Seamless Integration of Smart Objects into the Internet Using XMPP and mDNS/DNS-SD

Von der Fakultät für MINT - Mathematik, Informatik, Physik,
Elektro- und Informationstechnik
der Brandenburgischen Technischen Universität Cottbus–Senftenberg

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

vorgelegt von

Diplom-Informatiker (Dipl.-Inf.)
Ronny Klauck

Geboren am 09.08.1982 in Cottbus

Gutachter: Prof. Dr.-Ing. habil. Hartmut König
Gutachter: Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Gutachter: Prof. Dr. Jürgen Schönwälder

Tag der mündlichen Prüfung: 26.04.2016

Abstract

With the integration of smart objects into the Internet users should gain new possibilities to directly interact with their physical environment. This vision is called *Internet of Things (IoT)* and is enabled by the development of micro *Internet Protocol (IP)* stacks that allow one to directly connect smart objects to the Internet. IP alone cannot ensure a seamless integration because advanced services (e.g., service discovery, identity management) can only be provided at the application layer. The current development of application protocols for the IoT focuses on the *Machine-to-Machine (M2M)* communication and introduces specialized protocol gateways, smart object-specific code or data representations that hinder a seamless integration. This thesis deals with the seamless integration, discovery, and employment of smart objects into the *current Internet infrastructure* under *Human-to-Machine (H2M)* communication aspects by using and adapting already established protocols that have been standardized by the *Internet Engineering Task Force (IETF)*, such as the *Extensible Messaging and Presence Protocol (XMPP)*, *Multicast DNS (mDNS)*, and *DNS Service Discovery (DNS-SD)*. The proposed approach is called *Chatty Things*. So smart objects may become a natural part of the network making the IoT readily usable for (non-technical) users and network administrators providing them with the same level of usability that is predominant in the current Internet infrastructure.

The applicability of XMPP and mDNS/DNS-SD for smart objects has been evaluated with implementations of minimized, modular, and extensible software stacks for the IoT operating system *Contiki*. This includes a readily usable *Application Programming Interface (API)*, an essential set of XMPP extension protocols, a proposal for lightweight and user-friendly event notification, a standardized bootstrapping, and a seamless fallback mechanism for ad hoc use cases when infrastructure services are failing for XMPP-driven smart objects. Furthermore, this thesis presents optimizations for the used protocols to reduce the network traffic in low data rate smart object networks (e.g., sensor-specific groups, enhanced message compression mechanisms). To sum up, this thesis shows how XMPP and mDNS/DNS-SD can be used economically on smart objects for the seamless integration with low effort into the current Internet infrastructure to enable a transparent (H2M) interaction and service discovery for the IoT.

Contents

1 Motivation

Extending the current Internet with a *"things"-oriented concept* [1] will allow these *"things"* of the physical world and humans to directly interact with each other [2]. This vision was named *"Internet of Things"* by Kevin Ashton [3] and describes a technological progress in which wirelessly communicating constrained devices (e.g., smart objects [4]) will work in synergy to help improving our daily lives [2]. Connecting the real world with the Internet opens a new way of observation and a new view of detail on the underlying physical processes. The Internet of Things (IoT) vision enables new kinds of pervasive and ubiquitous Internet applications and services which can be used by humans to remotely control and monitor the environment, while reacting on events or interpreting real-time data of the physical world [5]. The integration of smart objects into the Internet enables an intuitive Human-to-Machine (H2M) interaction with the physical world through computer-based and mobile devices [6]. These objects belong to the Internet analogously ordinary computational devices today. The access to their provided information should simply be available for all Internet users via well-known applications and protocols [7].

There is a wide range of IoT applications covering many areas, such as retail, logistics, pharmaceutical, ubiquitous intelligent devices, ambient assisted living, environmental and social aspects. The monitoring of objects during their lifetime will offer a complete history of any item to protect consumer rights, to prove rightful ownerships, and to improve the quality management along the supply chain of goods. In the future goods may be transported automatically without user interaction directly from the producer to the consumer including intelligent decisions about the shortest path. These smart objects can assist human beings in the optimal use of drugs and can protect them from overdoses. Ubiquitous intelligent devices around us (e.g., smart clothes, smart books) will be able to exchange information with each other to dynamically adjust the climate control or to interact with entertainment systems in houses or cars. Intelligent homes can be realized without developing dedicated and expensive systems using smart objects to address and to control lamps, light bulbs, and every single smart device in an intelligent way. Smart hotel rooms will preconfigure themselves (e.g., temperature, lighting, tv channels)

when the traveler arrives because everyone and everything can be connected to the Internet of the future to exchange information. Early warning systems can be set up by connecting worldwide sensor information to detect and prevent catastrophes. The Internet of Things allows the design of smart and cost-efficiency communication systems for the disaster avoidance and management on top of hybrid and heterogeneous networked systems/sensors, which can be installed permanently or deployed ad hoc in emergency situations. Furthermore, smart objects can be used to reach and monitor inaccessible or remote locations to gather information for the rescue forces [2].

The Internet of Things offers a wide area of new possibilities for developing novel Internet services, but it poses a number of challenges that have to be solved [5, Sec. 4]:

- **Scalability:** The extension of the Internet with a large number of smart objects requires certain basic functionalities, e.g., communication and service discovery, which have to equally efficiently work in all given network environments without hindering the growth of the Internet at the same time;
- **Self-configuration:** The inclusion of smart objects into a new environment should be carried out, if procurable, without any user interaction or a complicated setup. Smart objects should configure themselves to automatically adapt to given situations;
- **Interoperability:** Since smart objects possess varying capabilities, they need standardized communication procedures to address and cooperate with each other in a vendor-independent manner;
- **Discovery:** Smart objects should be able to announce their availability and their advertised services in the given environment for automatically being identified or found by other objects or nearby users via discovery, look up, or name services;
- **Data Volumes and Data Interpretation:** There will be scenarios with rare communication, which produce small amounts of data, and real world scenarios, such as smart object networks or logistics, producing huge data volumes. To support users with useful information of their environment, filter mechanisms need to address data or events target-oriented;
- **Security and Personal Privacy:** Security mechanism are needed on each smart object to ensure privacy, while blocking unwanted information flows which may contain sensitive personal data;

- **Power Supply:** Ideally, smart objects should work very power-efficient and harvest energy from their environment, so that they do not need to be connected to a power supply, but the current development of batteries and energy harvesting technologies progresses only slowly. Therefore energy-efficient hardware (e.g., low power processors and communications modules) and software (e.g., protocol stacks) is required;
- **Wireless Communication:** For the communication between smart objects, low power wireless technologies are preferred to adapt to a low bandwidth because energy is the most concerning aspect of this device class.

A large number of different smart objects will emerge as new dynamic and global resources of the Internet for use by networks, services, and applications (cp. [8, Sec. 1.2]).

The implementation of the IoT vision is strictly coupled with the development of Internet Protocol (IP) based solutions for smart objects. IP is the dominating basic technology of the Internet used to couple different network infrastructures and has proven to be highly scalable [9, Sec. 1.2.3]. Connecting smart object networks directly to the Internet with IP would be an ideal way to ensure interoperability [10] because the development and use of diverse networking protocols can be omitted [11]. This allows a simplified connectivity model without using specialized protocol gateways, which are unavoidable when proprietary protocols are deployed. Instead, routers can be used, which are the standard way to connect networks in the Internet (cp. [12, 13]). Moreover, IP-based protocols enable users and programmers to reuse the experience and solutions that have widely been applied in the Internet for decades [1, 14]. Thus, standardization and interoperability are an important presumption for the Internet of Things. [7, Sec. 14].

1.1 Problem Statement

IP alone cannot ensure an automatic integration of smart objects due to varying protocols used at the higher layers in the IoT and Internet. Current IoT application protocols introduce new mechanisms or new dependencies, such as smart object-specific code or data representations, that differ from the established standards used in the Internet at the application layer for computational devices (cp. [12]). Therefore application protocol gateways are required that introduce additional complexity in terms of message translation and protocol version support [11, 15]. Message translation is typically time-consuming and failure-prone [16]. It reduces the flexibility, scalability, and end-to-end functionality from

the protocol [5] and security [17] point of view. In addition, gateways as single point of failures interrupt the communication among the smart objects when they fail. Protocol gateways are a limiting factor that should be omitted when integrating smart objects into IP-based infrastructures because they again separate smart objects from ordinary computational devices at the application layer and neutralize the existing integration of both worlds at the IP layer ("One Internet vs. Islands", [18]). Thus, a similar situation may repeat at the application level as at the IP level years ago [13]. Currently a seamless integration of smart objects into the Internet is, therefore, not possible. With IP as the underlying protocol, running established standards at the application layer can boost the handling/interaction with and the integration of smart objects [12].

Furthermore, a seamless integration strategy should also have the human user in mind, i.e., the Human-to-Machine (H2M) communication. Current IoT application protocols for the integration into the Internet focus on Machine-to-Machine (M2M) communication. The use of wireless-enabled mobile devices has grown exponentially during this decade and created scenarios in which access to a wide variety of services from ubiquitous resources is desired without a deep knowledge by the user [19]. Smartphones represent a programmable and flexible platform with a wide range of applications (apps), while at the same time leveraging from the human element when carried as ubiquitous and pervasive commodity hardware [20]. Instead of requiring users to learn new interaction schemes to access data from their environment, smart objects should seamlessly be integrated into the Internet infrastructure with known and standardized approaches. As users already use smartphones for their daily communication (e.g., Instant Messaging (IM) and chat) and stay in touch with their (human) environment, we prefer solutions that support the Internet's end-to-end principle and that integrate into the software a user is familiar with. Therefore, smart objects should become directly accessible by the Internet community via established Internet mechanisms (e.g., applications, protocols) [7, Sec. 10.3]. There is a broad range of approaches to overcome this issue. Web services and middlewares are two of them, but they provide similar features with an incompatible integration paradigm. This produces an unnecessary variety and intricacy for a smooth user interaction with smart objects in the IoT [11]. Such a scenario should be strictly avoided because it would lead to a fragmentation of the IoT in the worst case, prevent its growth, and lower its acceptance rate [2]. A common protocol stack as basic communication concept for all supported classes of IoT devices would easily enable a seamless integration of smart objects into the Internet (cp. [18, Sec. 3.1.2]). Smart objects cannot be divorced from the rest of the Internet if the Internet of Things should couple all connected devices and should allow them to discover

and communicate vendor-independent with each other, while respecting the end-to-end principle of the Internet.

Therefore, we prefer the use of an established application protocol for discovery, interoperability, addressability, and self-configuration ("arrive and operate", scalability) through a standardized scheme that complies with the conventional Internet domain [5]. Self-configuration and self-management are key-enablers to allow the seamless integration and automatic handling of billions of devices independent of the network environments and user interaction. Adding smart objects to any operational environment has to be possible without a long and difficult installation procedure because the growing amount of such devices in the IoT cannot be handled with manual setups (i.e., parameter-less bootstrapping and service discovery). Smart objects can be placed everywhere. If no infrastructure network (e.g., no fixed access point) is available or the devices are mobile, the smart objects should automatically adapt to their environment by forming ad hoc networks for routing information towards the infrastructure or to a dedicated smart object which is accessible by users (i.e., hybrid smart object networks) [2]. Since IoT "applications are typically event-based: the application performs most of its work in response to external events" [9, Sec. 1.2.2]. Thus, application protocols which are based on the publish-subscribe paradigm should be favored to support the nature of smart objects (i.e., "things"-oriented concept) and to realize an efficient notification of events from objects to human beings (cp. [21]). Beside the (technical) integration of smart objects at the network and application layer, the integration into the workflow and the life of the users is very important (i.e., usability) because smart objects communicate with their environment, e.g., with nearby people carrying computational devices and vice versa [5, Sec. 2]. To sum up, an established application protocol for the IoT should combine scalability, usability, efficiency, and flexibility to provide a common set of features as core (e.g., publish-subscribe, identity management, authentication, bi-directional communication) and as extension (e.g., service discovery, hybrid smart object networks). Only then the collaboration between all IoT-based devices and human beings can be realized without limiting the manifold possibilities for future demands and solutions, as required in [11].

1.2 Focus of this Thesis and Research Issues

In this thesis, an XMPP layer for IP-based smart objects is proposed that provides a minimized, modular, and standardized protocol stack to seamlessly integrate constrained

devices into the *current Internet infrastructure* and to enable an easy H2M interaction with the physical environment through familiar software in hybrid (ad hoc and infrastructure) networks. The *Extensible Messaging and Presence Protocol* (XMPP) [22], *Multicast DNS* (mDNS) [23], and *DNS Service Discovery* (DNS-SD) [24] are standardized by the Internet Engineering Task Force (IETF) and widely deployed in the Internet. XMPP itself is a set of flexible and open Extensible Markup Language (XML) technologies for real-time data stream and IM applications that are expandable through XMPP Extension Protocols (XEPs) to adopt to various environments and scenarios. From the network point of view, using XMPP as the default communication protocol allows us to implement pervasive networking without using a middleware or a protocol gateway. This approach will guarantee the end-to-end access for all classes of devices which should be supported in the IoT because the functionality is directly placed in the devices and their offered services can be accessed at the higher layers as required by [18, Sec. 3.1.3]. XMPP simplifies the interconnection of devices [5]. An important aspect for pervasive networking is that XMPP provides ad hoc Peer-to-Peer (P2P) communication with *XEP-0174 Serverless Messaging* [25] via mDNS and DNS-SD. These technologies are based on work of the IETF's Zeroconf working group that defined several standards in the field of service-oriented networking, also known as Bonjour. Both, XMPP and Bonjour, offer a rich variety of open source software for servers, clients, and libraries. They support several mobile and desktop operating systems. Thus, the implementation of IoT applications will strongly benefit from it. The time and cost needed to develop, test, and maintain IoT applications on smart objects can be reduced because existing tools can simply be reused [11]. Developers can use the Application Programming Interface (API) and concentrate on the real problem instead of struggling with the constrained resources of smart objects and new programming paradigms, while users will get in touch with complete and consumer-friendly IoT solutions.

