FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Content Aware Smart Routing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Jürgen Galler, BSc.
Matrikelnummer 0825384

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.Dr. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Günther Gridling

Wien, 7. Oktober 2013 _____ _____

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Content Aware Smart Routing

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

by

Jürgen Galler, BSc.
Registration Number 0825384

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof.Dr. Wolfgang Kastner
Assistance: Dipl.-Ing. Günther Gridling

Vienna, 7th October 2013 _____ _____

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Jürgen Galler, BSc.
Leopold-Figl-Straße 42, 4040 Linz

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

Hereby, I want to thank my advisors Dr. Wolfgang Kastner and Dipl.-Ing. Günther Gridling for their intense support throughout the thesis. In addition, I want to thank Carina, my girlfriend for her mental support, as well as her parents, Helmut and Waltraud for integrating me in their family and providing accommodation during the past two years. Finally, I want to thank my grandparents for their financial support throughout my studies.

Kurzfassung

In Bezug auf die Anzahl der Netzwerkknoten und die erforderliche Skalierbarkeit stoßen die gegenwärtigen Internet-Technologien bereits an ihre Grenzen. Heutzutage sind längst nicht nur mehr Computer, sondern auch Tablets, Smartphones und sogar kleinste Geräte, bis hinunter zu Sensoren oder Aktoren mit dem Internet verbunden. Durch die breite Systemlandschaft entstand in den letzten Jahren eine Vielzahl neuer Anwendungsgebiete, und damit verbunden eine Menge neuer Technologien, wie zum Beispiel IPv6, ZigBee, 6LoWPAN, NFC oder RFID. Aus dem Verbund dieser Technologien folgt das so genannte *Internet of Things* (IoT). Für dieses Internet der Dinge ist es jedoch oft problematisch, einen Dienst (ein Service) im Internet zu addressieren, da ein Client im Normalfall dafür den genauen Bezeichner (URI) wissen muss.

Diese Masterarbeit stellt ein Konzept vor, das die starke Bindung zwischen Service und Server lockert (unter Betrachtung von Sensor/Aktor Systemen). Hierfür werden speziell Techniken von IPv6 verwendet. Ein Client kann dabei eine nicht adressierte Anfrage an das Content Aware Network stellen. Ein spezieller Router (*Smart Router*) empfängt das Paket und leitet es an einen bestimmten Server weiter. Zu diesem Zweck beinhaltet die Anfrage ein *Semantic Tag*, das von den einzelnen Netzwerkknoten interpretiert werden kann. Um die korrekte Zieladdresse zu bestimmen, befragt der Smart Router einen *Resolver*, der einen semantischen Reasoner verwendet, um bekannte Fakten von einer Ontologie herzuleiten. Der Resolver wandelt die um Fakten erweiterte Anfrage in eine spezielle Anfrage an einen *Domain Server* um. Nachdem diese aufgelöst worden ist, wird sie an einen oder mehrere bestimmte *Server* weitergeleitet. Ein solcher Server kann schließlich eine weitere, nicht adressierte Anfrage absenden, um bestimmte Sensoren oder Aktoren anzusprechen. Zusätzlich zum inhaltsbasierten Weiterleiten von Anfragen kann das Content Aware Network auch nicht addressierte Datenpakete korrekt routen, die von Sensoren bzw. Aktoren nach einem bestimmten Ereignis gesendet werden (z.B. nach dem Überschreiten von bestimmten Schwellwerten).

Als Proof-of-Concept wurde ein Referenzsystem implementiert, das grundlegende Funktionalitäten demonstriert. Das vorgestellte Konzept wurde in Bezug auf Performanz, Sicherheit und Fehlertoleranz evaluiert. Der Durchsatz wurde anhand der benötigten Nachrichten gemessen und mit einigen alternativen Ansätzen verglichen. Zum Schutz vor Angriffen wurde die Anwendbarkeit von IPsec untersucht.

Abstract

Today's Internet has reached its scalability limits with respect to the number of participants and performance. Contemporary Internet devices are not just computers, but also tablets, smart phones and even small devices, like sensors and actuators. A variety of new technologies is inherent to these Internet appliances, like IPv6, ZigBee, 6LoWPAN, NFC or RFID. From the combination of these technologies a new paradigm arises, the so called *Internet of Things* (IoT). The problem of all these new services is how a service can be addressed. Typically, a client has to know the exact resource identifier (URI) in order to use a specific service hosted by a server somewhere in the Internet.

This thesis provides a concept which relaxes the strong binding between service and server by the use of IPv6 focusing on the addressing of sensors and actuators. A client can place an unaddressed request to the so called *content aware network*. A special router, called *smart router*, fetches the request and forwards it to a dedicated server, which is able to process the request. To this end, the request contains a semantic tag which is interpretable by participating network nodes. In order to determine the correct address of a server, the smart router inquires a *resolver*. This resolver uses a semantic reasoner utilizing a smart routing ontology in order to augment the request by known facts. The resolver converts the request to a more specific query for a *domain server*. There can be several domain servers for a variety of domains. After the initial primary request is sent to its target server, this server examines the request and performs one or more secondary requests in order to set the corresponding actuators or to fetch current values from sensors. This secondary request is also unaddressed and forwarded to the correct network nodes by use of the inherent knowledge of the content aware network. In addition to the correct routing of unaddressed client requests, the content aware network approach is also able to route unaddressed data packets sent from sensor/actuator-units. Such a data packet can be sent for example on the change of a node's internal state (e.g. on the exceeding of certain thresholds).

As a proof of concept, an elementary reference system is implemented. This implementation provides a reduced set of functionalities in order to evaluate the concept with respect to performance, security and fault tolerance. The content aware network's performance, measured in the amount of messages, is compared to some theoretical alternative approaches. In order to preserve security, IPsec is investigated.

Contents

List of Figures

List of Symbols

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
AH	Authentication Header
API	Application Programming Interface
CCN	Content Centric Network
CIA	Confidentiality, Integrity, and Availability
CNT	Connected Nodes Table
DHCPv6	Dynamic Host Configuration Protocol for IPv6
DL	Description Logic
DONA	Data-Oriented (and Beyond) Network Architecture
ERT	Executed Request Table
ESP	Encapsulating Security Payload
FFD	Full Function Device
FIB	Forwarding Information Base
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IoT	Internet of Things
IPsec	Internet Protocol Security
IPv4	Internet Protocol Version 4

IPv6	Internet Protocol Version 6
LR-WPAN	Low-Rate Wireless Personal Area network
MAC	Media Access Control
MHz	Megahertz
NDN	Named Data Networking
NDP	Neighbor Discovery Protocol
NFC	Near Field Communication
OSI-model	Open Systems Interconnection Model
OWL	Web Ontology Language
PAN	Personal Area Network
PID	Process Identifier
PIT	Pending Interest Table
PRT	Pending Request Table
QoS	Quality-of-Service
RDF	Resource Description Framework
RFD	Reduced Function Device
RFID	Radio Frequency Identification
S/A-System	Sensor/Actuator System
S/A-Unit	Sensor/Actuator Unit
SADB	Security Association Database
SART	S/A Response Table
SPARQL	SPARQL Protocol And RDF Query Language
SPD	Security Policy Database
SPI	Security Parameter Index
SQL	Structured Query Language

SSL	Secure Sockets Layer
SWRL	Semantic Web Rule Language
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TMR	Triple Modular Redundancy
TTL	Time To Live
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VPN	Virtual Private Network
W3C	World Wide Web Consortium
WPAN	Wireless Personal Area network

CHAPTER 1

Introduction

1.1 Motivation

We have reached an era where not only computers, but also smart phones, TVs, cars and even smaller devices like temperature sensors are connected to the Internet. Typically, for retrieving the desired content one has to know the exact URL on the Web. If someone does not know this unique identifier, search engines must be used. Therefore, for getting the desired information often an indirect way has to be taken. Then, when the desired data is found somewhere on the Internet, it may not be in exactly the desired format. The user has to examine data from several resources in order to aggregate all information to a required state.

Since these bypasses are not compliant to the user's aim, it is natural to search for alternatives. This master thesis handles the challenge of retrieving data from an IPv6 network by not knowing the exact location of the information with a closer examination of S/A-systems (Sensor/Actuator-System, i.e., a system which has the purpose of connecting sensors and actuators to a set of network components). To this end, a user sends a dedicated unaddressed request packet via its network interface. A special router (called *smart router*) catches the packet and handles the request. It is even possible to inquire aggregated information from several sensors (or to set multiple actuators at once) by a single request. The thesis also provides a reference implementation of the provided concept based on standard PC hardware and several Java and C programs.

For example, it is possible to use the provided technique in building automation systems or on sensor buses in a car. While the the suggested concept's field of application is dedicated to S/A-systems, it is also possible to apply the concept in other areas.

1.2 Problem Statement

In the scope of the IoT6 project [35], a concept for a smart routing technique in IPv6 based networks should be established. For this purpose, it should be possible not only to send packets with dedicated source and receiver addresses, but also to send unaddressed packets which should be routed based on the semantic tag of a packet. This semantic tag is a piece of machine-readable information which describes the content of a network packet. According to the functionality of a dedicated network component, such a component is able to understand and process a packet compliant to its semantic tag. To this end, the concept should describe all needed network components, as well as the techniques used for message exchange and semantic reasoning. Furthermore, it should be possible to represent a real topological structure within the network to augment semantic interpretation possibilities.

In addition to a self-contained concept, the thesis should provide an elementary reference implementation in order to evaluate and demonstrate the provided technique. The resulting network should be auto-configurable and tolerant to faulty sensor nodes or broken network links.

1.3 Aim and Methodological Approach

The thesis provides the concept combined with an analysis and a reference implementation of a content aware networking system. By the use of techniques acquired by this thesis, it should be possible to augment the features of IPv6 by a complete new routing taxonomy. Instead of shifting semantic interpretation of network packets to higher OSI layers, a full semantic processing of such packets is performed at layers 3/4 (network layer, transport layer).

While there are cases of routing, where other factors (e.g. security, or QoS) matter, the presented concept handles the task of finding a destination address to an unaddressed (semantic) packet. Security issues are evaluated in the end of the thesis. The final result is a working system of core functionality which serves as a reference design for the IoT6 project [35].

The initial step is an intensive literature research on the topics of Internet protocols, CCNs (Content Centric Networks), smart routing and the Semantic Web. After this research, a technical concept is developed, containing necessary hardware and software components to build a reference system. In this scope, available techniques are evaluated and an ontology is created. This ontology represents typical components of a content aware network with a closer examination on S/A-systems in order to enable reasoning on the semantic tag of a network packet. During the life cycle of the thesis, the created ontology gets augmented by more complex components, properties and rules.

Afterwards, a reference network is implemented which consists of required hardware and software components, like smart routers, resolvers, servers and clients. The thesis is concluded

by an evaluation of the reference network. This evaluation discusses performance, security and fault tolerance of the presented concept and compares the system with some alternative approaches.

1.4 Structure of the Work

After a short introduction in Chapter 1, some state-of-the-art techniques are presented in Chapter 2. This chapter summarizes several ubiquitous standards of the Internet of Things (IoT).

Chapter 3 explains the used technologies. It is shown by some examples how an OWL ontology is built up, how Internet sockets are used, or how kernel modules are implemented.

The concept of a content aware network is introduced in Chapter 4. First, some use cases are defined. Starting from these use cases, needed network components and their interaction are described. Sequence diagrams illustrate how client requests are handled, or how network nodes are connected to the network and automatically configured.

Chapter 5 presents a reference implementation of the chosen concept. This reference implementation serves as a proof of concept and provides a basic set of features needed by a content aware network. The chapter details all implemented components and packet types. The implementation's message sequences are explained in the remainder of this chapter.

The presented concept and its reference implementation are evaluated in Chapter 6. This chapter compares the content aware network to alternative approaches and discusses traffic throughput and caching, as well as security and safety issues.

The thesis is concluded by a summary and an outlook of future work in Chapter 7.

CHAPTER 2

State of the Art

This chapter points out the limits of IPv4 and introduces some state of the art techniques on the topics of network communication and Semantic Web.

2.1 Internet Oriented Visions

2.1.1 Limits of Current Internet Technology

In 2012, 2.4 billion people were connected to the Internet worldwide, compared to 361 million back in 2000 [18] and there is still a rising tendency. Contributing to this large amount of Internet users is certainly the still rising amount of smart phone users. Nevertheless, the set of available IPv4 [21] addresses runs out in the next few years [20].

In addition to this shortage of IPv4 addresses, current Internet communication is generally based on point-to-point connections. An Internet service is usually bound to a server with a fixed address somewhere on the Internet. This means that a host needs to know the exact URL of the server which provides a dedicated service. This attachment of service and server is no longer appropriate.

Furthermore, binding a service to a specific server comes mostly with the price of *single point of failures*. To avoid these, additional precautions in form of explicit redundancy must be made, since current Internet communication does not provide techniques for a built-in redundancy of services by default.

2.1.2 New Technologies

IPv6

In order to maintain Internet service despite the rapidly increasing number of Web-enabled devices, a successor of IPv4 was developed, the Internet Protocol, Version 6 (IPv6) [15]. With the much bigger address space it is possible to connect even simple devices (e.g. freezers, temperature sensors, electronic timetables at bus stops) to the Internet, enabling a whole new set of possibilities. In contrast to a 32-bit IPv4 address, an IPv6 address is 128-bits wide. Therefore, the IPv6 address space is 2^{96} times larger than the address space of IPv4. The IPv6 packet header is shown in Figure 2.1.

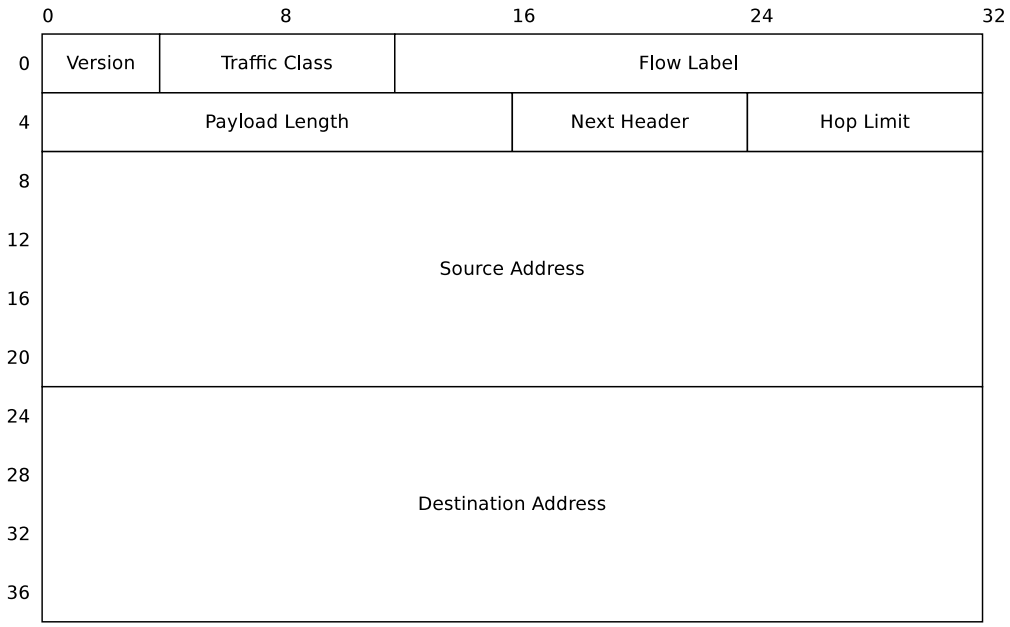


Figure 2.1: IPv6 packet header.

Version (4 bits): The `version` field specifies the version of the IP protocol. In case of IPv6, this field contains the constant 6.

Traffic class (8 bits): The `traffic class` field is used to classify packets in the manner of QoS (quality-of-service).

Flow label (20 bits): The `flow label` field can be used to inform routers about subsequent packets which should be sent over the same paths. This should avoid the reordering of packets.

Payload length (16 bits): The `payload length` field contains the length of the IPv6 packet's payload (in combination with all extension headers) in bytes.

Next header (8 bits): To specify the transport layer protocol, the `next header` field is used. This field has the same function as the `protocol` field of the IPv4 header. In addition to *TCP* (6) and *UDP* (17), the next header field can also specify protocols, like *ESP* (50) and *AH* (51) which are used for secure communication via IPsec (see Section 2.1.2)

Hop limit (8 bits): The `hop limit` field replaces the IPv4's `TTL` (time to live) field. It specifies how many routers the packet is allowed to pass, before it is discarded. Every router has to decrement the value of this field by 1.

Source address (128 bits): The `source address` field contains the IPv6 address of the packet's sender.

Destination address (128 bits): The `destination address` field contains the IPv6 address of the packet's receiver.

In addition to longer addresses, IPv6 also provides a set of new features. One of them is *IPv6 multicasting* which replaces the inflexible broadcasting functionality of IPv4. While broadcasting reaches always all participants of a network, multicasting can be configured in a way, such that only interested parties receive a multicast packet. To this end, an IPv6 device must be added to a dedicated multicast group by just binding the corresponding multicast address to one of its interfaces. Multicast addresses have the format `FF00::/8` (i.e., `FF<x><y>::<z>`), where `FF` indicates that this address is a multicast address. The next eight bits are reserved for flags (`x`: 4 bits) and the scope (`y`: 4 bits). The remainder of the address (`z`) is used to identify the multicast group. Important flag bit values are:

0b0000: permanent multicast address (assigned by the IANA)

0b0001: transient or dynamic assigned multicast address

The next four bits specify the scope of the multicast address. Some defined scopes are:

1: Node-local scope multicast addresses have the same functionality as the IPv4 loopback address (`127.0.0.1`). They never leave an interface.

2: Link-local scope multicast addresses are never forwarded by routers. These addresses are used to address the local subnet.

e: Global scope multicast addresses. They can be routed world wide.

0, f: Reserved scopes.

Examples for predefined multicast addresses are:

All nodes: `FF02::1`

All routers: `FF02::2`

Similar to IPv6 multicasting is *IPv6 anycasting*. Anycasting can be used to increase a service's availability by adding equivalent servers to a network. Each server has the same IPv6 address assigned. The router tables only have shortest routes stored, so only the nearest server is visible to a router. On the reception of an anycast packet, a router does a look-up in its routing table and forwards the packet automatically to just one (the nearest) server. On the failure of one server, the routes are automatically updated to an alternative server.

IPsec

As mentioned above, IPv6 also provides techniques to secure communication by encryption and authentication by the use of IPsec [36]. IPsec is a protocol suite specified by the IETF to secure network communication by authentication and encryption. Unlike SSL/TLS which is based on the transport layer of TCP/IP, IPsec works directly on the network layer. IPsec can protect communication paths

1. between two hosts,
2. between two security gateways, or
3. between a security gateway and a host.

Hosts implementing IPsec must support (1) and (3), while a security gateway must support all three protection types.

IPsec creates a security boundary (see Figure 2.2) in order to secure network communication. For any packet, an IPsec enabled device has to decide if the packet should be discarded, bypassed, or protected.

IPsec provides two protocols, to secure network communication, *IP Authentication Header (AH)* [24] and *Encapsulating Security Payload (ESP)* [25]. For an IPsec implementation, ESP support is mandatory, while AH is optional. AH offers integrity and data origin authenticity, while ESP also provides confidentiality. Both protocols can be used for access control by the use of key distribution and the use of an `SPD` (Security Policy Database) which manages traffic flows. In most cases, it is enough to support ESP. For key management and distribution, IKEv2 [3] is proposed (but an implementation can also use other techniques for key management).

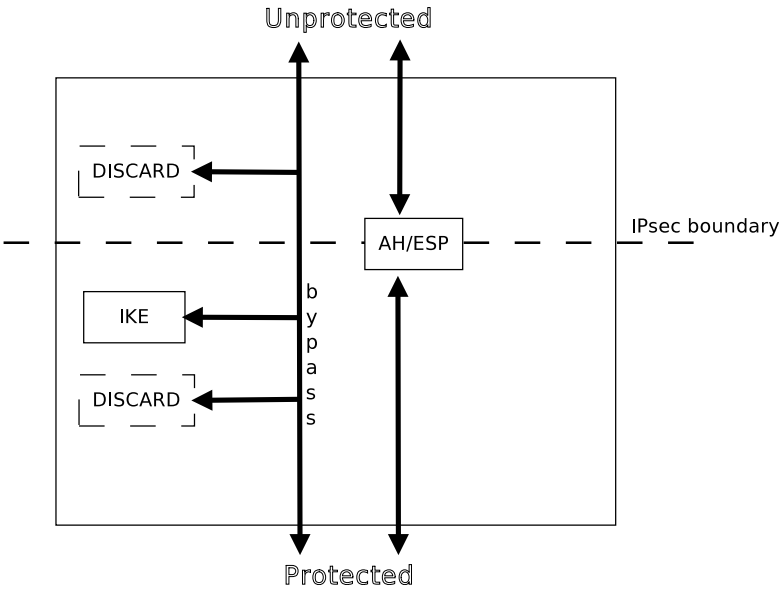


Figure 2.2: Network protection by IPsec [36].

IPsec supports two modes of operation: In *transport mode*, the IPsec header is located between the IP header and the transport header. Therefore, protection is provided for next layer protocols. Operating in *tunnel mode*, whole IP packets can be packed into the payload of an IPsec packet. This mode is used to create virtual private networks (VPNs).

IPsec creates security functions by the use of security associations. A security association contains a set of algorithms and keys to authenticate and encrypt a uni-directional traffic flow. For bi-directional communication, two security associations are needed. There is also the possibility to define security associations for a multicast group. In addition, a participant may support multiple security associations.

In order to describe the used protocols of a security association, an `SADB` (Security Association Database) is used. An outgoing packet has to provide an `SPI` (Security Parameter Index). Together with the destination address, an `SPI` uniquely identifies a security association contained in the `SADB` for a specific packet. For incoming packets, a similar procedure is performed to get decryption algorithms and keys from the `SADB`.

IEEE 802.15.4

IEEE 802.15.4 [28] is a standard based on the physical layer and the media access control layer of the OSI-model (see Figure 2.3). The standard is dedicated to Low-Rate Wireless Personal Area Networks (LR-WPANs). It is a basis of upper level standards, like ZigBee or 6LoWPAN which describe higher OSI-layers, not contained in the specification of IEEE 802.15.4.

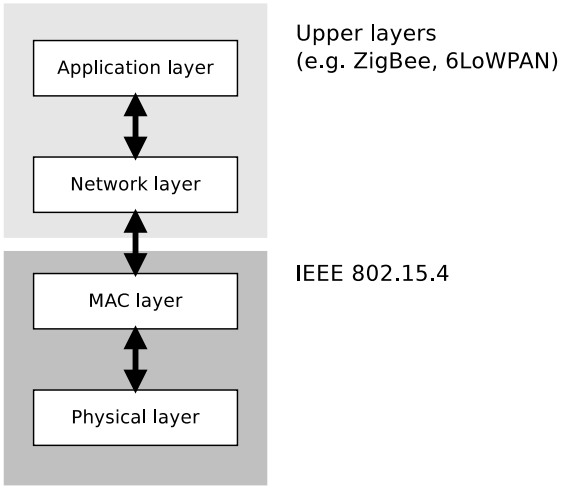


Figure 2.3: IEEE 802.15.4 protocol stack.

IEEE 802.15.4 works on one of these three unlicensed frequency bands:

- Three channels in the 868.0-868.6 MHz band (Europe)
- Up to 30 channels in the 902-928 MHz band (North America)
- Up to 16 channels in the 2400-2483.5 MHz band (worldwide)

There are two types of network nodes defined, namely *full function devices (FFD)* and *reduced function devices (RFD)*. An FFD is (in addition to normal operation) able to route packets. If an FFD manages the whole network, it is called a PAN coordinator. An RFD is a very simple device with low resources. Only elementary functions are implemented by an RFD in order to communicate with FFDs.

IEEE 802.15.4 supports two different topologies, namely *peer-to-peer* and *star networks*. Each of them needs at least one PAN coordinator. Peer-to-peer networks can create arbitrary links between any devices in range, while star networks are bound to the star pattern.

ZigBee

ZigBee [28] describes the network and application layers on top of an IEEE 802.15.4 network (see Figure 2.3). In contrast to Wi-Fi (IEEE 802.11), ZigBee networks can build hierarchical wireless mesh networks. This means that it is possible to deliver packets entirely wirelessly by multiple hops (e.g. a node sends a packet to another one which acts as a wireless router and forwards the packet according to a routing policy. After a few hops, the packet arrives at its final destination. The advantage of this approach is that far off destinations can be reached with low power devices.

To form a ZigBee network, a PAN coordinator must choose a channel which is free of disturbances and interferences. Afterwards, arbitrary nodes can send beacon requests in order to connect to the network. In the first phase, only the PAN coordinator answers the beacon, but when the network also contains FFDs (ZigBee routers), it is also possible that such a device responses to a beacon request.

6LoWPAN

IPv6 has the big drawback of a huge packet header. This means for small devices a severe overhead of memory space and bandwidth. 6LoWPAN (IPv6 over Low power Wireless Personal Area Networks) [27] provides a reduced IPv6 protocol stack optimized for low power wireless networks (based on IEEE 802.15.4) where the resources are too limited for standard IPv6 networks (e.g. for home and building automation or entertainment systems).

By careful inspection of packet headers, 6LoWPAN reduces the IPv6- and UDP headers to 7 bytes [34]. The reduced 6LoWPAN packet header is illustrated in Figure 2.4.

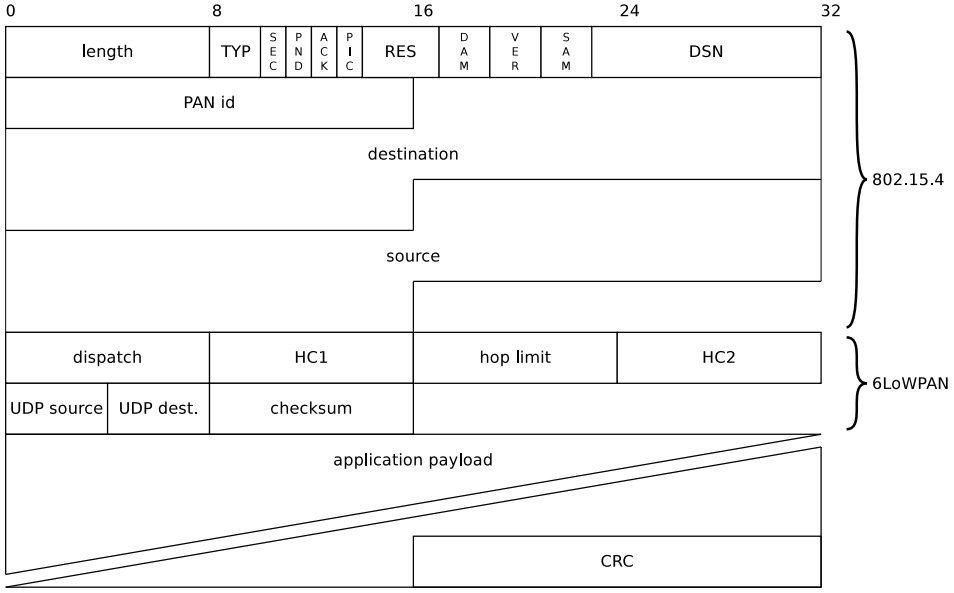


Figure 2.4: 6LoWPAN packet header [34].

