure 3.3, it does not relay the data itself, but registers the the publishing nodes at the different master nodes.

The multimaster package has the advantage that it can subscribe to multiple remote topics and that it is also able to do the same in the reverse direction. Unfortunately its configuration is still static and it does not reduce the used bandwidth by relaying the topic locally.

### Multimaster System from FKIE

The multimaster system from FKIE implements full multimaster capabilities for the node, topic and service registry.

The system uses the concept of synchronization nodes. On every machine, that runs a master node, an additional synchronization node is needed. The synchronization node contacts other synchronization nodes in the network and uses them to retrieve data from their local masters.

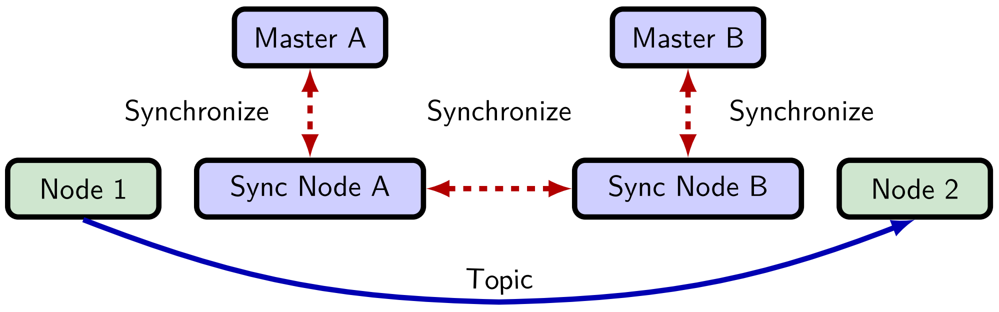


Figure 3.4.: The multimaster system from FKIE uses synchronization nodes to synchronize multiple masters nodes

To find each other inside the network, the FKIE implementation can either use IP multicasts or zeroconf networking. Zeroconf is a distributed alternative to the DNS system, with the option to discover other devices in the local network.

After the synchronization nodes have discovered each other, they register themselves with the remote synchronization nodes. The remote synchronization nodes then start to transmit changes of their master node to other registered synchronization nodes. Figure 3.4 shows the different connections, created by the system.

To get the current status of the local master node, the synchronization nodes start to poll their master node in a regular interval. It is not possible to request the whole state of the master node within a single request. Instead, the synchronization node has to make a separate request for every node and service, which is registered on the master. This can lead to a significant load for the master node. It also introduces an inherent latency between the time, a change is made on the master node and the time, the synchronization node starts to read the master state.

When a change is detected on the local master, the synchronization node sends all information needed to replicate this change on a remote master to the other synchronization nodes. They now use the ROS master API to replicate the change on their local master.

A node interested in a topic can now make a lookup at its local master and get the address of a potentially remote node, that is offering this topic or service. It then can use this information to open a direct connection to this remote node.

Another potential problem is the possibility of conflicting entries in the node namespace. If the local synchronization node tries to register a remote node under the same name as the local node, the local master sends a de-registration request to the local node. This can lead to sever problems, when there is no explicit regulation of the namespace.

### 3.3.2. Improved Multimaster Concept

Based on the experiences with the FKIE multimaster implementation, a new design has been developed to tackle the problems of the FKIE approach. These problems are a high load on the master nodes, latency and conflicting nodes with the same name connected to different masters.

#### Lowering the Load on the Master Nodes

To lower the load on the master nodes, a new interface to the master node is introduced. It allows other applications to subscribe to changes of the master node's state.

This eliminates the need to poll the master in a regular interval for changes. It also reduces the latency of the change notification significantly.

When subscribing to the changes for the first time, the whole state of the master is transmitted. To reduce the amount of data, that is transmitted for each subsequent change of the master, only differences to the last state are transmitted.

The synchronization nodes use this interface to get the status of remote master nodes. After discovering a remote master, they connect to it and register themselves for updates about changes of its state.

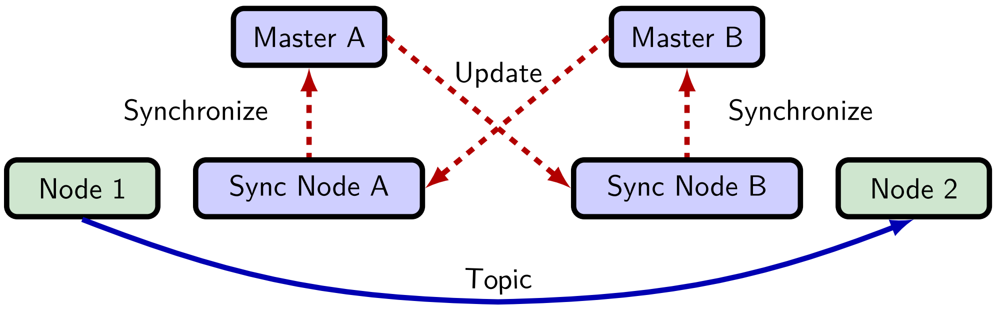


Figure 3.5.: Multimaster implementation with direct connections to remote master nodes.

This leads to the new network topology shown in Figure 3.5. There is no direct connection anymore between the different synchronization nodes. Instead there are now only connections from local masters nodes to remote synchronization nodes.

### Allowing Same Named Nodes on Different Master Nodes

To avoid cluttering the node namespace of all masters, nodes that are originally registered on a remote master node are put into a special region of the node namespace:

/remote/<remote master node name>

Figure 3.6 shows an example namespace with nodes having different locations in the namespaces of different master nodes. This is possible, since ROS does not transmit this information when one node contacts another node. It is purely used to identify a node at the local master node. The ROS Master API gives the synchronization nodes the freedom to register remote nodes at any location in the namespace.

```
/
└─ remote
│   └─ masterB
│   │   └─ sensor1
│   └─ masterC
│       └─ sensor1
└─ sensor1
```

Figure 3.6.: Example of a node namespace of a system with three master nodes. This is the node namespace seen from masterA. Although there are nodes with the same name registered at three different master nodes, there are no collisions in the namespace.

With a good choice of the names, the approximate location of a node in the system can be identified by looking at its path in the node namespace. If it is not in the /remote path, it is a local node located in the vicinity of the local master. If it is in the /remote path, it is located in the vicinity of the in the master node, mentioned in the path.

### Parameter Server Synchronization

The previous section covered the synchronization of nodes, topics and services. To synchronize two master nodes completely, the parameter server has to be synchronized too.

This poses a more complex problem than the synchronization of nodes, topics and service. Parameters are not linked to a specific master node. Therefore it is not implicit, from which master node a change to a parameter has originated. This can lead to situations where the change to a parameter starts to circulate endlessly in the multimaster system.

Another problem arises, when the same parameter is changed on two different parameter servers at the same time and then propagated trough the network. It is not clear, which version of the parameter will be stored in the end. It might even also lead to loops in the update mechanism. This can lead to severe problems in the consistency of the content of the parameter server.

These problems are already known in the field of (distributed) databases. It seems not advisable to try to implement a new database software for the parameter server. There are already software packages that can be used to create a distributed parameter server.

The current implementation of the FKIE multimaster does not support the synchronization of the parameter servers of the different master nodes. As the ROS community is currently discussing about moving the database of the parameter server to a redis [1] database, this thesis will not try to replicate the parameter server over different masters.

### Master Auto Discovery

Every node that wants to participate in the ROS network must know the hostname of one master node. It needs this information to open a connection to the master node and register itself with the master node.

When creating a larger network, with devices that might change their location inside the network, this can lead to a configuration problem. Every time the location is changed the nearest ROS master node should be used. This can only be done by changing the hostname of the preferred master.

One way to solve this, is to run a master node and a synchronization node on every machine, that runs normal ROS nodes and to choose the local host as the master node. The local master node then starts to contact other ROS master nodes in the network and connects the local nodes with the the rest of the network.

Although this is a working approach, it has several disadvantages:

- High computation load on the machine: Running a local ROS master and a synchronization node needs resources on the local machine. This can easily consume much more resources than the application node itself.
- High network load for the node: The local master node starts to open connections to all other master nodes in the network. This can lead to a high load for slow network connections.
- Cluttering the namespace: Each master node gets an entry in the /remote namespace of the other masters. This can lead to an unnecessary number of entries in the namespace.

To prevent these disadvantages, the concept of the proxy master is introduced. A proxy master is an application that runs on the same machine that also runs ROS nodes. The proxy master

[1] `http://redis.io/`

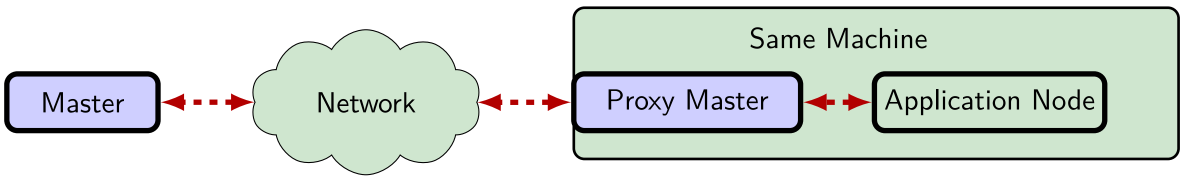


Figure 3.7.: The proxy master gives local nodes the option to automatically connect to an unknown master in the the local network.

presents itself as a ROS master node to the local nodes, running on the machine. When the first node connects to the proxy master, the proxy master looks for another master in the local network and forwards all data, coming from the local node, to the remote master node. As shown in Figure 3.7, the application node does not need to know the address of the remote master node.

This behavior has the advantage that the proxy master does not need to process the data coming from the node in any way. The proxy master just needs to forward the data to the remote master node and deliver the answer back to the local node. This behavior can be implemented with less resources than a complete ROS master node.

It also does not open connections to multiple other masters. Once it has selected one master in its vicinity, it only needs to open and close connections to it, when a local node wants to interact with its master.

The nodes will also appear to be connected to the remote master node and will therefore appear at this path in the namespace. This keeps the namespace clean and more useful.

The disadvantage of this concept gets clear, when the machine, running the proxy master, is moved to another location in the network. As the proxy master has no understanding of the ROS protocol, it can not move the connection of its local nodes to another master node. The proxy master must still forward all communication to the master node, it originally chose.

The proxy master seems to be an interesting solution, to minimize the amount of configuration needed, to get a ROS node up and running.

## 3.4. Namespace Regulations

The naming of nodes, topics and services in ROS is not governed by any specification or regulation. The names can be chosen according to what ever the designer of the nodes thought was best.

This can lead to problems, when designing a system in which many different devices are supposed to act together. There is a significant risk of two nodes trying to register two different topics under the same name. The naming of the nodes and topics would probably also not yield any meaning, leading to a cluttered namespace.