An important focus of this thesis is the optimization of the H2M interaction because XMPP was initially designed for the communication among humans. As smart object networks have an event-driven nature [9, Sec. 1], the publish-subscribe paradigm of XMPP is very well suited for the notification of events from smart objects to users (cp. [26, Sec. III.4]) without disturbing users in their daily workflow. The efficient event distribution cannot be handled well by Web-based approaches (e.g., HTTP/REST, cp. [27]). As the Hypertext Transfer Protocol (HTTP/1.1) [28] is based on the request-response paradigm, data changes are not automatically pronounced to interested entities and have to be checked periodically by each interested entity itself. This polling mechanism is inefficient and can cause a high network traffic, even if there is no data change (cp. [21]). Smart

objects can collect a huge amount of data from their environment [29] and thus a lot of information has to be pushed to the Internet which can confuse and overstrain ordinary users. Therefore, mechanism supporting users to easily subscribe to events and information they are really interested in via an XMPP chat client will be investigated, which allow users to filter and prioritize information. Further significant drawbacks of HTTP/REST-based approaches are that following links become insufficient for the search and the discovery of services in the IoT [27] and that no bi-directional communication between HTTP clients is possible without running a HTTP server additionally at the client side [30, Sec. 5]. In conjunction with mDNS/DNS-SD, the (automatic) discovery of smart objects and their services is investigated. This is still an open issue in the IoT, as it requires that smart objects describe themselves in a way human beings and computational devices can understand (cp. [27]). Furthermore, self-configuration and bootstrapping in hybrid network environments are validated as part of the XMPP layer for smart objects, i.e., helping to set up smart environments without any user interaction (including a simple and standardized fallback strategy based on *XEP-0174 Serverless Messaging*).

Due to the scarce resources of smart objects in terms of low memory, slow microcontrollers, and low data rate, the adaption of XMPP and mDNS/DNS-SD focuses on these parameters because of their initial design target for larger computer systems with nearly no limit of bandwidth and hardware resources. Therefore, the most essential functions for typical IoT appliances and H2M/M2M communication need to be prioritized, mapped to existing or possibly new XEPs, implemented, and tested for the efficient use of these protocols on smart objects. Redesigning XEPs, splitting up functionalities, and reducing redundant transmitted data will be therefore necessary. Following the building blocks concept, each XEP will be implemented as an independent module and can be chosen as an optional feature to complete the XMPP Core functions as needed during compile time of the minimized protocol stack. A special challenge is the implementation of XEPs as tiny modules with reduced code size and an economic use of message exchange (e.g., 127 bytes maximum packet size for IEEE 802.15.4), while extracting protocol behavior unnecessary for resource-constrained devices and analyzing several possibilities to reduce the XML message overhead with and without compression techniques. The minimized XMPP and mDNS/DNS-SD implementations have still to be standard-compliant despite maintaining a low memory and bandwidth profile. In this sense, an application-specific protocol support should be enabled that depends on the actual duties and scenarios of smart objects in the IoT vision but retains the strength of XMPP and its expandability though XEPs. Note that the avoidance of protocol gateways requires to prevent the introduction of smart

object-specific code and data representations, since all classes of IoT devices should be treated equally. To sum up, the implementation of XMPP and mDNS/DNS-SD with a low memory footprint leads to a general system design for the IoT vision.

This thesis validates the applicability of XMPP and mDNS/DNS-SD to smart objects in terms of memory efficiency, low data rate support, high flexibility, and H2M interaction to ensure the interoperability of smart objects with the Internet as well as the scalability of the complete XMPP-based system design. Thus, the research questions we address in this thesis are: (1) how to scale XMPP down to work as substrate on constrained devices for the Internet of Things; (2) what are the minimum memory requirements for using XMPP and mDNS/DNS-SD on constrained devices; (3) what are possible improvements for XMPP and mDNS/DNS-SD in order to achieve a low network traffic; (4) how to support human users filtering and getting an up-to-date view on all interested events to which a user has subscribed to; (5) how to search and discover smart objects using the established DNS protocol to bootstrap XMPP-driven smart objects without the need for any user interaction or manual pre-configuration. To foster the proposed approach we provide the following contributions to the Internet community: open source implementations of parts of the developed XMPP and mDNS/DNS-SD components for the Contiki operating system and active involvement in the standardization of discovering XMPP-driven (smart) objects for the IoT as XMPP Extension Protocol (XEP).

As this thesis is based on the benefits of IP for smart objects, it concentrates on the seamless integration into the existing Internet infrastructure and the H2M interaction at the application layer. Thus, research issues related to network layer and lower layers (e.g., multi-hop scenarios which are mostly used for M2M communication), power consumption measurements, and implementations of security functions are not in the scope of this thesis. Secured communication is an important aspect in the Internet of Things. XMPP supports secured communication, e.g., using the Transport Layer Security protocol (TLS) [31]. There is ongoing research on implementing the protocol for smart objects. The considerations of this thesis to use XMPP as a dedicated application protocol for seamlessly integrating smart objects into the Internet do not have any impact on this. Therefore, security aspects are not the main concern of this thesis.

1.3 Outline of this Thesis

An introduction of smart objects, their integration into the Internet, and their technical limitations is given in Chapter 2. Furthermore, currently developed application layer protocols and their drawbacks are discussed. It will be shown that a seamless integration of smart objects into the Internet cannot be achieved with these approaches.

Chapters 3, 4, 5, and 6 present the main contributions of this thesis. Chapter 3 presents our approach to scale XMPP down to work as substrate on highly constrained devices for the Internet of Things. For this, the Chatty Things approach is introduced that provides a standardized application layer and system architecture based on XMPP and mDNS/DNS-SD for a seamless integration of IP-based smart objects into the Internet at the application layer. It is discussed which essential set of XMPP features is required to make the H2M interaction readily usable for the IoT.

The following Chapters 4 and 5 focus on the minimization of the memory requirements for using XMPP and mDNS/DNS-SD on highly constrained devices and possible improvements for these protocols to reduce the network traffic. Therefore, the required technical steps and the architectural solutions for the adaption of XMPP and mDNS/DNS-SD on IP-based smart objects without the introduction of smart object-specific code and data representations are described. We present a minimized and modular implementation of an XMPP stack that provides a readily usable API, an essential set of XEPs for the IoT, and a proposal for lightweight event notification to support human users with a filter mechanism and an up-to-date view on all interested events to which a user has subscribed to. Moreover, we present a minimized mDNS/DNS-SD implementation with adjustments for smart objects to enable a standardized service discovery at the application layer for the IoT. Based on this we present a solution for Chatty Things to discover and to connect to XMPP servers in a given network without any pre-configured parameters. It further implements enhanced DNS message compression mechanisms to effectively reduce the number of exchanged IP packets in low data rate smart object networks.

In Chapter 6 we evaluate the performance of the prototypical implementations of the Chatty Things' main components. Chapter 7 completes this work with concluding remarks.

2 Connecting Smart Objects to the Internet

Smart objects [7] can be described from a hardware point of view as highly constrained devices equipped with sensors, actuators, a low power microprocessor with scarce memory (Kbytes of ROM and RAM), and a low power radio device (e.g., IEEE 802.15.4, low power WLAN), which often operate battery-powered [2, 4]. From the software point of view a smart object runs a (tiny) operating system (e.g., Contiki OS [32], Tiny OS [33]) and specific applications that define its behavior. The integrated sensors are used to detect characteristics of physical objects, whereas actuators can be used to actively interact with physical objects or systems of the real word [34, Sec. 3.2]. The communication between smart objects and the environment is an essential part of the IoT vision. Therefore, radio devices have to work very efficient to allow a long lifetime of battery-operated smart objects because sending and receiving data consume much more energy than for the processing of data [7, Sec. 11.3]. IEEE 802.15.4 [35] defines a low power, low data rate, and low cost wireless link standard at the physical and the MAC layer for the efficient communication over lossy links, which is heavily desired to ensure interoperability at higher layers [36, Sec. 4]. Therefore, IEEE 802.15.4 is an often used radio technology in conjunction with the realization of smart object networks [7, Sec. 9.2.2]. The combination of hardware components and the energy-efficient behavior of the IEEE 802.15.4 standard [37, Sec. 2.2] allow smart objects to sense, save, and transfer measured values as well as to make decisions about themselves, while communicating and interacting with other objects, devices, or systems [38, Sec. 3.1]. Smart objects require a small physical size to be embedded in everyday objects and their price should be low to create versatile things [13].

The following chapter describes the interconnection of smart objects with ordinary computational devices via IP in Section 2.1 and their technical limitations due to the heterogeneity of available hardware platforms in Section 2.2. Furthermore, Section 2.3 introduces the requirements for a seamless integration of smart objects at the application layer and reviews state of the art integration approaches and developed application layer protocols for the IoT. Section 2.4 discusses the drawbacks of the current approaches and the need for a seamless integration of smart objects at the application layer.

2.1 Using IP to Interconnect Different Classes of Devices

IP-based integration strategy for smart objects has become widely used [39], since it provides efficiency in terms of routing, message overhead, latency, and energy cost [36]. The routing between smart object networks and traditional IP networks is carried out by the border routers which convert 802.15.4 (i.e., 127 bytes maximum packet size) / 6LoWPAN[1] frames to Ethernet / IPv6 frames (i.e., 1280 bytes minimum MTU size) and vice versa. To efficiently embed IPv6 packets in 802.15.4 frames (e.g., fragment oversized packets, statelessly compress packet headers, forward packets via multi-hop wireless routes) to use IP in smart object networks the Internet standard RFC[2] 4944 [41] was proposed to allow a seamless integration of resource-constrained devices [5, Sec. 6]. Thus, smart objects can communicate natively with IP to other IP networks, with each other, and with any IP device respecting IP's end-to-end principle [36], as depicted in Figure 2.1.

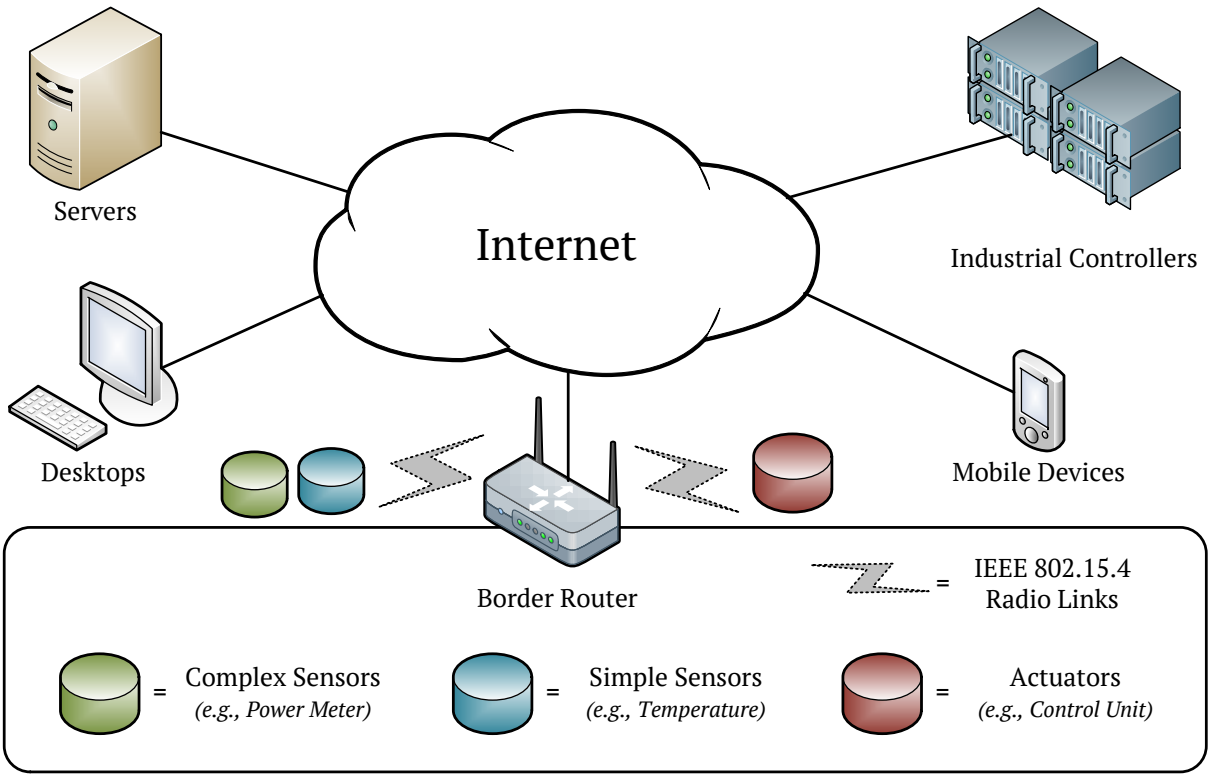

Figure 2.1: Interconnecting several classes of devices over IP (adapted from [36, Sec. 2])

[1] IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals (RFC 4919 [40]).
[2] Request for Comments (RFC) [Online] `http://www.ietf.org/rfc.html`.