The `version` field of the IPv6 header is always 6 and the `length` field is the equal to the `length` field of the IEEE 802.15.4 minus the IPv6 header length. The fields `flow label` and `traffic class` are never used. Hence, these four fields can be simply removed. Usually, the `next header` field points to either UDP or TCP. An 8-bit field is therefore a huge waste, so it is replaced by 2-bits within the `HC1` field of the 6LoWPAN header. In addition, IPv6

addresses can be computed out of IEEE 802.15.4's 64-bit MAC addresses [7], which makes the `source` and `destination` fields of the IPv6 header unnecessary. For the UDP header, the length also can be computed out of the IEEE 802.15.4's `length` field, which saves again 2 bytes. In addition, in most cases 4-bits are sufficient for UDP port numbers, so `source` and `destination UDP port` fields are shrinked to 4 bits.

6LoWPAN implementations are available for a variety of (embedded) operating systems (e.g. for Contiki or TinyOS). Since 6LoWPAN is based on IPv6, there are many standards and protocols available which can be used. Therefore, instead of starting from the beginning by implementing a new protocol, a developer can use one of many existing, tested and working implementations.

2.2 Things Oriented Visions

2.2.1 Automated Life

Modern technology allows almost everything to be automated. For example, consider shipment tracking. There is no human being involved when someone wants to retrieve the exact location of his parcel. Another example would be the cashless payment method that comes with your bank account. Again, no person is involved when you put your debit card into a ticket machine in order to pay a train ticket. To enable such automated processes for everyday use, a variety of technologies exist. This section describes two of them.

2.2.2 NFC

NFC (Near Field Communication) [29] is an international standard for short distance wireless communication based on induction. The maximum data rate is $424kbit/s$. A possible application of NFC is cashless payment. Modern credit cards have a small non-powered NFC chip included. In order to pay with such a card, it is no longer required to plug the card into a terminal. The only thing to do is just holding the card near to an NFC enabled reading device. NFC is also supported by high end smart phones. With such a smart phone, it is on the one hand possible to exchange data with other devices (telephone numbers, Wi-Fi settings, etc.), but it is also possible to use the smart phone for payment, like an NFC-enabled credit card. In addition, there are so called NFC-tags. These small non-powered chips, which contain a unique identifier, can be used for merchandise management systems, backtracking of shipped goods or similar applications.

2.2.3 RFID

Similarly to NFC-tags, RFID (Radio Frequency Identification) provides short distance wireless communication with powerless chips. The induction based RFID standard is used for authentication systems. To this end, an RFID system consists of an RFID-tag which is connected to

an antenna and an RFID reading device. The RFID-tag can be best compared to a key and the reading device is the lock of a specific door. There are a lot of different designs of RFID-tags: check cards, buttons, key tags, etc.

2.3 Semantic Oriented Visions

2.3.1 Content Awareness

First of all, the *Semantic Oriented Visions* have to deal with data collection, storage and addressing. Consider a set of resources which provide the same type of data, but differently formatted (e.g. three temperature sensors, where one measures the temperature in Celsius, one in Fahrenheit and the third one in Kelvin). The technologies included in the *Semantic Oriented Visions* must be able to correctly combine the resources and make decisions based on the gained information.

Furthermore, by the nature of the Internet's inherent point-to-point strategy of network communication, network technologies will have to deal with a large amount of traffic. Concerning this, it is important that future Internet communication provides more intelligent routing paradigms based on the semantic information of a packet. To this end, the *Semantic Oriented Visions* present machine readable languages for processing a chunk of information by so called semantic reasoners. Unfortunately present automatic semantic interpretation is only performed at higher layers of the OSI-model, limiting the possibilities of smart routing.

A dedicated program should be able to make decisions based on semantic information. In this context, *Content Aware* means that a machine is able to read and understand the meaning of a piece of information. For instance, a router with semantic interpretation possibilities (i.e., a smart router) should not only be able to route a packet based on its destination address, but also it should be possible to make routing decisions based on the content of a network packet. Network packets can be handled differently according to their type. Video streams can be processed with a higher priority than emails, since a router is able to examine the internals of a network packet. For this purpose, every packet must provide a piece of machine readable information, a so called *semantic tag*. Whether the complete packet is interpretable by a component or just a part of it depends on the application. To come back to the example, a video stream must only contain some machine readable meta-information, while for an email it may be also interesting if the mail body contains any keywords like *urgent*, or *important*.

2.3.2 Semantic Web

Today's Web is full of semi-structured or unstructured data. For a machine it is hardly possible to interpret and process the information contained in such chunks of human readable data. Therefore, it is necessary to add some meaning (i.e., meta information) to this data. It is important to structure information on the Web such that a piece of software can identify the type and meaning

of it in a way to combine and augment it with information from other resources. An example of such an application would be a smart search engine which is able to collect information from search results of several Web pages into one structured list. Due to the interpretable information it is possible to infer new facts and combine them in a machine readable search result.

The Semantic Web now provides the needed features to systematically interpret and process content on the Web. Structured data, combined with its meta information is stored within one or more ontologies. Due to the adding of meta information to data, the association of a service to a dedicated server is relaxed. Several ontologies can be combined arbitrarily in order to create new services. With redundant information within these ontologies, single point of failures can be avoided implicitly.

With RDF and OWL [39], the W3C [6] provides languages to build complex ontologies. There are several reasoners [4] [30] to infer not directly encoded facts and relations. By the use of query languages [40], it is possible to make decisions based on given ontologies.

For the use with S/A-systems several *sensor and observation ontologies* have been developed. These can be divided into more *observation-centric* respectively more *sensor-centric* ontologies. While the main goal of observation-centric ontologies is the representation of measurements, the focus of sensor-centric ontologies is the description of devices. The observation-centric *SENSEI* [9] ontology provides an information model to represent sensor values. To each value several parameters, like type, date, origin, sensor id and accuracy can be stored. The sensor-centric *MMI Device Ontology* [22] was developed to represent oceanographic sensors and samplers. It can be used to characterize devices in order to provide a discovery service of available sensors to Web-applications.

2.3.3 Content Aware Networking Approaches

Content Centric Networking (CCN)

Content Centric Networking (CCN) also entitled as *Named Data Networking (NDN)* is presented in [17]. CCN is a radically new approach of the way network communications should be established. It distinguishes between two types of packets, namely *interest* and *data packets* (see Figure 2.5). A data packet satisfies an interest packet, if the *content name* of the interest packet equals the prefix of the *content name* of the data packet. Like the addressing scheme of IP, content names are hierarchical (e.g. `/ThisRoom/projector` is a valid content name). Content validation can be established by providing digital signatures within data packets.

Every participant of the network is able to send an interest packet of a specific type. CCN routers have to forward the interests according to the *Forwarding Information Base (FIB)* and store them in a *Pending Interest Table (PIT)*. Interest packets can then be answered by arbitrary participants of the network by sending a corresponding data packet. It is possible that more

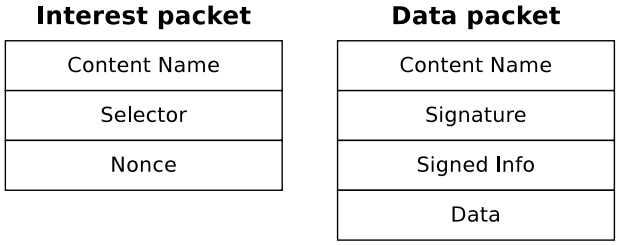


Figure 2.5: CCN packet types [17].

than one interest gets satisfied by one data packet. If an interest is satisfied, it is removed from a router's PIT. In addition, CCN routers cache data packets in so called *Content Stores* for a higher throughput.

The steps for handling an interest packet are the following: On the reception of an interest packet, a router has to perform a longest-match look-up for the content name on its Content Store. If there is a match, this means that the router has cached a suitable data packet. Therefore, the interest can be directly answered. Otherwise, a look-up in the router's PIT is made to determine if the router has already forwarded a similar request which was not satisfied yet. In this case, the interest is added to the corresponding entry in the PIT. Only when no matching entry in the Content Store or the PIT was found, the packet is forwarded according to the router's FIB. In this case, also a new entry is created in the PIT.

On the reception of a data packet, a router has to examine all pending interests. If there is a match, the data packet must be forwarded to all belonging faces (i.e., network interfaces or software processes). In addition, the data packet must be stored in the Content Store.

Unfortunately, the current reference implementation (CCNx) has some issues achieving the desired performance. To this end, [19] analyzes the performance of the CCNx implementation and simplifies its forwarding structure. Optimized concepts, issues and principles are presented which should increase the throughput of CCNx to up to 10 Gbps by the use of hardware acceleration.

Data-Oriented (and Beyond) Network Architecture" (DONA)

The "Data-Oriented (and Beyond) Network Architecture" (DONA) [16] proposes to replace the current DNS names with flat, self-certifying names and the DNS name resolution with a name-based anycast primitive which is located above the IP layer. Rather than naming a server on the Internet, some dedicated service or data is named by a pair of the cryptographic hash of the publishers public key `P` and a human-readable label `L`. A client asking for data with the name `P:L` eventually receives a triplet `<data, public key, signature>`. The client can check if the data really comes from the supposed sender. While DONA suggests an interesting

way of naming content on the Internet, its goal is not directly in the semantic representation of the content itself.

2.4 Putting Everything Together

When considering all these new technologies, one can observe that through their ensemble, a new paradigm of Internet communication arises: the so called *Internet of Things (IoT)*. It is obvious that IoT does not stand for a particular technique, but for a set of strategies and concepts to seize the interconnection of mostly low-power and resource-efficient Internet connected devices. For example, for modern shipment tracking it is possible to trace a parcel by attaching small tags on it (RFID- or NFC-tags). At any major location, this tag can be read. Since these reading devices are connected to the Internet, a customer is able to retrieve the actual position of the parcel. It is obvious that many technologies belonging to the IoT work in the background with no human involvement. These systems shall not disturb, but help people in various situations. The IoT combines three visions of contemporary devices: First, there are the *Internet oriented visions*. To this part protocol standards, like ZigBee or 6LoWPAN can be counted. Then, there are the *things oriented visions* which contain techniques like *NFC* or *RFID*. Finally, there are the *semantic oriented visions* [13] which are mostly based on the Semantic Web.

In the scope of the IoT, many concepts and theories about the future Internet exist. One of them is the Cognitive Net [26]. The Cognitive Net wants to extend the network infrastructure in a way, such that whole network streams can be routed at once, instead of sending every packet on its own path through the network. This should reduce routing overhead. In addition, it should be possible to define priorities for data streams. For example, a high definition video stream should have a higher priority than a file download. In the case of network overload, it should also be possible to spontaneously form mesh-networks out of existing infrastructure which is not primary used for Internet communication (like Bluetooth).

On the other hand, also projects exist which try to provide reference systems to serve as recommendations for future Internet communication. One example is the IoT-A, the Internet of Things - Architecture [33]. Another one is the IoT6 project [35] which is a European research project and part of the 7th Framework Programme [5]. The focus points of these projects are the investigation of IPv6 and inherent techniques. They try to integrate all relevant technologies (such as NFC, 6LoWPAN, ZigBee, Semantic Web, etc.) into one whole concept in order to seize the jungle of uncoordinated standards, projects and programming techniques. These projects must eventually deal with the semantic interpretation possibilities of a vast amount of different network nodes, like sensors, actuators, etc.

CHAPTER 3

Technologies

This chapter provides a description of used technologies and programming techniques. It should serve as a rough manual in order to understand the subsequent chapters.

3.1 Sockets

3.1.1 Overview

Communication between two applications, hosted on either the same or on two different nodes, is established by so called sockets. A socket is a bidirectional communication tunnel with two ports, one for each application. In this context, a special consideration to Internet sockets is taken. An Internet socket is used to communicate with a piece of software (a server or a client) somewhere on a remote computer. There are three types of sockets: *Datagram*, *stream* and *raw sockets*.

3.1.2 Socket Types

Datagram sockets

A datagram socket is used for connection-less communication, like it is used for UDP. It is possible to communicate with a lot of network partners over one datagram socket, but there is no assignment of messages to a dedicated communication cycle, since there is no message stream.

Stream sockets

Connection-oriented communication is done via a stream socket. Such a socket establishes a connection to a remote program, before communication is possible. Once a stream socket is

opened, it can only be used for exact one communication stream (with one network partner). Nevertheless, the big advantage of a stream socket is its assignment of messages to a dedicated communication cycle.

Raw sockets

There may be some cases where neither datagram nor stream sockets provide the needed functionality. Each of them may restrict the communication in a way, such that the desired behavior cannot be fulfilled. In this case, a raw socket has to be used. A raw socket provides just minimal functionality to the application side, which enables full access to all packet headers and the possibility to implement an alternative transport layer.

3.1.3 How to Use a Socket

This section provides some examples for the use of datagram socket communication over IPv6. The provided examples are all programmed in C on a Linux machine (Ubuntu 12.04.2 LTS), but since socket communication is provided by the operating system through system calls, the functions shall be similar in other programming languages.

Listing 3.1 shows a function to open a datagram socket. The function returns the ID (i.e., the file descriptor) of the created socket. In order to use UDP over IPv6, the parameters `PF_INET6`, and `SOCK_DGRAM` must be applied to the `socket()` function. Next, a valid source address structure must be bound to the socket (by using `bind()`. For this purpose, a specific port and IPv6 address may be set. It is also possible (like in the example), to use 0 as port and `in6addr_any` as IPv6 address. In this case the operating system uses any address of an arbitrary interface as source address and an arbitrary free UDP port. By calling `getsockname()` the actual socket address (IPv6 address in combination with an UDP port) can be determined.

```
1  int open(void) {
       socklen_t sin6len;
3      int sock;
       struct sockaddr_in6 src_addr;
5
       sin6len = sizeof(struct sockaddr_in6);
7      sock = socket(PF_INET6, SOCK_DGRAM,0);

9      if (sock < 0)
           return sock;
11
       memset(&src_addr, 0, sizeof(struct sockaddr_in6));
13     src_addr.sin6_port = htons(0);      /* any port */
       src_addr.sin6_family = AF_INET6;    /* use IPv6 */
15     src_addr.sin6_addr = in6addr_any;  /* any address assigned to this machine */

17     /* bind the socket to src_addr */
       bind(sock, (struct sockaddr *)&src_addr, sin6len);
19
```

```
    /* get the actual socket address */
21  getsockname(sock, (struct sockaddr *)&src_addr, &sin6len);

23  return sock;
}
```

Listing 3.1: Create a datagram socket

Listing 3.2 demonstrates a function which can be used to receive data from a network socket. The function uses `recvfrom()` to receive data from a specific socket. The function provides the received data in the pointer `buffer`. The address of the packet's sender is stored in `rx_src_addr`.

```
size_t rcv(int sock, void *buffer, size_t buffer_len, struct sockaddr_in6 *rx_src_addr)
    {
2   socklen_t sin6len;
    size_t rx_len;
4
    sin6len = sizeof(struct sockaddr_in6);
6
    /* fill buffer with received data */
8   rx_len = recvfrom(sock, buffer, buffer_len, 0, (struct sockaddr *)rx_src_addr, &
        sin6len);

10  /* clear not used section of the buffer */
    memset(((uint8_t *)buffer)+rx_len, 0, buffer_len-rx_len);
12
    return rx_len;
14 }
```

Listing 3.2: Receive data from a datagram socket

Listing 3.3 provides a function to send data to a dedicated IPv6 address and port over an existing socket. To use IPv6 addresses, `AF_INET6` must be used as address family. `inet_pton()` is used to convert IPv4 or IPv6 addresses from text (string) to binary form. Finally `sendto()` is used to send the contents of `buffer` to the specified destination address over an existing socket.

```
size_t send(int sock, char *dest_addr_s, int dest_port, void *buffer, size_t buffer_len
    ) {
2   socklen_t sin6len;
    struct sockaddr_in6 dest_addr;
4
    sin6len = sizeof(struct sockaddr_in6);
6
    /* clear destination address */
8   memset(&dest_addr, 0, sizeof(dest_addr));

10  /* set destination address type to INET6 */
```

```
    dest_addr.sin6_family = AF_INET6;
12
    /* the server IP address, in network byte order */
14  /* dest_addr_s contains the destination IPv6 address as string */
    inet_pton(AF_INET6, dest_addr_s, &dest_addr.sin6_addr);
16
    /* the port we are going to send to, in network byte order */
18  dest_addr.sin6_port = htons(dest_port);

20  /* buffer contains the data to send
    return sendto(sock, buffer, buffer_len, 0, (struct sockaddr *)&dest_addr, sin6len);
22 }
```

Listing 3.3: Send data over a datagram socket

Listing 3.4 closes an open socket by first calling `shutdown()` to stop communication in both directions. The second argument specifies the direction: 2 must be used for both, receiving and transmitting. Finally, `close()` destroys the file descriptor.

```
closeSocket(int sock) {
2   if(sock > 0) {
        shutdown(sock, 2);
4       close(sock);
    }
6 }
```

Listing 3.4: Close a datagram socket

3.2 Kernel Modules

3.2.1 Overview

Sometimes it is not enough to execute a program in user space. In this case, a kernel module can be created. A kernel module has to be registered at the kernel. After the registration it allows to execute code in kernel space. Kernel modules should be kept small and simple. They should avoid too much dynamic memory allocation. Instead, static memory allocation should be preferred when possible.

Listing 3.5 shows an example of a simple kernel module. `module_init()` is used to register a function which is called when the module is initialized (in this case `init()`). `module_exit()` is used to register a function which is executed when the kernel module is unregistered. `printk()` can be used to print info and error messages to `/var/log/kern.log`.

```
#include <linux/kernel.h>
#include <linux/module.h>

#define NAME  "[kernel-demo] "
#define ERROR(args...)    printk(KERN_ERR NAME args)
#define MESSAGE(args...) printk(KERN_INFO NAME args)

static int __init init(void)
{
  MESSAGE("Hook registered\n");

    /* return 0 for success*/
    return 0;
}

static void __exit exit(void)
{
  MESSAGE("Hook unregistered\n");
}

module_init(init);
module_exit(exit);
```

Listing 3.5: Demo kernel module (kdemo.c)

To build a kernel module, a special makefile has to be employed (see Listing 3.6). After building the kernel module by calling `make`, the module can be registered at the kernel by calling `sudo insmod kdemo.ko`. To unsubscribe the module, `sudo rmmod kdemo.ko` must be executed. Listing 3.7 entails the output of the kernel module printed to `/var/log/kern.log`

```
obj-m += kdemo.o

all:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Listing 3.6: Makefile for demo kernel module

```
Aug 30 10:00:07 A kernel: [125563.395241] [kernel-demo] Hook registered
Aug 30 10:00:33 A kernel: [125590.071105] [kernel-demo] Hook unregistered
```

Listing 3.7: Log output of demo kernel module

3.2.2 Netfilter

Netfilter hooks (see [23], [12] Chapter 19.3) implement the packet-filtering functionality in Linux. They can be used to monitor, filter and manipulate network packets before they are rerouted or arrive at the source application. Netfilter hooks have to be registered within kernel modules. It is possible to attach a hook at five different positions of the Linux packet filtering architecture (see Figure 3.1):

`NF_INET_PRE_ROUTING:` This is the first possibility to register the hook. All routing code is executed after this hook.

`NF_INET_LOCAL_IN:` All incoming packets to this computer pass these hooks.

`NF_INET_FORWARD:` All packets which just pass this machine are passed to these hooks. Before and after these hooks, the packets must pass the routing engine.

`NF_INET_LOCAL_OUT:` All outgoing packets which have their source at this computer must pass these hooks.

`NF_INET_POST_ROUTING:` All packets which are forwarded to other computers pass these hooks.

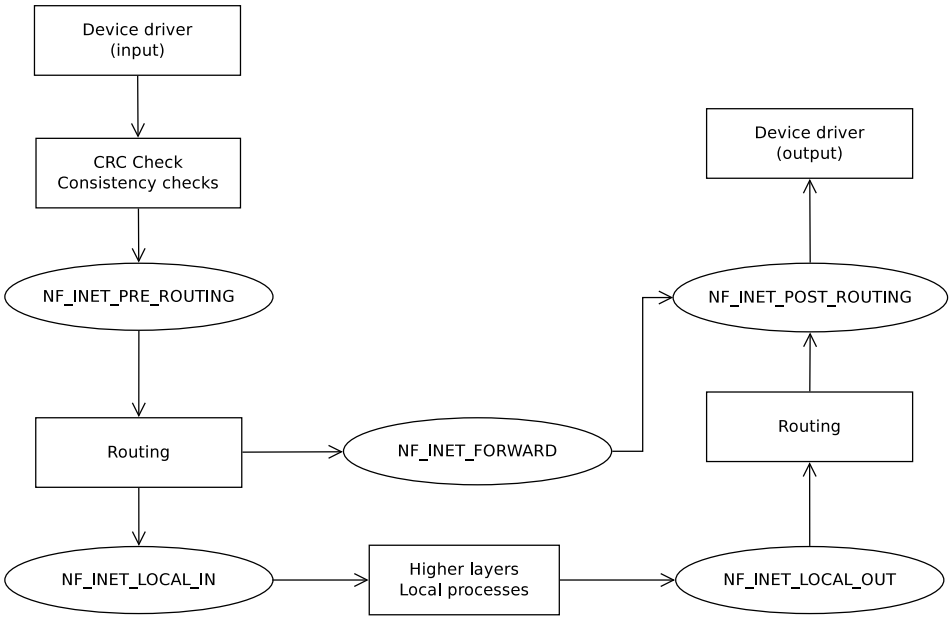


Figure 3.1: The packet filtering architecture of Linux [12].

Listing 3.8 shows an example to register a netfilter hook within a kernel module. `AF_INET6` is used to filter IPv6 packets. The priority is set to `NF_IP6_PRI_FIRST` which is the highest possible priority (`INT_MIN`). Priorities are sorted in ascending order. `hook_func()` is the function to call, when a packet passes the hook.

```
nfho.hook     = hook_func;           /* function to call */
nfho.hooknum  = NF_INET_PRE_ROUTING; /* called right after packet received */
nfho.pf       = AF_INET6;            /* IPv6 packets */
nfho.priority = NF_IP6_PRI_FIRST;    /* set to highest priority */
nf_register_hook(&nfho);             /* register hook */

MESSAGE("Hook registered\n");
```

Listing 3.8: Register a netfilter hook

Listing 3.9 shows a simple netfilter hook function which logs the source and destination address of any IPv6 packet to `/var/log/kern.log`.

```
unsigned int hook_func(unsigned int hooknum, struct sk_buff *skb, const struct
    net_device *in, const struct net_device *out, int (*okfn)(struct sk_buff *))
{
    struct ipv6hdr *ip_header;

    ip_header = (struct ipv6hdr *) ipv6_hdr(skb);

    if(!skb || !ip_header) {
        ERROR("Problem during packet filtering!");
        return NF_ACCEPT;
    }

    MESSAGE("GOT packet from %pI6 to %pI6\n", &(ip_header->saddr), &(ip_header->daddr))
        ;

    return NF_ACCEPT;
}
```

Listing 3.9: Netfilter hook function

3.2.3 Netlink

Kernel modules should be small and simple. Hence it is important for most applications to shift functionality to the user space. To this end, a communication mechanism is needed, such that a client program can exchange data with the kernel space. This functionality is provided by netlink sockets [11].

Kernel Module Functions

Listing 3.10 shows how to create a netlink socket on the kernel side. `netlink_kernel_create()` creates the socket and `netlink_register_notifier()` registers an event notifier which is used to get informed about netlink events (e.g. connection closed by user space program). `NETLINK_USER` must be set to an arbitrary unique id. This id is used to reference to the correct netlink socket from the client program. `nl_recv_msg` is the name of the receive function. When a connected process sends data to the kernel space, this function is called.

```
1 #define NETLINK_USER 31

3 // Notifier event of netlink socket
  static struct notifier_block nl_notifier = { nl_event, NULL, 0 };
5
  // Netlink socket
7 struct sock *nl_sk;

9 nl_sk = netlink_kernel_create(&init_net, NETLINK_USER, 0, nl_recv_msg, NULL,
      THIS_MODULE);

11 // Setup netlink notifier
  netlink_register_notifier(&nl_notifier);
```

Listing 3.10: Open a netlink socket (kernel space)

A registered event notifier (see Listing 3.11) is a simple function with three arguments. The second argument (`event`) contains the type of the raised event (e.g. `NETLINK_URELEASE` when a client shuts down the connection). The third argument contains a pointer to a dedicated `netlink_notify` structure which can be used to access information about the user space application (PID and used protocol) which initiated the event.

```
  static int nl_event(struct notifier_block *this, unsigned long event, void *ptr) {
2     struct netlink_notify *n = ptr;

4     [...]

6     return NOTIFY_DONE;
  }
```

Listing 3.11: Fetching netlink events (kernel space)

The registered receive function `nl_recv_msg()` (see Listing 3.12) has exactly one argument which is used to access the received data and the PID of the sender process.

```
static void nl_recv_msg(struct sk_buff *skb) {
  struct nlmsghdr *nlh;
  int user_pid;

  nlh = (struct nlmsghdr*)skb->data;
  MESSAGE("Netlink received msg payload: %s\n", (char*)nlmsg_data(nlh));
  user_pid = nlh->nlmsg_pid; /* pid of sending process */
}
```

Listing 3.12: Receive function for netlink sockets (kernel space)

Listing 3.13 shows how to close a netlink socket from kernel side. For this purpose, the function `netlink_kernel_release()` must be called. In addition to closing the socket, the netlink event notifier has to be unregistered by calling `netlink_unregister_notifier()`.

```
// close socket
netlink_kernel_release(nl_sk);

// unregister notifier
netlink_unregister_notifier(&nl_notifier);
```

Listing 3.13: Close a netlink socket (kernel space)

User Space Functions

To exchange data with a kernel module, a user space program also has to open a netlink socket by calling `socket()` (see Listing 3.14). Netlink communication is done through a raw socket (i.e., `SOCK_RAW`) with the communication domain `PF_NETLINK`. In order to communicate with the correct kernel module, `NETLINK_USER` must be set to the same value in both, the kernel module and the user space application. Source and destination socket addresses must be bound to the `AF_NETLINK` address family. While for the source address the PID has to be set to the correct ID of the user space process, the PID of the kernel module must be set to 0 by definition. In order to communicate with exactly one kernel module (unicast), `nl_groups` of the destination address must be set to 0.

```
#define NETLINK_USER 31

struct sockaddr_nl src_addr, dest_addr;
int sock_fd;

[...]

void netlink_create(void) {
  sock_fd = socket(PF_NETLINK, SOCK_RAW, NETLINK_USER);

  if(sock_fd<0)
```

```
    return;

  memset(&src_addr, 0, sizeof(src_addr));
  src_addr.nl_family = AF_NETLINK;
  src_addr.nl_pid = getpid(); // self pid

  bind(sock_fd, (struct sockaddr*)&src_addr, sizeof(src_addr));

  memset(&dest_addr, 0, sizeof(dest_addr));
  dest_addr.nl_family = AF_NETLINK;
  dest_addr.nl_pid = 0;    // For Linux Kernel
  dest_addr.nl_groups = 0; // unicast
}
```

Listing 3.14: Open a netlink socket (user space)

In order to send data to the kernel module, `sendmsg()` must be called (see Listing 3.15). To this end, a `nlmsghdr` structure must be filled with the correct destination address, data and user PID.