It is therefore beneficial for larger systems to have some rules, regulating the use of the namespace. Since there is probably no single set of rules, which will satisfy the requirements of all possible situations, the rules should be exchangeable between different systems.

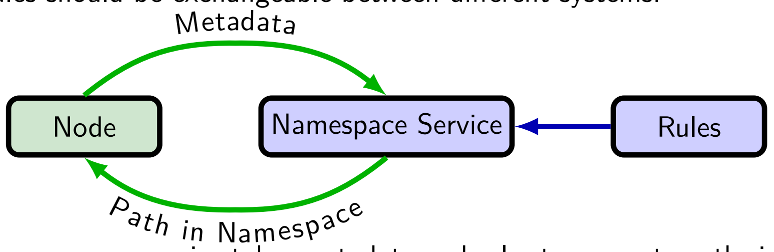


Figure 3.8.: The namespace service takes metadata and rules to generate paths in the namespace.

To provide the system with flexible rules to organize the namespace, a new service is introduced. This service accepts metadata from the nodes. This metadata contains an arbitrary set of parameters about the node and its topics and services. The newly introduced service then applies a set of rules to this metadata to generate paths in the namespace, that should be used by the node. Figure 3.8 shows how the service combines the metadata from the node and the rules, to form a path inside the namespace.

This system leads to a dynamic namespace, which can be shaped according to the needs of the local applications.

## 3.5. Bandwidth Reductions

The way ROS has implemented its publish/subscribe system makes it very robust against network failures and failing master nodes. This is caused by the direct connections that are opened from the subscribers to the publishers, with no other nodes in between.

Such an approach also causes a network load for each publisher which is proportional to the number ob subscribers it has. This can lead to problems with typical applications for Intelligent Environments. Here, many different applications might be interested in the topic of a single sensor.

When introducing the multimaster environment as outlined in section 3.3.2, there is additional overhead from the fully meshed network between the different master nodes. For each change in the state of a master node, a message is transmitted to every other master in the system. The amount of data and messages that have to be transmitted by each master therefore scales directly with the number of master nodes in the system. Figure 3.9 shows a system with three master nodes, where each master nodes as to update two synchronization nodes.

To reduce the amount of connections and bandwidth used by the system, it is beneficial to aggregate messages that are traveling along the same path and contain the same data.

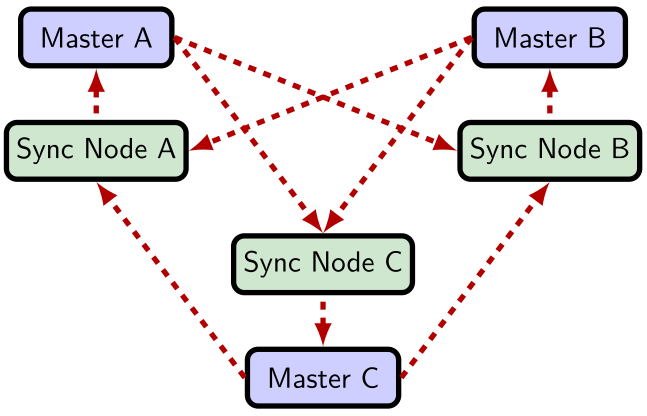


Figure 3.9.: Each master node has to update every other synchronization node.

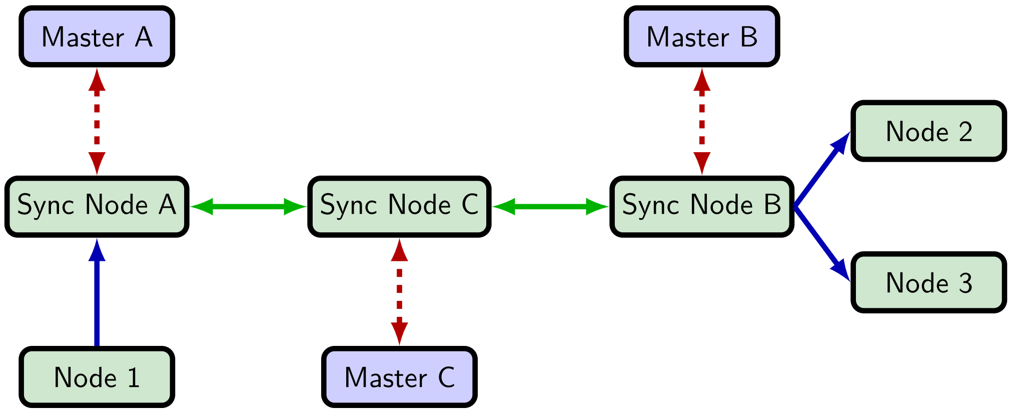


Figure 3.10.: In a routed multimaster setup, less connections are needed and the routed network can be used to forward topics.

This can be achieved by forming an overlay network, which is super imposed onto the actual network infrastructure. This network is formed by the connections between the different synchronization nodes. These connections are used to aggregate messages from other nodes and deliver them to their destination. As shown in Figure 3.10, the synchronization nodes start to publish the topics of other remote nodes in the system and subscribe to them when needed.

In this configuration, many nodes, connected to different masters, can connect to a remote node, without creating a high network load for the remote node. The messages from the remote node are routed in an efficient way between the different ROS networks and do not need to be transmitted redundantly. In addition, this routed network can be used to pass along changes of the state of the connected masters.

Although this can lead to enormous reductions in the bandwidth used by topics and the multimaster state transfer, it does not reduce the bandwidth needed to execute calls to ROS services. These must still be answered individually by the nodes. Services are therefore still registered with the remote nodes as the providers of services.

### 3.5.1. Hybrid Approach

While the approach with a routed network between synchronization nodes gives improvements in latency with low speed links and used bandwidth, the meshed approach has a lower latency with high speed links and better reliability. This leads to a concept which combines the two approaches into a hybrid approach. This approach uses a routed overlay network between different sites connected by slower links, and the meshed network inside regions which are connected by high speed local network links. It allows to connect multiple ROS networks via the Internet, wireless links or a corporate network.

## 3.6. Security

In 2012 the ROS industrial consortium[2] has been announced. The goal of this consortium is the promotion of ROS in industrial environments. Despite the possibility that this could lead to a widespread usage of ROS outside of research environments, ROS does not have any security mechanisms in place. In particular the following aspects of ROS leave it completely unprotected in a normal network:

- All messages between ROS nodes are unencrypted.
- There are no access restrictions for any part of the system.
- Anyone who knows the address of the ROS master can get full access to the system.

This becomes a severe problem, when making a ROS master accessible from the Internet. In a multimaster system, only a single master has to be accessed from the outside, to get almost full control over the ROS network. An attacker can perform the following actions:

- Subscribe to any topic inside the ROS network.
- Publish to any topic of the complete ROS network.
- Get full access to the parameter server of the local ROS master.
- Impersonate and shutdown any local node by creating a new node with the same name.
- Call any service inside the ROS network.

In the context of Intelligent Environments, these actions can be used to cause severe system malfunctions and breaches of privacy.

[2]ROS Industrial Consortium, `http://www.swri.org/4org/d10/msd/automation/ros-industrial-consortium.htm`

As ROS is mainly used in research environments, extensibility, performance and ease of administration is one of its key strengths. These goals can get very hard to achieve, when introducing security concepts. They can easily lead to massive administration overheads and performance losses.

While performance losses due to the needed encryption of data might be acceptable to gain more security for the system, the administration overhead might prove more severe. In its current state, ROS requires almost no configuration of the ROS master and only very little configuration for each node.

ROS is based around an anonymous publish/subscribe system. Nodes are expected to accept messages on specific topics and should not make their behavior dependent on the publisher of the message. The publishers are also expected not to change their behavior, depending on the nodes, receiving their messages. In the case of a multimaster system, they might not even have the possibility to access this information. This would make it the responsibility of the master node, to define which node is allowed to publish to a specific topic or listen to it.

### 3.6.1. Encryption of Node to Node Messages

ROS nodes communicate directly with each other. A publisher opens a connection to each subscriber. To form the basis of a secure system, the nodes have to be able to exchange encrypted messages. Since subscribed nodes might loose their authorization while being subscribed, no system wide key can be used, and each connection must be encrypted with a separate key. To secure each connection with a separate key, asymmetric cryptography can be used. For example, the Diffie-Hellman key exchange algorithm [48] is able to securely transmit a key between two parties over an insecure network. With this algorithm, publisher and subscriber can agree on a key which can be used afterwards with a symmetric algorithm.

This can provide the publisher and the client with a key to use, but does not prevent an attacker from performing a "man in the middle" attack, where the attacker poses as the subscriber towards the publisher and as the publisher towards the subscriber. This can only be prevented if the publisher and the subscriber have a way to verify that the data originally came from their counterpart and has not been altered.

Another option to securely transmit messages from the publisher to the subscriber is to use public key cryptography like RSA. In this scenario, the publisher and the subscriber first exchange their public keys. The publisher then uses its private key to sign messages and the public key of the subscriber to encrypt them. Only the subscriber can then decrypt the message with its private key and can check the integrity of the message with the public key of the publisher. This has the advantage that the public keys are public information and do not need to be kept a secret. Since anyone can publish a public key for another node, a system has to be put in place, which prevents attackers from injecting public keys for other nodes. Another disadvantage is that asymmetric cryptography is considerably slower than symmetric cryptography.

To prevent an attacker from providing the publisher with a fake public key for another node, the nodes have to be able to verify that a given public key is valid. This is can be achieved by including a trusted third party (a certificate authority), which signs the public keys of the nodes. Its public key can then be used to verify an unknown public key of another node. For this several steps have to be performed:

1. The node has to generate a private/public key pair.
2. The node securely transmits its identity (composed by its public key and identity information) to the certificate authority.
3. The certificate authority signs the identity with its own private key and sends the signed identity together with its own public key back to the node.
4. The node signs the received public key of the certificate authority with its own public key and sends it back to the certificate authority.
5. The certificate authority checks that the node received the correct public key and sends its acceptance over a secure channel back to the node (This prevents the node from receiving an illegal public key of the certificate authority).

The key aspects of this procedure are the initial transfer of the identity of the node to the certificate authority and the transfer of the acceptance back to node. These steps must be performed using a secure channel which can not be intercepted.