Sensors, actuators, and other embedded devices can be part of the same IP network as servers, industrial controllers, PCs, or smartphones without specialized gateways because the interoperability is provided at the network layer [11, 42]. This will allow the development of a new set of future applications which require the integration and the interoperability with existing network infrastructures and existing applications as well as the heterogeneity for technologies on the hardware, software, and communication level [43].

2.1.1 Drawbacks of Protocol Gateways

The current Internet is based on the IP end-to-end principle. First smart object networks were not based on IP [5, Sec. 2]. Multi-protocol gateways were used to interconnect these networks with IP networks, either encapsulating the traffic into IP packets or translating the protocols. Protocol translation is a complex task because network protocols differ in their used semantics, mechanisms, and logic. Protocol translation gateways may cause further limitations, such as incomplete address management, restricted support for a set of specialized protocols [44, Sec. 3.1], or even break the network models on both sides if their protocol paradigms are not mapped exactly. Management and failure analysis are very difficult and augment with the number of supported protocol translations which can only be handled by experts rather than by usual administrators. Moreover, multi-protocol translation gateways represent a networking bottleneck in terms of scalability, flexibility, and reliability. They introduce an undesirable state and a single point of failure in the network, while each protocol improvement entails changes in the gateways. With the use of IP in smart object networks, gateways are not needed and not recommended any more because they lead to a wrong architectural design of the IoT vision that will hinder the development of future innovative applications [7, Sec. 3.8].

2.1.2 IP is Lightweight and Low Power

With the development of memory-efficient and low power IP stacks like uIP(v6) [45], resource-constrained devices can implement IP using only a low memory footprint of a few Kbytes of ROM / RAM on low power 8-bit microcontrollers [46] and run over sub-milliwatt radio links enabling years of operation even for multi-hop scenarios [13]. The Contiki operating system for the Internet of Things integrates *uIP* [47] – a full RFC-compliant TCP/IP stack for 8-bit microcontrollers that is well known in the IoT community. Real world tests with a home-monitoring application have shown that IP-based smart objects can

achieve an average duty-cycle[3] of 0.65%, an average per-hop latency of 62 ms, and a data reception rate of 99.98% [36]. Furthermore, the cost of IP to transmit 1 byte is only 1.67 μJ, and the transmission of an IP packet is correspondingly 630 μJ [36]. These results were achieved through implementing useful optimization techniques for uIP, such as distributed TCP caching, spatial IP address assignment, header compression, and application overlay routing [46] in conjunction with optimizations on the link layer, such as duty cycling mechanisms (e.g., power-saving MAC protocols [42, Sec. 3.2], [49]), and efficient routing with effective link estimation [36]. To sum up, IP for smart object networks has become widely accepted [42] because it facilitates the integration of these devices into the Internet and it has shown that it can provide the same performance (e.g., energy consumption, data throughput) as using specialized protocols on resource-constrained smart objects [10]. In addition, IP provides scalability, stability, interoperability, and efficiency to enable the communication among billions of devices to set up large-scale networks. Combining the availability of small and low power IP stacks for smart objects IP is the promising choice for the IoT vision [13].

2.2 Heterogeneity of Smart Object Hardware Platforms

The multitude of available hardware platforms for smart objects needs hardware-independent solutions that provide an homogeneous access, such as the Contiki operating system and small IP stacks, to overcome their different characteristics (see Table 2.1). Unfortunately, these differences are not fully abstracted by Contiki and its integrated IP stack leading to technical limitations (i.e., supported IP packet and memory sizes) for adapting established application protocols. In the following, we discuss these limitations.

Table 2.1: Classes of constrained smart objects (in Kbytes, taken from RFC 7228 [4])

Name	**ROM**	**RAM**
Class 0	$<<$ 100	$<<$ 10
Class 1	$\sim$ 100	$\sim$ 10
Class 2	$\sim$ 250	$\sim$ 50

[3] A duty cycle is the time a smart object is active for sensing, computing, and sending data. The rest of the time it is in sleep mode to save energy [48, Sec. 3.4].

2.2.1 The Contiki Operating System

Contiki [32] OS is an open source operating system for the Internet of Things running on embedded hardware with constrained memory and computing resources with a typical configuration of 40 Kbytes of ROM / 2 Kbytes of RAM. Version 2.5 of the Contiki operating system was released in September 2011 and is used as basis for the implementation and the evaluation of the developed prototype in this thesis. Contiki's core system is based on an event-driven kernel with on-demand preemptive multi-threading to effectively share the low memory resources among all processes. To provide concurrency processes are implemented as event handlers that run till completion and return to the kernel when finished. A process is implemented either as an application program or as a service. Services provide functions that can be used by application programs. When running the Contiki system, processes and services can dynamically be loaded and unloaded. Inter-process communication is provided through the kernel by posting events.

In addition, the operating system Contiki provides a tool chain to facilitate software development and debugging. The software-based power profiling mechanism [50] can be used to determine the power-efficiency of existing smart object hardware platforms and software designs, while requiring only small code changes to the tested application. The COOJA cross-layer network simulator [51] helps to shorten the compile-run-debug cycle for the development and test of a set of smart objects combining low level simulation of the node hardware and high-level simulation of the node behavior. COOJA is able to run Contiki programs as compiled native code or simulates unmodified target platform firmware via the instruction level simulator MSPsim [52] for the MSP430 microcontroller [53]. Furthermore, COOJA supports cross-vendor interoperability simulation by running non-Contiki nodes (e.g., TinyOS-based nodes [54] or nodes implemented in Java) [55].

2.2.2 uIP: Low Power IP Stack

The uIP [56] stack supports the protocols ARP, IPv4/v6, Serial Line IP (SLIP), ICMP echo, UDP, and TCP. It implements all protocol features of RFC 1122 [57] for the host-to-host communication, but it skips certain mechanisms for the interaction between the application and the stack (e.g., IP options, multiple interfaces, TCP congestion control, out-of-sequence TCP data, data buffered for rexmit) to reduce the code size to only a few Kbytes. Implemented TCP/IP features of uIP are: IP and TCP checksums, IP fragment reassembly, multiple TCP connections, TCP options, variable TCP Maximum Segment

Size (MSS), Round Trip Time (RTT) estimation, TCP flow control, and TCP urgent data. Retransmissions have to be handled by the application because outgoing data are not buffered by the stack. Instead the corresponding application is called with a flag by uIP to resend the data. The IPv6 version of uIP is a fully tested (IPv6 Ready Phase 1 certified) and a low power IP stack due to power-efficient radio mechanisms, such as ContikiMAC [49], which allows smart objects to operate with IP-based networks.

The memory use of uIP depends on the application requirements: the amount of traffic, the number of simultaneous connections, and the choice of the Application Program Interface (API). For programmers, uIP provides two APIs. Protosockets is a BSD socket-like API implementation without full multi-threading, whereas the raw API is event-based and more low level than protosockets, but it uses less memory. The event-driven interface informs the application on top of uIP and is the recommended API for UDP-based communications. Protosockets is only available for TCP connections and advertises a transparent API to send data without handling retransmissions, acknowledgements, and to read data which is split into several TCP segments.

IP Message Size Limits

Contiki's uIP stack uses the layers below IP (e.g., Rime [58] for IPv4, 6LoWPAN [40] for IPv6) and their provided features (e.g., fragmentation) to efficiently route IP packets in a network, in the following for short *lower layer(s)*. An IP packet relies on the lower layer fragmentation skills, which again depend on the hardware platform and its built-in radio transceiver. This limits the supported IP packet size of Contiki and the allocated global packet buffer of uIP without using IP fragmentation.

Table 2.2 summarizes the maximum available sizes of an IP packet for each supported hardware platform and radio module, which were collected from the Contiki *platform* folder. If these values are exceeded, IP fragment reassembly must be enabled, which costs an additional amount of RAM and 700 bytes of code size. Experiments with lower layer fragmentation have shown that this mechanism is energy-efficient for request-response cycles, as there is no need to optimize the number of fragments [68, Sec. VI]. Devices like the AVR Raven or the Redbee Econotag can handle the lower layer fragmentation very well and thus support a higher IP packet size compared to other hardware platforms (e.g., Tmote Sky, Zolertia Z1). Therefore, it is recommended that applications respect the available IP payload size for sending application data to avoid the use of IP fragment

Table 2.2: Supported IP packet sizes of Contiki 2.5 (in bytes)

Hardware Platform / Radio Module	**IPv4**	**IPv6**
AVR Raven [59]	1300	1300
AVR ZigBit [60]	240	240
ESB [61]	110	110
MEMSIC IRIS [62]	128	240
STM32 [63]	140	140
MSB430 [61]	116	116
MEMSIC MICAz [64]	128	240
Redbee Econotag [65]	1300	1300
Tmote Sky / MEMSIC TelosB [66]	108	240
Zolertia Z1 [67]	108	140

reassembly. The Zolertia Z1 has one of the lowest IP packet sizes for IPv4/v6, while the Tmote Sky supports a medium IP packet size for IPv6, as depicted in Table 2.2. Thus, the two hardware platforms are good reference devices to test under which circumstances and limits messages of standard application protocols, which were originally designed for the use of an MTU size of 1280 bytes[4], fit into a single IP packet of the uIP stack.

Available IP Payload Sizes of Application Data

The available size of application data for sending a single IP packet depends on the maximum IP packet size supported (e.g., global packet buffer, `UIP_BUFSIZE`) of each device driver (see *core/net/uip.h*) subtracted by the following header sizes:

- **Link Level (LL) Header Size:** Offset to the IP header in the global packet buffer. This value is only used for Ethernet and set to 14. The default value is 0. It is defined in *core/net/uipopt.h*, but it can be overwritten by each hardware platform.
- **IP Header Size:** This value is set to 40 for IPv6 and to 20 for IPv4. It is defined in *core/net/uip.h*.
- **TCP Header Size:** The value is set to 20 as defined in *core/net/uip.h*.

[4]Minimal MTU for IPv6 which all links must handle [69, Sec. 5].

Table 2.3 shows the available TCP / UDP payload sizes for IPv4 and IPv6 of the Tmote Sky and the Zolertia Z1, which are determined by the corresponding device drivers and provided through the macro `UIP_APPDATA_SIZE` of the uIP API. The sizes of the link level header and the sizes of the TCP payload for each hardware platform were taken from the platform-specific *contiki-conf.h* configuration files in the Contiki *platform* folder.

Table 2.3: IPv4 and IPv6 header and payload sizes of uIP under Contiki for selected hardware platforms (in bytes)

Hardware Platform	**Header (LL / IP / TCP)**	**Payload (TCP / UDP)**
Tmote Sky (IPv4)	0 / 20 / 20	48 / 68
Zolertia Z1 (IPv4)	0 / 20 / 20	48 / 68
Tmote Sky (IPv6)	0 / 40 / 20	48 / 180
Zolertia Z1 (IPv6)	0 / 40 / 20	48 / 80

The available TCP payload size is preconfigured as TCP Maximum Segment Size (MSS), thus the TCP payload is limited by uIP on both hardware platforms and the used IP version. In contrast to UDP that supports lower layer fragmentation to enlarge the available IP payload size, TCP has a header overhead problem for each sent IP packet because TCP (in conjunction with protosockets) does not make use of this lower layer fragmentation for non of the selected hardware platforms. So the TCP, IP, and lower layer headers consume the largest part of each sent TCP packet over IEEE 802.15.4 links (cp. [37, Sec. 4]).

2.2.3 Focused Smart Object Hardware Platforms

For the development and evaluation of our software stacks and IoT applications, we focus on the Zolertia Z1 and the Tmote Sky/TelosB hardware platforms. Both represent typical constrained devices of class 0 (i.e., limited memory under 100 Kbytes of ROM and 10 Kbytes of RAM [4], cp. Table 2.1). They offer a variety of connectors (e.g., IEEE 802.15.4 and USB) for debugging, testing, and deployment of the firmware images. Moreover, these devices have comparable hardware specifications because both are based on the MSP430 16-bit microcontroller [53] family. The Zolertia Z1 [67] is based on a low power MSP430F2617 microcontroller with 92 Kbytes of ROM[5] and 8 Kbytes of RAM. The Z1 also has built-in sensors (e.g., temperature, accelerometer, battery level), support for

[5] 64 Kbytes maximum are available, since MSP430 has no 20-bit address space extension.

external sensors [70], and communication capabilities with an IEEE 802.15.4-compliant RF transceiver (Chipcon 2420[6] [71]) and a microUSB connector. The Tmote Sky/TelosB [66] is equipped with a MSP430F1611 microcontroller, 48 Kbytes of ROM / 10 Kbytes of RAM, built-in sensors (e.g., temperature, humidity, light) and uses the same radio module[7] as the Zolertia Z1.