```
#define MAX_PAYLOAD 1024 // maximum payload size

struct iovec iov;
int sock_fd;
struct msghdr msg;
struct sockaddr_nl dest_addr;

[...]

void netlink_send(char **data) {
  static struct nlmsghdr *nlh = (struct nlmsghdr *)malloc(NLMSG_SPACE(MAX_PAYLOAD));

  memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));
  nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
  nlh->nlmsg_pid = getpid();
  nlh->nlmsg_flags = 0;

  strcpy((char *)NLMSG_DATA(nlh), *data);

  iov.iov_base = (void *)nlh;
  iov.iov_len = nlh->nlmsg_len;

  msg.msg_name = (void *)&dest_addr;
  msg.msg_namelen = sizeof(dest_addr);
  msg.msg_iov = &iov;
  msg.msg_iovlen = 1;

  sendmsg(sock_fd,&msg,0);
}
```

Listing 3.15: Send data to a kernel module over a netlink socket (user space)

In order to receive data from the kernel side, `recvmsg()` must be called (see Listing 3.16). The received data is then filled into a `msghdr` structure. By the use of `NLMSG_DATA()`, the user space application can access a pointer to the data sent by the kernel module.

```
int sock_fd;
struct msghdr msg;

[...]

size_t netlink_rcv(char **data) {
  struct nlmsghdr *nlh;
  size_t size;

  recvmsg(sock_fd, &msg, 0);

  nlh = (nlmsghdr *)(msg.msg_iov->iov_base);
  size = nlh->nlmsg_len - sizeof(struct nlmsghdr);

  *data = (char *)NLMSG_DATA(nlh);

  return size;
}
```

Listing 3.16: Receive data from a kernel module by a netlink socket (user space)

A netlink socket is closed like any other socket by calling `close()` (see Listing 3.17).

```
int sock_fd;

[...]

netlink_close(void) {
  if(sock_fd > 0) {
    close(sock_fd);
    sock_fd = -1;
  }
}
```

Listing 3.17: Close a netlink socket (user space)

3.3 OWL (Web Ontology Language)

3.3.1 Overview

OWL [39] which is developed by the W3C [6] is a language to define ontologies used by the Semantic Web. OWL 2 is an extension of the original OWL language which augments expression capabilities.

An OWL ontology consists of *classes*, *object properties*, *data properties* and *instances*. A class contains meta information which is used by one or more instances. Classes and properties can be defined hierarchically (with the possibility of polymorphism), which means that child classes (respectively properties) inherit the characteristics from their parents.

An entity of a class is called instance. For example, a class can be *Human*, with two child classes *Man* and *Woman*. Some instances can be *Bob*, and *Alice*. *Bob* is an instance of *Man* and *Alice* is an instance of *Woman*.

Object properties are used to specify relations between instances. For example, the object property *marriedTo* can be applied to *Alice* with the value *Bob*. Hence Alice is married to Bob. For convenience, it is possible to specify such properties transitively or (which makes for this example more sense) symmetric. A semantic reasoner can then infer derived properties from such specifications.

Data properties are used to store some information to an instance. Consider the data property *age* which is of type integer. Applied to Bob it may has the value 34 and applied to Alice it has the value 30. Listing 3.18 shows the OWL ontology belonging to this example.

```
<?xml version="1.0"?>

<!DOCTYPE Ontology [
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<Ontology xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="http://www.semanticweb.org/people"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     ontologyIRI="http://www.semanticweb.org/people">
    <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
    <Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#"/>
    <Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#"/>
    <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
    <Declaration>
```

```
24          <Class IRI="#Human"/>
        </Declaration>
26      <Declaration>
            <Class IRI="#Man"/>
28      </Declaration>
        <Declaration>
30          <Class IRI="#Woman"/>
        </Declaration>
32      <Declaration>
            <ObjectProperty IRI="#marriedTo"/>
34      </Declaration>
        <Declaration>
36          <DataProperty IRI="#age"/>
        </Declaration>
38      <Declaration>
            <NamedIndividual IRI="#Alice"/>
40      </Declaration>
        <Declaration>
42          <NamedIndividual IRI="#Bob"/>
        </Declaration>
44      <SubClassOf>
            <Class IRI="#Man"/>
46          <Class IRI="#Human"/>
        </SubClassOf>
48      <SubClassOf>
            <Class IRI="#Woman"/>
50          <Class IRI="#Human"/>
        </SubClassOf>
52      <ClassAssertion>
            <Class IRI="#Woman"/>
54          <NamedIndividual IRI="#Alice"/>
        </ClassAssertion>
56      <ClassAssertion>
            <Class IRI="#Man"/>
58          <NamedIndividual IRI="#Bob"/>
        </ClassAssertion>
60      <ObjectPropertyAssertion>
            <ObjectProperty IRI="#marriedTo"/>
62          <NamedIndividual IRI="#Alice"/>
            <NamedIndividual IRI="#Bob"/>
64      </ObjectPropertyAssertion>
        <DataPropertyAssertion>
66          <DataProperty IRI="#age"/>
            <NamedIndividual IRI="#Alice"/>
68          <Literal datatypeIRI="&xsd;int">30</Literal>
        </DataPropertyAssertion>
70      <DataPropertyAssertion>
            <DataProperty IRI="#age"/>
72          <NamedIndividual IRI="#Bob"/>
            <Literal datatypeIRI="&xsd;int">34</Literal>
74      </DataPropertyAssertion>
        <SymmetricObjectProperty>
76          <ObjectProperty IRI="#marriedTo"/>
        </SymmetricObjectProperty>
78      <DataPropertyRange>
            <DataProperty IRI="#age"/>
```

```
80          <Datatype abbreviatedIRI="xsd:int"/>
        </DataPropertyRange>
82  </Ontology>

84

86  <!-- Generated by the OWL API (version 3.4.2) http://owlapi.sourceforge.net -->
```

Listing 3.18: OWL file representing the relations between Bob and Alice

3.3.2 Syntax

OWL 2 can be expressed in a variety of syntaxes:

RDF/XML: Any tool that supports OWL has to support the RDF/XML-syntax. Among others, it is used for data exchange between tools.

OWL/XML: The OWL/XML syntax is optimized for the use with XML tools. This syntax is used in the scope of this thesis.

Functional syntax: With the functional syntax it is easier to express the formal structure of an ontology.

Manchester syntax: The Manchester syntax is better readable for humans. In addition, it needs less space as an XML based syntax.

Turtle: Turtle is an optional syntax which is not standardized by the OWL working group. This syntax shall simplify the process of reading and writing RDF triples.

3.3.3 Profiles

OWL 2 also specifies subclasses of the language's structural specification which are better suitable for some applications. These subclasses are called *profiles* [37]. There are three kinds of profiles:

OWL EL: With restricting OWL to OWL EL, polynomial time algorithms can be used for standard reasoning purposes. OWL EL is mostly used for ontologies with lots of classes and properties, where expressive power can be interchanged for better performance.

OWL QL: OWL QL enables the use of normal database technologies in order to perform conjunctive queries in logarithmic space. OWL QL is suitable for lightweight (i.e., small amount of classes and properties) ontologies, or where it is necessary to inquire data by relational queries (e.g. by SQL). It is possible to inquire a huge amount of instances relatively quickly.

OWL RL: By the use of OWL RL, polynomial time reasoning is possible. Such algorithms can be implemented by using standard rule languages. It is useful for simple structured ontologies with a huge amount of individuals.

3.3.4 SPARQL

To query an ontology, SPARQL [40] can be used. Listing 3.19 demonstrates an example for retrieving all *marriedTo* relations.

```
PREFIX : <http://www.semanticweb.org/people#>
SELECT ?p1 ?p2
WHERE
{
  ?p1 a :Human .
  ?p2 a :Human .
  ?p1 :marriedTo ?p2 .
}
```

Listing 3.19: SPARQL query

The query applied to the ontology of Listing 3.18 returns the result shown in Table 3.1. Therefore, the reasoner correctly concluded that Alice is married to Bob and Bob is married to Alice.

p1	**p2**
Alice	Bob
Bob	Alice

Table 3.1: Result of the SPARQL query.

3.3.5 SWRL

Finally, it is possible to augment the ontology by SWRL rules [41]. Such a rule can be for example:

$$Human(?p1), Human(?p2), Human(?p3),$$
$$brother(?p1, ?p2), child(?p1, ?p3) \rightarrow uncle(?p3, ?p2)$$

This rule means that, if there exists a human `p1` which is has a brother `p2` and a child `p3`, then `p2` is an uncle of `p3`. The left part is called the *rule body* and the part right of the arrow

($\rightarrow$) is called the *rule head*. If all statements in the rule body are valid for a set of individuals, then the statements in the rule head are also valid.

In general, ontologies augmented with SWRL rules are not decidable, but in practice, the use of SWRL rules is restricted to DL-safe (Description Logic safe) rules. This subclass of SWRL retains decidability. Any SWRL rule is DL-safe if the rule head contains only variables which also occur in a *datalog atom* in the body [14]. Such a datalog atom is an atom with a predicate symbol which is not specified as a class or property in any axiom within the ontology.

For example, the rule above is DL-safe, since `p1`, `p2`, and `p3` are restricted to named individuals `Human(?p<x>)`. Hence, it is ensured that all variables are contained in datalog atoms.

CHAPTER 4

Concept Specification

This chapter presents a concept for a content aware network. To this end, possible use cases are defined from which all needed components are extracted. After describing these components, their interaction is illustrated by the use of sequence diagrams.

4.1 Use Cases

4.1.1 Overview

In order to specify the components needed for the content aware network, some use cases are defined which are shown in Figure 4.1.

4.1.2 Temperature of a Room

Assume that a facility manager wants to know the temperature of a specific room. To this end, he uses his computer to place a dedicated request to the content aware network.

Figure 4.2 illustrates the components needed to handle the temperature request. A special device, called smart router, has to decide to which network node this request must be forwarded. For this purpose, the smart router inquires a reasoning server, called resolver. The resolver then infers (by the use of a dedicated domain server) that a specific environmental server provides an exact and current temperature value of the requested room. After requesting this temperature value, the environmental server does a second request in order to retrieve the current values from all temperature sensors within the room. This request also reaches a smart router which determines (due to its knowledge about connected network nodes) that this type of request must be forwarded to one or more temperature sensors, located in the specified room. Since it is possible that the location also contains a terrace with outside temperature sensors, these outside sensors must be removed from the result set. Next, the smart router forwards the request to the

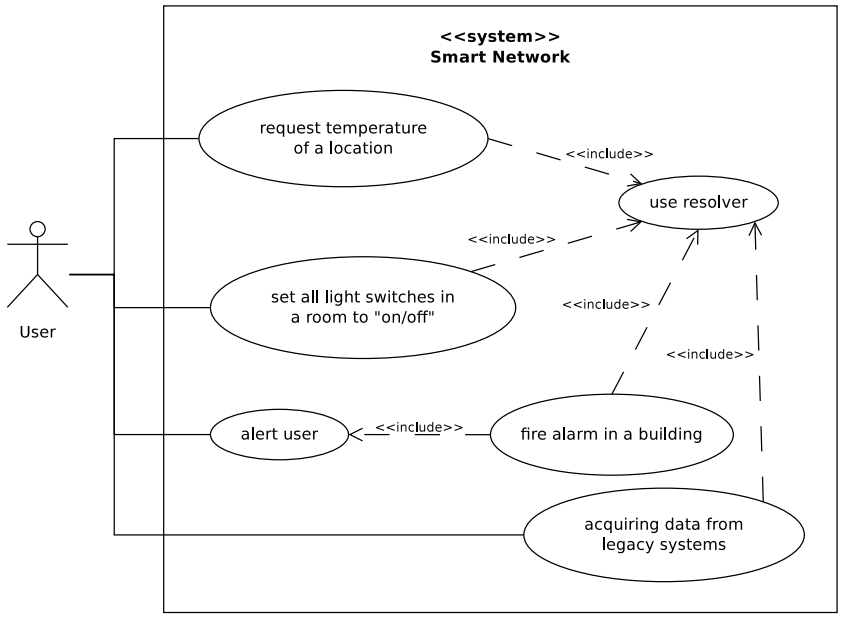


Figure 4.1: Use Case diagram.

final result set of temperature sensors. After all sensors have reported their current values to the environmental server, the server fuses these values by averaging and reports the result to the facility manager's computer.

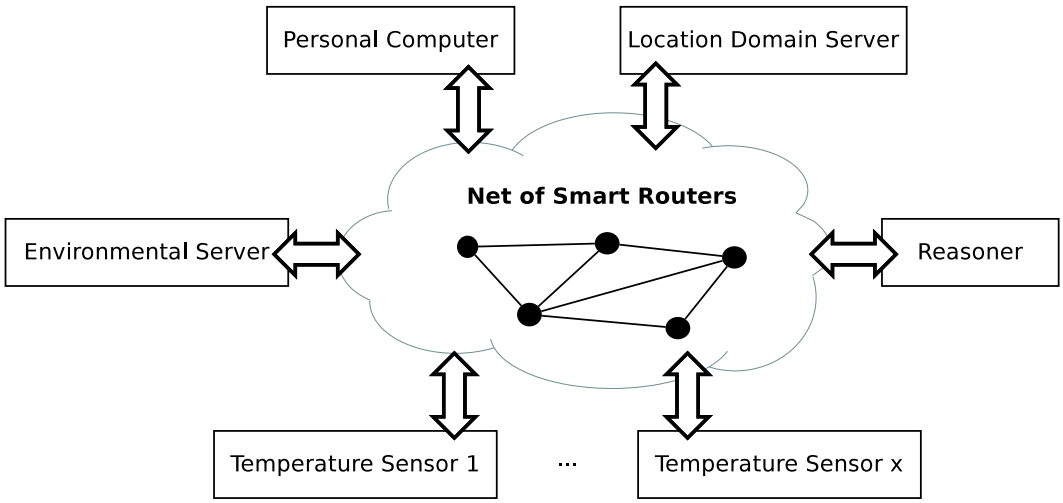


Figure 4.2: Components involved in a temperature request.

4.1.3 Weather in a City

Consider a user which wants to get a detailed weather report for his home city. The user takes his tablet computer and places a *weather* request to the content aware network.

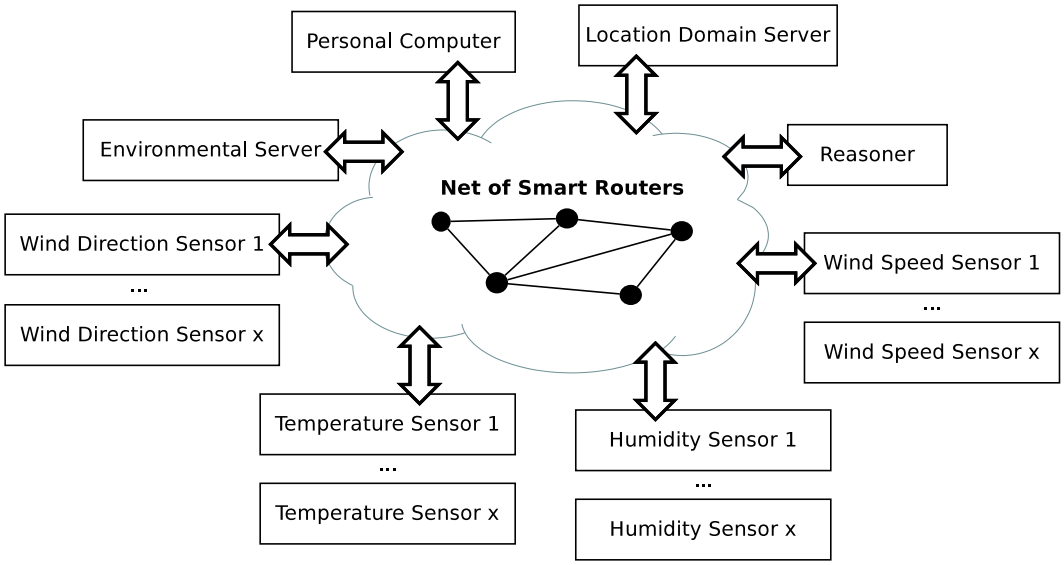


Figure 4.3: Components involved in a weather request.

The components involved in handling the request are shown in Figure 4.3. As above, a smart router receives the request and inquires a resolver in order to find a dedicated server which is able to process the request. The resolver augments the request with known facts and uses a location domain server in order to find a responsible server. Then, after forwarding the request to this server, the server has to place several additional requests in order to inquire some sensors. *Weather* is a concluded property which includes *outside temperature*, *outside humidity*, *wind speed* and *wind direction*. For each of these properties, a secondary request has to be made. After all sensors have reported their current values, the various sensor categories are fused and the results are sent back to the user's Tablet.

4.1.4 Fire Alarm

For a sensor, there can be one or more threshold values. If such a value is exceeded, the sensor sends an unaddressed alert message of a predefined severity level. Like in the examples above, the content aware network has to handle the alert and forward it to all corresponding network nodes. Figure 4.4 shows all components needed to handle temperature alerts.

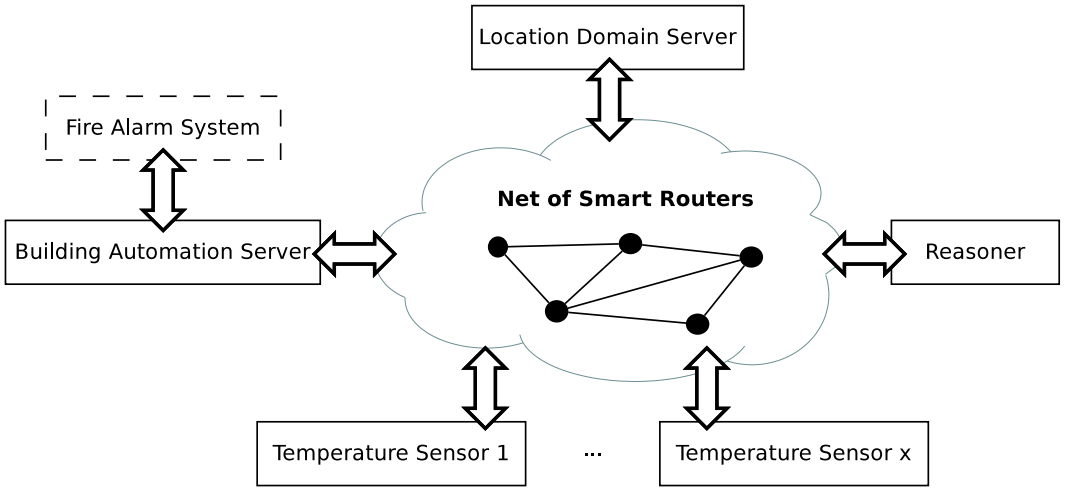


Figure 4.4: Components involved in handling temperature alert messages.

Now assume that two temperature sensors have exceeded their threshold values. Each single sensor sends an alert message which is finally forwarded to a dedicated building automation server. This server is able to cache alerts. Since the two alerts were within a specific period, the server has to alert a global fire alarm for the location to which these sensors belong to.

4.1.5 Light Control

Assume that someone wants to turn on all lights at his *location*. For this purpose, a request is sent to the content aware network by the use of a smart phone. According to the configuration, the meaning of the term *location* is either determined by a location property or by other facts (i.e., location of the client's subnet). Figure 4.5 shows the components which are involved in handling this request.

As above, a smart router catches the packet and forwards its semantic tag to a resolver. The resolver then determines (by the use of a dedicated domain server) the required network address which is in this case the address of a building automation server. After forwarding the request to this server, the server then generates another request in order to switch on all lights in the given domain.

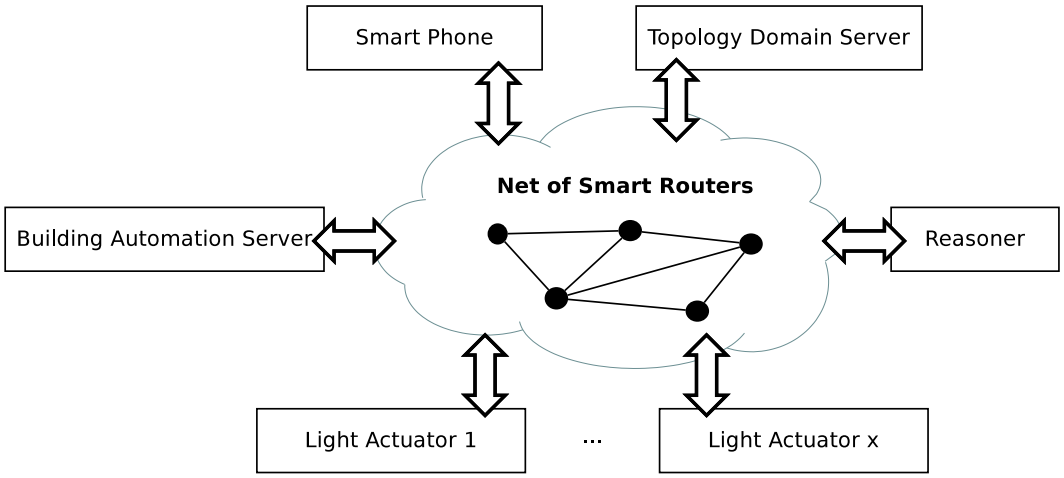


Figure 4.5: Components needed to turn on all lights within a specific domain.

4.1.6 Acquiring Information from Legacy Systems

Already available systems do not provide machine readable semantic data needed by a content aware network. For this purpose, a special device called *legacy gateway* is used (see Figure 4.6). The content aware network can access such a gateway like any other S/A-unit. The gateway takes control over protocol translations and configuration handling. A gateway can host one or more legacy sensors or actuators.

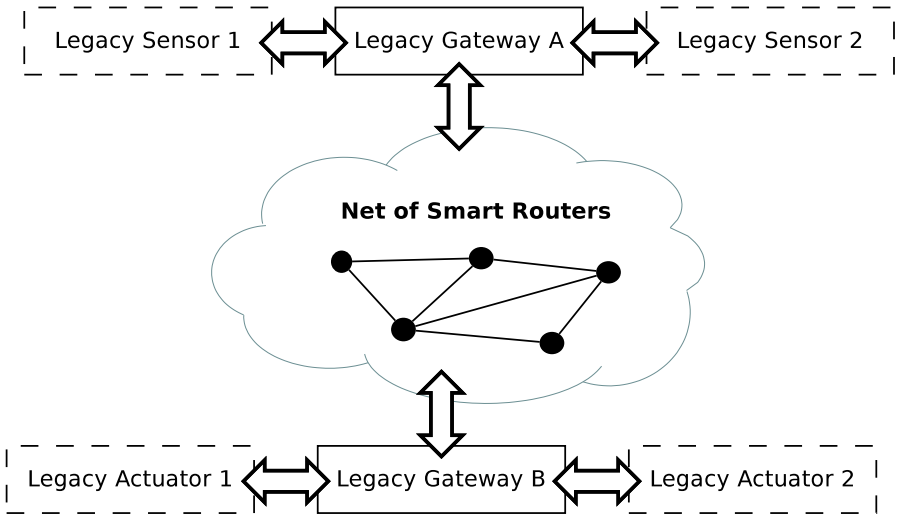


Figure 4.6: Components needed to connect legacy S/A-systems.

4.2 Component Specification

4.2.1 System Structure

Based on the use cases in Section 4.1, a content aware network must contain several components in order to provide the required functionality. Figure 4.7 shows the structure of such a typical content aware network. The various network components of the Figure are explained in the following subsections.

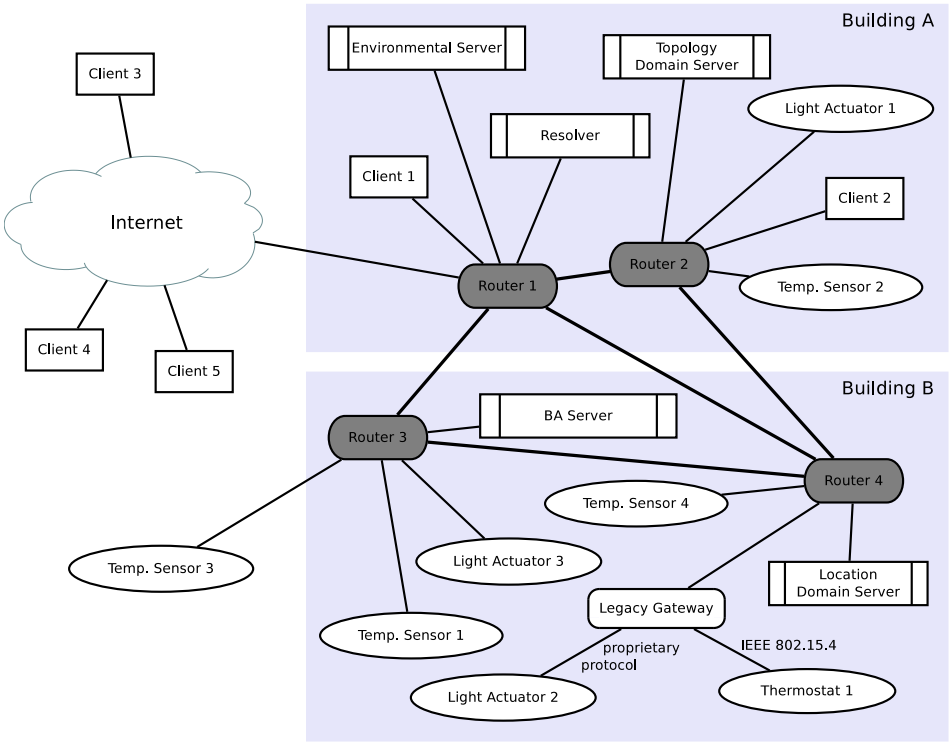


Figure 4.7: Typical network structure.

4.2.2 Client

A *client* is a participant of the network, sending either requests as standard IPv6 packets or as unaddressed packets which contain a semantic content description. A client may be a standard PC, a tablet computer, a mobile phone or any other networking device.

In the case of unaddressed requests, a client attaches a semantic tag to the packet, which contains its own IPv6 address and port, a random message id and an XML-formatted query. This

query specifies the domain of the request (subject), as well as some properties to which the final S/A-units must match.

An example for an XML-formatted query sent from a client is shown in Listing 4.1 (namespaces are omitted):

```
<query>
    <subject>Vienna<subject>
    <properties>
        <property type="object" name="temperature" />
    </properties>
</query>
```

Listing 4.1: XML-formatted query sent from a client

4.2.3 Smart Router

In addition to "normal" routing, a *smart router* is capable of handling unaddressed packets. These unaddressed packets are packets which should be routed according to their semantics, instead of specifying a dedicated destination address. In general, there are following cases for the use of unaddressed packets

Primary requests: A primary request is sent from clients (see Section 4.2.2) to the content aware network. Packets sent in the scope of a primary request are called *primary query packets*. An example for such a request is *get the current temperature in Vienna*. Primary query packets are always forwarded to dedicated servers (see Section 4.2.7).

Secondary requests: These requests are sent from a server in order to process a primary request. Packets sent in the scope of a secondary request are called *secondary query packets*. An example for such a request is *get the current values from all temperature sensors within a specific building*. A secondary query packet is always routed to a set of on demand S/A-units (see Section 4.2.4).

Data packets: These packets are sent from a multicasting S/A-unit when its internal state changes (e.g. when the measured value exceeds a specific threshold value)

By combining above types of unaddressed packets, there are two different strategies for handling such packets:

Information pull strategy: The information pull strategy is used for client requests. It specifies the combination of a primary and a set of corresponding secondary requests.

Information push strategy: The information push strategy is used to handle data packets sent from S/A-units.

While data packets are sent from S/A-units, a query packet is mostly used to request semantic information (e.g. aggregating or fusing sensor values, topology information, etc.). A smart router processes query packets, as well as data packets and forwards them to the correct destination. There may be more than one network node handling a specific packet type. In this case, either one of the destinations has to be chosen by the smart router, or the packet has to be forwarded to all nodes according to the packet information.

In order to be able to forward unaddressed packets, a smart router must have knowledge about connected network nodes within its domain. For this purpose, other network nodes have to report their aliveness to the content aware network in a predefined interval. This is done by sending a connect message which contains the node's (maybe already extended) XML-formatted configuration (see Listing 4.2). A smart router may also augment such a configuration by known facts (e.g. specify in which room a device is located). Finally this augmented configuration must be forwarded to interested servers.

In addition to normal routing tables, a smart router must provide the following tables:

CNT: The `CNT` (Connected Nodes Table) stores configurations of connected network nodes. On the reception of a connect message, the smart router stores the augmented configuration in this table. On the reception of secondary requests, this table is used to find appropriate S/A-units which match the request's semantic tag. To this end, a reasoner may be used to augment the semantic tag of the request. Table entries are of the format `<address, timestamp, configuration>`, where the configuration field contains XML-formatted settings from S/A-units. The table gets updated on the reception of connect messages of new or already connected nodes. Outdated entries are removed on every look-up, where each entry is checked against its timestamp. Since connect messages are sent in periodic intervals, the timestamp field should always contain the latest value for online nodes. Hence, old entries can be simply detected and removed. In this case, a (negative) connect message must be sent in order to inform other network components about the offline node.

ERT: The `ERT` (Executed Request Table) keeps track of already processed requests. This table is needed, since a smart router forwards all secondary requests to its neighbor routers. A smart router cannot know, if one of these other routers has already received the request. The table entries have the format `<source address, message id>`. On the reception of a secondary request, the entries are checked. A matching entry indicates that the smart router has already processed the request, which means that it can be discarded. If no matching entry is found, the request is processed, and the source address and message id are stored in the `ERT`. The table has to be implemented as a ring buffer. This implicitly removes outdated entries.

Finally, a smart router may be used for address assignment of connected network nodes. For this purpose, NDP or DHCPv6 may be utilized.

4.2.4 S/A-Unit

An *S/A-unit* (sensor/actuator unit) is a network node which contains one or more sensors or actuators. An S/A-unit may have a default configuration encoded as machine readable XML-data. This configuration may be augmented by smart routers or dedicated servers (e.g., an environmental server may extend the configuration of some temperature sensors). While a dedicated server configures an S/A-unit's behavior (e.g. threshold values, units, etc.), a smart router extends the configuration by known facts (such as the location).

Depending on its configuration, the behavior of a connected S/A-unit can be one of the following:

On demand: The network node is idle by default. Only when another network component inquires the node's state, an answer (in form of a response packet) is sent to this component. This behavior is used by the information pull strategy.

(Periodic) multicast: A network node configured as a periodic multicasting device reports its state in (equidistant or non-equidistant) intervals. The report is caught by the nearest smart router which can then forward the information to a dedicated source by inquiring a reasoner. This behavior is used by the information push strategy.

It is also possible that a component implements both types of behaviors (e.g. temperature sensor: periodic multicasting is used for fire alarms and on demand behavior is used for handling client requests).

An example for a configuration of a temperature sensor is shown in Listing 4.2 (namespaces are omitted).