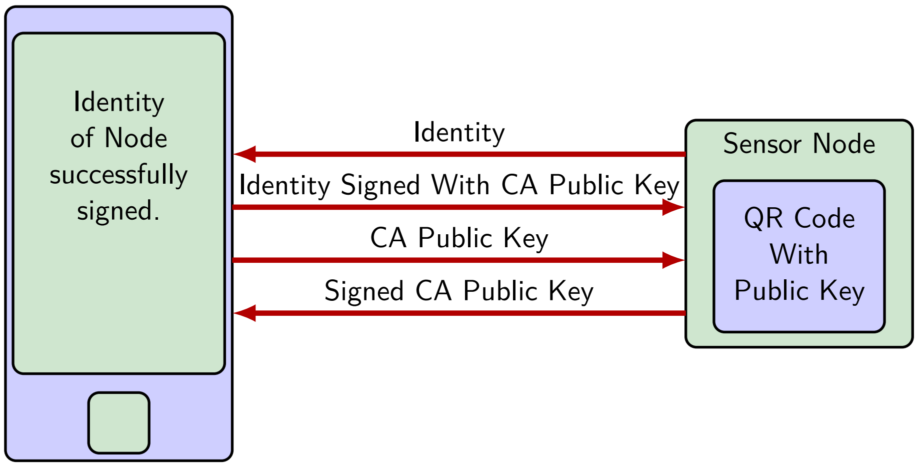


Figure 3.11.: The smartphone signs the identity of the node and shows the user if the process did succeed.

Figure 3.11 shows an example, where the mobile phone of a user serves as the certificate authority. It can use its camera to scan the node's public key from a QR Code or read it from an NFC tag.

This auxiliary channel [27] is used to securely transfer the key between the two unauthenticated devices. The node can then send an identity, signed with its private key, to the phone. The phone can check if the identity belongs to the node and sign it. It then can display to the user if the node received the correct public key and instruct the user i.e. to press a button on the node or shake it a number of times. This can be used by the node to verify the correct reception of the public key.

After the keys have been exchanged and signed, the node can verify the identity of other nodes and send its identity to other nodes:

1. The subscriber sends its identity to the publisher.
2. The publisher verifies the identity with the public key of the certificate authority, stores it, and sends its own identity to the subscriber.
3. The subscriber checks the identity of the publisher and stores it.

After the identities have been exchanged they can be used to securely exchange messages:

1. The publisher signs the message with its own private key and encrypts the signed message with the public key of the subscriber.
2. The subscriber decrypts the key with its own private key and checks the signature with the public key of the publisher.

While this method does work, it also needs both sides to execute two asymmetric cryptography steps for each message. To still be able to use a symmetric algorithm, the publisher can generate a key and send it to the subscriber as the first message. Afterwards this key can be used for communication between the publisher and the subscriber with a symmetric algorithm.

To prevent replay attacks, the messages should contain a counter, which can be used to discard messages which have been captured and resent by an attacker. The symmetric key should also be changed on a regular basis.

### 3.6.2. Authentication and Authorization

Apart from setting up a secure channel between the publisher and the subscriber, a system is needed to manage which nodes are allowed to access which topics.

One way to achieve this is to again use the certificate authority to sign a list of topics to which a node can publish or subscribe to. By providing other nodes with parts of this list, they can decide if they will accept messages from this node or send messages to it.

#### Issuing Access Grants

The smartphone of the user can again play a central role in this scenario. The smartphone can receive requests for rights from the nodes. These requests include a list of requests for rights to access specific paths in the namespace. The smartphone can then either decide on its own, if the node is allowed to access these topics, or prompt the user to allow access to specific topics. After a decision has been made, the phone can send a list of signed access grants back to the node. The node can now use this list, to authorize itself when accessing a topic covered by an access grant in the list. The access grant must be sent to another node, when the two nodes try to exchange data on this topic for the first time. The access grant of the remote node can then be stored and the signature of the certificate authority does not need to be checked again, thus saving bandwidth and power.

#### Revoking Access Grants

A node which receives a signed access grant can use this grant anytime in the future. In the meantime the certificate authority might decide to revoke this grant. To communicate a revoked access grant a revocation list, which contains a reference to every revoked access grant, can be kept in the system. The reference can consist of a hash of the access grant. The revocation list can either be kept with the certificate authority and get queried each time, a topic is requested, or regularly get transmitted to the nodes in the system. In the later situation, only references to revoked access grants which contain rights to which a node has access to, need to be sent to this node. As the smartphone of the user might not be able to send this list to every node, the ROS master node can receive the list and make it available.

Using this list of revoked access grants, a node can quickly decide, whether it will accept a connection with another node, which is trying to publish or subscribe to a specific topic.

Another option is to couple access grants with an expiration date. After this date, the access grant looses its validity and a new one has to be requested from the certificate authority again. As the smartphone of the user might not be available every time, this task should be performed by the ROS master node. It can sign an access right again if it has not been revoked. To perform this action it also needs access to the private key of the certificate authority and a list of revoked access grants.

### 3.6.3. Implementation with Wireless Sensor Nodes

This system provides basic security mechanisms, which heavily rely on the use of asymmetric cryptography. Although asymmetric operations are only executed, when a new connection between two nodes is established, they still present a heavy computational burden for small devices.

Currently systems which use elliptic curve cryptography [49] can be implemented for 32 Bit micro controllers using a few kB of RAM and ROM. While these specifications might not yet be the baseline for nodes in Wireless Sensor Networks, technological advancements will push these devices into the reach of cryptographic systems like this.

In the meantime, the boarder router which is used by most Wireless Sensor Networks, can be used to terminate all asymmetric cryptography and only use symmetric cryptography inside the Wireless Sensor Network as already mentioned in section 2.2.1.

## 3.7. Privacy

The anonymous publish/subscribe nature of ROS does not interact well with the requirements of a system which protects the privacy of its users. Even with the addition of a security concept, nodes with access to a specific topic experience no limitations regarding the data transmitted via this topic or under which circumstances the data can be used.

What the nodes in the ROS system ultimately do with the data is highly application specific and depends on the transmitted data. Although nodes can be implemented not to use information that is not intended for them, a much stronger approach is to not transmit this data in the first place. Access grants mentioned in section 3.6.2 can be used to achieve this. By specifying conditions inside an access grant, a publisher can decide if a subscriber has the rights to access a specific message. Examples for such conditions can be the current time or the content of a specific field in the message. Additionally, the access grant can specify that certain fields in a message must be cleared or somehow filtered before the message can be transmitted to the subscriber. For example could the position of a user be sent with a different accuracy to a node which regulate the heating of the building, than to a node which displays current events on a display near the user.

By carefully issuing such access grants and designing the structure of published messages, the flow of information can be tailored to the privacy needs of the users. The way in which the system is configured might be the most critical aspect of it. Systems in which this step proves too difficult for the users might end up either being rejected completely or not being configured properly, resulting in loss of privacy or a dysfunctional system.

By using a system with conditional access grants, the user needs to have a fairly good knowledge of the structure of the messages and what kind of information a topic actually carries. To simplify this step, standards have to be found, which can be used to provide the user with a more abstract view on how his data is used by the system. This step might prove even more difficult than the actual technical implementation of conditional access to topics.

# Chapter 4.

# Implementation Details

This chapter covers the implementations of the concepts mentioned in chapter 3.

## 4.1. IPv6

ROS encapsulates almost all access to the actual network inside its client libraries and the implementation of the maser node. The libraries used for almost every node are written for C++ and Python. The master node and supporting libraries are written in Python.

Both of these languages already have support for IPv6, but it is necessary to tell the application to use IPv6 instead of IPv4 for its operations. The concept of sockets is used to handle network communication. The dominant implementation of sockets follows the BSD socket API. This API is written in the C programming language. ROS directly uses this API in its C++ client library. The Python programming language uses the same socket API as its underlying network library and provides the programmer with almost the same API, as is used in C and C++. The next part of this section will therefore only provide references to the C++ implementation of the IPv6 implementation. The changes needed for the Python implementation are analogous to these changes.

### 4.1.1. Selection of Networking Mode

Changing the structure of the low level networking code poses a big chance of introducing subtle and hard to track down errors into the code. This might lead to a problem when, trying to get these changes applied to the officially distributed version of ROS.

To increase the chance of getting the changes accepted by the maintainers of ROS, it was decided to disable the new functionality by default. This makes it easier to keep the exact same behavior of the system, when no IPv6 support is needed. To activate the IPv6 portion of the code, an environmental variable needs to be set. If this variable is set, the master node starts to accept IPv6 connections and the nodes start to try to establish connections using IPv6.

### 4.1.2. Storing IPv6 Addresses

As IPv6 addresses are four times larger than IPv4 addresses, new and larger data structures are needed to store these addresses. To prevent the code from needing two different sets of data structures, new storage structures are used. These storage structures can either store IPv4 or IPv6 addresses and have an additional parameter, setting the actual type of the address.

### 4.1.3. Listening for Connections

To listen for new connections, a socket has to be created and put into a listening state. When creating a socket, it has to be specified, if the socket is an IPv4 or IPv6 socket and on which address it should listen. The choice of the protocol type is made by evaluating the environmental variable. The socket is created for a specific address family by passing either the constant AF_INET (for IPV4) or AF_INET6 (for IPV6) during its creation. ROS already has some logic, to decide on which address it should listen for new connections. This address can also be overridden by environmental parameters. When no special address is specified, ROS tries to bind the socket to any interface or address that the host currently has. To signal this to the socket, the IPv4 address '0.0.0.0' or the IPv6 address '::' has to be passed to the socket. ROS has been modified to choose either one of these two addresses, based on the state of the environmental variable.

#### XML-RPC and IPv6

The current implementation if the Python library for XML-RPC does not support the creation of an XML-RPC server which is listening for incoming IPv6 connections. To support this, the constructor of the 'XMLRPCServer' class had to be duplicated and changed to support IPv6.

### 4.1.4. Resolving Names

When opening a connection to another node, first its name must be resolved to an IP address. This is usually done with the gethostbyname() function. It returns a list of IPv4 addresses, which can be used to contact a host. To also support IPv6 addresses, the new getaddrinfo() function has to be used. It returns a list of IPv4 and IPv6 addresses, that can be used. The code then has to loop through these addresses and decide, either to use the IPv4 or the IPv6 address of a host.

### 4.1.5. Opening Connections to Other Nodes

When a new connection is opened, a new socket has to be created and instructed to make a new connection to a specific IP address. The application also has to make sure, to provide the correct type of address to socket.

## 4.2. Multimaster

The implementation of the multimaster system for ROS makes use of many ROS components. This allows it to adapt to possible changes of ROS in the future, without needing major redesigns of the architecture.

Two different kinds of multimaster systems have been implemented. The fully meshed type for use in LAN systems and the routed type for use over single connections with limited bandwidth.

As mentioned in section 3.3.2 both are based heavily on the concept of the multimaster system from FKIE. To implement an interface that allows to subscribe to the state of the master node, the API of the master node has been extended.

The master exposes its API via the XML-RPC protocol and is written in Python. To extend the functionality, new methods need to be defined and exported via the XML-RPC interface.

The master node keeps an internal list with all subscribed nodes. When the state of the master changes, it goes through every entry in this list. When the list of publishers for a certain topic changes, the master sends an update to all nodes, which are subscribed to this topic. This is done by calling the XML-RPC interface of the subscribing nodes.