2.3 Integration Approaches at the Application Layer

The integration of smart objects does not stop at the network layer because for a seamless integration a solution at the application layer is needed to advertise advanced services (e.g., discovery, identity management) for the implementation of IoT applications (cp. [42, 72]). In the following, we give an overview on currently used concepts and application protocols on top of IP, which were designed for or adopted to smart object networks with the promise to fulfill these requirements of the IoT.

2.3.1 Web of Things

As the web is the most used Internet service today, the idea behind the Web of Things is to make smart objects directly accessible via the Hypertext Transfer Protocol (HTTP/1.1) [28] on top of TCP over 6LoWPAN [73]. This can be done either by running a web server on smart objects or by using gateways which translate HTTP requests to proprietary protocol requests for accessing the sensed data of smart objects placed behind this gateway. Both variants have been implemented and successfully tested for resource-constrained smart objects. The first variant has the advantage that no additional translation mechanism is needed to seamlessly integrate smart objects into the web [27]. The second variant has the drawbacks argued in Section 2.1.1. RESTful web services allow the access to HTTP-driven smart objects via the Representational State Transfer (REST). This is a lightweight architectural model providing resource abstraction through Uniform Resource Identifiers (URI) [42, Sec. 3]. URIs link resources and offer smart objects in a simple way to use services of the resources by following their links [27]. Discovering and identifying the required resources via links is inefficient because a link can become outdated, i.e.,

[6] It has a buffer size of 128 bytes.

[7] The differences under Contiki (cp. Table 2.2) may related to different board layouts, the integration and the choice of the other used third party hardware components.

smart objects provide contextual information, whereas a context can often move from one to the other [27]. Moreover, a link is defined by a long string of characters that is hard to remember when accessing smart objects *spontaneously* [48]. Short links are not an optimal solution for this problem [74], since they require an additional shortener service acting as a translator from long to short links and vice versa similar to a redirector. This causes performance and several security issues, such as phishing [75] or spam [76], because intermediated services advertised by third parties are required.

REST uses the request-response paradigm of HTTP (e.g., the HTTP operations `GET`, `PUT`, `POST`, `DELETE`), which allows clients to request information from a server and get the answer as a response. This paradigm fits well for control-oriented applications, but it has a real drawback for scenarios (e.g., event-driven and streaming applications) in which changes in the sensed data have to be sent immediately to interested devices or human beings. Monitoring-oriented applications require a form of asynchronism because smart objects should be able to send updates of the physical world in real-time or when results were computed rather than being polled periodically (cp. [27, 77]). There are many technologies to overcome this issue of HTTP (e.g., server-sent events [78], pubsubhubbub [79]), but these solutions are workarounds and can only be used in an application-specific manner (e.g., supported by a web browser). The Web of Things can be seen as one possible direction for the integration of smart objects at the application layer, but it has the drawback of a non-appropriate event handling for larger scale implementations (cp. [21, 80]). Protocols based on the push mechanism offer a more suitable solution for real-time event handling and communication, but they have not been considered as research topic to realize the IoT vision up to now because web services have been the research focus in the last years (cp. [77, Sec. 1.3]).

2.3.2 Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) [81, 82, 83] has been developed by the IETF Constrained RESTful Environments (CoRE) working group and has been standardized since June 2014. It is a REST-based transfer protocol designed for M2M communications with a very low message overhead, reduced response times, and a set of basic services (e.g., reliability, discovery) in comparison to HTTP. In contrast to HTTP, CoAP uses UDP as basis transport protocol and introduces a compact binary message format. Reliability is implemented with an own approach at application layer by marking a message as confirmable in the CoAP header. For the discovery of CoAP instances, the CoRE link

format [84] was invented, which is a dedicated description for available CoRE resources, their attributes, and link relationships. CoAP instances can be interconnected with the Internet via CoAP-to-HTTP protocol gateways. Thus, CoAP accepts advantages, e.g., well-known interaction methods and reliability, and disadvantages, e.g., request-response and discovery of resources via web linking, of the HTTP/REST approach (cp. Section 2.3.1). Moreover, CoAP relies on smart object-specific code and data representations, while reinventing TCP-like features at the application layer. According to [85, Sec. 3.3.2], a TCP reinvention must be omitted, since most application layer protocols are based on TCP and need to be redesigned to work with UDP. This endangers the interoperability of the Internet according to the former IETF chair *Russ Housley* because current standards are not embraced [86]. To make CoAP a more suitable solution for the Internet of Things (cp. [80]) further protocol extensions are currently being investigated by the CoRE working group, such as pushing of information [87] or a TCP transport [88]. The latter solves integration issues of CoAP into current infrastructures regarding UDP blocking by firewalls, Network Address Translation (NAT), or the unawareness of CoAP in middleboxes.

2.3.3 Message Queuing Telemetry Transport for Sensors (MQTT-S)

The Message Queuing Telemetry Transport (MQTT) protocol [89] has been designed for the efficient M2M communication of sensors and implements a lightweight broker-based publish-subscribe paradigm. Specification 3.1 of MQTT provides features, such as three Quality of Service (QoS) levels, command messages, or dynamic topics. Since MQTT could hardly be implemented on smart objects, a low bandwidth version, called Message Queuing Telemetry Transport for Sensors (MQTT-S) [90], has been introduced to fit the needs of smart object networks. However, MQTT-S is not compatible with the used transport protocol (i.e., MQTT-S uses UDP and MQTT uses TCP), message format, and feature-set (e.g., only support of QoS level-1) with MQTT. For the interconnection of MQTT and MQTT-S clients MQTT-S protocol gateways, MQTT-S forwarder, and MQTT broker are needed. The scalability of the architecture highly depends on the performance and the connection management overhead of the gateway (e.g., transparent or aggregating) [91]. Further drawbacks of MQTT(-S) are that there exist only a few implementations for computational devices and that the protocols are not an Internet standard, which definitely will lower their acceptance rate.

2.3.4 Devices Profile for Web Services (DPWS) for Embedded Devices (uDPWS)

The Devices Profile for Web Services (DPWS) for embedded devices (uDPWS) [92] provides web services for embedded microcontrollers to integrate them in existing infrastructures. Therefore, uDPWS uses standardized protocols like IP, UDP/TCP, and the Simple Object Access Protocol (SOAP). Services like dynamic discovery, subscribing to services, and receiving events from web services are supported by uDPWS. The main drawback at the moment is that DPWS is only available for systems running Microsoft's Windows operating system and Microsoft's Universal Plug'n'Play (UPnP) stack. Thus, an application- and network-independent announcement of services to couple different classes of devices cannot be implemented.

2.3.5 Sensor Web Enablement (SWE)

Sensor Web Enablement (SWE) [93] is a standard developed by the Open Geospatial Consortium (OGC). SWE is a generic framework for a platform- and protocol-independent interaction between sensors to realize complex scenarios, such as traffic, environmental, and industrial process monitoring. It, however, requires a large infrastructure support [93, Sec. 4.1]. Appliances of SWE depend on a complex middleware for the sensor network management. Different versions of the middleware can disturb the cooperation if needed updates fail. In addition, the middleware must be available for nearly every used operating system to ensure a frictionless integration of a wide range of device classes.

2.4 Towards a Seamless Integration of Smart Objects

uIP provides the possibility to support many standardized high-level services on smart objects [12]. This interoperability at the application layer is crucial for the integration of smart objects into existing Internet infrastructures because already available software and configuration tools can be used, testing efforts can be reduced, while at the same time easing installation and deployment. The reason is that all IP devices can immediately interoperate with each other without additional translation mechanisms or dedicated software support [7, Sec. 3.1]. Unfortunately, the further direction of the Web of Things is influenced by the ongoing development of CoAP/MQTT-S. Both bring back already

solved issues from the network layer to the application layer. Protocol gateways separate traditional computational devices (e.g., use of HTTP, MQTT) from smart objects (e.g., use of CoAP, MQTT-S) and new protocols are explicitly introduced to work only in resource-constrained environments (e.g., use of smart object-specific code or data representations, many features are limited [88, 94]). Gateways represent bottlenecks. They are single points of failure for the whole network traffic to and from the smart object network introducing an undesirable state to the network. When a protocol gateway fails, the interaction with the smart objects is disrupted because no fallback mechanisms for an ad hoc communication with ordinary computational devices are foreseen (i.e., no support of hybrid smart object networks). Latest scalability measurements of CoAP for its web integration show that using CoAP-to-HTTP gateways is significantly slower compared to web platforms that speak CoAP directly [80]. Thus, a frictionless integration of smart objects into current infrastructures and a scalable IoT architecture cannot be realized at the same time using protocol gateways (cp. [44, Sec. 3.1]).

The reduction of protocol gateways makes it important to avoid the introduction of smart object-specific code and data representations, especially on computational devices, because this will break already available software. This is a real issue for current application protocols, such as CoAP, because their focus is on the data transport. Moreover, MQTT and MQTT-S are not standardized by the IETF. This can lower their acceptance rate or lead to a fragmentation of the IoT [11]. Currently, there are only a handful of prototypical implementations and simulation results of CoAP and MQTT-S, but no experiences with a widely deployed and well tested real world scenario in terms of scalability and actual integration cost. This will limit the evolution, the innovation, and the interoperability of the IoT (cp. [7, Sec. 14.1]). To illustrate the wide range of upcoming IoT applications we describe two different use cases:

Smart Home, Smart Office, and Smart Hotel Room. Home automation should support consumers in their daily routines ranging from a simple environment control (e.g., lighting, temperature, safety, comfort) to assisted living [7, Sec. 23.2]. Unfortunately, home automation produces high acquisition and installation costs because it lacks standardization [7, Sec. 23.1]. In contrast to smart homes, a smart office and a smart hotel room can be seen as a temporary place to stay. These places need to adapt very fast to the needs and the preferences of their alternating users. Today a smartphone can be considered as the central element of a modern mobile user. Such a mobile device can be used to interact with the environment to store and push preferences of users to any stationary sensor, actuator,

or smart object without the need of tracking a user's profile. In the vision of [2] such a scenario should allow users to query their current location, temperature, or local weather and to get noticed via a mobile device about the local weather forecast or friends who stay already at the same building. A smart office will provide useful information on events of the building or the outer environment to the user. It can automatically adjust the workplace to the user's preferences (e.g., printer settings, room temperature, presence status) or to changing conditions (e.g., solar altitude). Nowadays, typically used technologies for smart homes are proprietary, not IP-enabled ones [95]. They require gateways and protocol adapters. Latest approaches like the *openhab* project [96] are based on IP, but still require skilled technical users for the installation and integration. Thus, these solutions are not readily usable for the masses, yet, which require an easy setup and configuration to reach a high acceptance rate [7, Sec. 23.3.9].

Flexible Post-Disaster Management. In [213, 214] we have proposed a flexible post-disaster management system which couples smart objects and computational devices with cloud technology. Post-disaster management is never a fixed task, especially when deployed sensors and the communication infrastructure get destroyed (by earthquakes or tsunamis). Thus, in a short time a new infrastructure has to be established for this temporary challenge, which should provide an uncomplicated system with an easy setup and comprehensible data access. The applications and the underlying technology should enable communication and data sharing inside the local Peer-to-Peer (P2P) user group as well as sharing the data with other users over the Internet to provide a global view on self- and remotely-sensed data. Sensor data measured by rescue forces will assist crisis management in various ways, namely in getting accurate incident reports and in making adequate real-time situation-dependent choices. A further combination of sensed data, place and time of the sensing event, and the specific sensing entity (e.g., fire fighter, rescue specialist) enables an even wider range of applications, such as environmental monitoring, marking of dangerous spots for non-involved bystanders, or traceable scanning of sites for endangered civilians. The system design should provide the flexibility to change decisions about which sensor types to integrate, where to place the smart objects, how to access them in-field, or how to interconnect the smart object network to any external network (i.e., the Internet) alongside and after the deployment. Smart objects can be used to extend smartphones with a wider array of sensors, while using such smart objects for scenarios in which the area cannot safely be reached or where rescue forces are not allowed to stay over a longer time period (e.g., next to radioactive zones).