```
<configuration>
    <connect>
        <connected>true</connected>
        <interval type="seconds">180</interval>
    </connect>
    <actuators />
    <sensors>
        <sensor>
            <type>temperature</type>
            <unit>Celsius</unit>
            <range>
                <min>-50.0</min>
                <max>50.0</max>
            </range>
            <accuracy type="double">0.5</accuracy>
            <behavior>
                <onDemand />
```

```
18              </behavior>
            </sensor>
20      </sensors>
        <net>
22          <ip version="6">fe80::1234</ip>
            <port type="UDP">1234</port>
24      </net>
        <location outside="true">
26          <city>Vienna</city>
            <coordinates>
28              <latitude>48.2004993</latitude>
                <longitude>16.3708093</longitude>
30          </coordinates>
        </location>
32  </configuration>
```

Listing 4.2: XML configuration of a temperature sensor

The `connect` field specifies if the node is already connected and defines the interval for sending connect messages (see Section 4.3.2). In addition, the S/A-unit's configuration includes several `sensor` or `actuator` entries, one for each sensor or actuator. These fields specify specific properties of a sensor or actuator. Some of these properties may be modified by a dedicated server (e.g., `behavior`), but some of them are specific to an S/A-type and cannot be modified (e.g. `range` or `accuracy`). The `net` field contains network specific settings, like the IP address and port. Finally, the location is stored in the `location` field. `net` and `location` can be initially set by a smart router to some default settings, but a dedicated server is able to change these settings to more specific values. If some property is not known (e.g. the location after the initial power-up of an S/A-unit), the corresponding field is left empty. This indicates that a smart router or a dedicated server has to augment these fields with some default settings. For instance, a default location can be determined by the location of the nearest smart router. Therefore, this smart router appends its own location to the configuration when receiving a dedicated connect packet.

4.2.5 Resolver

A *resolver* has the purpose of augmenting requests. A resolver inquires an ontology by using a semantic reasoner. Such a reasoner is able to infer not directly encoded facts (e.g. transitively defined locations, where a network node is situated). For this purpose, a resolver has knowledge about a smart routing ontology. Communication with a resolver is done by anycasting (see Section 2.1.2). After the detection of the type of request, the resolver augments the (maybe incomplete) XML-formatted request by known facts. To this end, the resolver uses a semantic reasoner (e.g. this reasoner can be queried by the use of SPARQL queries). After the known facts are added, the request is transformed to a query for a specific domain server. For example, the resolver detects that *Vienna* is an outside location and therefore augments the initial request

temperature in Vienna to the complete request *outside temperature in Vienna* and forwards it to a location domain server.

In addition, a resolver is also interested in connect packets sent from other network nodes (S/A-units, servers or legacy gateways) of specific types. Such connect packets are then transformed to instructions for domain servers in order to update their databases.

4.2.6 Domain Server

A *domain server* has knowledge about a specific domain. There can be many domain servers for a variety of domains. A domain server always handles detailed requests which were augmented by a resolver before. Like with resolvers, communication with domain servers of a dedicated type is done by anycasting (see Section 2.1.2). There are two possibilities to implement a domain server:

Passive domain server: The resolver adds all known facts to a request. Hence, the domain server does not need reasoning to process a request. A passive domain server can be a simple database server which can be queried by the use of *SQL*.

Active domain server: The resolver has only knowledge about a minimum ontology which is used to augment requests by obvious facts. This ontology can be extended by several domain ontologies which are then used to complete a request (i.e., returning a set of IPv6 addresses) by a dedicated domain server.

Since the resolver needs already a semantic reasoner to augment requests, in most cases the passive variant is sufficient. Only in special cases, where the resolver's ontology does not provide all needed meta information, active domain servers are needed. Examples for (passive) domain servers are:

Topology domain server: A topology domain server has detailed knowledge about the network topology. An example for a request, which a topology domain server is able to process is: *all temperature sensors within a specific subnet*.
A (simplified) SQL query can look like:

```
SELECT node.ipv6address
FROM node
WHERE node.type = 'temperatureSensor'
AND node.netId = '<ID>'
```

Location domain server: A location domain server is able to process requests about a specific location. To this end, such a server exactly knows which network node is situated in a specific location. Location domain servers are able to process requests like *outside temperature of Vienna*, or *weather of Vienna*.
A (simplified) SQL query can look like:

```
SELECT node.ipv6address
FROM node
WHERE node.type = 'environmentalServer'
AND locatedIn(node.ipv6address, 'Vienna') = 1
```

Here, `locatedIn()` is a stored procedure which checks recursively if the server is located in a specific location.

4.2.7 Server

A *server* handles addressed or unaddressed primary requests. The server has detailed information about the semantics of a specific primary query packet. With this information the server then may send a secondary request. This request eventually reaches a set of sensors or actuators. In the case of aggregating sensor values, the server applies a fusing function on the current sensor values as specified by the semantics of the request (e.g. an averaging function). In the case of actuators, the server forwards the desired actuator states to these nodes. Finally, the server sends back the requested information to the client.

An example for a request to an environmental server is *outside temperature of Vienna*. To this end, the server detects that it has to query outside temperature sensors located in Vienna and submits therefore the (secondary) request *outside temperatureSensor of Vienna*. In most cases, the secondary request has the same object properties as the primary request, but a different subject specified.

A more complex example is the request *weather of Vienna*, where a server has to query not only temperature sensors, but also humidity, wind speed and wind direction sensors by sending dedicated secondary requests.

In order to keep track of the membership of responses from secondary requests to primary requests, any server must contain at least following tables:

PRT: The `PRT` (Pending Request Table) stores the semantic tag of received primary requests. The table's entries have the format `<message id, request>`, where the `request` field contains all information from the primary request (i.e., source address, port and semantic tag). When all S/A-units have sent their responses to the server, the corresponding

entry in the PRT is used to find the origin of the primary request (i.e., the client). At this point the entry is removed and the final response is sent to the requesting client.

SART: The SART (S/A Response Table) contains responses from S/A-units. Its entries have the format <message id, response>. When all values belonging to a specific primary request are received, or after a specific timeout, the corresponding entries of this table are fused. To this end, all responses have the same message id as the initial primary request. After fusing the values and sending the result back to the client, the corresponding entries of this table are removed.

To register the server at other components, it has to send a connect message containing its configuration in predefined intervals to the content aware network. An example for an XML-formatted configuration for a server is shown in Listing 4.3.