A stale, crashed or unreachable node could potentially cause a major holdup, if all hosts would be processed sequentially. To prevent this, the master node keeps a pool of threads, each of these threads handles a single connection. This significantly reduces the latency from the occurrence of a change in the master to the report of a change, if a few of the connected nodes can not be reached.

The state of the master can change rapidly when starting new nodes on the local machine. The rate of these changes can potentially be much faster than the master can send updates to a remote node. It therefore caches all changes to its state until the last update of its state has been registered by the subscribing nodes, reducing the number of updates and the overall latency, when running a complex system.

This makes it easy for a node to keep track of the state of a master node, after it has registered itself with the XML-RPC API of the master node. However, it does not cover the possibility of the master node crashing and restarting. In this event, the subscribing node does not get called again with updates to the state of the master. To handle this situation, the master node keeps a counter of changes to its internal state. This counter called the version of the master node is incremented with each change that is made to its state. The master version is embedded in every update that the master node sends to the subscribed nodes. These nodes compare the version of the update with a version of the master, which they obtain on a regular basis separately from the updates sent by the master node. How the subscribing nodes obtain the current version of the master is depending on the actual implementation of the multimaster system and is covered later. When the version obtained through updates from the master does not change during a certain period of time and is not equal to the version obtained separately, the connection to the master is deemed broken and the node subscribes again to changes of the master state. Through this newly opened connection, the node can once again obtain the full current state of the master node and receive the same version through the update, as it has obtained through the separate channel.

When the state of the master node changes, it calls the update() operation of each subscribed synchronization node with a list of changes.

### 4.2.1. Fully Meshed Multimaster System

The fully meshed multimaster system is intended to be a system with zero configuration requirements. This is achieved using UDP multicasts to announce the presence of master nodes in the local network. On each machine which is running a master node, an extra synchronization node is started. The synchronization node starts to announce the existence of its local master. It sends this data in a regular interval in UDP packets to a known IPv6 multicast address. The scope of this address is limited to the local segment of the network. It also subscribes itself to this multicast address, to receive the announcements of other synchronization nodes.

The announcement packets contain the URL of the XML-RPC interface of the master node and the version of the master node. The synchronization nodes use the URL information to subscribe to the state of the remote master node. After the first connection has been established, they use the value of the master version as a heart beat of the remote master node and to check if the subscription to the state of the master node is still working.

This forms the fully meshed network, where each synchronization node connects to each remote master node on the local network.

## 4.3. Proxy Master

The proxy master is implemented as a small Python program which listens on the local loop-back interface for incoming connections. It forwards these connections to one of the master nodes which are advertised by the synchronizations nodes of the fully meshed multimaster system.

During its runtime, the proxy master keeps a list of all advertised masters and sorts this list depending on the link quality to the synchronization nodes. The quality is determined by the rate of received announcement packets over a time window. Synchronization nodes which have poor connections will show packet loss and get sorted to a lower position in the list.

When the first local node tries to contact the proxy master, the proxy master forwards this connection to the first master node in its list. It stores the combination of the connection to the local client and the connection to the remote master node in a list and uses this combination for all further communication with these two nodes. As the proxy master can not distinguish between different local nodes, from now on it will always use the master node, it chose the first time, for all future connection attempts by local nodes.

## 4.4. Namespace Rules

For the namespace service, a method is needed to generate paths in the namespace from the metadata of nodes. A technology is needed to transform data according to a set of rules. For this task, the XSLT technology seemed to be the best fit.

An XSLT processor uses an XML stylesheet containing rules and an XML document with data and outputs the transformed document, applying the stylesheet to the document. The output of the XSLT processor is not limited to XML and can be any format, specified by the stylesheet.

To use this technology for generating paths in the ROS namespaces, a set of rules has to be developed in the form of a stylesheet and the metadata of the nodes must be available as an XML document.

ROS already uses XML documents to describe the configuration of a node. These files are called .launch files. Besides allowing the specification of the the name of the node and the parameters to be loaded into the parameter server, they also have configuration options to remap topics. When a topic gets remapped, the ROS libraries will change every occurrence of this topic in the node with its remapped value. This is done during runtime and allows to reconfigure the topics to which the nodes are publishing to or subscribing to.

By adding the metadata of nodes to their .launch files, the XSLT stylesheet can transform the metadata in the .launch files into remapping directives. The stylesheet can also change the name of the node and its place in the ROS namespace.

XML itself supports namespaces. With namespaces, tags can have the same name, without confusing programs parsing the document, since the tags are in different namespaces. To embed arbitrary additional data in a ROS .launch file, every additional metadata is put into a special namespace which is recognized by the namespace service.

As the node starts to execute after its .launch file has already been processed by the ROS libraries, the transformation of its .launch file must happen before it gets executed. This is achieved by a small helper application that accepts a .launch file with embedded metadata, sends it to the transformation service and then launches the node using standard ROS procedures.

Another option is to load the metadata of a node as a parameter of the node into the parameter server during start-up. This has the advantage, that other nodes can inspect the metadata of this node. In this case, the node has to call the namespace service on its own, every time it tries to publish or subscribe to a topic. The complete metadata of the node is send to the namespace service, together with the name of the topic, that the node wants to publish. The service will then reply with a single text string, which specifies which new name should be used for this topic. A disadvantage of this approach is that the node can only contact the namespace service, after it has initialized itself and has already assumed a location in the node namespace.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<node>
  <owner>schneider</owner>
  <name>htc_desire</name>
  <group>smartphones</group>

  <topic>
    <name>accelerometer</name>
    <function>sensor</function>
    <mobile>true</mobile>
  </topic>

  <topic>
    <name>temperature</name>
    <function>sensor</function>
    <location>building1/room3</location>
  </topic>

</node>
```

(a) Example node metadata.