These varying requirements can only be fulfilled either through a set of specialized protocols each designed for a dedicated use case (e.g., M2M, network management) or using a single but *highly extensible* application protocol. First approaches with a dedicated application protocol use HTTP (cp. Section 2.3.1) and the Simple Network Management Protocol (SNMP). So Schönwälder et al. [16, 39] use SNMP as the sole management protocol for all device classes to show that the adaptation of a widely used standard application protocol to smart objects is feasible, thus simplifying the management of such devices in the same way as with traditional computers in IP networks. SNMP was designed as a simple management protocol with a limited feature set. It misses aspects of active collaboration and the extensibility for new functions and future demands. A significant drawback of HTTP/1.1 is that it cannot handle well the efficient event distribution (cp. [27]). In addition to HTTP and SNMP, the standardized *Extensible Messaging and Presence Protocol* (XMPP) [22] had been proposed as a further candidate protocol, but it has never been analyzed for this, in particular whether it can directly be implemented on smart object hardware and whether it can be efficiently used in low data rate networks (cp. [26, 77]). XMPP is a set of flexible and open Extensible Markup Language (XML) technologies standardized by the Internet Engineering Task Force (IETF) and widely deployed in the Internet to implement Instant Messaging (IM), real-time user collaboration, and Voice over IP (VoIP) applications. The XML-based message structure can easily be extended with additional functions by protocol extensions, so-called XMPP Extension Protocols (XEPs) [97], to support a wide range of use cases. We discuss the applicability of XMPP for the Internet of Things in detail in Section 3.1.

Existing research on the use of XMPP in the IoT can be divided into publish-subscribe architectures, which couple sensors and actuators with an XMPP network, and specific implementations of XMPP for resource-constrained hardware platforms running Contiki. An approach to realize an XMPP-based architectures for large-scale sensing and actuation, called *Sensor Andrew*, has been introduced in [98]. The goal of this project is to create a highly scalable and extensible network which consists of several classes of devices (e.g., sensors, actuators, desktops) and supports a wide range of applications for point-to-point and multicast messaging as well as data logging. XMPP components are used as basic protocol to achieve interoperability and to reuse already established technologies. On top of it, a so-called Sensor Over XMPP (SOX) library was implemented as a sublayer to integrate sensors and actuators via protocol gateways (i.e., SOX adapters). Thus, the resource-constrained hardware platforms do not directly implement an XMPP stack, since they send only SOX messages with a JID of the receiving XMPP entity to a protocol

gateway (cp. [98, Sec. V.B]) which converts the message and forwards it to entities of the XMPP network. Another similar approach [99] uses so-called sensor bots (e.g., smartphones) as gateways to retrieve customized and squeezed down XML messages from embedded devices via USB connections at the MAC layer. The sensor bots are used to transform these messages to XMPP-compatible messages and forward them to the XMPP network at the application layer. This approach also lacks a direct implementation of the XMPP stack on the embedded devices. These two approaches introduce the same issues as discussed earlier in this section because they decouple different classes of devices at the application layer through gateways. Initial steps towards an XMPP stack for resource-constrained devices were made with the *uXMPP* [100] and the *XMPPClient for mbed* [101]. Each of them presents a lightweight XMPP client implementation with only a rudimentary XMPP Core function set and no optimizations for smart object networks. A difference between the two prototypes is that uXMPP (see Chapter 4) is based on the Contiki OS and uses the uIP stack, whereas the *XMPPClient for mbed* uses Ethernet frames which provide a larger MTU size for IP packets.

Further research can be found in the area of custom XMPP Extension Protocols (XEP). The project *OpenSpime* [102] provides XEPs for digital signatures, encryption, authority claiming, data reporting, and seeking for the communication of physical devices. Dedicated XEPs for IoT scenarios are being developed by Peter Waher[8] and focus on large-scale M2M communication for industrial-like environments. The IoT XEPs allow sensor data interchange (*XEP-0323*), provisioning of services, access rights and user privileges (*XEP-0324*), remote control (*XEP-0325*) and introduce a so-called *Concentrator* (*XEP-0326*), i.e., a gateway running an XMPP client who translates sensor data into XML messages. These XEPs rely heavily on the *Efficient XML Interchange (EXI)* format (*XEP-0322*) to transmit detailed XML messages containing additional, yet redundant meta data for the raw sensor data. This meta data is needed to interpret the raw sensor data once, but it is also included in each message, thus increasing message size. A direct implementation of these XEPs on resource-constrained devices cannot be realized because these XEPs require too much use of bandwidth and memory. Thus, these XEPs do not solve the problem of integrating smart objects seamlessly into the Internet. To solve the problem, it requires a reduced set of XEPs fitting in the memory of smart objects, allowing a low bandwidth usage in smart object networks, and providing useful features for the IoT at the same time. Such a solution is presented in the following chapters.

[8] [Online] `http://wiki.xmpp.org/web/Tech_pages/IoT_XepsExplained`.

3 Chatty Things

In order to solve the problem of the seamless integration of smart objects into the Internet at the application layer we present in this chapter an approach that uses XMPP as the underlying communication protocol to interact with different device classes. It is called *Chatty Things* [215]. This introduced style of communication via XMPP extends IP-based smart objects in the Internet of Things to *Chatty Things*, as depicted in Figure 3.1.

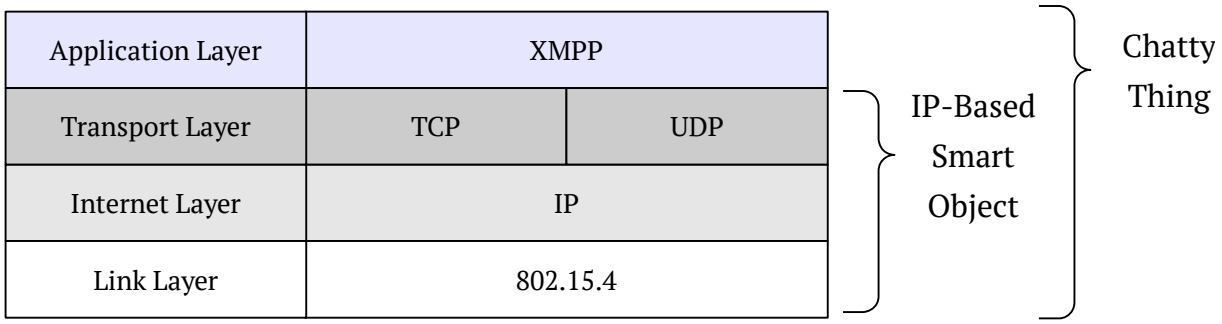

Figure 3.1: Extending IP-based smart objects to Chatty Things

3.1 Extensible Messaging and Presence Protocol (XMPP) and the IoT

XMPP is a XML-based protocol that has been designed for supporting collaborative applications, such as instant messaging and chat. It has proven to be highly scalable (e.g., 100k+ servers, 50+ million clients in 2007) because it streams XML data over long-lived TCP connections in a decentralized network for the use of inter-domain messaging [103, 104]. XMPP provides useful features as core functionalities, such as identity management, authentication, inter-domain messaging, and bi-directional communication, which are required for the seamless integration of smart objects and the development of IoT applications. A feature comparison of XMPP with CoAP and MQTT(-S) is given

in Table 3.1. A vital aspect of XMPP is that the XMPP Standards Foundation (XSF)[1] provides a continuous maintenance of the XMPP protocol family allowing system designers to benefit from all aspects of sustainability and expandability. XMPP supports various application types beside the usual message or presence propagation. Examples are ad hoc grid computing [105], intercloud directory and exchange [106], multi-agent system platform [107], or pervasive social computing applications [108]. A diversity of systems ranging from social networks (e.g., Movim [109]) to unified communication solutions (e.g., Cisco [110], Microsoft [111]) and computational devices ranging from desktop computers to mobile entities can easily be connected through XMPP. Furthermore, XMPP offers a rich variety of open source software [112] for servers, clients, and libraries supporting several operating systems thus reducing development costs. It ensures a high extensibility for various IoT scenarios and future demands.

Table 3.1: Feature comparison of currently developed IoT application protocols with XMPP

Feature	**XMPP**	**CoAP**	**MQTT(-S)**
IETF Standard	Since 2004	Since 2014	No
Use of Protocol Gateway	No	Yes	Yes
Inter-Domain Messaging	Yes	No	No
Message Bus	Publish-Subscribe	Request-Response	Publish-Subscribe
Message Format	XML / EXI	Binary	Binary
Message Overhead	High / Low	Lowest	Low
Transport Protocol	TCP	UDP	TCP (UDP)
Security	TLS	DTLS	TLS (No)
Authentication	SASL	No	Password
Identity Management	Yes [30]	No	No
Service Discovery	Only XEPs	Yes	Yes
Available Implementations	Many	Few	Few
Available Protocol Extensions	Many	Few	Few

The main advantage of XMPP is that it allows both the interaction with a server infrastructure (XMPP Core) and alternatively the ad hoc communication (P2P) via the *XMPP*

[1]The XMPP Standards Foundation (XSF) is an independent, nonprofit standards development organization with the goal to define extensions (open protocols) to XMPP.

extension (XEP) *0174 Serverless Messaging* [25]. The latter enables ad hoc communication through the use of *Multicast DNS* (mDNS) and *DNS Service Discovery* (DNS-SD) without any connection to an XMPP server. This provides a fallback mechanism for IoT applications when XMPP servers are unavailable. The high flexibility and extensibility of XMPP is deeply rooted in XML. XML enables cross-platform and cross-language communications [38, Sec. 5]. The downside of XML is its high memory and processor usage due to complex message handling and parser implementation (cp. [18, Sec. 3]). Recent researches have shown that XML-based protocols can efficiently be used in combination with message compression on smart objects in low data rate networks [85, Sec. 3]. An overview of feasible compression methods for smart objects, such as Efficient XML Interchange (EXI), is given in Section 4.2. XMPP's publish-subscribe paradigm can help users to filter and prioritize information. Publish-subscribe is organized as an interaction of components that publish messages and subscribe to classes of messages they are interested in [113]. This enables XMPP entities to efficiently interact with each other based on their own context, whereas the events are automatically distributed. It provides a bandwidth- and energy-efficient event distribution [114] in which only data changes need to be transmitted to the interested entities. This mechanism guards users from situations where they are overwhelmed with information because the collected data of the IoT will be huge [29]. In contrast, a poll mechanism can cause a high network traffic because interested entities have to regularly query all the corresponding entities when they want to be informed about the latest data (e.g., Web of Things, CoAP, cp. Section 2.3). Figure 3.2 depicts these differences and shows an optimized message flow for the use of push notification.

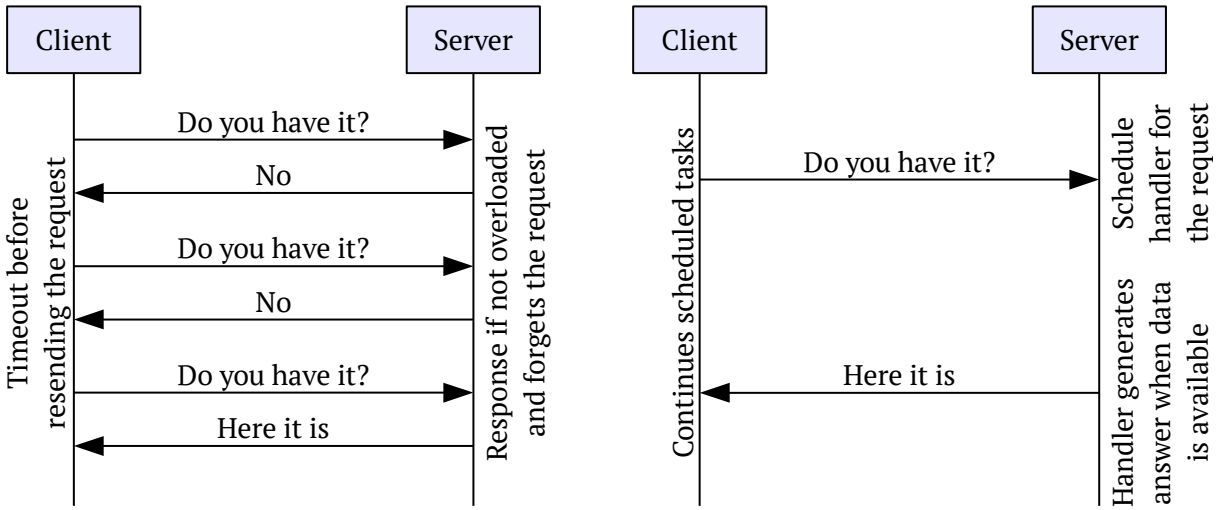

Figure 3.2: Poll (left) in comparison with push (right) (adapted from [26, Sec. III])

Beside the message overhead that can be compensated through EXI, XMPP outperforms its competitors in terms of standardization, inter-domain messaging, ease of integration, extensibility, and distribution. Depending on the requirements and challenges of the IoT, XMPP can be seen as a sort of least common denominator of the mostly needed protocol features for enabling a seamless integration of smart objects into the established Internet and a high interoperability of different classes of devices.

3.1.1 XMPP Core/IM, Address Format, and Session

The XMPP Core [22] (RFC 6120) specifies the use of long-living TCP connections for the exchange of Extensible Markup Language (XML) elements between XMPP entities over XML streams. As long as the XML stream is established, any number of XML elements, so-called XML *stanzas*, can be transmitted in an efficient manner and in near real-time. XMPP Core defines three types of XML stanzas (listed in the following). Each stanza has five common attributes: 'to', 'from', 'id', 'type', and 'xml:lang' to describe the rules of the client-to-server and the server-to-server streams:

- **Message:** Is used to push information from one XMPP entity to another. The 'to' attribute specifies the receiving XMPP entity.
- **Presence:** Is a publish-subscribe broadcast service to efficiently publish presence information (i.e., network availability information) from an XMPP entity to all subscribed entities. No 'to' attribute is set by the XMPP client for direct processing by the server (e.g., broadcast presence to other entities).
- **IQ:** Is a request-response mechanism which enables to directly query information and to receive the response from another XMPP entity similar to HTTP.