```
<configuration>
    <connect>
        <connected>true</connected>
        <interval type="seconds">180</interval>
    </connect>
    <acceptedPropertySets>
        <propertySet>
            <properties>
                <property type="object" name="temperature" />
            </properties>
            <allowsAdditionalProperties value="true" />
        </propertySet>
    </acceptedQueries>
    <net>
        <ip version="6">fe80::5678</ip>
        <port type="UDP">1212</port>
    </net>
    <location>
        <building>Building A</building>
        <room>E01.23</room>
    </location>
</configuration>
```

Listing 4.3: XML configuration of an environmental server

The fields connect, net and location have the same meaning as in Listing 4.2. The entry acceptedPropertySets describes which types of client queries are accepted by this server. To this end, the field allowsAdditionalProperties specifies whether a client's query can contain additional property fields, like

<property type="data" name="isOutside" />.

By sending connect packets containing such configurations, a domain server is able to find dedicated servers which are able to handle a specific client request.

4.2.8 Legacy Gateway

To connect legacy systems to the network, a *legacy gateway* is used. A legacy gateway is represented to the outer port as a normal S/A-unit. Internally it resolves the communication for one or more legacy systems. A legacy gateway has therefore the capability of minimal routing. In addition, it adds semantics to the connected systems.

The configuration of a legacy gateway is done like the configuration of a normal S/A-unit. To this end, the legacy gateway must support a default configuration for its connected sensors and actuators. This configuration must entail all default information for the connected S/A-types (e.g. type, unit, range, etc.). Since a simple sensor or actuator does not support a semantic configuration, the legacy gateway must map the behavior of its connected S/As to match the configuration. This includes for example unit conversion and S/A monitoring (i.e., periodic polling of S/A states) for the use of the information push strategy.

Possible default configurations can be stored in a file, or, for larger configuration sets, in a database. Therefore, the only thing which has to be done manually is to specify the types of connected sensors and actuators. The corresponding default configuration is then fetched from this database. In addition to the normal configuration of an S/A-unit, like in Listing 4.2, a configuration for a legacy gateway must also contain protocol information (i.e., information for communicating with sensors or actuators)

To resolve requests, a legacy gateway has to check the packet's semantic tag. In the case of actuators, it has to set all its matching actuators to the requested state and send a corresponding response message. In the case of sensors, all matching sensor values must be fused according to the gateway's internal configuration (e.g. by averaging, retrieving maximum or minimum values) and the result has to be sent back to the requesting server.

4.3 Networking

4.3.1 Overview

This section deals with used network protocols, packet types, the behavior of network components, and the techniques used for content aware smart routing.

To enable semantic parsing of network packets, several packet types are specified:

- Connect packet
- Configuration packet
- Query packet
- Response packet
- Data packet

Connect packets are sent from *S/A-units*, *servers* and *legacy gateways* after an initial power up or (to report about a network node's liveness) in predefined time intervals. A connect packet comprises the node's type, as well as an XML-based configuration. This configuration can be augmented by a smart router or a server. If a server alters the configuration of a network node, it sends a configuration packet to this unit. The payload of this packet contains also such an XML configuration. After receiving a configuration packet, a network node has to modify its internal state according to this configuration. This may result in sending a new connect packet in order to inform other network components about the node's modified configuration.

Query, Data and Response packets are used for common operation. Query packets containing primary requests are used for the information pull strategy. They are mostly sent by a client which is requesting some data from the content aware network. Secondary requests are sent by servers in order to process primary requests. Replies to any kind of requests are sent as response packets which contain an XML-formatted payload. Data packets are used for the information push strategy. They are sent by an S/A-unit after the appearance of a dedicated event to provide some information to the content aware network. In order to reference to the correct message sequence, requesting and data packets have to support a unique message identifier (*message id*). All subsequent packets (responses, derived requests, etc.) have to use this message id.

4.3.2 Connecting (Content Aware) Network Nodes

If a new content aware network node (*S/A-unit*, *server* or *legacy gateway*) is connected to the network, a special protocol has to be employed in order to register this component at other network nodes. In general, this protocol sequence must be performed in predefined intervals.

On the initial power-up, the new network node first gets its network address assigned by the auto-configuration feature of IPv6. Afterwards, it sends its factory defined configuration to the content aware network as shown in Figure 4.8 (1). The first smart router catches the packet and augments the configuration by known facts (e.g. the router's location). The smart router stores the newly connected node in its `CNT` (Connected Nodes Table). Now the smart router forwards

the semantic tag of the connect packet to a resolver in order to find interested servers (2). The resolver extends the request by known properties (gained from its ontology), transforms it to an SQL query and forwards it to a dedicated topology domain server (3). The topology domain server returns a set of IPv6 addresses of interested servers to the smart router (4) which then forwards the connect packet to these addresses (5). In addition, the resolver has to inform the domain servers about the newly connected node in order to update their databases (not shown in the figure). Finally, the servers may also modify the node's configuration and send it back to the smart router (6), which forwards it to the node (7).

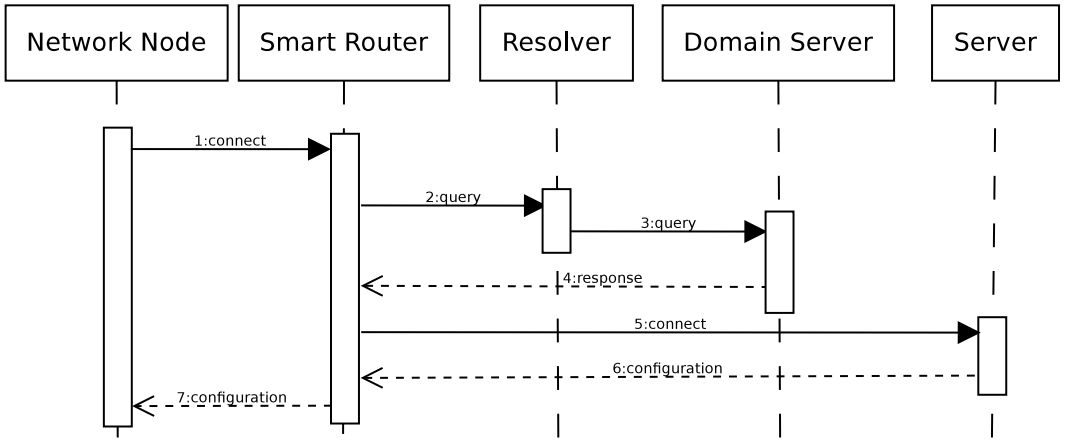


Figure 4.8: Messages sent after connecting a new S/A-unit.

In addition to this initial sequence, a connect packet is sent in predefined intervals in order to report about a node's liveness. As above, the node may get reconfigured by one or more servers. Finally it is possible that a server sends just a configuration message to a (already connected) node (e.g. because of a modification of the network topology).

4.3.3 Behavior of S/A-units

Information Pull Strategy

The information pull strategy is used for unaddressed requests (e.g. when a client requests the temperature of a room). It is distinguished between *primary* and *secondary requests*. All messages sent in the scope of this strategy can be counted to either one of these requests. While a primary request is sent initially by a client requesting information from the content aware network, a secondary request is initiated by a server in succession to a primary request.

Figure 4.9 shows the sequence of all relevant messages needed by the information pull strategy. First, the client sends an unaddressed primary request to the content aware network (1). The nearest smart router detects the request and reroutes its semantic tag to a resolver (2). Next, the resolver extends the request with known facts from its ontology. After determining the correct domain, the resolver transforms the request in order to forward it to a dedicated domain server (3). Then, the domain server processes the request. Afterwards, the result, containing a set of server IPv6 addresses, is returned as a response packet to the smart router (4). To conclude the primary request, the smart router forwards the primary request to all of the servers contained in the response packet's payload (5).

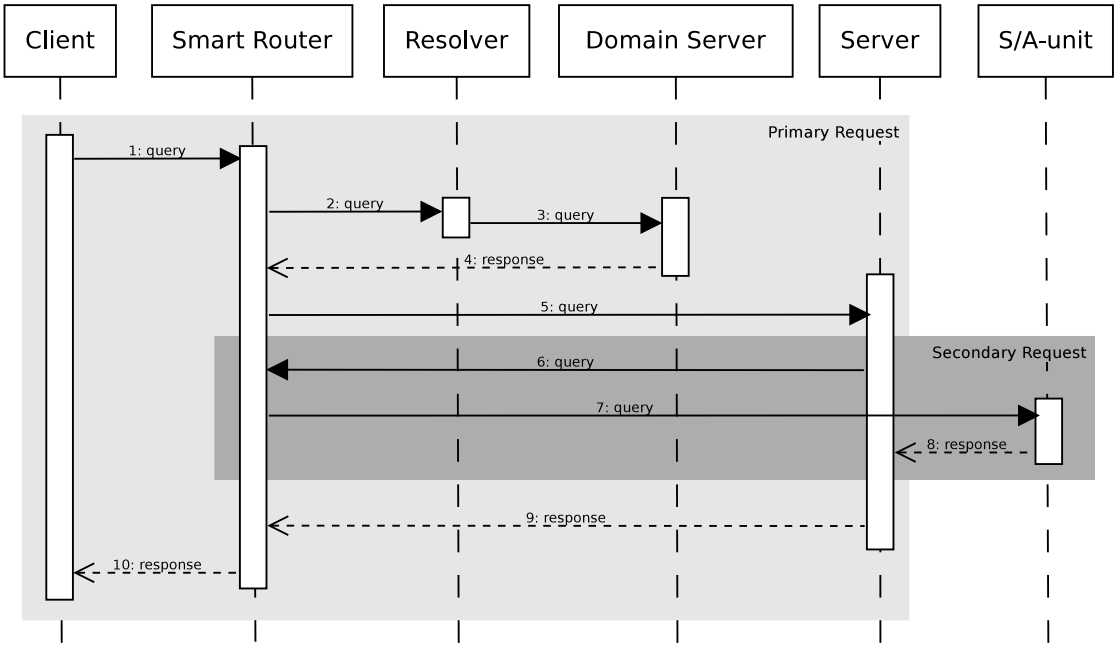


Figure 4.9: Messages sent in the information pull strategy.

After receiving and parsing a primary request, a server sends an associated secondary request (6). This request retrieves all relevant S/A-units. The next smart router forwards this secondary request to the desired S/A-units (7), as well as to other connected smart routers (not shown in the figure) which recursively forward the request. To this end, the smart router may use a resolver, as in the steps (2-4), if the request needs to be augmented (not shown in the figure). After performing the tasks specified in the packet, the S/A-units return the result to the corresponding server (8). In the case of sensor nodes, the server aggregates the results of all units (e.g. by averaging) and returns the aggregated result (9), which is forwarded to the client by the next smart router (10). In the case of actuator nodes, the server sends an acknowledgement (9),

which is also forwarded to the client by the next smart router (10).

Information Push Strategy

The information push strategy (see Figure 4.10) is used by S/A-units for multicasting of state changes (e.g. for threshold alerts of sensor nodes). To this end, an S/A-unit sends an unaddressed data packet to the content aware network (1). The next smart router catches the packet and forwards its semantic tag to a resolver (2). The resolver augments the semantic tag by known facts from its ontology and forwards it to a (topology) domain server (3). After the (topology) domain server returns the addresses (4), the smart router reroutes the initial data packet to all addresses contained in the resolver's response (5). According to the type of the data packet and the server's configuration, the server may execute some tasks, initiates state changes of other S/A-nodes (by sending a secondary request like in the information pull strategy), reports the state change to other systems, or simply caches the S/A-unit's current state for faster responding to future primary requests.

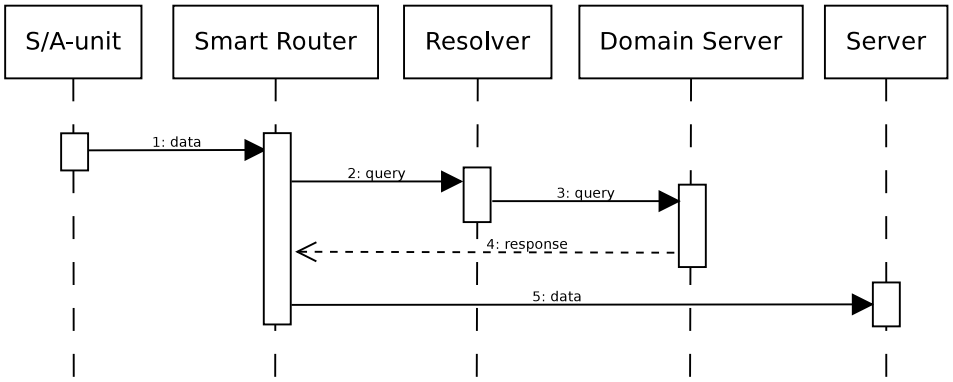


Figure 4.10: Messages sent in the information push strategy.

4.3.4 Range of Recipients

The range of recipients of an unaddressed packet can be limited by the following techniques:

Filtering properties: By the use of filtering properties, a resolver can limit the range of recipients naturally (i.e., using the `locatedIn` property)

Hop limit: The *hop limit* field of the IPv6 header replaces the `TTL` (time to live) field of the IPv4 header. It specifies how many routers the packet is allowed to pass before it has to reach its destination. Hence, an unaddressed packet's recipients can easily be limited by the amount of hops (i.e., all temperature sensors within a maximum hop distance of 3).

(Semantic) Hop count: This custom field is similar to the hop limit field mentioned above. The difference is that it is zero when an unaddressed packet is initially sent from an arbitrary network node. The value is incremented at each smart router by one, so the decision-making of dropping or allowing packets is not done at the sender but within the network infrastructure.

4.4 Sensor Fusion

4.4.1 Overview

Fusion of sensor values at dedicated servers can be done in different ways. One of the simplest forms is averaging. The problem of calculating the mean of a set of sensor values is that wrong measured values distort the result. To this end, more sophisticated fusing functions can be applied.

4.4.2 Confidence-Weighted Averaging

Confidence-weighted averaging [8] is based on (positive valued) penalties for unrealistic observations, which is done by the use of a confidence marker. This confidence marker is calculated by the reciprocal value of the statistical variance of a sensor. In the best case, the variance is 0, which means that the observation has the highest confidence. Following formula calculates a confidence-weighted mean of a set of sensor values:

$$\overline{x}_{fused} = \frac{\sum_{i=1}^{n} x_i \frac{1}{\mathbb{V}[S_i]}}{\sum_{i=1}^{n} \frac{1}{\mathbb{V}[S_i]}}$$

Here, n is the number of sensors, $\mathbb{V}[S_i]$ is the variance and x_i is the measured value of sensor i. Although it represents the best case, a variance of zero (i.e., $\mathbb{V}[S_i] = 0$) should not be used in order to avoid division-by-zero issues. For best results, heterogeneous sensors (i.e., sensors, which measure the same quantity in a different way) should be used.

The problem of *confidence-weighted averaging* is the determination of a sensor's variance. A solution to this is to perform several observations subsequently and calculate the empirical variance from these measurements by the typical formula:

$$\sigma^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2$$

On the one hand, this means that for a single sensor value multiple observations must be performed. On the other hand, a single faulty sensor which always provides the same (wrong) value does not get penalized.

4.4.3 Adaptive Algorithm

The *adaptive algorithm* provides another possible fusing function. This iterative algorithm first calculates the mean of all observations. Next, the measurement with the highest deviation is discarded. In the next iteration only the remaining sensor values are fused. The total amount of iterations can be either limited hard coded, or the algorithm stops, when the largest deviation is lower than a specified maximum value m. Figure 4.11 shows an example of four sensor values. Since x_4 has the largest deviation from the mean x_{avg}, the sensor value is discarded in the first iteration. In the next iteration, all sensor values are within the specified maximum interval $2 \cdot m$. Therefore, the calculated average value is accurate enough and the algorithm stops.

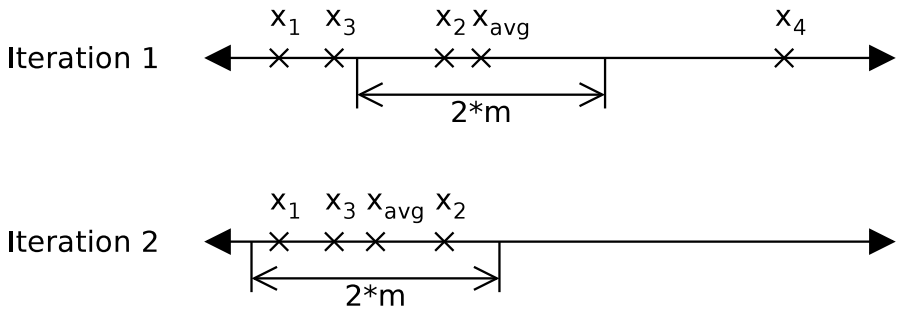


Figure 4.11: Adaptive algorithm - x_4 is discarded.

The advantage of this algorithm is its simplicity, but it may take some iterations before a highly accurate solution is found.

CHAPTER 5

Reference System Implementation

This chapter deals with implementation details of the reference system. The reference system provides an elementary set of features needed for *Content Aware Networking*. The implementation is based on a set of standard PC applications programmed in C and Java.

5.1 Proof-of-Concept

The concept describes in a very general manner how a content aware network is able to process unaddressed packets. While for the purpose of a proof of concept some not absolutely necessary features are omitted, some other restrictions are made in advantage to the performance of the network.

Since for the functional principle of the content aware network it makes no difference if the information push or pull strategy is used, only the more complex information pull strategy is implemented. To add the functionality of the information push strategy, just another packet type (data packets) must be supported by the network components.

The reference system uses an ontology which describes all needed (meta-)information (see Appendix A). The resolver also includes the functionality of the domain servers. While for large real-life networks it is feasible to separate these components for more flexibility, the reference implementation provides just a small set of locations and network nodes. Therefore, it is not necessary to add extra domain servers. As a side effect, this simplification also reduces the amount of transmitted network packets and hence the overhead of the content aware network.

The concept requires that also *servers* and *legacy gateways* send connect packets. In the case of the reference implementation, these connect packets are limited to *S/A-units*. The topology of smart routers and servers is stored within the ontology and can hence be used for reasoning by

the resolver. In addition, in the reference implementation connect packets are only processed by smart routers. Therefore, the forwarding of connect packets to servers is not implemented.

To make use of the reference implementation, a smart (demo) temperature sensor is implemented, providing random values. Since no other S/A-units are implemented, semantic tags are not encoded in XML, but directly encoded in the packet headers in a way that is more space efficient than XML. In addition, client requests are not encoded in XML, but there are fixed fields to add one subject and one object property. For the purpose of flexibility, the concept allows multiple object properties in one primary request. In general, this can also be achieved by specifying concluding object properties within the used ontology and is therefore not implemented. One possibility to specify concluding properties is by the use of SWRL rules. For example, the following rule can be used to request a fused temperature value which must be highly accurate:

$$\begin{aligned} &Location(?loc), Server(?serv), \\ &\quad temperature(?loc, ?serv), \\ &\quad\quad highAccuracy(?serv) \rightarrow highAccuracyTemperature(?loc, ?serv) \end{aligned}$$

5.2 Semantic Representation

To facilitate fast responses to unaddressed packets, a special language has to be employed. For this purpose, OWL [39] [38] is chosen. OWL is an XML-based language for defining ontologies which is standardized by W3C [6]. In addition, there are several reasoners (e.g. FaCT++ [30], Pellet [4]) and query languages (e.g. DL Query [32], SPARQL [40]) available. For the purpose of smart routing, a resolver contains a semantic reasoner to access OWL ontologies.

Appendix A contains the ontology used by the reference system implementation. Since the implementation's resolver also includes features of the concept's domain servers, the ontology contains also domain specific data. Figure 5.1 shows the class hierarchy used in the ontology. In general there are two super classes, namely *Location* and *Networknode*. A location can be more specified as *Room*, *Building*, or *City*. A location may be inside another location, which is specified by the *locatedIn* object property (see Figure 5.2). By its nature, this property is recursively specified. For this property there also exists an inverse (*locatedInInv*) which references all child locations. A network node may be a *server*, a *sensor*, an *actuator* or a *smart router*. A network node may also be within a specific location. Therefore, it may have specified the *locatedIn* property as well. In addition, a network node has a network address, defined by the *ipv6address* data property and a network node can be connected to one or more other network nodes, which is specified by the *connectedTo* object property. In order to get the right server or S/A-unit for a specific request type and location, some additional object properties are de-

fined: *temperature* and *humidity* retrieve a temperature or a humidity server for a given location, while *temperatureSensor* and *humiditySensor* retrieve sensor nodes which are able to measure temperature.

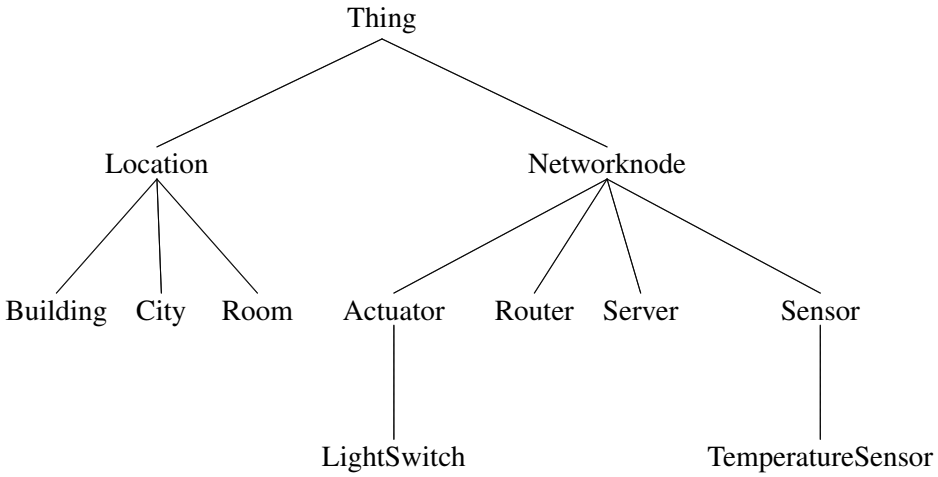


Figure 5.1: Class definitions contained in the ontology.

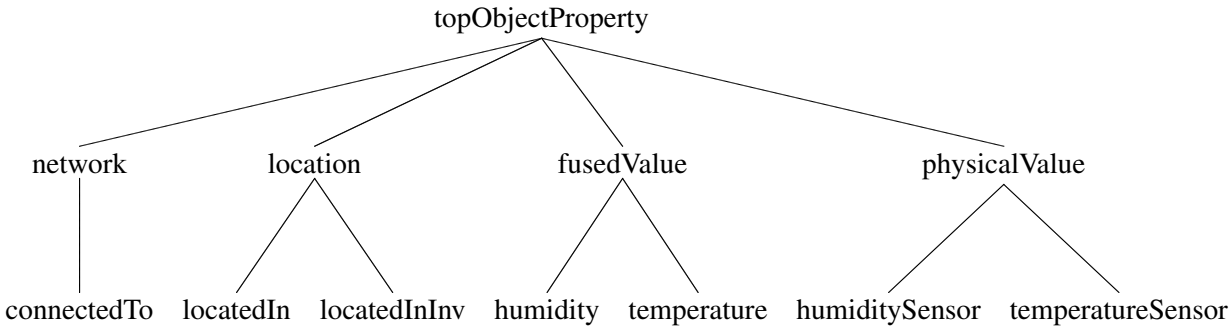


Figure 5.2: Object properties contained in the ontology.

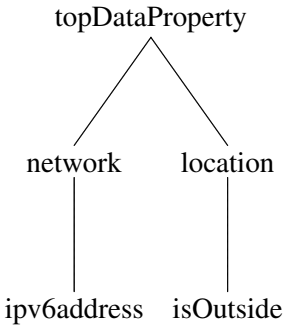


Figure 5.3: Data properties contained in the ontology.

In addition to the representation of spatial and topological properties of locations and network nodes, this file also include SWRL rules used by the reasoner. Such a rule is for example:

$$Location(?loc), Location(?superLoc),$$
$$Server(?serv), locatedIn(?loc, ?superLoc),$$
$$temperature(?superLoc, ?serv) \rightarrow temperature(?loc, ?serv)$$

This rule implements a transitive relation for the `temperature` property based on the `locatedIn` property. For non-transitive properties no such rule is required.

5.3 Smart Packet Types

5.3.1 Common General Header

For a controlled message exchange several smart packet types are defined. All of them have a common general header which is shown in Figure 5.4

Figure 5.4: Common semantic header of 6 bytes length.

The `message type` field defines the type of the smart packet. Possible values are:

1: connect packet
2: query packet

3: response packet sent from a resolver
4: response packet sent from a server
5: response packet sent from an S/A-unit
other: undefined

The `source port` field is set to the receiving port of the source of the packet. It is used by the receiver to answer a packet. The `message id` field, which should be set to an arbitrary 16-bit number, is used to reference to requests sent from a client by several components of the system. After the client has set the message id for its query packet, all related packets should use the same message id. The `hop count` field (see Section 4.3.4) should be set initially to 0 for each packet. Every smart router increments this value by 1 in order to count how many smart routers the packet has passed yet. Network components, such as smart routers or servers may accept or deny packets with hop counts larger than a specific value.

5.3.2 Connect Packet

A connect packet is sent from an S/A-unit after an initial power up, as well as after a change of its configuration. In addition, an S/A-unit may send connect packets in predefined intervals to refresh the smart routers' `CNTs`. The header of a connect packet consists of the common general header (see Figure 5.4) and the additional fields shown in Figure 5.5.

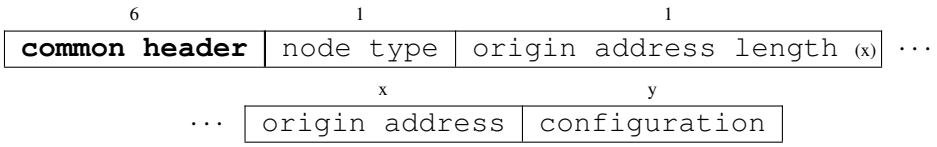


Figure 5.5: Semantic header of connect packets.

The `node type` field defines the type of the connected S/A-unit. The following node types are possible:

1: sensor node
2: actuator node
other: undefined

The next two fields `origin address length` and `origin address` specify the IPv6 address of the connected S/A-unit in abbreviated string formatted notation. This notation is used instead of the binary representation due to less space requirements for short addresses and better readability when observing message exchanges with a protocol sniffer. These fields are required, since forwarding packets of smart routers results in sending new packets. Therefore,

the original IPv6 source address field contains the address of the forwarding router. Due to the fact that smart routers may want to know from which other router a packet was forwarded, this approach has been chosen.

Unlike to the concept's XML-formatted configuration, for the actual implementation, the `configuration` field has the format shown in Figure 5.6.

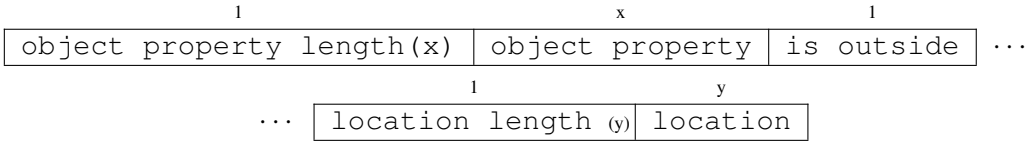


Figure 5.6: Configuration of a temperature sensor.

The fields `object property length` and `object property` define the accepted object property value. This value is specific to a type of S/A (e.g. in the reference implementation *temperatureSensor* is used for getting current values from temperature sensors).

The remaining fields may be left empty by an S/A-unit and will be overwritten during the configuration process: `is outside` defines if the node is inside or outside of a building and `location` is set to a string representing the place where an S/A-unit is located (e.g. "Vienna", "Office").

5.3.3 Query Packet

There are two main types of query packets, namely *primary* and *secondary query packets*.

Unaddressed primary query packets are sent in the scope of initial requests (i.e., primary requests), like tuples <object property, location> (e.g. <temperature, Vienna> to inquire the current temperature value of Vienna) or triples <object property, location, value> (e.g. <light, Office, on> to set all lights in the Office to the state "on").

Usually unaddressed secondary query packets are always sent from servers in the scope of a secondary request in order to address a set of S/A-units.

Additionally there are types of query packets used by smart routers to communicate with a resolver. These packets are *property*, *sub location* and *connected nodes requests*. A property request is used to get supported properties of a server, a sub location request retrieves all child locations belonging to a location, and a connected nodes request lists all connected network components of a specific type (servers, smart routers, resolvers, etc.) which are connected to a specific network component. The header of a query packet is shown in Figure 5.7.

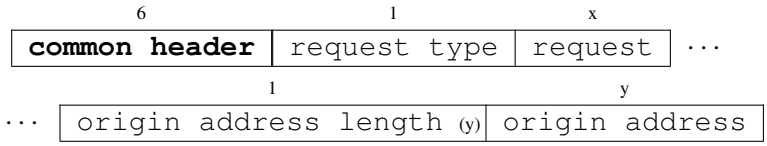


Figure 5.7: Semantic header of query packets.

As mentioned above, there are five kinds of supported request types of query packets:

1: primary request
2: secondary request
3: property request
4: sub locations request
5: connected nodes request
other: undefined

According to the concept, the query field contains XML-formatted data, but for the purpose of less space and resource requirements, the query field in the implementation is of the format specified in Figure 5.8.

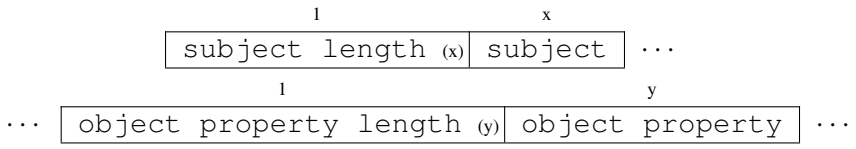


Figure 5.8: Request field of query packets.

The fields `subject length` and `subject` specify the subject this request is about (e.g. a location like *Vienna*) whereas the fields `object property length` and `object property` specify a restriction of destination nodes (e.g. *temperature* for fused temperature values).

The fields `origin address length` and `origin address` store the IPv6 address in abbreviated string notation of the sender of the request. Like above, the address is not stored in binary form because of better readability and less space requirements for short addresses.

5.3.4 Response Packet

Replies to query packets are sent as response packets. The packet header of a response packet is shown in Figure 5.9.

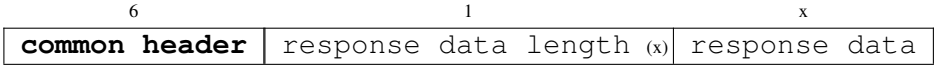


Figure 5.9: Semantic header of response packets.

According to the concept, the response data shall be XML-formatted, but for the purpose of less space and resource requirements, data records are divided by special characters. E.g. this is used by the implementation of the resolver when a request of IPv6 addresses delivers multiple results.

5.4 Network Components

5.4.1 Smart Router

The configuration of a smart router for standard IP packets is done by Linux iptables, provided by the packet filtering framework *Netfilter* [23] (like for "normal" routers). The additional smart routing paradigm is provided by a custom software component. This software component consists of two sub-assemblies: A kernel module (written in C) and a user space application (written in C++).

Kernel Module

The kernel module implements a *Netfilter Hook* [23] to catch all semantic packets (see Section 3.2.2). The hook is registered at `NF_INET_PRE_ROUTING` in order to filter all incoming packets. A semantic packet is always an IPv6 packet sent to a dedicated UDP port (in the case of the used configuration: 4444), no matter which destination the packet has (in fact the destination is not known by the sender). If a semantic packet is found, it is marked as `NF_STOLEN`, which means that the packet is accepted but not forwarded to its destination (which is in fact not known yet). The stolen packet is then sent to the user space application by the use of *Netlink Sockets* [11].

User Space Application

The user space application receives all semantic packets from the kernel module. According to the packet type, one of the following procedures is applied (see Figures 5.10 and 5.11):

Connect packets: When receiving a connect packet, missing semantic information of the packet is augmented by known facts (such as `location`, `isOutside`). Then the newly connected S/A-unit is stored in a special table called `CNT` (Connected Nodes Table). Potential existing entries may get overwritten.