```
/smartphones/schneider/htc_desire/sensors/accelerometer
```

(b) A transformed topic name.

Figure 4.1.: Example of transformed node metadata.

Appendix A shows how to use this system. The rules provided in the appendix consider the location and class of a device, to generate paths inside the ROS namespaces. Figure 4.1 shows example metadata provided by a node and a generated path for a topic of this node.

## 4.5. Routed Multimaster System

To reduce the bandwidth used by the fully meshed multimaster system, and to be able to reach beyond the local network, where the automatic discovery with multicasts is not possible anymore, the routed multimaster system is developed.

The system changes the way in which the states of the different masters are synchronized and introduces the possibility to route topics through an overlay network.

After start-up, a routed synchronization node registers itself to the status changes of the local master node. The same protocol is used as mentioned in section 4.2.

To be able to reach synchronization nodes which are not located in the local network, multicast packets can not be used. They will most likely not be forwarded beyond the local network. Only directly addressed unicast packets are usually allowed to pass the boundaries of networks. The synchronization nodes of the routed multimaster system use unicast UDP packets to make other synchronization nodes aware of their existence. This requires at least one out of two nodes to have the address of the remote node, to open a connection to the remote node. A list of nodes to which a synchronization node tries to open a connection has to be supplied via a command line option.

After two synchronization nodes have learned of their existence, they register a topic with their local master, which will be used to carry messages to the remote synchronization node. These topics are named after the following schema: /multimaster/{local hostname}-to-{remote hostname}. They then register the topic name which is used by the remote host at their local master and set the URI of the remote synchronization node as the publisher of this topic. This allows the two synchronization nodes to subscribe to this topic and use standard ROS messaging to communicate with each other. This method was chosen instead of a direct TCP connection between the two nodes to use the already well working APIs of ROS and the possibility to easily get access to the data with other nodes.

The two synchronization nodes now synchronize which topics are published at their local master node. After they receive this information, they start to publish these topics at their local master.

When a local node start to subscribe to one of the topics published by the synchronization node, the synchronization node sends a subscription message to all other synchronization nodes to which it is connected. They will also relay this message to all other connected synchronization nodes, propagating the request through the whole overlay network. If there are local publishers of this topic, the synchronization nodes subscribe to this topic locally.

When a local publisher publishes a message and the synchronization node is subscribing to this topic, the synchronization node forwards the message to all other synchronization nodes that are subscribed to this topic. If a synchronization node receives such a message, it publishes this message if local nodes are subscribed to the topic. It also forwards the message to all other synchronization nodes, that declared interest to this topic.

After all local subscriptions to a topic have ceased and no other synchronization node has declared interest for this topic, the synchronization node stops to request this topic from other remote synchronization nodes. This way only nodes which need to relay messages for other nodes still have to forward these messages.

ROS protects the message type of topics with a checksum. This checksum might not be known to all synchronization nodes, when starting to publish the topic to other local nodes. Therefore the messages exchanged between the synchronization nodes also contain the checksum of the type of the message, which is embedded. The synchronization nodes can use this information to publish the embedded message locally, without knowing the structure of the message type.

Although this system offers several advantages over the fully meshed system, it can not reproduce every aspect of a normal ROS system. The system does not register the remote nodes at the local master node and every message from a remote node seems to originate from the synchronization nodes. While this is a change in how the ROS systems behaves, it should not alter the functionality of a well designed system, since the ROS documentation discourages the use of information about the origin of a message. Another feature of ROS, that can not be reproduced fully, are latched topics. When a node publishes a latched topic, every new subscriber will automatically receive the last published message, when a new connection is opened. This can only partly be reproduces, as the synchronization node can not send multiple messages to a newly connected node. Instead, it has to send latched topics from other remote nodes as regular messages to the local subscribers. This can lead to local nodes receiving the same latched topic more than once. Depending on the implementation of the local nodes, this might change their behavior.

Appendix B shows how to setup a routed multimaster system. Appendix C shows how to setup a mixed system with routed and unrouted multimaster systems.

# Chapter 5.

# Evaluation

To evaluate the different parts of the system, they were set up on different computers running the Ubuntu 12.04.1 LTS operating system. Memory usage, latency and bandwidth usage were measured and are compared in the following sections.

## 5.1. Memory Usage

The memory usage of a ROS system is especially interesting, when looking at the concept of the proxy master. One of its main purposes is to reduce the amount of used memory compared to a full ROS master node implementation with a running synchronization node.

When running a full multimaster ROS system, 54.5 MB of RAM is utilized. This includes the roscore, rosmaster, rosout and multimaster processes. For every node which connects to the master, about 4.5 kB of additional RAM is used.

When running a proxy master, 8.9 MB of RAM is utilized and no additional RAM is used with multiple nodes using the proxy master.

This equals to a reduction of 84% in RAM usage.

## 5.2. Latencies and Bandwidth Usage

To measure the latencies, introduced by the routed multimaster system, a test bed was set up, using 8 BeagleBoards and 3 PandaBoards. These small computer boards are equipped with an ARM Cortex-A8 (BeagleBoard) or an ARM Cortex-A9 (PandaBoard) and operate with a 1 GHz clock. The BeagleBoard has 512 MB of RAM, the PandaBoard has 1 GB of RAM. Both boards have a 100 MBit Ethernet connection, which was used during the following measurements.

Timestamps generated by nodes on the board could not be used, to accurately measure the time a message need to be transferred from one board to another board. This was not possible, since the clocks of the different boards have large tolerances and time service protocols like NTP were not able to provide a jitter free clock with the required accuracy. To still be able to measure the timing of the sending and receiving events, the nodes changed the status of a spare digital output on the boards, when a new packet was sent or received. A logic analyzer was used to capture the events without introducing additional jitter into the measurement.

Multiple Ethernet switches were used and care was taken to not transfer the same message multiple times over the same link, to prevent the switches from becoming a bottleneck.

To simulate long range, low bandwidth links, four wireless routers were configured to act as two bridges to limit the bandwidth of the link and to introduce additional latencies. For this purpose, four TP-LINK TL-WR703N devices, running OpenWRT, were used.

To test the multimaster system, the size of the packets and the frequency of the transmissions was varied in exponential steps.

### 5.2.1. Direct Connections in a 100 MBit/s Network

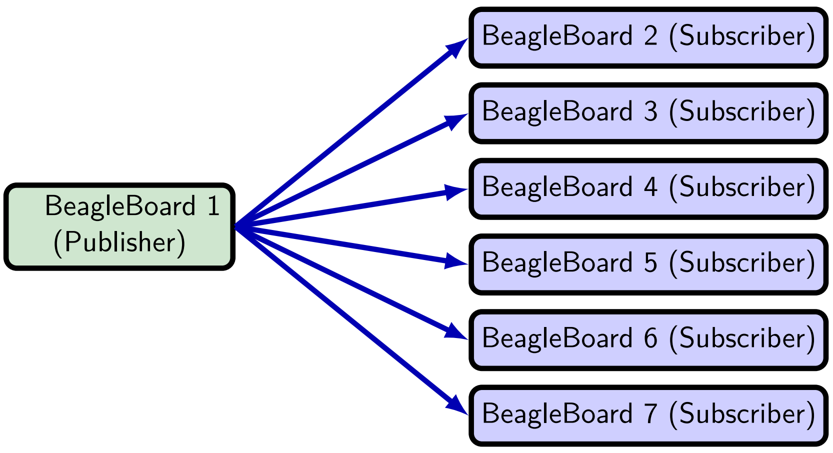

BeagleBoard 1 (Publisher)
BeagleBoard 2 (Subscriber)
BeagleBoard 3 (Subscriber)
BeagleBoard 4 (Subscriber)
BeagleBoard 5 (Subscriber)
BeagleBoard 6 (Subscriber)
BeagleBoard 7 (Subscriber)

Figure 5.1.: Test setup for direct 100 MBit/s evaluation: Six BeagleBoards subscribe directly to one publisher.

During this test one BeagleBoard published its messages directly to 6 other BeagleBoards. All boards were connected with a 100 MBit/s Ethernet switch. Figure 5.1 shows the logical test setup.

Figure 5.5 shows the measured latencies. The graph shows the mean time it took for a message to arrive at all boards. The graph also shows the time span in which 96 percent of all messages were received.