To secure XML streams the XMPP Core supports the Transport Layer Security protocol (TLS) [31]. However, the memory requirements of existing TLS implementations limit its applicability for smart objects. First approaches for implementing TLS with a low memory footprint have been published in [39, 115, 116]. For the authentication of an XMPP-specific profile, the Simple Authentication and Security Layer protocol (SASL) [117] is required. SASL supports a variety of standardized mechanisms, such as *PLAIN* [118], *CRAM-MD5* [119], *DIGEST-MD5* [120], *SCRAM-SHA-1* [121] or *ANONYMOUS* [122].

An XMPP entity is addressed via a unique Jabber Identifier (JID) [123] that consists of a local part (e.g., client name, chat room name), a domain part (e.g., server name),

and an optional resource part (e.g., device name, location). With a successful negotiation (i.e., client connects to server with same domain part), the client retrieves its contact list, called *roster*, publishes and receives the presence information to/from all XMPP entities bookmarked in the roster. Then messages can be exchanged with various XMPP entities of the same or a foreign domain similar to the email system (cp. [26, Sec. I]). XMPP IM [124] (RFC 6121) defines additional features for the XMPP Core for Instant Messaging (IM) and presence information exchange, include the subscription and the roster management.

3.1.2 XMPP Localization in Hybrid Network Environments

The localization of XMPP entities in ad-hoc and infrastructure environments (i.e., hybrid networks) can be implemented by combining XMPP Core and *XEP-0174 Serverless Messaging*. Entities with Internet access directly register and authenticate at an XMPP server, whereas entities in ad-hoc environments find each other via *XEP-0174* (see Figure 3.3). The uniqueness of the Jabber Identifier (JID) is guaranteed by the XMPP server when an XMPP entity is connected to its domain. In contrast to the infrastructure mode, the ad hoc XMPP network cannot ensure an unique JID because *XEP-0174 Serverless Messaging* is based on the third party protocols Multicast DNS (mDNS) [23] and DNS Service Discovery (DNS-SD) [24]. mDNS's functionality is to resolve domain names without the help of any unicast Domain Name System (DNS) server by sending DNS messages to a multicast group. DNS-SD announces a detailed service information (e.g., availability time, access protocol, IP address, port) via so-called DNS resource records to other devices in the network in a way that applications and users can simply look up or wait for available or needed services. Each entity analyzes the released records inside the multicast group and each of them generates a list of entities based on the received records. This means IP addresses and JIDs are generated randomly.

To ensures that XMPP entities are accessible in the two networks by the same JID an agent is used to connect ad hoc networks with the conventional XMPP infrastructure. The agent provides physical links between different network access technologies and relays connections from each entity of the ad hoc network to the XMPP server. This has the advantage that each entity directly authenticates itself at the XMPP server. Thus, a dedicated entity that exclusively acts as gateway is not required and the existing XMPP infrastructure can be used (for details we refer to [213]). Figure 3.3 illustrates the network structure of the localization system. The localization system was used in the *uBeeMe* platform [216, 217]

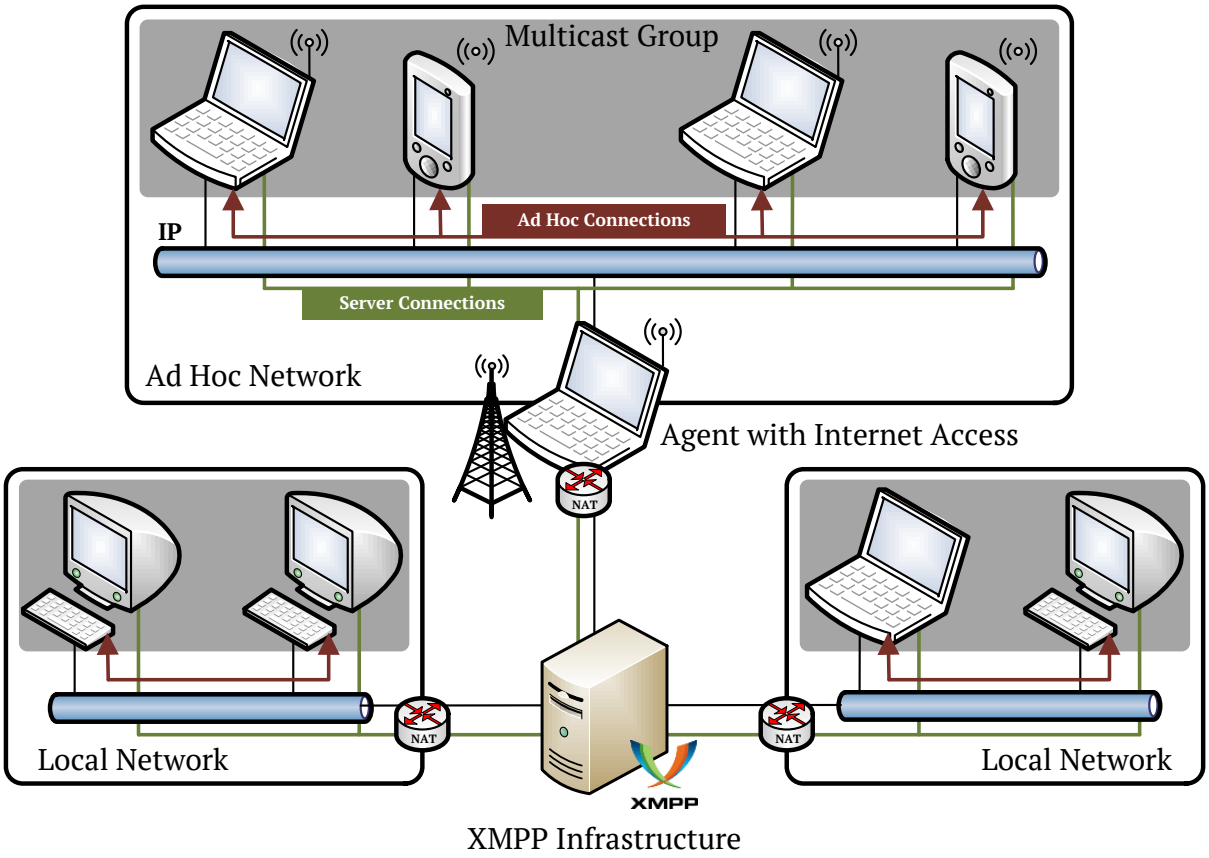

Figure 3.3: Connecting *XEP-0174* (ad hoc) clients with an XMPP infrastructure

to enable mobile collaborative applications. It can be seen as a predecessor of a possible XMPP layer for the IoT, since the agent can be easily extended to support network access technologies for smart objects (e.g., IEEE 802.15.4 radio links). For the seamless integration of smart objects into the Internet, it only requires to directly run an XMPP client on these objects without introducing smart object-specific code and data representations.

3.2 Chatty Things Approach

The Chatty Things approach aims at supporting the collaboration of smart objects with computational devices in IP-based network with a special focus on the Human-to-Machine (H2M) communication. The interaction between the different classes of devices in the IoT is handled through XMPP, as depicted in Figure 3.4. XMPP ensures the interoperability with existing service infrastructures and with currently used software. So all kind of devices can

directly *chat* with each other. Chatty Things are based on a modular XMPP (client) stack. In the minimal configuration only an implementation of the XMPP Core/IM is needed to enable basic management and communication functions (e.g., identity management, message exchange, status updates) as common services on all IoT devices. Optional XMPP features, such as *XEP-0045 Multi-User Chat (MUC)* and *XEP-0174 Serverless Messaging*), ensure high extensibility and can be implemented on top of XMPP Core/IM on-demand depending on the actual duties and scenarios.

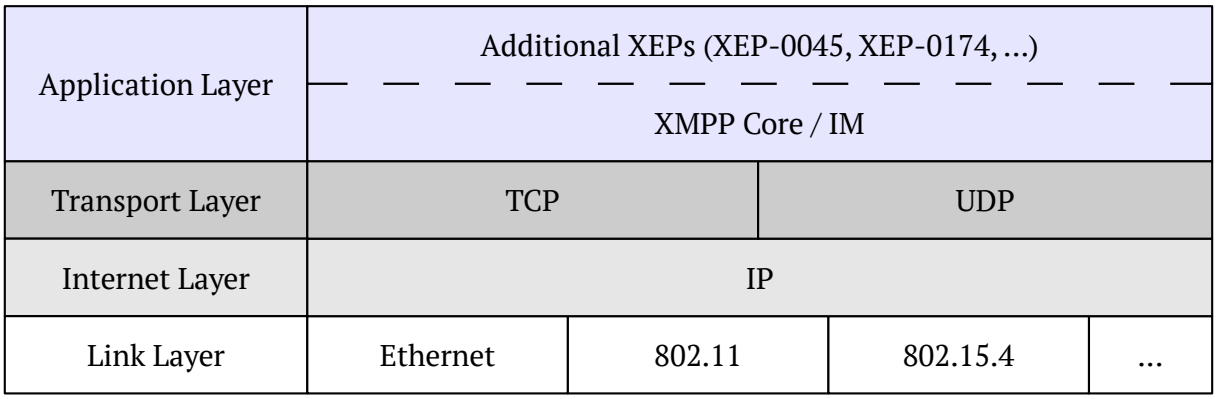

Figure 3.4: XMPP layer (and protocol extensions) for the IoT

Since the integrated sensors of smart objects are used to detect characteristics of physical objects (cp. Chapter 2), the sources for the provided information about the environment or about its status are restricted to the integrated sensors. Thus, the advertised services of smart objects are closely linked to their integrated sensors. Therefore, the services in Chatty Things, which a smart object can advertise to users and objects, are defined by the integrated sensors. The data provided by these sensors are grouped according to their semantics (sensor-specific grouping) to simplify the access. This approach will be presented in Section 3.3. In this way, the user interaction with the environment is simplified by browsing through the XMPP network for offered services and subscribing to them for upcoming events and information.

3.2.1 System Architecture for the IoT

For the seamless integration of Chatty Things, the currently established infrastructure of the Internet is used, i.e., the existing public XMPP server infrastructure and XMPP client software. Both have become widely accepted as the basic software instruments for real-time communication that can be installed on nearly every operating system to

provide users an easy access to XMPP-driven devices. Additionally, private XMPP servers can be deployed for local XMPP domains. In an XMPP infrastructure XMPP servers are Internet-connected or interconnected with each other by a domain-based network. The XMPP network consists of a decentralized client-server architecture enabling an inter-domain communication similar to the email system, as depicted in Figure 3.5.

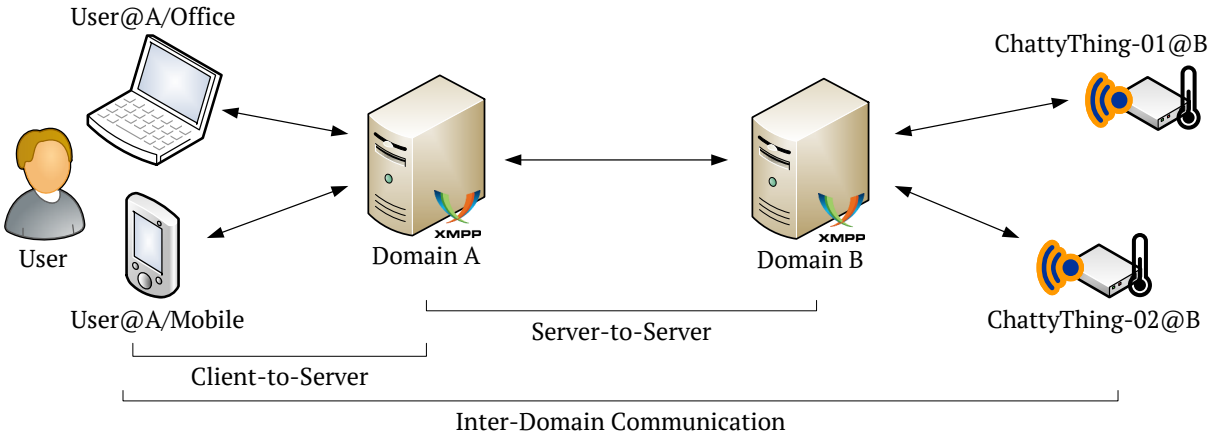

Figure 3.5: Decentralized client-server architecture of XMPP

XMPP clients are able to send messages to their domain-specific XMPP servers, while servers from foreign XMPP domains can also communicate with each other to forward messages. A single point of failure or overload situations can be prevented because a server malfunction only affects a certain domain and not the whole XMPP network. This decentralized client-server structure strengthens the **scalability** and the **reliability** of our system architecture because it can be extended with local XMPP servers or connected to remotely available once. Traffic bottlenecks can be bypassed by a direct integration of Chatty Things in IP-based networks (i.e., one-hop to local XMPP server). A basic anchor point in our system architecture is the use of the *General Purpose Access Point (GPAP)* [125] which provides physical links between different network access technologies for smart objects (e.g., IEEE 802.15.4 radio links) and ordinary computational devices (e.g., IEEE 802.11 radio links). So it interconnects different classes of devices over IP links. At the application layer all classes of devices can communicate locally and can be accessed remotely (i.e., authorized access from the Internet) at the same time through XMPP.