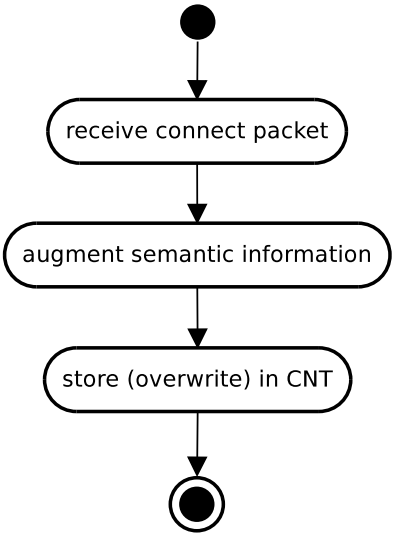


Figure 5.10: Activity diagram for processing connect packets on smart routers.

Primary requests: In the first step, primary requests (i.e., primary query packets) are forwarded to a dedicated IPv6 anycast address [10] to which all resolvers are listening. According to the principle of anycasting only the nearest resolver receives the packet.
Depending on the `object property` and the `subject` (see Section 5.3.3) the resolver answers with the addresses of one or multiple servers which are able to handle the request. Finally, the request is forwarded to these addresses.

Secondary requests: First, the smart router does a lookup in its `ERT` (Executed Request Table). If the secondary query packet (identified by its `message id`) was already processed before, it is discarded. Otherwise the request is stored in the `ERT`.
Next, the request is forwarded to all neighbor smart routers (these may also be smart routers where one or more ordinary routers are between). The addresses of these routers are determined by sending a *connected nodes request* to a resolver. While it would be possible to use a dedicated IPv6 multicast address (`ff02::<x>`) to which all smart routers are listening, this approach was chosen because of a greater flexibility (e.g. the ontology can contain rules such that the reasoner automatically limits the set of router addresses).
Finally, the request is forwarded to all connected S/A-units which match the semantic tag of the request. These addresses are determined by a lookup in the `CNT`.

In contrast to the concept, where table entries of the `CNT` are of the format `<address, timestamp, configuration>`, in the reduced implementation, instead of XML-formatted data, the configuration contains the fields `subject`, `object property` and `is outside`. The reference system does not remove outdated entries. Therefore, the timestamp field is not needed.

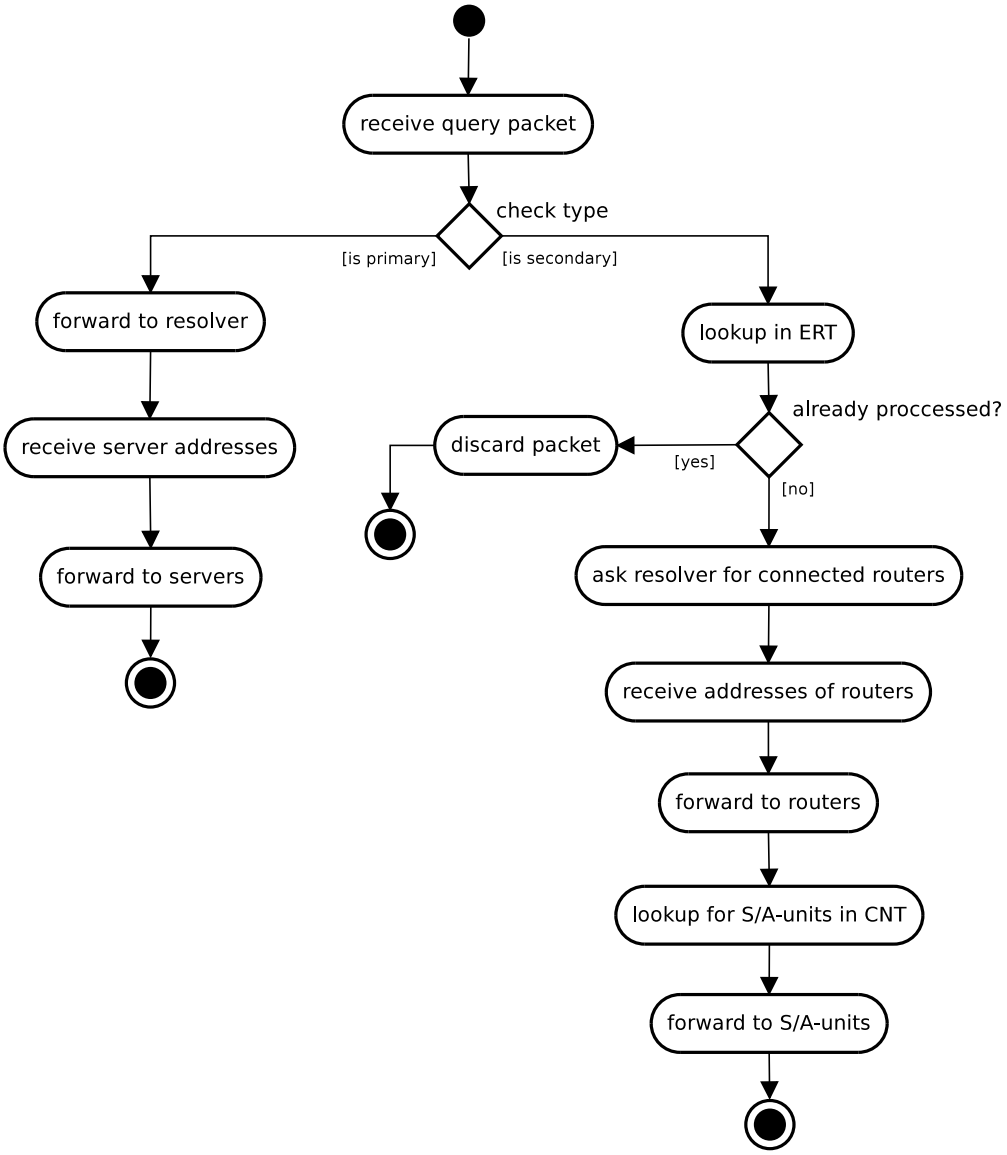


Figure 5.11: Activity diagram for processing requests on smart routers.

5.4.2 Resolver

The resolver is implemented in Java. It uses the OWL API [31] in combination with the Pellet reasoner [4] to access the ontology listed in Appendix A. The resolver is used for processing various semantic requests, like *property*, *sub location* and *connected nodes requests* (see Section 5.3.3). Since there are no domain servers in the reference implementation, the resolver also implements the functionality of location and topology domain servers. To this end, the ontology utilized here also contains servers and smart routers.

Resolvers listen to a dedicated IPv6 anycast address. This ensures that a query packet is always sent to the nearest resolver. If the nearest resolver is disconnected, the anycasting principle automatically updates routing tables to other resolvers.

5.4.3 Server

In order to provide elementary functionality, an *Environmental Server* is implemented, which is able to handle `temperature` requests (i.e., a request where the object property is *temperature*) by fusing multiple sensor values. The environmental server is also implemented in Java.

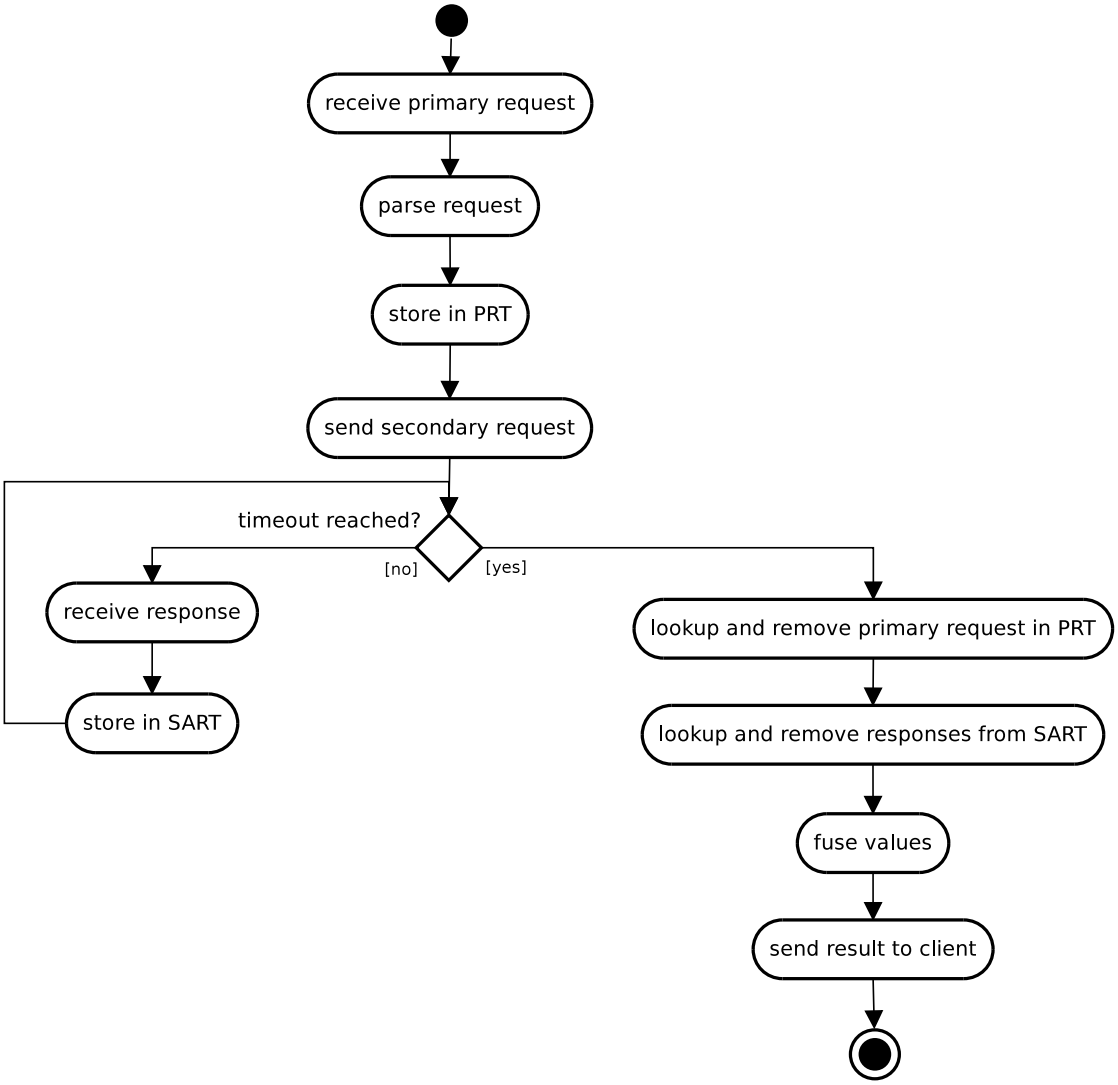


Figure 5.12: Activity diagram for processing primary requests on (environmental) servers.

Figure 5.12 illustrates the sequence of typical activities a (environmental) server has to process on the reception of a primary request. First, the server parses the request and stores it in its `PRT` (Pending Request Table). Afterwards, a secondary request with the same subject, but with a replaced object property (e.g. `temperature` becomes `temperatureSensor`) is generated. This request is then sent to the nearest smart router. Afterwards, a timeout is specified. During this interval responses from S/A-units are stored within the server's `SART` (Sensor/Actuator Response Table). By reaching the timeout, the sensor values are fetched from the `SART` and the corresponding pending request is fetched from the `PRT`. Now, the sensor values are fused and sent back to the client (which is determined by considering the semantic tag of the pending primary request).

5.4.4 S/A-Unit

In order to make use of the reference system, a demo temperature sensor is implemented. This Java application waits for *temperatureSensor* requests (i.e., a request where the object property is `temperatureSensor`) typically sent from a server. When such a request is received, a random temperature value is generated and the sender's address is extracted from the query packet. Finally, a response packet containing the (generated) temperature value is sent back to the source address.

5.4.5 Client

The client (implemented as a Java application) is used to access the content aware network. It waits for two keyboard inputs, namely a *subject* and an *object property*. With these two inputs a primary request is generated. Afterwards this request is sent to the network (without specifying a specific receiver's IPv6 address but with setting the UDP port to 4444), which initiates a sequence of message exchanges (see Section 5.5.3) within the content aware network. After the network has proceeded the request, the client receives a response packet containing the results (e.g. a fused temperature value).

5.5 Message Exchange

5.5.1 Protocols

For the message exchange between the network components, standard UDP packets over IPv6 connections are used. The payload of such a packet consists of the common general semantic header (see Section 5.3.1) followed by the semantic header for the dedicated packet type (see Sections 5.3.2, 5.3.3, 5.3.4). The packet is concluded by an optional payload of various length.

5.5.2 Connecting New Sensor Nodes

The (reduced) sequence of messages transmitted after connecting a new S/A-unit is shown in Figure 5.13. First, the S/A-unit sends a connect packet to the network (1). The next smart router catches the packet and augments the semantic tag by known facts (such as `location`). If some fact is not known yet, the smart router can optionally request this fact from a resolver (2) (e.g. the location of the router). The resolver then answers to this request either with the requested fact or with an error message (3). Finally, the smart router stores the augmented semantic tag in its `CNT` for processing future secondary requests.

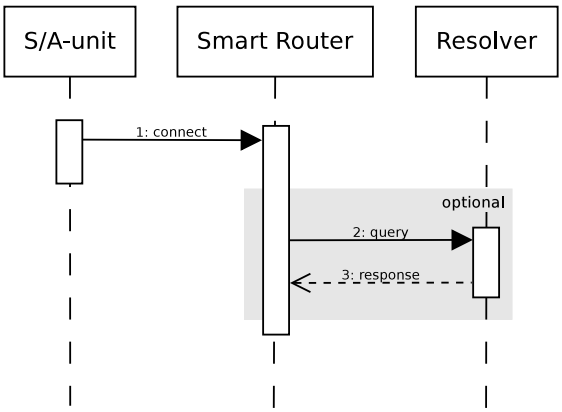


Figure 5.13: Messages sent after connecting a new S/A-unit.

5.5.3 Handling Client Requests

Figure 5.14 shows a typical sequence of UDP messages after a client has initiated a primary request. After a primary query packet is sent by the client (1), the next smart router catches the packet and queries a resolver for one or more server addresses (2). Next, the resolver answers either with a set of IPv6 addresses or with an error message if no address was found (3). Afterwards, the smart router forwards the primary request to these (server) addresses (4).

A server which received a primary request then initiates a secondary request (5). Like before, the next smart router fetches the request and forwards it to the connected S/A-units which match the semantic tag of the request (6). In addition, the request is forwarded to the connected smart routers (not shown in the figure), which also reroute the request to their connected S/A-units and smart routers (see Section 5.4.1). Every S/A-unit sends the result of the request (packed in a response packet) back to the requesting server (7). After fusing all responses at the server, the final result is sent back to the client (8). In contrast to the concept, where the final response is also caught by a smart router, the response message of the implementation is transparent to smart routers, since it is sent as an ordinary UDP message with a dedicated receiver.

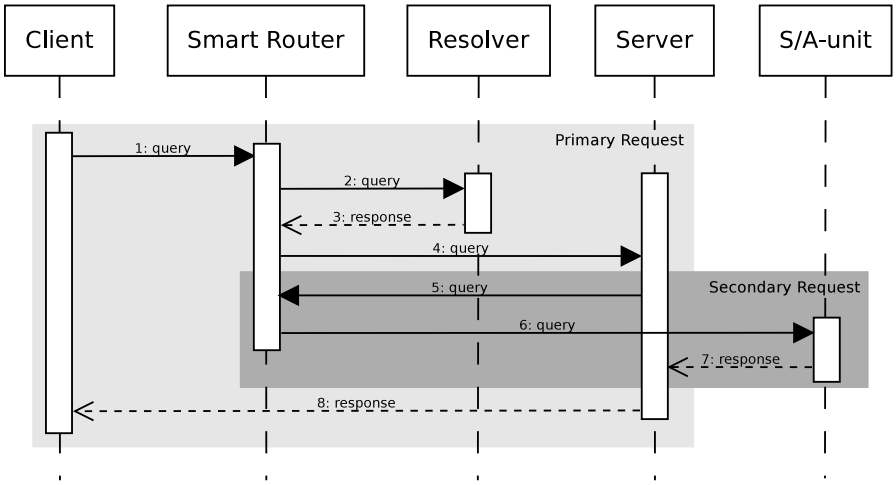


Figure 5.14: Messages sent during the cycle of a client request.

5.6 Component Description

5.6.1 System Structure

Figure 5.15 shows the interaction of the most important components included in the system.

5.6.2 Packet Types

These common classes are needed by all network participants and are implemented in C++ and in Java.

SmartPacket

This class is used to represent the common general header (see Section 5.3.1). It is the super class of all other packet types. It entails all needed fields and provides convenience methods to parse and create a binary representation of a packet.

SmartConnectPacket

This class is an inherited class from `SmartPacket`. It adds all specific fields needed for a connect packet (see Section 5.3.2) and extends the convenience methods.

SmartRequestPacket

The `SmartRequestPacket` is also inherited from `SmartPacket`. It adds the fields required for a query packet (see Section 5.3.3) and provides adapted convenience methods.

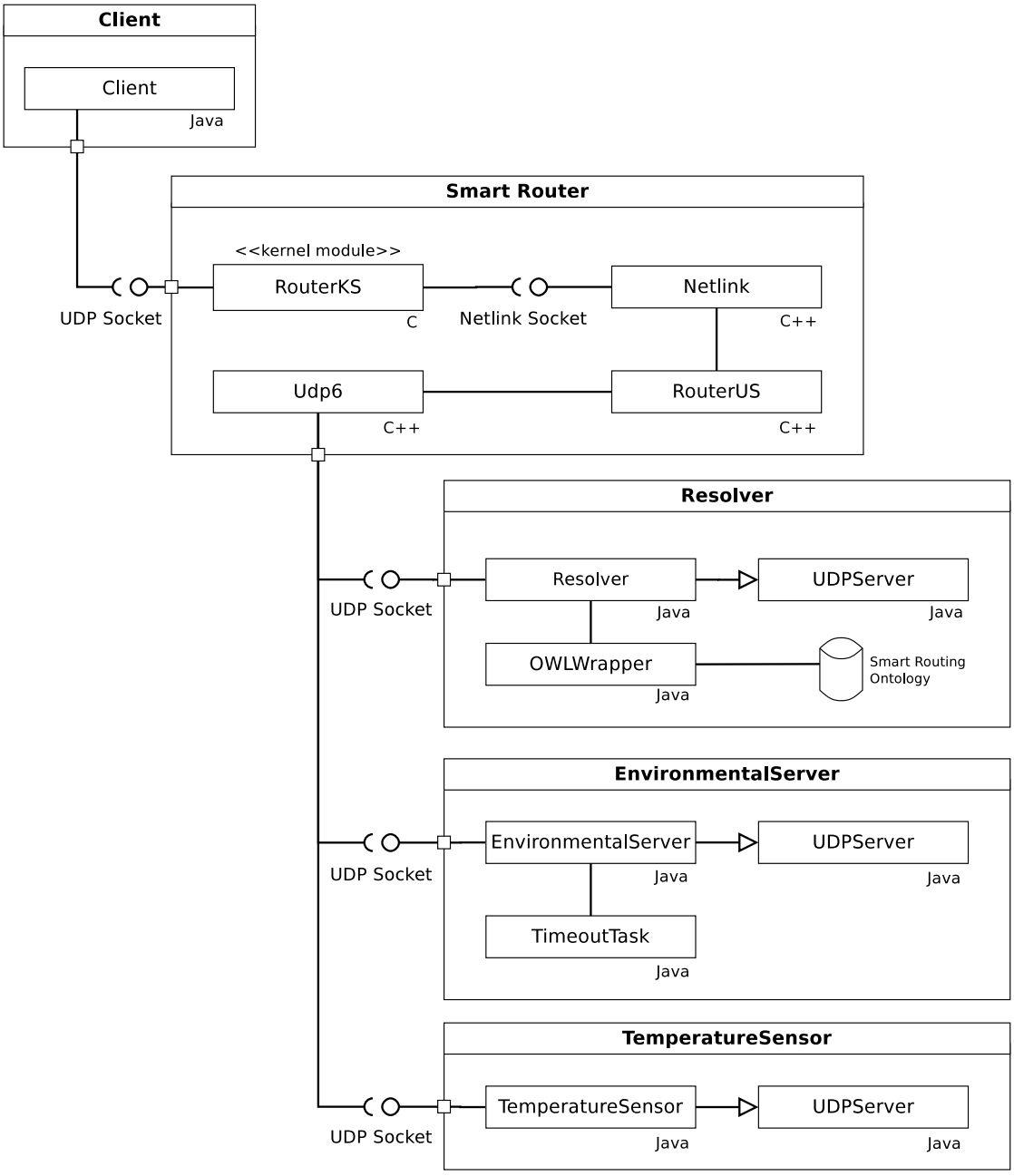


Figure 5.15: Component interaction.

SmartResponsePacket

Like the classes before, `SmartPacket` is the parent class of `SmartResponsePacket`. This class adds a field for the response data, as well as augmented convenience methods.

5.6.3 C/C++ (SmartRouter)

RouterKS

`RouterKS` is a kernel module used by a smart router, which filters unaddressed UDP packets from the network stream by implementing a netfilter hook and forwards them to the user space application by the use of netlink sockets.

RouterUS

The class `RouterUS` is the main class of the smart router's user space application. It is used to handle connect packets, primary and secondary requests. The class contains following important members:

`Netlink *netlink`
: Reference to a netlink object in order to receive fetched packets from the kernel module.

`Udp6 *udp`
: Reference to a Udp6 object, which is used for sending and receiving UDP packets over a IPv6 connection.

`list<SmartConnectPacket *> connected_nodes`
: List of all directly connected S/A-units (equals to the concept's `CNT`).

`list<uint16_t> pendingRequests`
: List of all processed secondary requests (equals to the concept's `ERT`)

Netlink

This class is used to establish a netlink socket between the user space application and the kernel module. Methods for receiving (`rcv()`) and sending (`snd()`) are provided.

Udp6

This class is used to handle a datagram socket for sending (`send()`) and receiving (`rcv()`) UDP packets by the use of an underlying IPv6 connection.

5.6.4 Java (Client, Resolver, EnvironmentalServer, TemperatureSensor)

UDPServer

This is an abstract super class used to create a server which accepts UDP packets. To this end, the class contains a `DatagramSocket`. The server continually listens on a datagram socket for new packets by the use of an abstract method `listenSocket()`. UDP Messages can be sent by the use of convenience methods, accepting either `SmartPacket` objects, strings, or byte arrays.

OWLWrapper

The `OWLWrapper` is used to access the OWL API. It has access to the ontology by the use of an instance of `OWLOntology`, as well as to the Pellet reasoner (`PelletReasoner`). The wrapper transforms requests of several types to queries for the Pellet reasoner.

Resolver

This class is the main class for the Resolver. It inherits the functionality of `UDPServer`. It waits for requests addressed to the resolver's anycast address and processes them by the use of an instance of `OWLWrapper`. For this purpose, the resolver examines the `request type` field of the received packet and calls the corresponding method of the `OWLWrapper` instance.

EnvironmentalServer

This is the environmental server's main class. It is also derived from `UDPServer` and waits for primary requests. This class contains following important member variables:

`protected ConcurrentHashMap<Integer, Request> pendingRequests`
: This hash map contains all pending primary request (equal to the concept's `PRT`).

`protected ConcurrentHashMap<Integer,`
`CopyOnWriteArrayList<SmartResponsePacket» pendingResponses`
: This array list entails all responses from S/A-units (equal to the concept's `SART`).

On the reception of a primary request, the `SmartRequestPacket` is wrapped in a `Request` and stored in the `pendingRequests` map. In addition, a timeout is specified by the use of a `Timer` object.

Request

This class acts as a container for a `SmartRequestPacket`. In addition to the packet, origin address and port are stored, since they are not part of a `SmartRequestPacket`.

TimeoutTask

The class `TimeoutTask` is needed in order to initiate the fusing function of the `EnvironmentalServer` on the arrival of the timeout's deadline. To this end, it has access to the `EnvironmentalServer`.

TemperatureSensor

The class `TemperatureSensor` implements an S/A-unit handling temperature requests. On the reception of a secondary request, a random temperature value is created and returned to the requesting node (i.e., the requesting server). `TemperatureSensor` is also derived from the `UDPServer`.

Client

This is a simple single class program which reads in a subject and an object property from the console and initiates a primary request. Afterwards, it waits for the reception of the server's response packet, containing a fused sensor value.

CHAPTER 6

Evaluation

This chapter evaluates the presented concept and its reference implementation with respect to performance, security and fault tolerance. Since not all features are implemented in the reference system, some distinctions are made between the concept and the implemented system when needed.

6.1 Alternative Approaches

6.1.1 Overview

In order to discuss the content aware network approach, it is necessary to know how alternative (classical or content aware) systems would be implemented. Depending on the knowledge of the client, several approaches are possible:

The integrated server approach: Like in the presented concept a dedicated server is responsible for fusing sensor values or propagating target states to actuator nodes. All requests are directly addressed to the integrated server. Therefore, this approach eliminates the need for smart routers and resolvers.

The domain server approach: A domain server has also the capabilities of dedicated servers. Therefore, a (domain) server can directly process primary requests and create the corresponding secondary requests, instead of first forwarding the request to a dedicated server.

The multicasting approach: For the multicasting approach no dedicated server is used. S/A-units of a dedicated type listen to the same IPv6 multicast address. A client has (hard coded) knowledge about various multicast groups. Therefore, by sending one multicast message, all S/A-units can be addressed at once. Sensor fusion is done at the client application.

6.1.2 The Integrated Server Approach

The integrated server approach merges the functionalities of resolver, domain server, smart router and all servers into one network component, called *integrated server*. This multi-functional device takes control over all tasks of a content aware network in both, the *information pull* and *information push* strategies. Each major task, the reception of a client request, the collection of sensor values, the distribution of actuator target states, as well as the handling of data packets is performed directly by the integrated server. Since all communication belonging to the content aware network is done with this device, IPv6 anycasting can be used instead of fetching matching packets out of the network traffic at (smart) routers.

An integrated server can be either responsible for all S/A-units or only for a specific type of S/A-units. The difference is that for the first case only one anycast address exists for all servers, and for the second case a multicast address exists for each server type. This means that a client must have knowledge about all possible server types and must address a request correctly to one of the predefined anycast addresses.

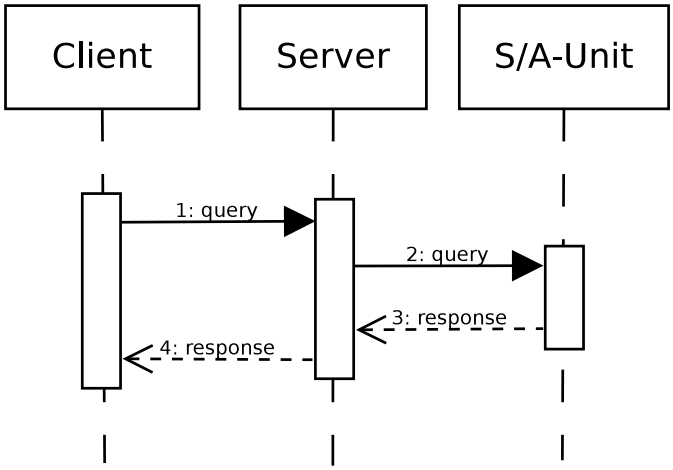


Figure 6.1: Message sequence of the information pull strategy in the scope of the integrated server approach.

Figure 6.1 shows the sequence of messages sent by the information pull strategy in the scope of the integrated server approach. After a primary request (1), the integrated server parses the request, infers semantic information by using a reasoner and then initiates a secondary request to all matching S/A-units (2). Since the server manages connected S/A-units on its own, S/A-units can be addressed directly by their own IPv6 addresses. When all S/A-units have returned their response to the integrated server (3), the server fuses these responses according to a predefined policy and returns the result to the client (4).

The information push strategy is simplified to exactly one message (see Figure 6.2), namely the data packet, sent from an S/A-unit.

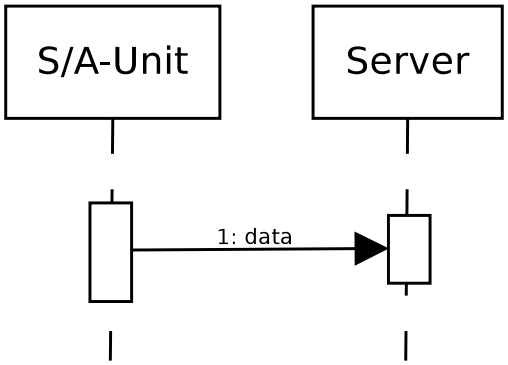


Figure 6.2: Message sequence of the information push strategy in the scope of the integrated server approach.

This approach gets along with just one content aware network component. This component comprises the entire functionality needed by a content aware network, which reduces network traffic. This implies, however, that the device must be extremely powerful in order to manage the computation intensive tasks of reasoning and sensor fusion. Just one type of component must handle all messages, even in large networks with thousands of messages. Another drawback of this approach is that a client has to know the exact (anycast) address of the integrated server. Especial in case of the variant with dedicated integrated servers, a client must send a request to the exact anycast address belonging to the intended type of integrated servers.

6.1.3 The Domain Server Approach

The domain server approach eliminates the need of dedicated servers. All detailed request handling and sensor fusion is done at the specific domain server. This server directly handles primary requests and data packets.

Figure 6.3 shows the sequence of messages sent in the scope of the domain server approach when a client initiates a primary request (in the scope of the information pull strategy). After the client requested information from the content aware network (1), the nearest smart router forwards this request to the next resolver (2). Like in the chosen concept, the resolver completes primary requests by using its ontology. The difference now is that the completed request is directly processed by the corresponding domain server (3), instead of forwarding it to a dedicated server. For this purpose, the whole primary request has to be forwarded, not just a transformed query (e.g. an SQL query). The (domain) server then can use a detailed database to handle the request. With this information, the server is able to send a secondary request in order to address

the corresponding S/A-units (4). From there the message sequence is the same as in the chosen concept.

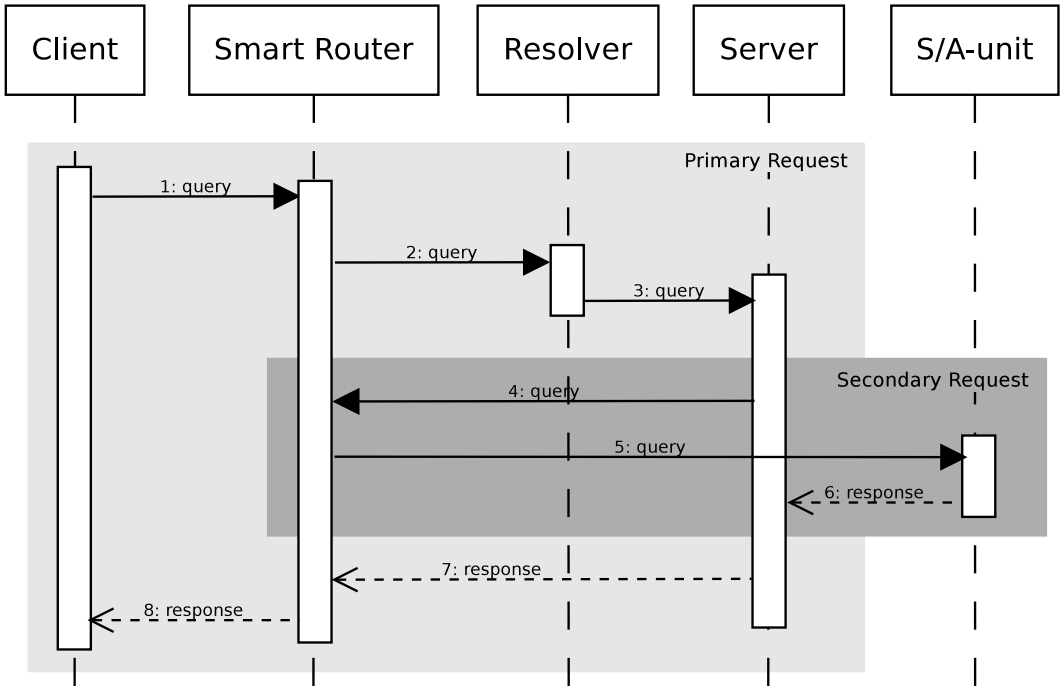


Figure 6.3: Message sequence of the information pull strategy in the scope of the domain server approach.

Like in the information pull strategy, data packets (1) in the scope of the information push strategy (see Figure 6.4) are directly forwarded to a resolver (2) which augments the data packets by the use of its ontology. Afterwards, the resolver forwards this augmented data packet to a domain server (3) which is directly able to handle the type of packet.

This approach includes the functionality of the dedicated servers directly into domain servers. For a typical network, there may be a few domains, but a lot of different services. Therefore, it is better to put the specific processing of packets into dedicated network servers, instead of implementing this functionality in an "all-in-one" domain server. In addition, it is possible that for a single service multiple domains are of interest. By the use of this approach, a workaround has to be made by implementing the same service many times at different kinds of domain servers.

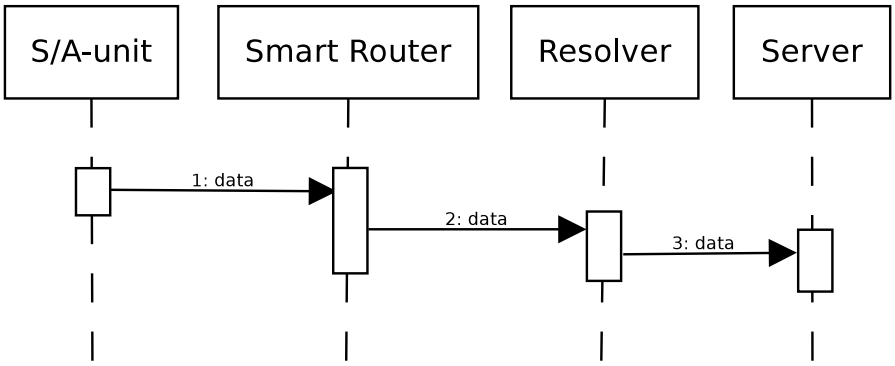


Figure 6.4: Message sequence of the information push strategy in the scope of the domain server approach.

6.1.4 The Multicasting Approach

The multicasting approach only uses clients and S/A-units. All functionalities needed to fuse sensor values are provided by the client itself. The ability to address multiple S/A-units at once is inherited from IPv6 multicasting by the definition of several multicast groups for each single type of S/A-unit. This approach is only usable for the information pull strategy, since no server is available for handling data packets in the scope of the information push strategy. When this feature is needed, a dedicated server must be added to the network.

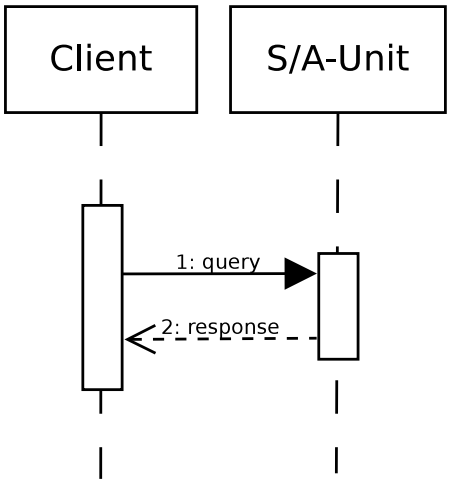


Figure 6.5: Message sequence of the information pull strategy in the scope of the multicasting approach.

Figure 6.5 shows the sequence of messages sent in the scope of the information pull strategy by the multicasting approach. A client directly sends a request to a set of S/A-units by addressing the packet to the correct multicast group (1). Every addressed S/A-unit returns its response directly to the client (2). Finally, the client itself interprets and fuses the received results on its own.

The big drawback of this approach is the lack of flexibility. It is hardly possible to limit the set of receivers according to predefined properties. The only techniques which can be used are defining several multicast groups and using the `hop limit` field of the IPv6 header. Nevertheless, this approach is useful to compare the presented concept with.

6.2 Performance

6.2.1 Number of Packets

Figure 6.6 shows a typical network topology when using the integrated server approach. Assume that the client wants to know the average temperature of all sensor nodes. The client must therefore send a dedicated request to the integrated server. This server has knowledge about all temperature sensors within a predefined network segment. Next, the server has to send four separate requests to these sensor nodes. Every node puts its current value into a response packet and returns it to the server. Finally, the integrated server fuses the values and returns the result to the client.

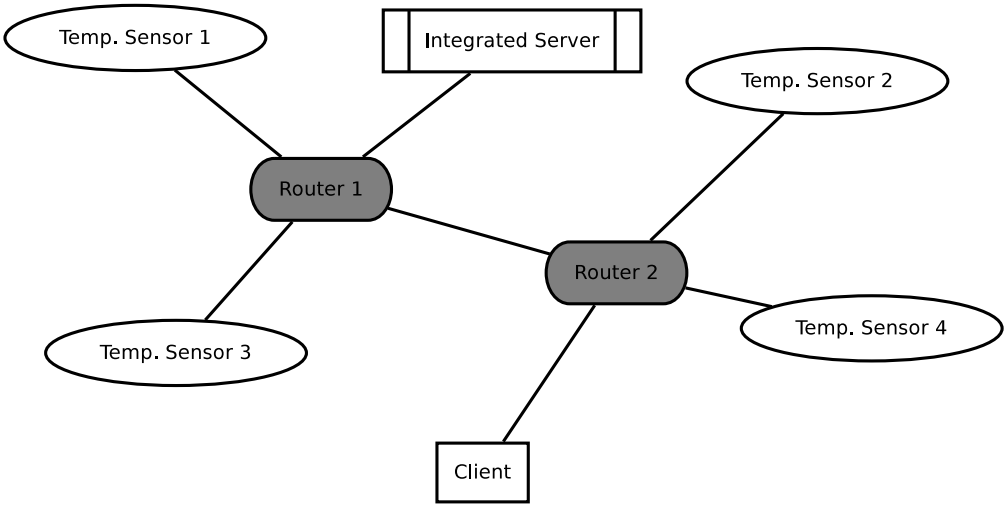


Figure 6.6: Network structure of the integrated server approach.

In total there are

$$m_{integrated} = 2 + 2 * n = 2 * (1 + n)$$

packets sent (where n is the amount of sensor nodes), when using this approach. According to this equation, $m_{integrated} = 10$ messages are sent in a network with four matching S/A-units when using the integrated server approach.

Figure 6.7 shows the setup of a typical network when the domain server approach is used. When requesting information from this network, a client has to send a primary request. This request is then forwarded to the nearest resolver by the next smart router. The resolver adds semantic information to the request and forwards it to a dedicated server, which initiates a secondary request. This secondary request is then split up at the network's smart routers in order to arrive at all corresponding S/A-units. After the S/A-units have reported their results to the server, the server fuses the results and sends a response packet back to the client.

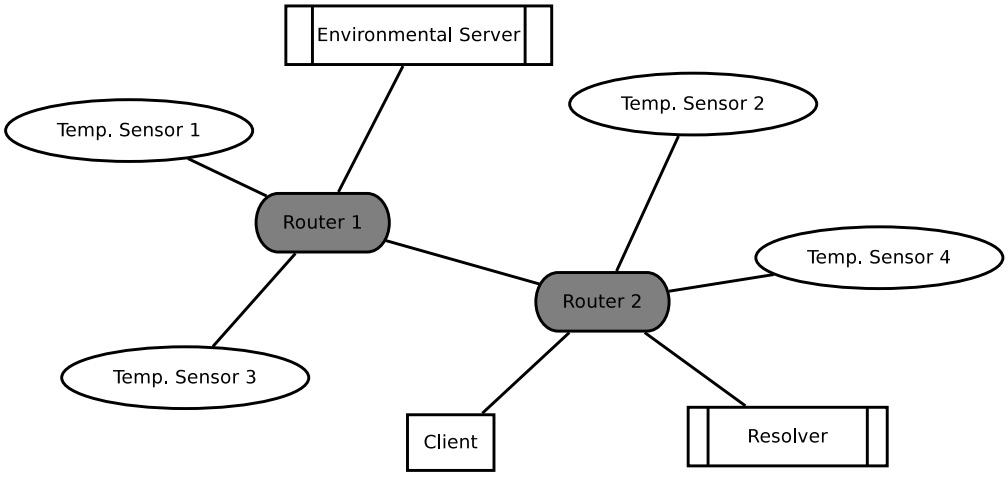


Figure 6.7: Network structure of the domain server approach.

At a first glance, there are

$$m_{domain} = 6 + 2 * n$$

messages sent in the scope of the information pull strategy when using this approach (where n is the amount of sensor nodes). Since the client's request is caught by the nearest smart router, in an optimal content aware network (a network, where no classic routers exist), the first two messages concern different network links. Therefore, the two messages can be combined in the formula above. In addition, the secondary request can be compared to IP multicasting, since the

request is only sent once and then split up at the smart routers. This means that a secondary request passes a single network segment just once. Finally, the server's response is in fact only one message, instead of two (7, 8). Therefore, the formula above can be simplified:

$$m_{domain,opt} = 5 + n$$

For the example of four matching S/A-units (see Figure 6.7), $m_{domain,opt} = 9$, which is slightly better than $m_{integrated} = 10$. For $n > 3$ this approach requires fewer messages than the integrated server approach, since

$$m_{integrated} - m_{domain,opt} = 2 + 2 * n - (5 + n) = n - 3$$

Figure 6.8 shows a typical network topology when using the multicast approach. By using the multicast approach, a client sends its request directly to all S/A-units by specifying the correct multicast address. Afterwards, every S/A-unit returns its current value straightforward to the client which finally fuses these values.

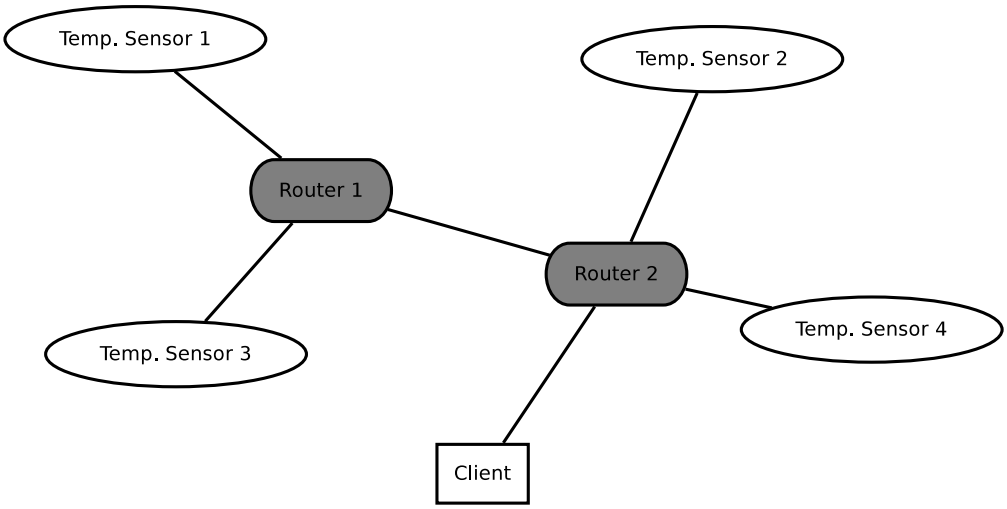


Figure 6.8: Network structure of the multicast approach.

Therefore, there are

$$m_{multicast} = 1 + n$$

messages sent in total when using this approach (where n is, like above, the amount of sensor nodes). It is obvious that (for the price of less flexibility) this approach needs half as

much messages compared to the integrated server approach. In addition, this approach needs constantly 4 messages less than the domain server approach. For the example of four matching S/A-units (see Figure 6.8), $m_{multicast} = 5$ messages are sent when using the multicast approach.

Now consider the example of requesting the average temperature when using the chosen concept (with the use of domain servers), respectively the concept's reference implementation (without the use of domain servers) provided by this thesis. According to Figure 4.9, the following message sequence is executed during the information pull strategy when considering the general concept:

First, the client sends an unaddressed temperature request. This request is caught by the next smart router which has to reroute it (send a new packet) to a resolver. This resolver augments the request and then forwards it to a dedicated domain server. The next packet sent is the response to the smart router. Finally, the smart router forwards the initial primary request to a dedicated sensor which then initiates a corresponding secondary request. At the next smart router this request is split up and sent to all matching S/A-units within the specified range. After the responses of the S/A-units are sent back to the server, the server aggregates the values and returns the fused value to the client.

At a first glance, following formula is valid for the amount of messages:

$$m_{concept} = 8 + 2 * n$$

But this formula is not exact, because some messages are in fact the same but sent over different network links. Consider the messages (1) and (5). The first one is the request sent from the client to the smart router. Since the smart router is the first router, to which the packet is passed, in an ideal content aware network (one, where no classical routers exist), this packet goes over exactly one network link. Since (5) is the same message as (1), but only forwarded by the smart router, these two messages concern a single network link at most one time. The same argument can be applied with (6) and (7). In addition, since the secondary request is in fact similar to multicasting, the combination of (6) and (7) affects a single network link at least one time. Finally, (9) and (10) are in fact also just one message. Therefore, this optimized formula is valid for the amount of messages when considering the presented concept:

$$m_{concept,opt} = 6 + n$$

Finally, consider the information pull strategy of the implemented reference system (see Figure 5.14). Since this system does not contain any domain servers, a resolver can directly respond to a request initiated from a smart router. This reduces the amount of messages again by 1. Therefore, the final formula which is valid for the reference system is:

$$m_{ref,opt} = 5 + n$$