For small messages, the latency does not fall bellow 2 ms. When the message size grows larger than 1000 Bytes, the latency starts to reach 10 ms. This is a result of the maximum size of one

Ethernet frame of 1500 Bytes. Messages which are larger than this need to transmit at least two packets.

At a rate of 1 message per second, the largest message that could be transmitted had a size of 1 MB.

### 5.2.2. Indirect Connections in a 100 MBit/s Network

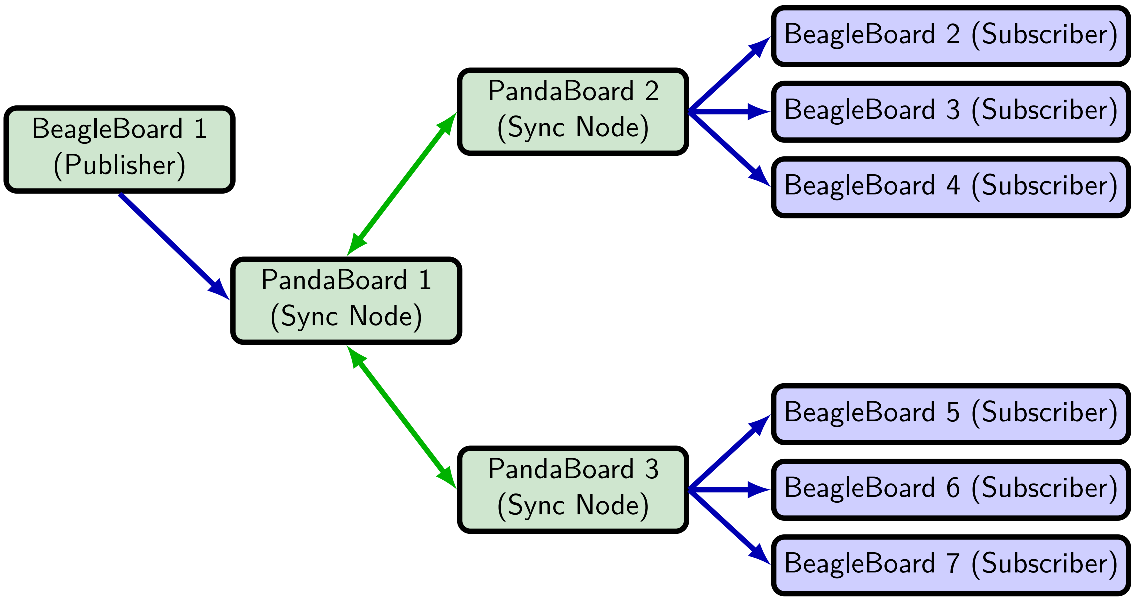

BeagleBoard 1 (Publisher)
PandaBoard 1 (Sync Node)
PandaBoard 2 (Sync Node)
PandaBoard 3 (Sync Node)
BeagleBoard 2 (Subscriber)
BeagleBoard 3 (Subscriber)
BeagleBoard 4 (Subscriber)
BeagleBoard 5 (Subscriber)
BeagleBoard 6 (Subscriber)
BeagleBoard 7 (Subscriber)

Figure 5.2.: Test setup for indirect 100 MBit/s evaluation: The three synchronization nodes route the messages to the nodes.

During this test the direct connections from the publisher to the subscribers was replaced by a network of synchronizations nodes. Figure 5.2 shows the logical test setup. The synchronization nodes were formed by the PandaBoards. One synchronization node subscribed to the messages of the publisher. It forwarded these messages through the routed multimaster network to the remaining two synchronization nodes. Of these, each published the messages to three local subscribers.

Figure 5.6 shows the resulting latencies. The average latency for small messages does not go below 7 ms. When sending messages larger than 10 kB, the latency starts to increase above 10 ms.

The largest message that could be transmitted at 1 message per second was 2 MB big.

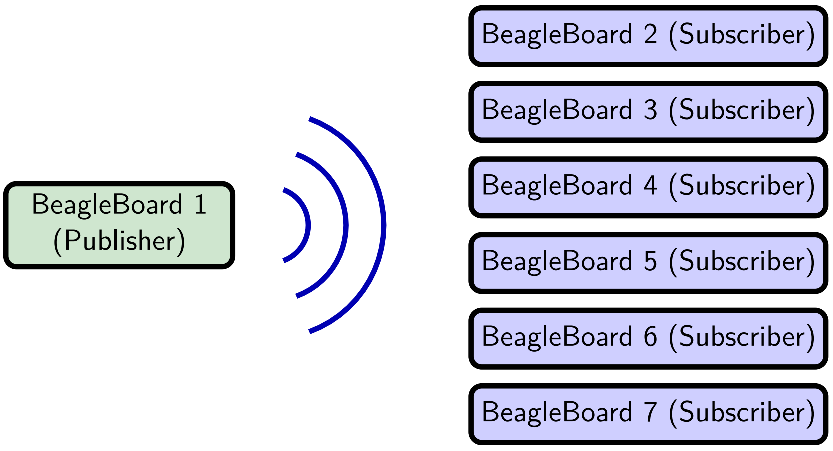

BeagleBoard 1 (Publisher)
BeagleBoard 2 (Subscriber)
BeagleBoard 3 (Subscriber)
BeagleBoard 4 (Subscriber)
BeagleBoard 5 (Subscriber)
BeagleBoard 6 (Subscriber)
BeagleBoard 7 (Subscriber)

Figure 5.3.: Test setup for direct 300 kBit/s evaluation: Six BeagleBoards directly subscribed to one publisher. All subscribers share the same wireless link.

### 5.2.3. Direct Connections in a 300 KBit/s Network

During this test one publisher, running on a BeagleBoard, was publishing messages directly to 6 subscribers, running on other BeagleBoards. Figure 5.3 shows the logical setup for this test. The BeagleBoard of the publisher was connected using a wireless bridge, while all subscribers were attached to the same Ethernet switch.

Figure 5.7 shows the measured latencies. The wireless bridge introduces a larger variance in measured latencies. The average values are now also higher, ranging in the tens of ms. The largest message transmitted was a 5 kB message at 1 messages per second.

### 5.2.4. Indirect Connections in a 300 kBit/s Network

For this test, the connections between the synchronization nodes were replaced by independent wireless links. Figure 5.4 shows the logical setup of the experiment.

Figure 5.8 shows the recorded latencies. Messages below a size of 2000 Bytes fall into a region between 10 ms and 20ms. The variance of the measured latencies is smaller than in the test with direct connection.

The largest message that could be transmitted at least once per second was 20 kB large.

### 5.2.5. Results

The use of the routed multimaster network allowed all tested configurations to send bigger messages at higher rates. This is a result of the reduced data rate, which is appearing at the network

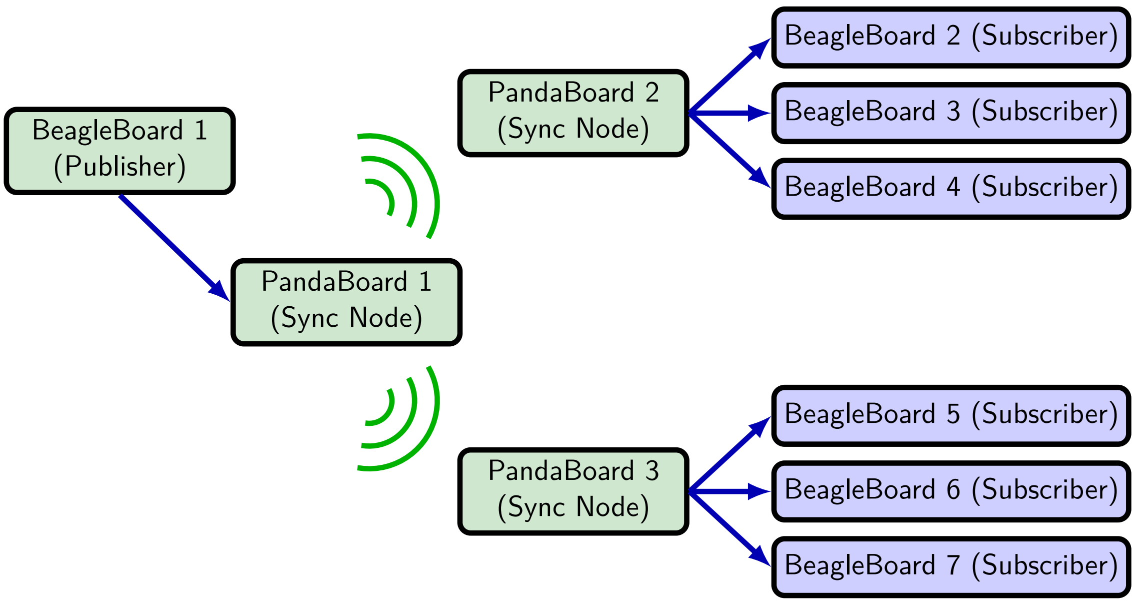

BeagleBoard 1 (Publisher)
PandaBoard 1 (Sync Node)
PandaBoard 2 (Sync Node)
PandaBoard 3 (Sync Node)
BeagleBoard 2 (Subscriber)
BeagleBoard 3 (Subscriber)
BeagleBoard 4 (Subscriber)
BeagleBoard 5 (Subscriber)
BeagleBoard 6 (Subscriber)
BeagleBoard 7 (Subscriber)

Figure 5.4.: Test setup for indirect 300 kBit/s evaluation: Three synchronization nodes route the data to the nodes. The links between the synchronization nodes are wireless.

interface of the original publisher.

In the case of the connection via Ethernet, the system introduced an additional latency. This was not observable while testing with wireless links, which already have a higher latency. For these links, the variance in latency was reduced.

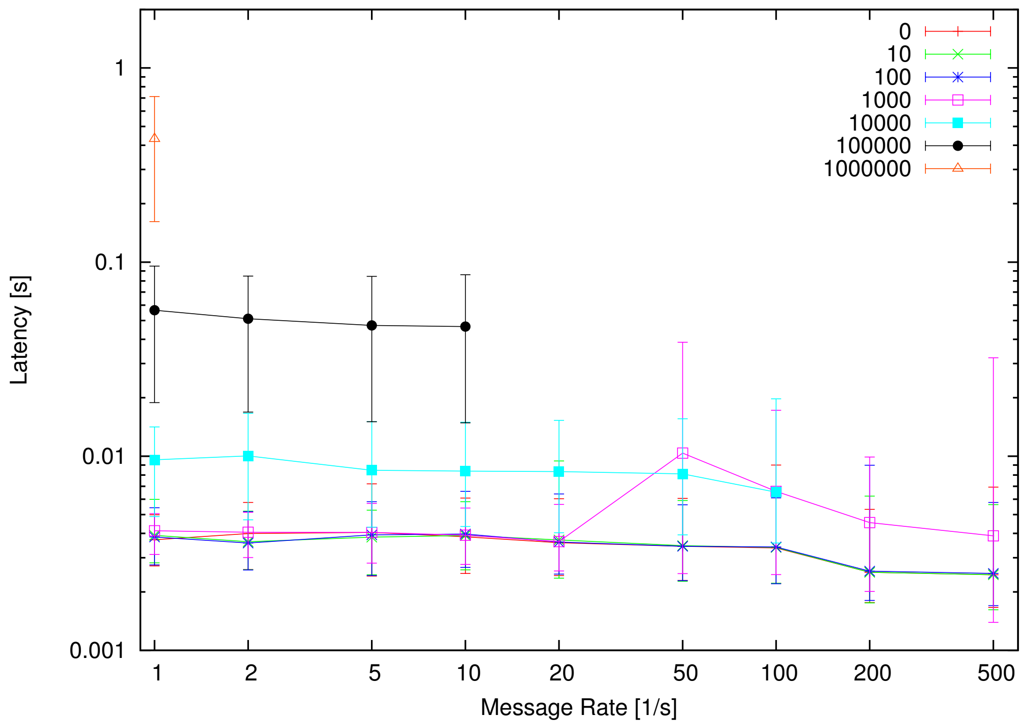


Figure 5.5.: Latency with direct connections in a 100 MBit/s network with message sizes between 0 Bytes and 1 MBytes.

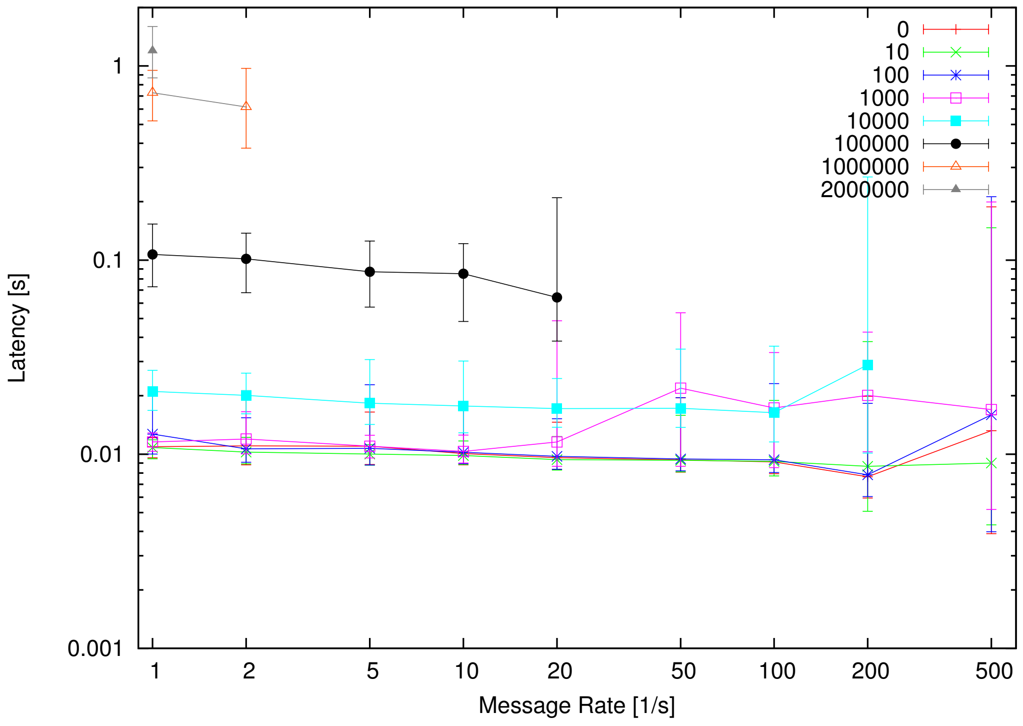


Figure 5.6.: Latency with indirect connections in a 100 MBit/s network with message sizes between 0 Bytes and 2 MBytes.

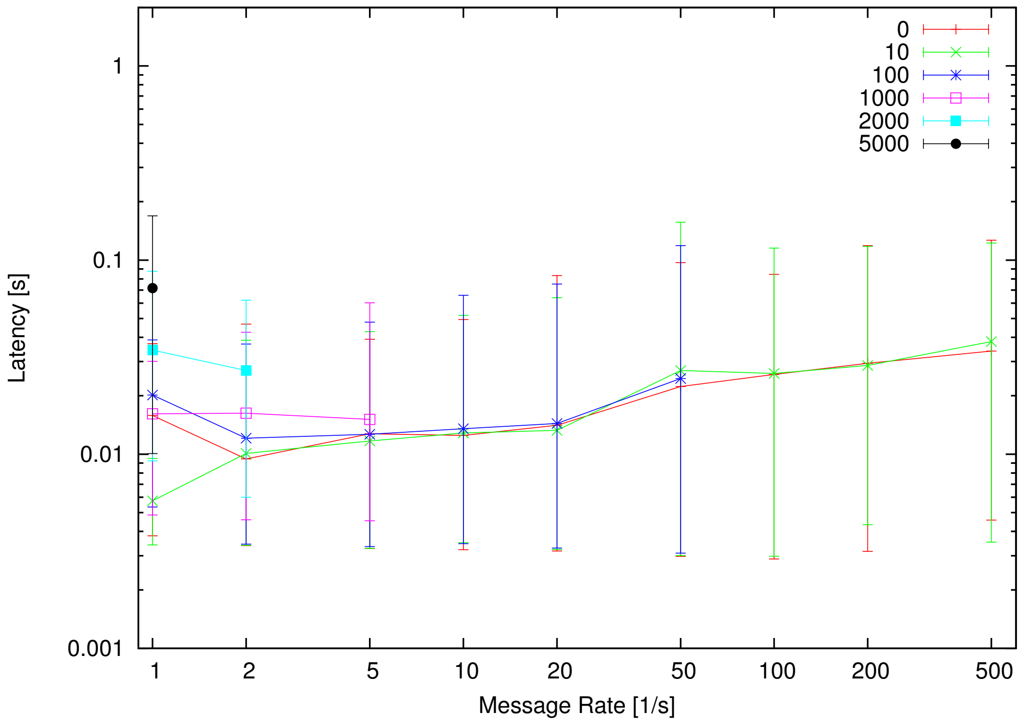


Figure 5.7.: Latency with direct connections in a 300 kBit network with message sizes between 0 Bytes and 5 kBytes.

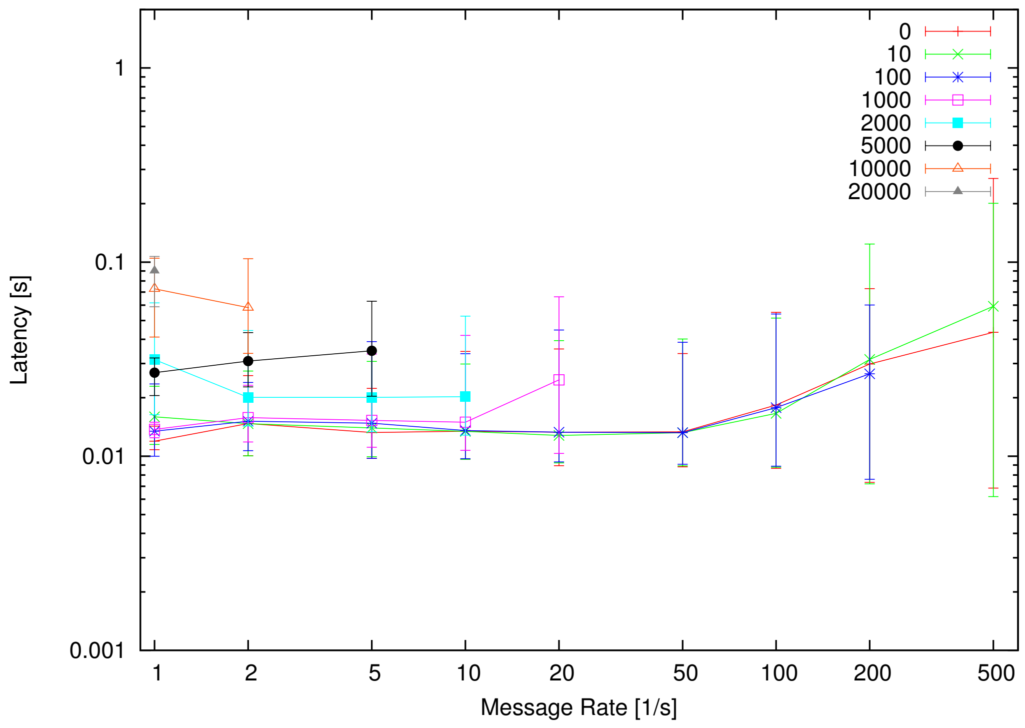


Figure 5.8.: Latency with indirect connections in a 300 kBit/s network with message sizes between 0 Bytes and 20 kBytes.

# Chapter 6.

# Conclusion

This thesis introduces a set of tools to improve the usability of ROS in larger installations for Intelligent Environments. The key improvements are lifting the restriction to a single master node, introducing a system to apply rules to the namespaces and reducing the memory and bandwidth usage for smaller devices.

In real world applications, security and privacy play a big role. Together with an implementation of security and privacy features, interfacing with the users smartphone, the system can be put to use there. Together with its strengths in simulation and ease of use, ROS can become an alternative to specialized middlewares for Intelligent Environments.

ROS carries properties in its design which can not be changed by enhancing it with external tools. This includes the need for every ROS master node to hold the complete namespace in its memory. Together with the flat design of the multimaster implementation, every change to the namespace system must be propagated through the whole system, to be available to every node. This can lead to an excessive exchange of messages between the different synchronization nodes. There is the possibility to restrict, which parts of the namespace are exported by a master to the rest of the system, but this just adds another parameter which has to be carefully adjusted, to not disturb the system. A better way would be to introduce a hierarchical system, where the master nodes form a tree structure which only forwards the parts of the namespace, which are actually necessary to satisfy the needs of the different parts of the system. The DNS system can be regarded as an example, where a massive namespace is managed by a hierarchical structure.

Another problem of ROS is the verbosity and complexity of its XML-RPC protocol. It lays a heavy burden on nodes in Wireless Sensor Networks. Although the resources available on these devices will increase over time, they are better spent on increasing the security and availability of a Wireless Sensor Network. Extending ROS to support a wider range of data formats like JSON [50], BinaryPack [51] or Protocol Buffers [52] paired with HTTP could help to extend the reach of ROS to very small devices.

Future work to improve ROS for Intelligent Environments should therefore focus on implementing a security concept, an alternative to to XML-RPC interface and a more distributed way to provide the namespaces and parameter server to the nodes.

ROS is in a unique position as a middleware. It enjoys high community participation, a regular release interval, good simulation environments and, together with the above mentioned enhancements, the capability to reach from the smallest devices up to the high powered service robots in future homes, hospitals or care homes for the elderly.

# Appendix A.

# Using the Namespace Service

The source code for the namespace service can be downloaded from a subversion repository located at `https://svn.vmi.ei.tum.de/vmi-ros-pkg/branch/ros_comm6-mm/nameservice_tum/`.

The namespace service consists of a single node, which takes three parameters, containing XSLT rules:

1. The ~rules parameter contains rules to transform the metadata of nodes into names of nodes, topics or services.
2. The ~launchfilerules parameter contains rules to transform complete ROS .launch files mixed with metadata.
3. The ~xslrules parameter contains rules, which are used to pre-transform the other two rules. These rules are used to ease the development of the rules contained in the ~rules and ~launchfilerules parameters.

## A.1. Starting the Service

The repository contains example rules inside the rules directory. The nameservice.launch file in the launch directory loads these rules into the parameter server and starts the namespace service.

## A.2. Transformation of Node Metadata

Figure A.1 shows example metadata for a node. Figure A.3 shows an example for rules, which can transform this metadata into names for nodes and topics.

The metadata contains the name of the device, who owns it and what group of devices it belongs to. It also contains the name, function and physical location of its topics. If a topic can not be associated with a fixed physical location, it is defined as a mobile topic.

The example rules take this metadata and generates paths in the namespaces.

### A.2.1. Node Name

When the owner of a device is known, its name is generated as:

```
/<group of the device>/<name of the owner>/<name of the device>
```

If the owner is unknown the following name is generated:

```
/<group of the device>/<name of the device>
```

Default values are used when necessary attributes of the node are not specified.

### A.2.2. Topic and Service Names

The rules for the paths of topics and services differentiate between mobile and stationary topics. Stationary topics are arranged according to the location of the topic, e.g.:

```
/building1/room3/sensors/temperature
```

Mobile topics are put into the namespace, depending on the node's owner, group and name, e.g.:

```
/smartphones/schneider/htc_desire/sensors/accelerometer
```

By default, a topic is assumed to be stationary and at an unknown location. The same rules apply to services.

### A.2.3. Calling the Service

Figure A.2 shows how a node calls the service and uses the received path to announce its topic. It reads the node's metadata from the parameter server, sends it to the namespace service and receives the topic name it should use for the specified topic.

## A.3. Transformation of .launch Files

To automate the transformation of the topic names, the namespace service can transform complete .launch files. Figure A.4 shows an example .launch file before and after transforming it. The service can handle multiple nodes in a single .launch file. The launchnode script in the nodes/ directory accepts such a .launch file, sends it to the namespace service and executes it.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<node>
  <owner>schneider</owner>
  <name>htc_desire</name>
  <group>smartphones</group>

  <topic>
    <name>accelerometer</name>
    <function>sensor</function>
    <mobile>true</mobile>
  </topic>

  <topic>
    <name>temperature</name>
    <function>sensor</function>
    <location>building1/room3</location>
  </topic>

</node>
```

Figure A.1.: Example node metadata.

```python
rospy.wait_for_service('get_topic_name')
get_topic_name = rospy.ServiceProxy('get_topic_name', GetTopicName)
metadata = rospy.get_param('~metadata', "")
pub = None
try:
  name = get_topic_name('accelerometer', metadata)
  pub = rospy.Publisher(name.topicname, std_msgs.msg.String)
except rospy.ServiceException, e:
  print "Service did not process request: %s"%str(e)
```

Figure A.2.: Example code to call the namespace service.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="no"/>
  <xsl:strip-space elements="*"/>
  <parameter name="node-owner" path="//node/owner" default=""/>
  <parameter name="node-group" path="//node/group" default="unknown-group"/>
  <parameter name="node-name" path="//node/name" default="unknown-node"/>

  <xsl:template match="/">
    <xsl:if test="$type='topic'">
      <xsl:apply-templates select="node/topic"/>
    </xsl:if>
    <xsl:if test="$type='node'">
      <xsl:apply-templates select="node"/>
    </xsl:if>

  </xsl:template>

  <xsl:template match="/node">
    <name>
      <!-- node/group To what kind of devices does this node belong? -->
      <xsl:text>/</xsl:text><xsl:value-of select="$node-group"/>

      <xsl:if test="$node-owner!=''">
        <!-- node/owner Who own this device? -->
        <xsl:text>/</xsl:text><xsl:value-of select="$node-owner"/>
      </xsl:if>
      <!-- node/name What is the name of the device where this topic resides? -->
      <xsl:text>/</xsl:text><xsl:value-of select="$node-name"/>
    </name>
  </xsl:template>


  <xsl:template match="/node/topic | /node/service">
    <parameter name="location" default="unknown-location"/>
    <parameter name="function" default=""/>
    <parameter name="mobile" default="false"/>

    <xsl:if test="name/text()=$name">
      <name>
        <!-- Is this topic mobile?-->
        <xsl:if test="$mobile='true'" >
          <!-- node/group To what kind of mobile devices does this node belong? -->
          <xsl:text>/</xsl:text><xsl:value-of select="$node-group"/>

          <xsl:if test="$node-owner!=''">
            <!-- node/owner Who own this device? -->
            <xsl:text>/</xsl:text><xsl:value-of select="$node-owner"/>
          </xsl:if>
          <!-- node/name What is the name of the device where this topic resides? -->
          <xsl:text>/</xsl:text><xsl:value-of select="$node-name"/>
        </xsl:if>

        <!-- is this topic stationary? -->
        <xsl:if test="$mobile!='true'">
          <!-- topic/location Where is this topic located? -->
          <xsl:text>/</xsl:text><xsl:value-of select="$location"/>
        </xsl:if>

        <!-- topic/function Does this topic belong to a general kind of device? -->
        <xsl:if test="$function!=''">
          <xsl:text>/</xsl:text><xsl:value-of select="$function"/><xsl:text>s</xsl:text>
        </xsl:if>

        <xsl:text>/</xsl:text><xsl:value-of select="$name"/>
      </name>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Figure A.3.: Example XSLT rules to transform node metadata. The rules in x̃slrules are used to transform the <parameter> tags into valid XSLT.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<launch xmlns:ns="urn:schemas-tum-de:vmi:ns">
  <node name="htc_desire" pkg="nameservice_tum" type="testnode2" output="screen">
    <ns:owner>schneider</ns:owner>
    <ns:group>smartphones</ns:group>
    <ns:topic type="~accelerometer">
      <ns:name>accelerometer</ns:name>
      <ns:function>sensor</ns:function>
      <ns:mobile>true</ns:mobile>
    </ns:topic>
    <ns:topic type="~output2">
      <ns:name>wallwasher</ns:name>
      <ns:function>lamp</ns:function>
      <ns:location>n1/raum3</ns:location>
    </ns:topic>
  </node>
  <node name="htc_desire2" pkg="nameservice_tum" type="testnode2" output="screen">
    <ns:owner>schneider</ns:owner>
    <ns:group>smartphones</ns:group>
    <ns:topic type="~accelerometer">
      <ns:name>accelerometer</ns:name>
      <ns:function>sensor</ns:function>
      <ns:mobile>true</ns:mobile>
    </ns:topic>
  </node>
</launch>
```

(a) The original .launch file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<launch xmlns:ns="urn:schemas-tum-de:vmi:ns">
  <node name="htc_desire" pkg="nameservice_tum" type="testnode2" output="screen" ↪
       ↩ ns="/smartphones/schneider">
    <remap from="~accelerometer" to="/smartphones/schneider/htc_desire/sensors/accelerometer"/>
    <remap from="~output2" to="/n1/raum3/lamps/wandfluter"/>
  </node>
  <node name="htc_desire2" pkg="nameservice_tum" type="testnode2" output="screen" ↪
       ↩ ns="/smartphones/schneider">
    <remap from="~accelerometer" to="/smartphones/schneider/htc_desire2/sensors/accelerometer"/>
  </node>
</launch>
```

(b) The transformed .launch file.