The GPAP in our system architecture is implemented through commodity router hardware that runs embedded Linux systems like *OpenWrt* [126] to implement the physical interconnection. OpenWrt routers are low priced (i.e., at the same price as smart object hardware), widely deployable (e.g., WLAN access points), and support a wide range of router hardware.

Already established routers can be upgraded to run OpenWrt and to act as GPAP. Routers are equipped with USB adapters to physically link IEEE 802.15.4 and Ethernet (mapped as network interfaces in the operating system). They also support additional software packages. In the following we call them *enhanced router(s)*. An enhanced router runs the XMPP server *Prosody* [127] and the *Avahi* [128] daemon. Thus, the GPAP and the XMPP server can be assembled only in a single hardware unit. Prosody XMPP servers can be used to deploy local and private XMPP domains or to advertise public XMPP domains for an environment of specific interests to users. The Avahi daemon is responsible for the forwarding of mDNS/DNS-SD messages to the different network interfaces that are interconnected by the router to enable service discovery and parameter-less bootstrapping of smart objects in the local network. Figure 3.6 depicts the generic system architecture in an exemplary use case (e.g., smart home).

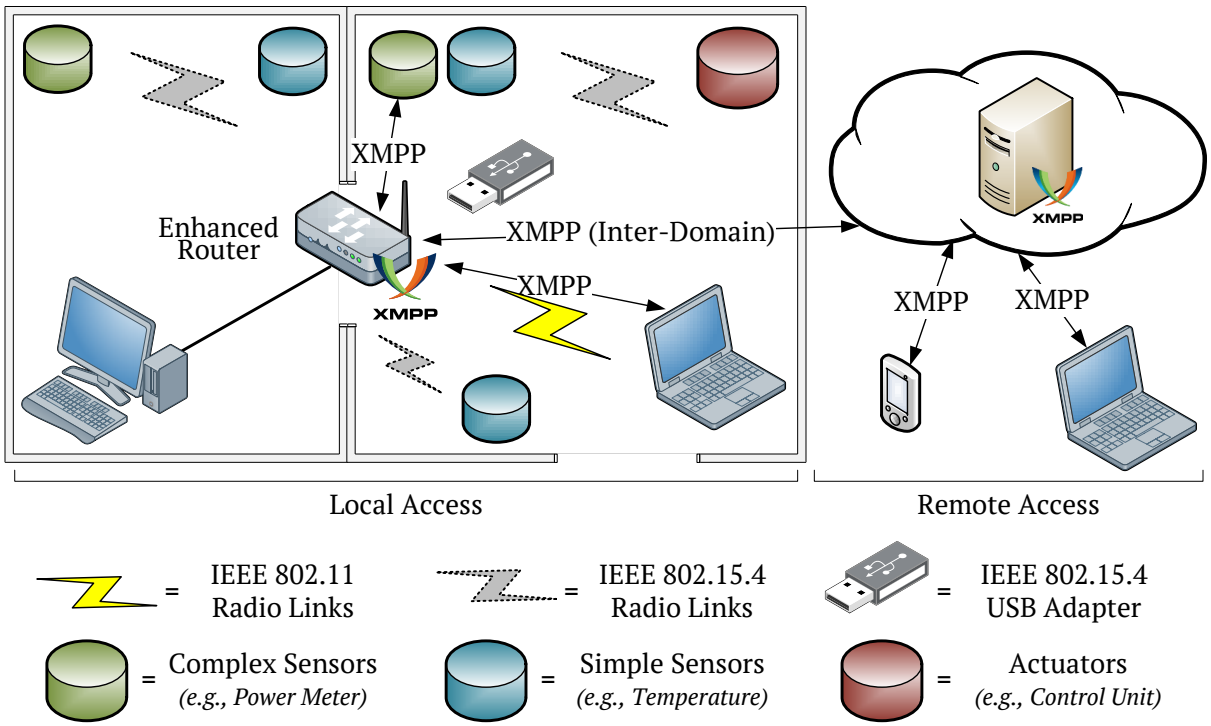

Figure 3.6: System architecture for a smart home use case

As XMPP servers are interconnected with each other similar to the email system, multi-hop scenarios can be bypassed. The reason to bypass multi-hop scenarios can be manifold. Since multi-hop provokes a higher energy consumption over long distances [129] and a bandwidth share for each involved relaying smart object, its influence on the whole network performance can be huge [130]. Furthermore, a large number of smart objects acting as

relays are required to pass a long distance. Interconnecting enhanced routers via (W)LAN allows us to pass long distances and forward data to the Internet. The advantage is that popular and efficient standards designed for a wide range communication can be used and that a hierarchical system via XMPP domains can be realized (see Figure 3.7). To pass the short distance each enhanced router covers the communication range of IEEE 802.15.4 radio links (i.e., a single router is able to interconnect several smart objects in a typical room of a household, cp. [21]).

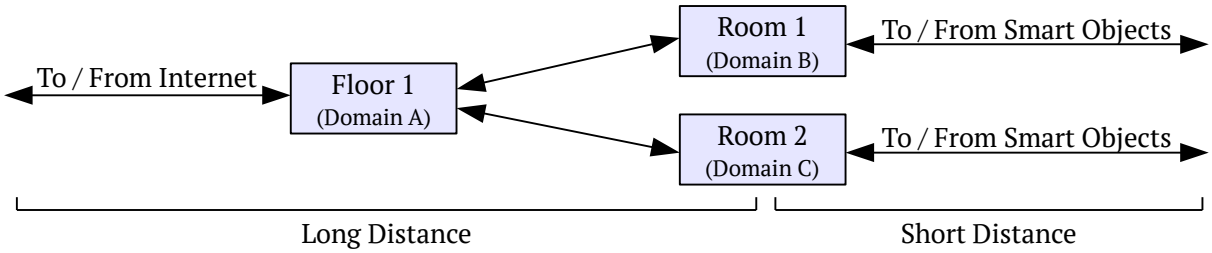

Figure 3.7: Hierarchical system of XMPP domains in a smart home use case

Fallback Mechanism: XEP-0174 Serverless Messaging

As users should have the possibility to interact with nearby Chatty Things independently of the availability of XMPP servers (i.e., infrastructure services), *XEP-0174 Serverless Messaging* (i.e., ad hoc services) is an integral part of the Chatty Things approach. This additional feature can be activated to support hybrid smart object network environments, as depicted in Figure 3.8. *XEP-0174* can be used then as a fallback mechanism if an infrastructure network (e.g., connection to the smart object network via a border router) is not present, so that an ad hoc communication can be set up directly between a computational device and a nearby Chatty Thing. If no further enhanced router is in the range of Chatty Things in case of a router failure, users can communicate ad hoc via IEEE 802.15.4 USB adapters from computational devices to directly access Chatty Things. The use of *XEP-0174 Serverless Messaging* requires an implementation of mDNS/DNS-SD for the discovery of XMPP entities via DNS messages (cp. Section 5.1). For the announcement of *XEP-0174 Serverless Messaging*, an XMPP entity advertises its online status with four DNS resource records (i.e., `SRV`, `TXT`, `A(AAA)`, and `PTR`). The value of the field name of the `SRV` record is set to the JID of the entity. When an entity joins the ad hoc network, it sends a `PTR` record (`_presence._tcp.local`) immediately to the multicast group. Now the other entities reply with their information (one `SRV`, `TXT`, `A(AAA)`, and `PTR` record per entity). If an entity wants to leave the network it has to send a `PTR` record

with Time-To-Live (TTL) set to zero. So users can discover nearby Chatty Things and can then initiate an XML stream to a Chatty Thing through a direct TCP connection.

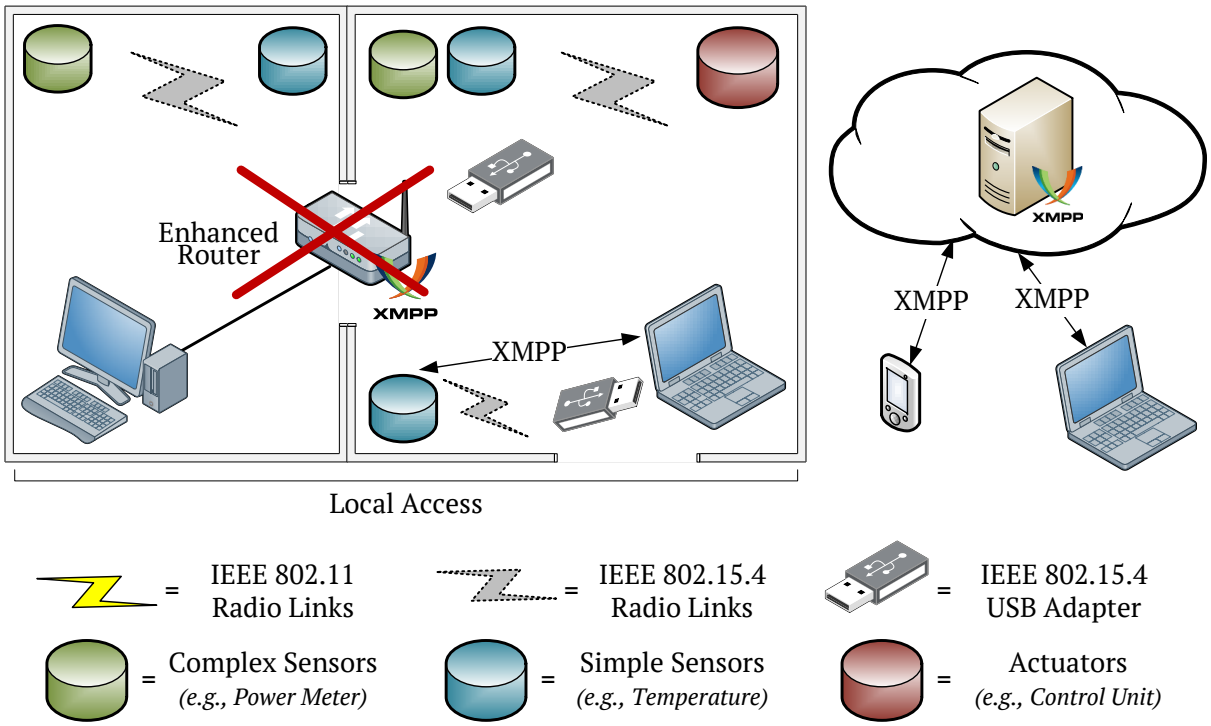

Figure 3.8: Fallback mechanism of the system architecture

Bootstrapping Chatty Things

Connecting a Chatty Thing to a server requires the IP address and the port number of the XMPP server. Since smart objects are supposed to be used in different network environments and to automatically adapt themselves, no pre-configured IP addresses can be used. This diversity of infrastructure and ad hoc networks requires an intelligent bootstrapping for Chatty Things to support hybrid network environments. At the moment, there is no standard for this process. To enable a bootstrapping without fixed and hard-coded start-up parameters (e.g., IP address or domain of the XMPP Server, JID of the XMPP entity), Chatty Things must attempt to find an XMPP server in their given network. For this, we propose the service discovery of an XMPP server using mDNS/DNS-SD similar to finding nearby Chatty Things using *XEP-0174 Serverless Messaging*. Network configuration and management is simplified with mDNS and DNS-SD because entities can detect each other inside a network without a previous distributed configuration or a

prior mutual acknowledgment [131]. The extension of a network with additional devices is simplified due to the fact that all devices are able to explore their network vicinity for available services and running applications. An XMPP server announces its availability through publishing four DNS records using the service type 'xmpp-client'[2]. If no XMPP server can be discovered in the given network during bootstrapping, Chatty Things have to enable *XEP-0174* as fallback mechanism. The overall process is called *parameter-less bootstrapping* in this thesis and described in detail in Section 5.2.1.

H2M Communication

Chatty Things particularly supports the Human-to-Machine (H2M) communication because it enables human beings to directly interact with their environment through standard chat clients (e.g., XMPP clients) – a software all Internet users are familiar with today – in a way they already know from interacting with their friends in social networks or chat rooms (cp. [26, Sec. III.4]). This allows an intuitive information handling and a centralized notification area for the users on their ordinary computational devices. So users are able to access data from smart objects and to communicate with them using standard technologies. These technologies are available for every operating system nowadays, e.g., for discovering objects and their services, for organizing objects in a contact list (e.g., an XMPP roster), for subscribing to topics that users are interested in (e.g., what happens in the user's neighborhood) or for interconnecting objects to automatically control a monitored environment (i.e., M2M). Location-aware content can directly be advertised to users who pass Chatty Things by adding them temporary to a user's roster. Bookmarking Chatty Things in the XMPP roster allows users to easily manage and group information as well as to permanently receive relevant updates of their environment similar to presence updates of their human friends. This is a huge benefit in contrast to the Web of Things approach that presents information on a website and provides the browse-ability of resources "by clicking on links" because links are inappropriate for the discovery of billions of devices [27]. Users benefit from easy-to-use and easy-to-learn (chat client) software to participate in the IoT because no special training, software, or hardware is needed to interact with their environment from commodity hardware. From the application development and administration point of view, only a single protocol has to be maintained. This boosts the development of consumer-friendly applications for the H2M interaction via a standardized

[2]Reserved name for client-to-server connections by Internet Assigned Numbers Authority (IANA) as described in RFC 6120 [22].

communication scheme because it allows programmers and administrators to use existing tools and handling expertise, while reducing integration cost and compatibility tests.