It is obvious that in case of larger systems with more S/A-units, the reference system produces less network traffic than the integrated server approach (the break-even point is at $n = 3$), while the domain server approach needs the same amount of messages regardless of the amount of S/A-units. Compared to the multicasting approach, the reference system constantly needs 4 more UDP messages. But for the price of these four messages, the system becomes a lot more flexible and convenient to the user.

Now consider the case, where the client request involves multiple S/A-types (e.g. requesting information about the current wind conditions involves *wind speed* and *wind direction* sensors). The formulas for the various approaches are listed below, where s_i is the amount of units of S/A-type i and n is the total amount of S/A-units (i.e., $n = \sum s_i$). t specifies the amount of different S/A-types involved in the request.

$$\begin{aligned}
m_{mult,integrated} &= 2 * \left(1 + \sum s_i\right) &&= 2 * (1 + n) \\
m_{mult,domain,opt} &= 4 + t + \sum s_i &&= 4 + t + n \\
m_{mult,multicast} &= t + \sum s_i &&= t + n \\
m_{mult,concept,opt} &= 5 + t + \sum s_i &&= 5 + t + n \\
m_{mult,ref,opt} &= 4 + t + \sum s_i &&= 4 + t + n
\end{aligned}$$

It is obvious that the integrated server approach needs the same amount of messages regardless of the amount of involved S/A-types. The domain server approach, the multicasting approach, as well as the chosen concept's approach and the reference system need $t - 1$ more messages, where t is the amount of S/A-types.

For measured quantities which change slowly, an S/A-unit may be configured as a periodic multicasting device. By the use of the configuration's inherent information push strategy, a server can be informed about state changes. This state information can be stored for future client requests. Hence, in this case there is no need for additional secondary requests. A client's request is therefore reduced to the following constant amount of messages:

$$\begin{aligned}
m_{push,concept,opt} &= 5 \\
m_{push,ref,opt} &= 4
\end{aligned}$$

6.2.2 Caching

In order to increase network performance, network nodes may cache information gained from previously sent packets. In this section, caching at various network components is discussed.

For a smart router it is feasible to cache any topology information received from a resolver (respectively domain server). For this purpose, a smart router may contain a special table to store destination addresses to semantic tags. Such a table is of the format `<hash(t), a>`, where `t` is the semantic tag of a query packet and `a` is a set of IPv6 addresses of interested network nodes. `hash()` is a hash function, providing a unique identifier for each semantic tag.

At a first glance, it is also possible to store results of primary requests at smart routers. Since every sequence of messages belonging to a dedicated primary request must provide the same `message id`, a smart router can implement the following functionality: On the reception of a primary request, the hash of the request's semantic tag (`t`) is stored in a dedicated request table. When the response belonging to a previously stored request reaches the smart router, this response (`r`) is added to the corresponding entry of the request table. The request table may have the format `<hash(t), r>`. To this end, a smart router must also examine the semantic tag of response packets. Since only the nearest smart router receives a client's primary request, it only makes sense for smart routers on network segments containing clients to employ this caching technique.

On closer examination, this caching technique has a big drawback when processing state change requests (i.e., requests where any states of actuators have to be changed). Caching such requests can result in missed state changes: Consider a system with two smart routers. Assume that smart router 1 processes (and therefore caches) the request *set all lights in the Office to "on"*. Now, smart router 2 gets the request to set all lights "off". Since smart router 1 has cached the first request, a final request, intended to set the lights "on" again would actually result in doing nothing.

This problem can also be found in shared memory multiprocessor systems under the term *cache coherency* [2]. Various protocols are available to maintain coherency, like *Synapse*, *Berkeley*, or *Illinois*. They are all based on updating remote caches at the modification of a processor's own cache. For the case of smart routing, this would mean an overhead in terms of network packets. But in most cases additional message sequences burn up the increase of performance gained from caching primary requests. Therefore, smart routers should only cache requests where no actuators must be set. To this end, a smart router would have to have detailed knowledge about the content of a (primary) request. But this knowledge is predestined to resolvers and servers. Giving a smart router the possibility to fully examine the contents of query packets would result in too much computational overhead. Hence, this caching technique is not directly applicable.

Another possibility to cache information is at resolvers. Using a semantic reasoner can be resource intensive for large ontologies. Instead of always inquiring an ontology for the same requests, processed requests can be cached. When receiving the same request again, a resolver can therefore respond faster. This type of cache is valid as long as the used ontologies are not altered. A resolver must therefore keep track on the modification of the ontology and clear its cache right after any change.

Caching at passive domain servers makes no sense, since contemporary database systems already provide high performance caches. On the other hand, for active domain servers the same caching techniques which are possible for resolvers can be applied.

Finally, a server can cache results of previous secondary requests. When receiving a primary request, the server must examine which type of secondary requests should be applied. In the case of just requesting current values from sensor nodes, the server can cache processed requests. Since the server has already knowledge about the internals of a query packet (compared to a smart router), the drawbacks mentioned above do not come into effect.

In addition, a server can cache data packets sent from S/A-units in the scope of the information push strategy. By the knowledge of the current state of these S/A-units, there is no need to send additional secondary requests to process a primary request.

6.2.3 Timeouts

The reference implementation's environmental server waits for a predefined time for responses of S/A-units before sensor values are averaged and sent back to the client. It is hard to specify a minimum timeout, because IP network communication can not be exactly timed. Therefore, the timeout has to be set to the worst case, which results in always waiting for the worst case deadline. There are two possibilities to get better average case results:

1. According to the concept, a smart router forwards the configuration of connected S/A-units to all interested servers. Therefore, a dedicated server would know how many S/A-units exist of a specific type. When requesting the state of a set of S/A-units, the server knows exactly, when all S/A-units have reported their current values. A timeout is only needed as a backup, when an S/A-unit failed.

2. When forwarding a secondary request, a smart router can report the amount of recipients to the requesting server. Since there might be more than one smart router which forwards the request to different S/A-units, a server must be able to handle and aggregate such reports of several smart routers.

6.3 Security

6.3.1 Overview

Like any distributed system, a content aware network is vulnerable to attacks and intrusion attempts. To secure network communication against such threats, possible attack scenarios are discussed. Finally, a solution is presented which handles the discussed vulnerabilities.

The basic concepts of security are *Confidentiality*, *Integrity*, and *Availability*. These three principles are called the *CIA-Triad* [1] (see Figure 6.9).

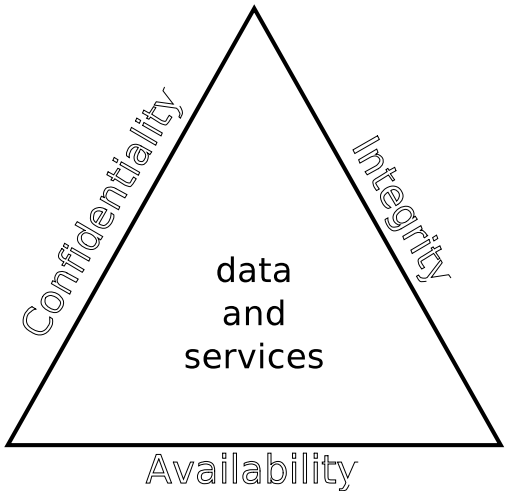


Figure 6.9: The CIA-Triad.

Confidentiality is about the protection of private data from viewing by non-authorized parties. Confidentiality is necessary to maintain privacy. Absolute secrecy of personal passwords is a requirement for confidentiality.

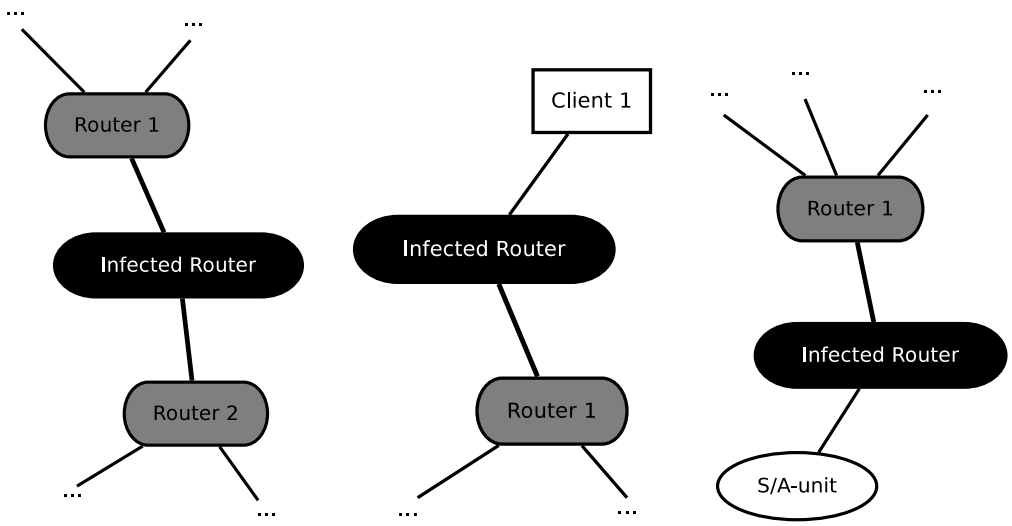
Integrity is about the protection of private data against unauthorized modification or deletion. To this end, not just prevention against undesirable modification of data must be provided, but also mechanisms to reverse changes.

Availability is the third part of the CIA-Triad. Availability is about the possibility to access personal data at any time when it is needed.

6.3.2 Vulnerabilities

Man-in-the-Middle

A Man-in-the-Middle attack can be performed by a manipulated router. The router seems to act like a normal smart router, but in reality it eavesdrops or manipulates network communication. Such a router can harm a content aware network concerning any principle of the CIA-Triad. According to the segment to which such a component is attached, several vulnerabilities exist:



(a) The manipulated router is located inside the content aware network.

(b) The manipulated router is located between the client and the content aware network.

(c) The manipulated router is located between an S/A-unit and the content aware network.

Figure 6.10: Man-in-the-Middle attack variants

(a) If the manipulated router is located inside the content aware network (see Figure 6.10a), the device affects a whole set of network participants. It is possible to eavesdrop, manipulate or drop any type of network packets, even data packets (e.g. consider a manipulated or even dropped data packet which is sent due to a fire alarm)

(b) If the manipulated router is attached directly between a client and the content aware network (see Figure 6.10b), the device may eavesdrop, manipulate or drop primary requests or responses to primary requests.

(c) Located on a segment between an S/A-unit and the rest of the content aware network (see Figure 6.10c), a manipulated router may manipulate current values or configurations. Like

in (a), it is also possible to compromise data packets in the scope of the information pull strategy.

Malicious Resolver (or Domain Server)

Malicious resolvers or domain servers can affect the availability and the integrity of a content aware network. Any request to such a device can be manipulated in a way, such that no or wrong results are returned. In the case of returning no data, the content aware network is simply not able to process unaddressed packets. Therefore, the network's availability is harmed. In the case of topology requests, it is even possible to return IP addresses of other malicious components, which affects the network's integrity.

Malicious Server

A malicious server can affect the system's integrity by reporting wrong results to primary requests. In addition, a malicious server may send unauthorized secondary requests, which can be used for setting wrong actuator states or eavesdropping sensor values.

Malicious S/A-Unit

Malicious S/A-units mainly affect the system's integrity. Considering the information pull strategy, a malicious sensor node may return fake current values to secondary requests, resulting in incorrect fused values. A malicious actuator node may set its state arbitrarily, which can be fatal for some systems (e.g. consider a malicious actuator node which regulates a valve in a nuclear power plant).

When the information push strategy is considered, a malicious S/A-unit may send wrong data packets, which can result in unintended behavior of servers and other network nodes (e.g. a wrong fire alarm).

Furthermore, a malicious S/A-unit can also harm the system's confidentiality, since such nodes can also be used to eavesdrop the internal state of sensors and actuators.

6.3.3 Preserving Security

Requirements

In order to secure a content aware network, it is required to encrypt network communication. If all communication is encrypted, it is harder to perform attacks and intrusion attempts. In addition, only authorized participants of the network shall be allowed to send packets. Messages, sent from unauthorized network components must be dropped immediately at smart routers.

IPsec

It is possible to use IPsec (see Section 2.1.2) for securing packet transmissions and component authentication to build a secure content aware network. To this end, IPsec's transport mode in combination with ESP can be utilized. In order to only encrypt traffic belonging to the content aware network, correct selectors (see [36] Section 4.4.1.1.) have to be used. For primary requests, it is enough to select UDP packets with the dedicated destination port 4444. Since communication with resolvers (and domain servers) is done by IPv6 anycasting, selectors for resolver requests and responses can be bound to the corresponding anycast address. All other selectors for the other network nodes have to be applied to the specific combination of source and destination IPv6 addresses and UDP ports.

Since IPsec supports not only encryption, but also authenticity (with the help of protocols like IKEv2), not only modification threats can be handled, but also a component's identity can be verified. This is done by signing messages with the sender's private key. Through distributing a node's public key to other participants of the network, a receiver can check if a specific packet was sent by a trusted sender.

What remains is the elimination of availability threats. Since for the availability of a system it makes no difference, whether a malicious network nodes cuts a network connection or a benign node fails, the experiences of Section 6.4 can also be used to handle availability threats.

6.4 Safety and Fault Tolerance

6.4.1 Definition

Fault tolerance means that a system should maintain its functionality despite of a maximum set of potential malfunctions. In contrast to *security* which is about the protection of a system from the outer world, *safety* deals with the handling of unintended behavior of system components (e.g. protection from software bugs or hardware errors). To this end, possible faults of a content aware network are discussed.

6.4.2 Broken Network Links

In general, IP networks provide multiple routes between pairs of network participants. A routing table contains entries for every possible path. Due to routing protocols, these entries are continuously updated. Therefore, in the case of broken network links, automatically alternative paths are chosen by a router. Since a content aware network is based on standard IPv6 communication, this powerful feature is already built in.

6.4.3 Broken (Smart) Routers

In case of a dead router, all network nodes directly connected to this router are cut off from the network. All other network nodes can be accessed by alternative routes, if there exists one. A solution to this problem can be found in adding multiple servers, resolvers and S/A-units (which handle the same tasks) to different segments of the network. In case of a separation of two parts of the network due to the failure of a smart router, there are multiple network components available in both separated network segments in order to maintain the functionality.

6.4.4 Broken Resolver (or Domain Server)

Communication with resolvers (and domain servers) is done by anycasting, which is a feature provided by IPv6. Therefore, all resolvers are listening to the same IPv6 address. By the principle of anycasting, a packet sent to such an address arrives only at one destination. Since this feature automatically detects online network participants belonging to a dedicated anycast group, it is simply possible to add x resolvers to the network in order tolerate $x - 1$ dead resolvers. For domain servers of a specific type the same argumentation holds.

6.4.5 Broken Server

The content aware network natively supports multiple servers responsible for the same primary requests. According to the semantic tag of a packet, a smart router has to forward the packet to one out of all available servers. Since all network nodes have to send connect packets in predefined intervals, a smart router detects an offline server by checking a timestamp field in its `CNT`. Therefore, by just providing multiple servers for the same tasks, primary requests can simply be handled by an alternative server.

Data packets sent during the information push strategy, have to be forwarded to all interested servers. For particular cases, it would be beneficial to use redundant servers handling the same data packets. These servers have to communicate with each other in order to make sure that the tasks to be performed only get executed once. In the case of two redundant servers this can be done by selecting one of them as master and the other one as slave server. On the reception of a data packet, the master server has to inform the slave server that it is alive and able to perform the intended tasks. If this message does not arrive at the slave server until a predefined deadline, the slave server assumes that the master is dead and executes the tasks on its own.

6.4.6 Broken S/A-Unit

Dead sensor nodes are not problematic, when multiple sensor nodes of a specific type are available, since sensor values are fused at servers. Therefore, the worst case scenario for broken sensor nodes is that a server does not receive data from all sensors, resulting in fusing the values from less sensors.

Dead actuator nodes can be handled also by redundant nodes. In this case it is important for secondary requests to specify absolute target values (e.g. set the target temperature to $+25\ ^{\circ}C$), instead of relative values (e.g. increase the target temperature by $+5\ ^{\circ}C$).

In general, faulty S/A-units can be detected by TMR (Triple Modular Redundancy) (see Figure 6.11). Instead of just providing one S/A-unit for a specific purpose, a set of three redundant units is provided. By voting it is possible to detect one faulty S/A-unit when the other two are working correctly.

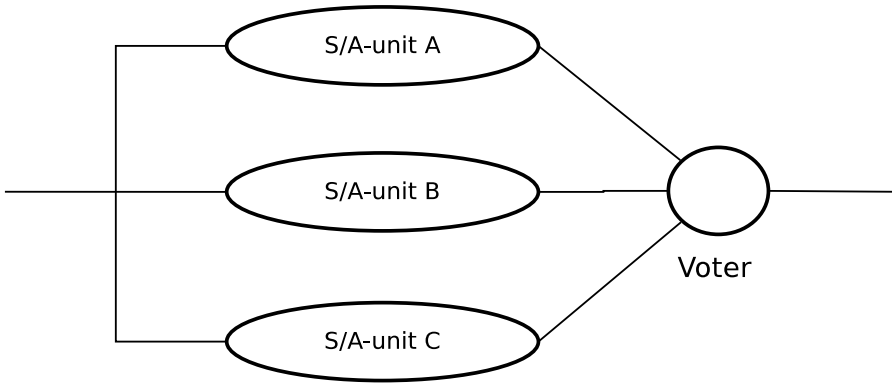


Figure 6.11: TMR in S/A-units.

Since a dedicated server can implement sophisticated fusing functions, which detect wrong sensor values (see Section 4.4), it is not necessary to implement TMR for sensor nodes. For instance, the adaptive algorithm automatically removes sensor values where the deviation from the mean is larger than a specified interval. A server can keep track of such faulty sensor values and, in case of repetition, a system administrator can be informed about the broken sensor node.

6.5 On the Way to a Complete Ontology

The provided ontology (see Appendix A) is optimized for the use of the reference system. A real-life ontology must have some modifications. In general, there is no need for instances of network nodes within in the ontology, since these are provided by dedicated domain servers. On the other hand, additional properties must be provided for specific cases (e.g., there must be the possibility to restrict a request to a minimal accuracy, or to specify all cities within a certain range).

In general, the resolver's smart-routing ontology is used to augment a request's object properties by some inferred properties fetched from the request's subject. In contrast to the implementation, this request does not need to be completely resolved to a destination address. The destination

address is found by converting the augmented request to a dedicated format which is readable by a domain server (e.g. SQL for passive domain servers).

When needed, this smart-routing ontology can be augmented by more sophisticated ontologies which can be interpreted by active domain servers or even dedicated servers. Such additional ontologies can be observation-centric (like *SENSEI* [9]), sensor-centric (like the *MMI Device Ontology* [22]), or a combination of the both. By the use of such an ontology it is possible to combine physical sensor values in order to obtain virtual sensor values or measurements over time. A virtual sensor is a sensor with no direct observation possibilities. The sensor value is virtually (by software) gained by the fusion of several other sensor types. An example of virtual sensing is the determination of a car's position within a tunnel by the use of the last received GPS coordinates and the car's speed and direction of movement. Measurement over time can be used for example to get average values within a specific time interval (e.g. average temperature of Vienna in July).

The process of resolving a request to a certain set of network addresses can be divided into several parts. First, the resolver must parse the request. Afterwards, the ontology must be inquired by the use of a semantic reasoner. This reasoner must initially find the corresponding instance to the request's subject within the ontology. Next, the subject's object properties are used to augment the request. Finally, the resolver converts the request to a dedicated query for a domain server.

Now, this server either queries a database (passive domain server), or inquires an extended ontology (active domain server). In the first case, a simple SQL query provided by the reasoner delivers the correct set of addresses. In the second case, the extended ontology must also contain instances of objects (e.g. servers) in order to obtain the needed addresses.

6.6 Interpreting Natural Language

In the actual concept client requests are based on a subject in combination with some object properties and an optional target state to set actuators (e.g. for the subject *Vienna* and the object property *temperature*, a client wants to receive the current outside temperature of the city Vienna). For more flexibility, a preliminary stage within a resolver could parse simple English phrases, like *get temperature in Vienna*, where the subject *Vienna* and the object property *temperature* are extracted. For this purpose, the set of accepted phrases should be kept simple in order to preserve a minimum of required performance. For actuators, such a phrase could be *set emergency lights in Building A to "on"*. The keyword *set* indicates that this request is designated to set some actuators. Therefore, the subject *Building A*, the object properties *light* and *emergency* and the target state *on* are extracted from the phrase.

CHAPTER 7

Conclusion

This chapter provides a short summary and an outlook of future work on the topic of content aware networking.

7.1 Summary

In the scope of this thesis, a concept for content aware networking (with a focus on S/A-systems) in IPv6 networks is presented. In this context, content aware means that network components have the capability to parse and process packets according to their semantic meaning. The advantage of this approach is the presented routing mechanism which is not based on destination addresses but on the semantic tag of a network packet which is interpretable by a piece of software. In order to make decisions based on such a semantic tag, techniques provided by the Semantic Web are used. Semantic reasoning helps augmenting information by known, but not directly encoded facts. To this end, ontologies, stored in OWL files, are used.

The concept distinguishes between the *information pull* and the *information push strategy*. The information pull strategy is used to handle client requests. Such a request can be for example the query of the current temperature of a specific location. To this end, the content aware network resolves this (unaddressed) primary request and forwards it to a dedicated environmental server. The environmental server then generates a secondary request in order to address all temperature sensors belonging to the requested location. When all sensors have reported their current values, the server averages these values and reports the result to the client.

The information push strategy is used for event messages, sent from S/A-units. If an S/A-unit has to report a change of its state, it sends an unaddressed data packet to the network. By interpreting the semantic tag of this packet, the network forwards this packet to a set of interested servers which may then perform further actions (e.g. setting actuators, or forwarding the

information to other systems).

In order to provide the feature of smart, content based routing, several network components are presented which are additional to standard networking devices. A smart router has - apart from normal routing of standard IP packets - the capability of detecting and fetching packets belonging to the content aware network. A resolver is another network component which has access to an ontology containing meta-information of the network. With the help of a semantic reasoner, a resolver can augment information contained in semantic tags. After extending the semantic tag of a network packet and transforming it to a dedicated query, it is then forwarded to a specific domain server. This domain server has full knowledge about a the packet's domain. With this knowledge it is able to return IPv6 addresses of dedicated servers which can handle the packet. Such servers can process client requests of the information pull strategy or data packets of the information push strategy. For this purpose, these servers can access a set of S/A-units (i.e., network components which contain sensors and/or actuators).

The thesis also provides a reference implementation of the presented concept. This reference implementation is a proof of concept with elementary functionality. For this purpose, some features of the concept are simplified (e.g. the functionality of domain servers is included in the resolvers).

The experiences found in the implementation of this reference system can be used to implement a complete content aware network which provides all features of the presented concept. To this end, the thesis provides a full evaluation of both, the concept and the reference implementation. This evaluation discusses the approach with respect to performance, security, safety and fault tolerance. In addition, advantages and disadvantages of some alternative approaches are presented.

7.2 Future Work

The actual reference system provides elementary functionality needed by the information pull strategy. The next step would be implementing a full system containing all presented network components, where the semantic tags are completely encoded in XML. This full system should also implement the information push strategy.

In order to secure network traffic, IPsec can be utilized. As the evaluation (see Section 6.3) shows, IPsec's transport mode in combination with ESP would fit the needs of a secure content aware network.

Caching at smart routers, resolvers and servers can increase throughput. To this end, conventional caching techniques may have to be adapted. Since information in S/A-systems is often short-lived, it would need to be investigated, whether caching of specific information is feasible.

Adding a human interface which is able to parse English phrases would help to increase the

usability of a content aware network. To preserve a minimum level of performance, accepted phrases must be kept small and simple.

The reference implementation is based on sending UDP packets. One further step is to shift semantic interpretation to lower levels of the OSI-model. Enabling semantic interpretation on lower levels enables a whole set of new possibilities (e.g. instead of defining a dedicated UDP port for semantic packets, a new protocol family can be defined). Also, replacing IP with wireless technologies, like 6LoWPAN or ZigBee, is worth of further investigation.

APPENDIX A

OWL Ontology Used for the Reference System