Figure A.4.: The original and the transformed .launch file.

# Appendix B.

# Setting Up a Routed Multimaster Network

The source code for the routed multimaster system is located in the subversion repository at https://svn.vmi.ei.tum.de/vmi-ros-pkg/branch/ros_comm6-mm/multimaster_tum_routed/.

To use this software, a modified version of the ros_comm package has to be installed. It is located at https://svn.vmi.ei.tum.de/vmi-ros-pkg/branch/ros_comm6-mm/ros_comm6/. This package can be install using the method described at http://www.ros.org/wiki/ros_comm6.

To start a synchronization node, the command in Figure B.1 is used. It takes a list of remote hosts as parameter. The synchronization node will then try to open connections to these hosts. If another synchronization node is running on one or more of these hosts, a connection will be established. The synchronization node will also accept connections from hosts, which are not in this list.

After a connection has been formed, all remote topics will be made available and routed through the system as needed.

```
rosrun multimaster_tum_routed multimaster.py host1 host2
```

Figure B.1.: Command to launch a routed synchronization node, which connects to host1 and host2.

# Appendix C.

# Setting Up a Hybrid Multimaster Network

The source code for the meshed multimaster system is located in the subversion repository at `https://svn.vmi.ei.tum.de/vmi-ros-pkg/branch/ros_comm6-mm/multimaster_tum/`.

When started, it will automatically discover other meshed synchronization nodes in the local network and open connections to it. Figure C.1 shows the command to launch the synchronization node.

To form a hybrid network, one host in the local network needs to run a routed synchronization node as well a meshed synchronization node at the same time.

```
rosrun multimaster_tum multimaster.py
```

Figure C.1.: Command to launch a meshed synchronization node.

# List of Figures

# Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, April 1965.

[2] F. Lewis, "Wireless sensor networks," *Smart Environments: Technologies, Protocols, and Applications*, pp. 11–46, 2004.

[3] M. Kranz, A. Schmidt, and P. Holleis, "Embedded interaction: Interacting with the internet of things," *IEEE Internet Computing*, vol. 14, pp. 46 – 53, March-April 2010.

[4] H. Ishii, "Tangible user interfaces," 2007.

[5] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 94–104, 1991.

[6] B. CONRAD BZURA, *THE EMERGING ROLE OF ROBOTICS IN PERSONAL HEALTH CARE: Bringing Smart Health Care Home*. PhD thesis, Worcester Polytechnic Institute, 2012.

[7] L. Capra and D. Quercia, "Middleware for social computing: a roadmap," *Journal of Internet Services and Applications*, pp. 1–9, 2012.

[8] K. Ashton, "That 'internet of things' thing." `http://www.rfidjournal.com/article/view/4986`, June 2009.

[9] M. Kranz, A. Möller, and L. Roalter, "Robots, objects, humans: Towards seamless interaction in intelligent environments," in *1st International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2011)*, (Algarve, Portugal), pp. 163–172, SciTePress, March 2011.

[10] M. Kranz, L. Roalter, and F. Michahelles, "Things That Twitter: Social Networks and the Internet of Things," in *What can the Internet of Things do for the Citizen (CIoT) Workshop at The Eighth International Conference on Pervasive Computing (Pervasive 2010)*, May 2010.

[11] A. Kansal, J. Hsu, S. Zahedi, and M. Srivastava, "Power management in energy harvesting sensor networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 4, p. 32, 2007.

[12] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki-a lightweight and flexible operating system for tiny networked sensors,” in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pp. 455–462, IEEE, 2004.

[13] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, *et al.*, “Tinyos: An operating system for sensor networks,” *Ambient intelligence*, vol. 35, 2005.

[14] H. Hagras, V. Callaghan, M. Colley, G. Clarke, A. Pounds-Cornish, and H. Duman, “Creating an ambient-intelligence environment using embedded agents,” *Intelligent Systems, IEEE*, vol. 19, no. 6, pp. 12–20, 2004.

[15] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Nahrstedt, “A middleware infrastructure for active spaces,” *Pervasive Computing, IEEE*, vol. 1, no. 4, pp. 74–83, 2002.

[16] E. Aitenbichler, J. Kangasharju, and M. Mühlhäuser, “Mundocore: A light-weight infrastructure for pervasive computing,” *Pervasive and Mobile Computing*, vol. 3, no. 4, pp. 332–361, 2007.

[17] A. Ranganathan, J. Al-Muhtadi, S. Chetan, R. Campbell, and M. Mickunas, “Middlewhere: a middleware for location awareness in ubiquitous computing applications,” in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pp. 397–416, Springer-Verlag New York, Inc., 2004.

[18] A. Katasonov, O. Kaykova, O. Khriyenko, S. Nikitin, and V. Terziyan, “Smart semantic middleware for the internet of things,” in *Proceedings of the 5-th International Conference on Informatics in Control, Automation and Robotics*, pp. 11–15, 2008.

[19] B. Williams and P. Nayak, “Immobile robots ai in the new millennium,” *AI magazine*, vol. 17, no. 3, p. 16, 1996.

[20] L. Roalter, A. Möller, S. Diewald, and M. Kranz, “Developing Intelligent Environments: A Development Tool Chain for Creation, Testing and Simulation of Smart and Intelligent Environments,” in *Proceedings of the 7th International Conference on Intelligent Environments (IE)*, pp. 214–221, july 2011.

[21] M. Naor and U. Wieder, “A simple fault tolerant distributed hash table,” *Peer-to-Peer Systems II*, pp. 88–97, 2003.

[22] OASIS, “Devices profile for web services specification.” `http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01`, July 2009.

[23] E. Avilés-López and J. García-Macías, “Tinysoa: a service-oriented architecture for wireless

sensor networks," *Service Oriented Computing and Applications*, vol. 3, no. 2, pp. 99–108, 2009.

[24] M. Familiar, J. Martínez, and L. López, "Pervasive smart spaces and environments: A service-oriented middleware architecture for wireless ad hoc and sensor networks," *International Journal of Distributed Sensor Networks*, vol. 2012, 2012.

[25] G. Anastasi, E. Bini, and G. Lipari, "Extracting data from wsns: A service-oriented approach," *Methodologies and Technologies for Networked Enterprises*, pp. 329–356, 2012.

[26] G. Nain, E. Daubert, O. Barais, and J. Jézéquel, "Using mde to build a schizophrenic middleware for home/building automation," *Towards a Service-Based Internet*, pp. 49–61, 2008.

[27] R. Mayrhofer and I. Ion, "Openuat: The open source ubiquitous authentication toolkit," *month*, 2010.

[28] L. O. Nicolas Falliere, "W32.Stuxnet Dossier." `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`, 2011.

[29] J. Al-Jaroodi, A. Al-Dhaheri, F. Al-Abdouli, and N. Mohamed, "A survey of security middleware for pervasive and ubiquitous systems," in *Network-Based Information Systems, 2009. NBIS'09. International Conference on*, pp. 188–193, IEEE, 2009.

[30] S. Yamada and E. Kamioka, "Access control for security and privacy in ubiquitous computing environments," *IEICE transactions on communications*, vol. 88, no. 3, pp. 846–856, 2005.

[31] X. Jiang, J. Hong, and J. Landay, "Approximate information flows: Socially-based modeling of privacy in ubiquitous computing," *UbiComp 2002: Ubiquitous Computing*, pp. 176–193, 2002.

[32] G. Moritz, E. Zeeb, F. Golatowski, D. Timmermann, and R. Stoll, "Web services to improve interoperability of home healthcare devices," in *Pervasive Computing Technologies for Healthcare, 2009. PervasiveHealth 2009. 3rd International Conference on*, pp. 1–4, IEEE, 2009.

[33] U. Hunkeler, H. Truong, and A. Stanford-Clark, "Mqtt-s—a publish/subscribe protocol for wireless sensor networks," in *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, pp. 791–798, IEEE, 2008.

[34] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli, "Context-aware middleware for resource management in the wireless internet," *Software Engineering, IEEE Transactions on*, vol. 29, no. 12, pp. 1086–1099, 2003.

[35] D. Salber, A. Dey, and G. Abowd, "The context toolkit: aiding the development of context-

enabled applications," in *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, pp. 434–441, ACM, 1999.

[36] A. Toninelli, S. Pantsar-Syväniemi, P. Bellavista, and E. Ovaska, "Supporting context awareness in smart environments: a scalable approach to information interoperability," in *Proceedings of the International Workshop on Middleware for Pervasive Mobile and Embedded Computing*, p. 5, ACM, 2009.

[37] K. Kjær, "A survey of context-aware middleware," in *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pp. 148–155, ACTA Press, 2007.

[38] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2, no. 4, pp. 263–277, 2007.

[39] M. Chaqfeh and N. Mohamed, "Challenges in middleware solutions for the internet of things," in *Collaboration Technologies and Systems (CTS), 2012 International Conference on*, pp. 21–26, IEEE, 2012.

[40] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, 2009.

[41] L. Roalter, M. Kranz, and A. Möller, "A middleware for intelligent environments and the internet of things," in *Ubiquitous Intelligence and Computing* (Z. Yu, R. Liscano, G. Chen, D. Zhang, and X. Zhou, eds.), vol. 6406 of *Lecture Notes in Computer Science*, pp. 267–281, Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16355-523.

[42] J. Postel, "Internet protocol." `http://tools.ietf.org/html/rfc791`, 1981.

[43] S. Deering and R. Hinden, "Internet protocol, version 6 (ipv6) specification." `http://tools.ietf.org/html/rfc1883`, 1995.

[44] G. Huston, "The changing foundation of the internet: confronting ipv4 address exhaustion," *The Internet Protocol Journal*, vol. 11, no. 3, pp. 19–36, 2008.

[45] S. Thomson, Bellcore, T. Narten, and IBM, "Ipv6 stateless address autoconfiguration." `http://tools.ietf.org/html/rfc2462`, 1998.

[46] R. Hinden, Nokia, and S. Deering, "Ip version 6 addressing architecture." `http://tools.ietf.org/html/rfc4291`, 2006.

[47] C. Metz, "Ipv4-mapped addresses on the wire considered harmful." `http://tools.ietf.org/id/draft-itojun-v6ops-v4mapped-harmful-02.txt`, 2003.

[48] W. Diffie and M. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, vol. 22, no. 6, pp. 644–654, 1976.

[49] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[50] D. Crockford, "The application/json media type for javascript object notation (json)." `http://tools.ietf.org/html/rfc4627`, 2006.

[51] C. Bormann, "The binarypack json-like representation format." `http://tools.ietf.org/html/draft-bormann-apparea-bpack-00`, 2012.

[52] Google, "Protocol buffers." `http://code.google.com/apis/protocolbuffers/`.