```
<?xml version="1.0"?>


<!DOCTYPE Ontology [
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>


<Ontology xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="http://www.iot6.eu/2013/routing#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     ontologyIRI="http://www.iot6.eu/2013/routing#">
    <Prefix name="" IRI="http://www.iot6.eu/2013/routing#"/>
    <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
    <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
    <Prefix name="xml" IRI="http://www.w3.org/XML/1998/namespace"/>
    <Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#"/>
    <Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#"/>
    <Annotation>
        <AnnotationProperty abbreviatedIRI="rdfs:label"/>
        <Literal datatypeIRI="&rdf;PlainLiteral">Smart Routing Ontology</Literal>
    </Annotation>
    <Annotation>
        <AnnotationProperty abbreviatedIRI="rdfs:comment"/>
```

```
        <Literal datatypeIRI="&rdf;PlainLiteral">OWL ontology for smart routing</
            Literal>
32  </Annotation>
    <Declaration>
34      <Class IRI="Actuator"/>
    </Declaration>
36  <Declaration>
        <Class IRI="Building"/>
38  </Declaration>
    <Declaration>
40      <Class IRI="City"/>
    </Declaration>
42  <Declaration>
        <Class IRI="HumiditySensor"/>
44  </Declaration>
    <Declaration>
46      <Class IRI="LightSwitch"/>
    </Declaration>
48  <Declaration>
        <Class IRI="Location"/>
50  </Declaration>
    <Declaration>
52      <Class IRI="Networknode"/>
    </Declaration>
54  <Declaration>
        <Class IRI="Room"/>
56  </Declaration>
    <Declaration>
58      <Class IRI="Router"/>
    </Declaration>
60  <Declaration>
        <Class IRI="Sensor"/>
62  </Declaration>
    <Declaration>
64      <Class IRI="Server"/>
    </Declaration>
66  <Declaration>
        <Class IRI="TemperatureSensor"/>
68  </Declaration>
    <Declaration>
70      <ObjectProperty IRI="connectedTo"/>
    </Declaration>
72  <Declaration>
        <ObjectProperty IRI="fusedValue"/>
74  </Declaration>
    <Declaration>
76      <ObjectProperty IRI="humidity"/>
    </Declaration>
78  <Declaration>
        <ObjectProperty IRI="humiditySensor"/>
80  </Declaration>
    <Declaration>
82      <ObjectProperty IRI="locatedIn"/>
    </Declaration>
84  <Declaration>
        <ObjectProperty IRI="locatedInInv"/>
```

```
86      </Declaration>
        <Declaration>
88          <ObjectProperty IRI="location"/>
        </Declaration>
90      <Declaration>
            <ObjectProperty IRI="network"/>
92      </Declaration>
        <Declaration>
94          <ObjectProperty IRI="physicalValue"/>
        </Declaration>
96      <Declaration>
            <ObjectProperty IRI="temperature"/>
98      </Declaration>
        <Declaration>
100         <ObjectProperty IRI="temperatureSensor"/>
        </Declaration>
102     <Declaration>
            <DataProperty IRI="ipv6address"/>
104     </Declaration>
        <Declaration>
106         <DataProperty IRI="isOutside"/>
        </Declaration>
108     <Declaration>
            <DataProperty IRI="location"/>
110     </Declaration>
        <Declaration>
112         <DataProperty IRI="network"/>
        </Declaration>
114     <Declaration>
            <NamedIndividual IRI="BuildingA"/>
116     </Declaration>
        <Declaration>
118         <NamedIndividual IRI="BuildingB"/>
        </Declaration>
120     <Declaration>
            <NamedIndividual IRI="EnvironmentalServerLinz"/>
122     </Declaration>
        <Declaration>
124         <NamedIndividual IRI="EnvironmentalServerVienna"/>
        </Declaration>
126     <Declaration>
            <NamedIndividual IRI="Linz"/>
128     </Declaration>
        <Declaration>
130         <NamedIndividual IRI="Office"/>
        </Declaration>
132     <Declaration>
            <NamedIndividual IRI="Router1"/>
134     </Declaration>
        <Declaration>
136         <NamedIndividual IRI="Router2"/>
        </Declaration>
138     <Declaration>
            <NamedIndividual IRI="Router3"/>
140     </Declaration>
        <Declaration>
```

```
142         <NamedIndividual IRI="Router4"/>
    </Declaration>
144     <Declaration>
        <NamedIndividual IRI="Shop"/>
146     </Declaration>
    <Declaration>
148         <NamedIndividual IRI="StorageRoom"/>
    </Declaration>
150     <Declaration>
        <NamedIndividual IRI="Vienna"/>
152     </Declaration>
    <SubClassOf>
154         <Class IRI="Actuator"/>
        <Class IRI="Networknode"/>
156     </SubClassOf>
    <SubClassOf>
158         <Class IRI="Building"/>
        <Class IRI="Location"/>
160     </SubClassOf>
    <SubClassOf>
162         <Class IRI="City"/>
        <Class IRI="Location"/>
164     </SubClassOf>
    <SubClassOf>
166         <Class IRI="HumiditySensor"/>
        <Class IRI="Sensor"/>
168     </SubClassOf>
    <SubClassOf>
170         <Class IRI="LightSwitch"/>
        <Class IRI="Actuator"/>
172     </SubClassOf>
    <SubClassOf>
174         <Class IRI="Room"/>
        <Class IRI="Location"/>
176     </SubClassOf>
    <SubClassOf>
178         <Class IRI="Router"/>
        <Class IRI="Networknode"/>
180     </SubClassOf>
    <SubClassOf>
182         <Class IRI="Sensor"/>
        <Class IRI="Networknode"/>
184     </SubClassOf>
    <SubClassOf>
186         <Class IRI="Server"/>
        <Class IRI="Networknode"/>
188     </SubClassOf>
    <SubClassOf>
190         <Class IRI="TemperatureSensor"/>
        <Class IRI="Sensor"/>
192     </SubClassOf>
    <ClassAssertion>
194         <Class IRI="Building"/>
        <NamedIndividual IRI="BuildingA"/>
196     </ClassAssertion>
    <ClassAssertion>
```

```
198         <Class IRI="Building"/>
            <NamedIndividual IRI="BuildingB"/>
200     </ClassAssertion>
        <ClassAssertion>
202         <Class IRI="Server"/>
            <NamedIndividual IRI="EnvironmentalServerLinz"/>
204     </ClassAssertion>
        <ClassAssertion>
206         <Class IRI="Server"/>
            <NamedIndividual IRI="EnvironmentalServerVienna"/>
208     </ClassAssertion>
        <ClassAssertion>
210         <Class IRI="City"/>
            <NamedIndividual IRI="Linz"/>
212     </ClassAssertion>
        <ClassAssertion>
214         <Class IRI="Room"/>
            <NamedIndividual IRI="Office"/>
216     </ClassAssertion>
        <ClassAssertion>
218         <Class IRI="Router"/>
            <NamedIndividual IRI="Router1"/>
220     </ClassAssertion>
        <ClassAssertion>
222         <Class IRI="Router"/>
            <NamedIndividual IRI="Router2"/>
224     </ClassAssertion>
        <ClassAssertion>
226         <Class IRI="Router"/>
            <NamedIndividual IRI="Router3"/>
228     </ClassAssertion>
        <ClassAssertion>
230         <Class IRI="Router"/>
            <NamedIndividual IRI="Router4"/>
232     </ClassAssertion>
        <ClassAssertion>
234         <Class IRI="Room"/>
            <NamedIndividual IRI="Shop"/>
236     </ClassAssertion>
        <ClassAssertion>
238         <Class IRI="Room"/>
            <NamedIndividual IRI="StorageRoom"/>
240     </ClassAssertion>
        <ClassAssertion>
242         <Class IRI="City"/>
            <NamedIndividual IRI="Vienna"/>
244     </ClassAssertion>
        <ObjectPropertyAssertion>
246         <ObjectProperty IRI="locatedIn"/>
            <NamedIndividual IRI="BuildingA"/>
248         <NamedIndividual IRI="Linz"/>
        </ObjectPropertyAssertion>
250     <ObjectPropertyAssertion>
            <ObjectProperty IRI="locatedIn"/>
252         <NamedIndividual IRI="BuildingB"/>
            <NamedIndividual IRI="Vienna"/>
```

```
254     </ObjectPropertyAssertion>
        <ObjectPropertyAssertion>
256         <ObjectProperty IRI="locatedIn"/>
            <NamedIndividual IRI="EnvironmentalServerLinz"/>
258         <NamedIndividual IRI="BuildingA"/>
        </ObjectPropertyAssertion>
260     <ObjectPropertyAssertion>
            <ObjectProperty IRI="locatedIn"/>
262         <NamedIndividual IRI="EnvironmentalServerVienna"/>
            <NamedIndividual IRI="BuildingB"/>
264     </ObjectPropertyAssertion>
        <ObjectPropertyAssertion>
266         <ObjectProperty IRI="temperature"/>
            <NamedIndividual IRI="Linz"/>
268         <NamedIndividual IRI="EnvironmentalServerLinz"/>
        </ObjectPropertyAssertion>
270     <ObjectPropertyAssertion>
            <ObjectProperty IRI="locatedIn"/>
272         <NamedIndividual IRI="Office"/>
            <NamedIndividual IRI="BuildingA"/>
274     </ObjectPropertyAssertion>
        <ObjectPropertyAssertion>
276         <ObjectProperty IRI="connectedTo"/>
            <NamedIndividual IRI="Router1"/>
278         <NamedIndividual IRI="Router4"/>
        </ObjectPropertyAssertion>
280     <ObjectPropertyAssertion>
            <ObjectProperty IRI="connectedTo"/>
282         <NamedIndividual IRI="Router1"/>
            <NamedIndividual IRI="Router3"/>
284     </ObjectPropertyAssertion>
        <ObjectPropertyAssertion>
286         <ObjectProperty IRI="connectedTo"/>
            <NamedIndividual IRI="Router1"/>
288         <NamedIndividual IRI="Router2"/>
        </ObjectPropertyAssertion>
290     <ObjectPropertyAssertion>
            <ObjectProperty IRI="locatedIn"/>
292         <NamedIndividual IRI="Router1"/>
            <NamedIndividual IRI="Office"/>
294     </ObjectPropertyAssertion>
        <ObjectPropertyAssertion>
296         <ObjectProperty IRI="connectedTo"/>
            <NamedIndividual IRI="Router2"/>
298         <NamedIndividual IRI="Router4"/>
        </ObjectPropertyAssertion>
300     <ObjectPropertyAssertion>
            <ObjectProperty IRI="connectedTo"/>
302         <NamedIndividual IRI="Router2"/>
            <NamedIndividual IRI="Router1"/>
304     </ObjectPropertyAssertion>
        <ObjectPropertyAssertion>
306         <ObjectProperty IRI="locatedIn"/>
            <NamedIndividual IRI="Router2"/>
308         <NamedIndividual IRI="BuildingA"/>
        </ObjectPropertyAssertion>
```

```
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="connectedTo"/>
        <NamedIndividual IRI="Router3"/>
        <NamedIndividual IRI="Router4"/>
    </ObjectPropertyAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="connectedTo"/>
        <NamedIndividual IRI="Router3"/>
        <NamedIndividual IRI="Router1"/>
    </ObjectPropertyAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="locatedIn"/>
        <NamedIndividual IRI="Router3"/>
        <NamedIndividual IRI="Shop"/>
    </ObjectPropertyAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="connectedTo"/>
        <NamedIndividual IRI="Router4"/>
        <NamedIndividual IRI="Router2"/>
    </ObjectPropertyAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="connectedTo"/>
        <NamedIndividual IRI="Router4"/>
        <NamedIndividual IRI="Router3"/>
    </ObjectPropertyAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="connectedTo"/>
        <NamedIndividual IRI="Router4"/>
        <NamedIndividual IRI="Router1"/>
    </ObjectPropertyAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="locatedIn"/>
        <NamedIndividual IRI="Router4"/>
        <NamedIndividual IRI="BuildingB"/>
    </ObjectPropertyAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="locatedIn"/>
        <NamedIndividual IRI="Shop"/>
        <NamedIndividual IRI="BuildingB"/>
    </ObjectPropertyAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="locatedIn"/>
        <NamedIndividual IRI="StorageRoom"/>
        <NamedIndividual IRI="Vienna"/>
    </ObjectPropertyAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="temperature"/>
        <NamedIndividual IRI="Vienna"/>
        <NamedIndividual IRI="EnvironmentalServerVienna"/>
    </ObjectPropertyAssertion>
    <DataPropertyAssertion>
        <DataProperty IRI="isOutside"/>
        <NamedIndividual IRI="BuildingA"/>
        <Literal datatypeIRI="&xsd;boolean">false</Literal>
    </DataPropertyAssertion>
    <DataPropertyAssertion>
```

```
366         <DataProperty IRI="isOutside"/>
            <NamedIndividual IRI="BuildingB"/>
368         <Literal datatypeIRI="&xsd;boolean">false</Literal>
        </DataPropertyAssertion>
370     <DataPropertyAssertion>
            <DataProperty IRI="ipv6address"/>
372         <NamedIndividual IRI="EnvironmentalServerLinz"/>
            <Literal datatypeIRI="&xsd;string">fe80::150</Literal>
374     </DataPropertyAssertion>
        <DataPropertyAssertion>
376         <DataProperty IRI="ipv6address"/>
            <NamedIndividual IRI="EnvironmentalServerVienna"/>
378         <Literal datatypeIRI="&xsd;string">fe80::150</Literal>
        </DataPropertyAssertion>
380     <DataPropertyAssertion>
            <DataProperty IRI="isOutside"/>
382         <NamedIndividual IRI="Linz"/>
            <Literal datatypeIRI="&xsd;boolean">true</Literal>
384     </DataPropertyAssertion>
        <DataPropertyAssertion>
386         <DataProperty IRI="isOutside"/>
            <NamedIndividual IRI="Office"/>
388         <Literal datatypeIRI="&xsd;boolean">false</Literal>
        </DataPropertyAssertion>
390     <DataPropertyAssertion>
            <DataProperty IRI="ipv6address"/>
392         <NamedIndividual IRI="Router1"/>
            <Literal datatypeIRI="&xsd;string">fe80::221:6aff:fe6a:dbc</Literal>
394     </DataPropertyAssertion>
        <DataPropertyAssertion>
396         <DataProperty IRI="ipv6address"/>
            <NamedIndividual IRI="Router2"/>
398         <Literal datatypeIRI="&xsd;string">fe80::221:6aff:fe6b:dbc</Literal>
        </DataPropertyAssertion>
400     <DataPropertyAssertion>
            <DataProperty IRI="ipv6address"/>
402         <NamedIndividual IRI="Router3"/>
            <Literal datatypeIRI="&xsd;string">fe80::221:6aff:fe6c:dbc</Literal>
404     </DataPropertyAssertion>
        <DataPropertyAssertion>
406         <DataProperty IRI="ipv6address"/>
            <NamedIndividual IRI="Router4"/>
408         <Literal datatypeIRI="&xsd;string">fe80::221:6aff:fe6d:dbc</Literal>
        </DataPropertyAssertion>
410     <DataPropertyAssertion>
            <DataProperty IRI="isOutside"/>
412         <NamedIndividual IRI="Shop"/>
            <Literal datatypeIRI="&xsd;boolean">false</Literal>
414     </DataPropertyAssertion>
        <DataPropertyAssertion>
416         <DataProperty IRI="isOutside"/>
            <NamedIndividual IRI="StorageRoom"/>
418         <Literal datatypeIRI="&xsd;boolean">false</Literal>
        </DataPropertyAssertion>
420     <DataPropertyAssertion>
            <DataProperty IRI="isOutside"/>
```

```
422         <NamedIndividual IRI="Vienna"/>
            <Literal datatypeIRI="&xsd;boolean">true</Literal>
424     </DataPropertyAssertion>
        <SubObjectPropertyOf>
426         <ObjectProperty IRI="connectedTo"/>
            <ObjectProperty IRI="network"/>
428     </SubObjectPropertyOf>
        <SubObjectPropertyOf>
430         <ObjectProperty IRI="humidity"/>
            <ObjectProperty IRI="fusedValue"/>
432     </SubObjectPropertyOf>
        <SubObjectPropertyOf>
434         <ObjectProperty IRI="humiditySensor"/>
            <ObjectProperty IRI="physicalValue"/>
436     </SubObjectPropertyOf>
        <SubObjectPropertyOf>
438         <ObjectProperty IRI="locatedIn"/>
            <ObjectProperty IRI="location"/>
440     </SubObjectPropertyOf>
        <SubObjectPropertyOf>
442         <ObjectProperty IRI="locatedInInv"/>
            <ObjectProperty IRI="location"/>
444     </SubObjectPropertyOf>
        <SubObjectPropertyOf>
446         <ObjectProperty IRI="temperature"/>
            <ObjectProperty IRI="fusedValue"/>
448     </SubObjectPropertyOf>
        <SubObjectPropertyOf>
450         <ObjectProperty IRI="temperatureSensor"/>
            <ObjectProperty IRI="physicalValue"/>
452     </SubObjectPropertyOf>
        <InverseObjectProperties>
454         <ObjectProperty IRI="locatedInInv"/>
            <ObjectProperty IRI="locatedIn"/>
456     </InverseObjectProperties>
        <SymmetricObjectProperty>
458         <ObjectProperty IRI="connectedTo"/>
        </SymmetricObjectProperty>
460     <TransitiveObjectProperty>
            <ObjectProperty IRI="locatedIn"/>
462     </TransitiveObjectProperty>
        <ObjectPropertyDomain>
464         <ObjectProperty IRI="connectedTo"/>
            <Class IRI="Networknode"/>
466     </ObjectPropertyDomain>
        <ObjectPropertyDomain>
468         <ObjectProperty IRI="humiditySensor"/>
            <Class IRI="Location"/>
470     </ObjectPropertyDomain>
        <ObjectPropertyDomain>
472         <ObjectProperty IRI="temperature"/>
            <Class IRI="Location"/>
474     </ObjectPropertyDomain>
        <ObjectPropertyDomain>
476         <ObjectProperty IRI="temperatureSensor"/>
            <Class IRI="Location"/>
```

```
478     </ObjectPropertyDomain>
        <ObjectPropertyRange>
480         <ObjectProperty IRI="connectedTo"/>
            <Class IRI="Networknode"/>
482     </ObjectPropertyRange>
        <ObjectPropertyRange>
484         <ObjectProperty IRI="humiditySensor"/>
            <Class IRI="HumiditySensor"/>
486     </ObjectPropertyRange>
        <ObjectPropertyRange>
488         <ObjectProperty IRI="locatedIn"/>
            <Class IRI="Location"/>
490     </ObjectPropertyRange>
        <ObjectPropertyRange>
492         <ObjectProperty IRI="temperature"/>
            <Class IRI="Server"/>
494     </ObjectPropertyRange>
        <ObjectPropertyRange>
496         <ObjectProperty IRI="temperatureSensor"/>
            <Class IRI="TemperatureSensor"/>
498     </ObjectPropertyRange>
        <SubDataPropertyOf>
500         <DataProperty IRI="ipv6address"/>
            <DataProperty IRI="network"/>
502     </SubDataPropertyOf>
        <SubDataPropertyOf>
504         <DataProperty IRI="isOutside"/>
            <DataProperty IRI="location"/>
506     </SubDataPropertyOf>
        <SubDataPropertyOf>
508         <DataProperty IRI="network"/>
            <DataProperty abbreviatedIRI="owl:topDataProperty"/>
510     </SubDataPropertyOf>
        <DataPropertyDomain>
512         <DataProperty IRI="ipv6address"/>
            <DataSomeValuesFrom>
514             <DataProperty IRI="ipv6address"/>
                <Datatype abbreviatedIRI="xsd:string"/>
516         </DataSomeValuesFrom>
        </DataPropertyDomain>
518     <DataPropertyRange>
            <DataProperty IRI="isOutside"/>
520         <Datatype abbreviatedIRI="xsd:boolean"/>
        </DataPropertyRange>
522     <DLSafeRule>
            <Body>
524             <ClassAtom>
                    <Class IRI="Location"/>
526                 <Variable IRI="urn:swrl#loc"/>
                </ClassAtom>
528             <ClassAtom>
                    <Class IRI="Location"/>
530                 <Variable IRI="urn:swrl#superLoc"/>
                </ClassAtom>
532             <ClassAtom>
                    <Class IRI="Server"/>
```

```
534                 <Variable IRI="urn:swrl#serv"/>
            </ClassAtom>
536             <ObjectPropertyAtom>
                    <ObjectProperty IRI="locatedIn"/>
538                 <Variable IRI="urn:swrl#loc"/>
                    <Variable IRI="urn:swrl#superLoc"/>
540             </ObjectPropertyAtom>
                <ObjectPropertyAtom>
542                 <ObjectProperty IRI="temperature"/>
                    <Variable IRI="urn:swrl#superLoc"/>
544                 <Variable IRI="urn:swrl#serv"/>
                </ObjectPropertyAtom>
546         </Body>
            <Head>
548             <ObjectPropertyAtom>
                    <ObjectProperty IRI="temperature"/>
550                 <Variable IRI="urn:swrl#loc"/>
                    <Variable IRI="urn:swrl#serv"/>
552             </ObjectPropertyAtom>
            </Head>
554     </DLSafeRule>
</Ontology>
556


558
<!-- Generated by the OWL API (version 3.4.2) http://owlapi.sourceforge.net -->
```

Listing A.1: OWL file representing spatial semantics (functional syntax)

Bibliography

[1] Jason Andress. *The basics of information security : understanding the fundamentals of InfoSec in theory and practice*. Waltham, Mass. : Syngress, 2011.

[2] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. In *ACM Transactions on Computer Systems (TOCS)*, volume 4, pages 273–298. ACM, 1986.

[3] Ed. C. Kaufman. RFC4306 - The Internet Key Exchange (IKEv2) Protocol, `http://tools.ietf.org/html/rfc4306`. Accessed: 2013-09-06.

[4] Clark and Parsia. Pellet reasoner, `http://clarkparsia.com/pellet/`. Accessed: 2013-04-02.

[5] European Commission. Fp7 - 7th european framework programme, `http://ec.europa.eu/research/fp7/index_en.cfm`. Accessed: 2013-08-15.

[6] W3C (World Wide Web Consortium). `http://www.w3.org`. Accessed: 2013-08-07.

[7] M. Crawford. Transmission of IPv6 Packets over Ethernet Networks, `http://tools.ietf.org/html/rfc2464`. Accessed: 2013-09-12.

[8] Wilfried Elmenreich. Fusion of Continuous-Valued Sensor Measurements using Confidence-Weighted Averaging. In *Journal of Vibration and Control*, 2007.

[9] Barnaghi et al. Sense and sensability: Semantic data modelling for sensor networks, `http://personal.ee.surrey.ac.uk/Personal/P.Barnaghi/doc/SENSEI-paper-CCSR.pdf`. In *Proceedings of the ICT Mobile Summit*, pages 1–9, 2009.

[10] C. Partridge et al. RFC1546. Host Anycasting Service, `http://tools.ietf.org/html/rfc1546`. Accessed: 2013-08-13.

[11] Gowri Dhandapani et al. Netlink Sockets - Overview, `http://qos.ittc.ku.edu/netlink/html/index.html`. Accessed: 2013-08-11.

[12] Klaus Wehrle et al. *The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel*. Prentice Hall, 2004.

[13] Luigi Atzori et al. The Internet of Things: A survey. In *Computer Networks*, volume 54. Elsevier B.V, 2010.

[14] Pascal Hitzler et al. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 2009.

[15] S. Deering et al. RFC2460. Internet Protocol, Version 6 (IPv6) Specification, `http://tools.ietf.org/html/rfc2460`. Accessed: 2013-08-07.

[16] T. Koponen et al. A Data-Oriented (and Beyond) Network Architecture. In *SIGCOMM'07*. ACM, 2007.

[17] Van Jacobson et al. Networking Named Content. In *CoNEXT'09*. ACM, 2009.

[18] Miniwatts Marketing Group. World internet user statistics and world population stats, `http://www.internetworldstats.com/stats.htm`. Accessed: 2013-08-19.

[19] Patrick Crowley Haowei Yuan, Tian Song. Scalable NDN Forwarding: Concepts, Issues and Principles. 2012.

[20] Geoff Huston. IPv4 Address Report, daily generated, `http://www.potaroo.net/tools/ipv4/index.html`. Accessed: 2013-05-22.

[21] DARPA internet program. RFC791. Internet Protocol, Protocol Specification, `http://tools.ietf.org/html/rfc791`. Accessed: 2013-08-07.

[22] Marine Metadata Interoperability. MMI Device Ontology, `https://marinemetadata.org/references/sensorsontmmi`, 2008. Accessed: 2013-08-07.

[23] Nicolas Bouliane Jan Engelhardt. *Writing Netfilter modules, `http://inai.de/documents/Netfilter_Modules.pdf`*. Accessed: 2013-08-11.

[24] S. Kent. RFC4302. IP Authentication Header, `http://tools.ietf.org/html/rfc4302`. Accessed: 2013-09-06.

[25] S. Kent. RFC4303. IP Encapsulating Security Payload (ESP), `http://tools.ietf.org/html/rfc4303`. Accessed: 2013-09-06.

[26] Antonio Liotta. The Cognitive Net Is Coming. In *Spectrum*, volume 50. IEEE, 2013.

[27] Geoff Mulligan. The 6LoWPAN Architecture. In *EmNets2007*. ACM, 2007.

[28] Draintree networks. IEEE 802.15.4 and ZigBee resources, http://www.daintree.net/resources/index.php. Accessed: 2013-09-12.

[29] NFC-Forum. www.nfc-forum.org. Accessed: 2013-08-29.

[30] University of Manchester. FaCT++ reasoner, http://owl.man.ac.uk/factplusplus/. Accessed: 2013-04-02.

[31] University of Manchester. The owl api, http://owlapi.sourceforge.net/. Accessed: 2013-08-13.

[32] Stanford University School of Medicine. DL Query, http://protegewiki.stanford.edu/wiki/DLQueryTab. Accessed: 2013-08-07.

[33] Internet of Things Architecture. http://www.iot-a.eu. Accessed: 2013-08-29.

[34] OpenWSN. 6LoWPAN, https://openwsn.atlassian.net/wiki/display/OW/6LoWPAN. Accessed: 2013-09-12.

[35] IoT6 Project. http://www.iot6.eu. Accessed: 2013-08-07.

[36] S. Kent, K. Seo. RFC4301. Security Architecture for the Internet Protocol, http://tools.ietf.org/html/rfc4301. Accessed: 2013-09-06.

[37] W3C. OWL 2 Web Ontology Language Profiles (Second Edition), http://www.w3.org/TR/2012/REC-owl2-profiles-20121211/. Accessed: 2013-09-16.

[38] W3C. OWL Tutorial, http://www.w3schools.com/rdf/rdf_owl.asp. Accessed: 2013-04-10.

[39] W3C. OWL (Web Ontology Language) 2, http://www.w3.org/TR/owl2-overview/. Accessed: 2013-08-07.

[40] W3C. SPARQL, http://www.w3.org/TR/rdf-sparql-query/. Accessed: 2013-08-07.

[41] W3C. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, http://www.w3.org/Submission/SWRL/. Accessed: 2013-08-07.