3.2.2 Essential Set of XEPs

It is not necessary to implement all XEPs and each XEP feature for Chatty Things. A reasonable set of XEPs is sufficient for supporting the H2M communication. The following XEPs are of particular interest for Chatty Things and have been adapted from the proposal of Hornsby et al.'s selection [26, Sec. III, Table 2]:

- ***XEP-0030 Service Discovery*:** Allows the discovery of extended information (e.g., identity, capabilities, advertised features, supported XEPs, joined chat rooms) about XMPP entities [132];
- ***XEP-0045 Multi-User Chat (MUC)*:** Defines a many-to-many chat for XMPP entities in which chat room names represent currently available topics [133];
- ***XEP-0050 Ad Hoc Commands*:** Allows users to initiate a command session and to interact with an automated process through an XMPP client [134];
- ***XEP-0060 Publish-Subscribe*:** Defines a generic publish-subscribe framework, which enables XMPP entities to broadcast extended information (i.e., publish) to all interested entities (i.e., subscribe) via pushing event notifications [135];
- ***XEP-0174 Serverless Messaging*:** Allows two XMPP entities to establish an XML stream without the need of an XMPP server (i.e., ad hoc communication) [25].

Compared to this selection of XEPs, Hornsby et al. [26] propose *XEP-0166 Jingle* instead of *XEP-0050 Ad Hoc Commands*. We have not chosen *XEP-0166 Jingle* because we do not see a need for signalizing multimedia sessions for Chatty Things, since Jingle is mainly used for the H2H interaction (e.g., voice or video chat, file transfer). *XEP-0050* instead allows the implementation of an easy-to-use H2M interaction and control in future IoT scenarios. Another important point is the availability of the preferred XEPs in current XMPP servers and clients to achieve a high acceptance rate of the XMPP stack with current XMPP-based systems and software solutions. The selection process of Hornsby et al. [26] did not consider this aspect. Therefore, we analyzed the supported XEPs of popular and widely used XMPP servers and clients (for desktop and mobile systems). The

results[3] are summarized in Table 3.2 and Table 3.3 and show that most XMPP servers implement the favored XEPs for IoT appliances.

Table 3.2: Supported XEPs of popular XMPP servers

XMPP Server	**XEP-0030**	**0045**	**0050**	**0060**	**0174**
Prosody IM [127]	Yes	Yes	Yes	Yes	Implementation on client side only
ejabberd [136]	Yes	Yes	Yes	Yes	
Openfire [137]	Yes	Yes	Yes	Yes	
Tigase [138]	Yes	Yes	Yes	Yes	

In contrast to XMPP servers, XMPP clients less support the selected XEPs. While the desktop XMPP clients (e.g., Pidgin, Psi) implement in general more features in form of XEPs, mobile and cross-platform XMPP clients (e.g., Xabber for Android[4], iChat/iMessage for OS X[5] and iOS[6], XMPPFramework for iOS, libpurple[7], and Smack API for desktop and Android operating systems) concentrate on supporting *XEP-0030 Service Discovery* and *XEP-0045 Multi-User Chat*. One exception are iChat/iMessage that use Wide-Area DNS Service Discovery (DNS-SD) [139] instead of *XEP-0030*. It is conspicuous that *XEP-0060 Publish-Subscribe* is not or only partially implemented by all analyzed XMPP clients. The reason is that its specification and implementation is quiet complex. Instead, XMPP clients implement *XEP-0163 Personal Eventing Protocol* that uses only a minor part of *XEP-0060*, while in all other cases *XEP-0060* seems to be avoided. Furthermore, it seems that *XEP-0050 Ad Hoc Commands* has still not become widespread in mobile XMPP clients. As *XEP-0050* is not very usable for H2H communication in mobile scenarios nowadays (i.e., it is a manual process by humans), we believe that it may become interesting for H2M interaction use cases in IoT, such as the remote controlling of Chatty Things. The issue with *XEP-0174 Serverless Messaging* is that it has a high implementation effort, since it relies on the third party protocols mDNS and DNS-SD (cp. Chapter 5). In addition, *XEP-0174* has been developed for an one-to-one ad hoc communication when no XMPP

[3]Implemented version and function set can differ for each XEP, since these information are not documented in detail in the corresponding data sheets.
[4]Linux-based operating system for mobile devices.
[5]Operating system for desktop devices developed by Apple.inc (e.g., MacBook, iMac, Mac Pro).
[6]Operating system for mobile devices developed by Apple.inc (e.g., iPhone, iPod).
[7]Feature-rich and open source XMPP library written in C which is maintained by the Pidgin team.

server is available or needed. In the Internet XMPP servers are usually reachable so that this type of communication is seldom needed.

Table 3.3: Supported XEPs of popular XMPP clients

XMPP Client	**XEP-0030**	**0045**	**0050**	**0060**	**0174**
Pidgin (libpurple) [140]	Yes	Yes	Yes	Partial	Yes
Psi [141]	Yes	Yes	Yes	No	No
XMPPFramework [142]	No	Yes	No	Partial	No
Xabber [143]	Yes	Yes	No	No	No
iChat/iMessage	No (DNS-SD)	Yes	No	Partial	Yes
Smack [144] API	Yes	Yes	No	Partial	Yes [145]

Our decision for a basic XEP set has been based on the following criteria: availability, complexity of the message structure to reduce bandwidth, and code footprint. Therefore, we selected only the following XEPs for our minimized XMPP stack: *XEP-0045 Multi-User Chat*, *XEP-0050 Ad Hoc Commands*, and *XEP-0174 Serverless Messaging*. The reason for skipping *XEP-0030* and *XEP-0060* is their heavy use of complex XML message structures and iq stanzas, which rely on the request-response mechanism (cp. Section 3.1.1) that we want to bypass for automated processes and event notification. The push notification is more suitable (cp. Section 2.4). The proposed sensor-specific grouping approach allows us to further reduce the number of XEPs by replacing *XEP-0060* through a slim *XEP-0045* implementation (cp. Section 3.3).

3.2.3 Addressed Use Case Scenarios

In the following we show how the use cases introduced in Section 2.4 can be implemented using Chatty Things. We assume all devices, ranging from smart objects to ordinary computational devices, run an XMPP stack to ensure that these devices can simply be accessed via an XMPP interface. Such a solution was also introduced by the open Home Automation Bus (openhab) project [96] during the work on this thesis. The XMPP interface is used to control things of a household via standard XMPP chat clients. In contrast to our vision, chat requests are mapped to the internally used Java-based Open Services Gateway

initiative (OSGi) runtime environment [146], which can be seen as a dedicated middleware connecting only smart objects (cp. discussion of the drawbacks of CoAP in Section 2.3.2). The two use cases have in common that our proposed XMPP-based system architecture is used to seamlessly integrate Chatty Things and other XMPP-driven devices into the Internet directly via enhanced (OpenWrt) routers.

Smart Home, Smart Office, and Smart Hotel Room. With Chatty Things, a point is reached where devices can easily be integrated by non-technical users in their homes. These devices detect automatically nearby devices and search the network for services they are interested in and then subscribe to them, and advertise their own services. Thus, smart homes can be set up and upgraded at moderate cost. Users will gain access to information of the environment they live in and get the possibility to interact with it. Moreover, devices from different vendors can seamlessly be coupled and integrated making the extension of new smart home services really simple without relying on a proprietary and costly solution from a dedicated vendor. Figure 3.6 depicts an example of the possible home automation use case. Rooms of a building can be separated by a small power outlet connected via an access point for IEEE 802.15.4, comparable to the enhanced router (e.g., OpenWrt or *Sheevaplug* [147]), to bypass multi-hop for Chatty Things and to monitor and (access) control a building with the help of a domain-based room management. Current solutions and ideas for smart offices or hotel rooms either focus on a smart room control system [148] or need to track the behavior of the guests [149]. The benefit of an XMPP-based smart hotel room is that no tracking of guest profiles is necessary because the preferences of the guest are stored on the user's mobile device. They are only shared with the hotel room if the guest agrees to it (protection of privacy). For the privacy of smart offices and smart hotel rooms, separate XMPP domains have to be used. This ensures that only Chatty Things of the given room (domain) can be accessed and controlled by the user. A smart hotel room should allow a hotel guest to push its preferences directly to all surrounding Chatty Things for the adjustment of the temperature, the favorite radio or tv show, and the dimming of the light during the user steps into the booked room. The check in and checkout process can also be simplified through coupling the booking number to the user's unique JID allowing remote access of the guest to push the preferences to the booked room before arriving or allowing an automatic notification to the reception and the cleaning staff when the guest leaves the room.

Flexible Post-Disaster Management. Combining sensored objects with cloud technology is a new trend [150], [151], which replaces a Wireless Sensor Network (WSN) by a hybrid system composed of powerful, but portable hand-held devices with the ability to integrate smart objects (through a one-hop communication). A main advantage of this approach is that time-critical sensor data can be processed instantaneously. Using a plain WSN for this has several drawbacks [152]: energy is a big concern for typical sensor nodes, so data need to be transported hop-by-hop to a network sink causing delays in real-time data streams. The processing capabilities of resource-constrained devices are also limited compared with more powerful platforms. In our approach the integration has to be performed through communication with Chatty Things over smartphones to avoid the limitation of sensing capabilities that is typical for energy- and resource-constrained smart object networks. A consequence of our design choice is a limitation of the system's life time compared with pure WSNs, but monitoring critical infrastructures and supporting post-disaster management is only a temporary challenge (e.g., days to weeks). Despite the shorter life time, we still gain the important benefit of flexibility and the ability to transfer and process larger data quantities in time-critical situations.

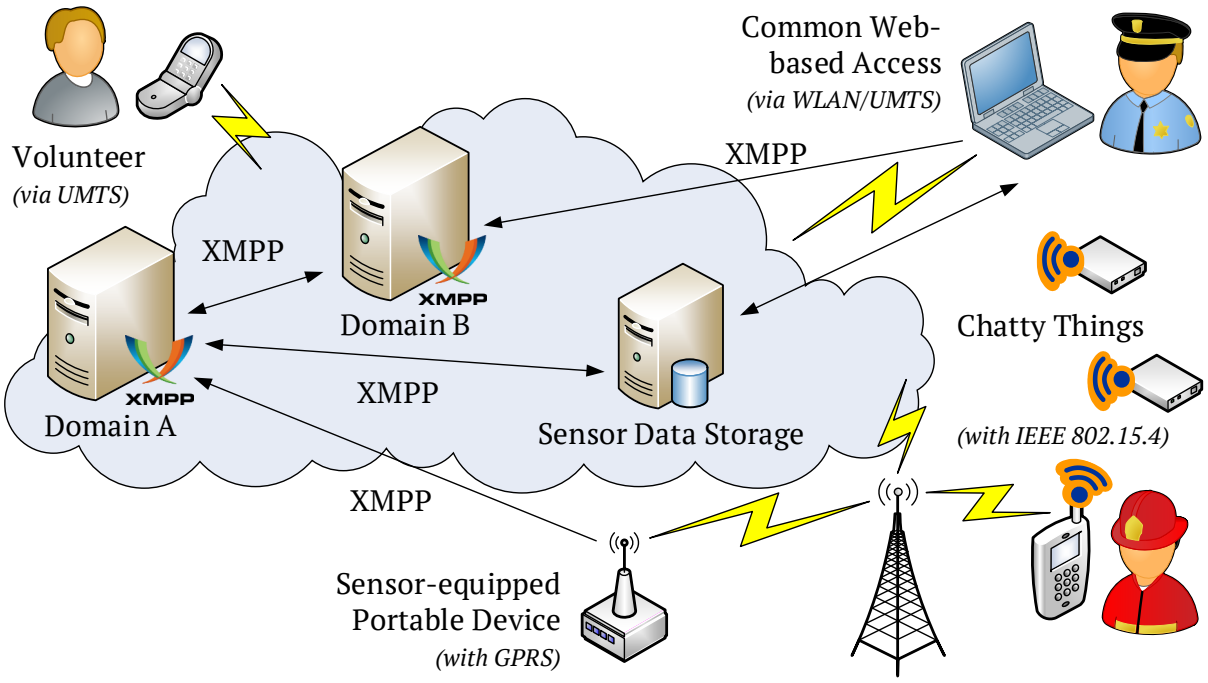

Figure 3.9: System reference model of the flexible post-disaster management

Our proposed system consists of sensor-equipped portable devices, Chatty Things (with short range communication), stakeholders with different (sensor-equipped) devices, and cloud services, as depicted in Figure 3.9. In our design hand-held devices and Chatty Things are connected either directly to the Internet or through a one-hop communication via XMPP over another hand-held device to deliver a real-time data stream with a high resolution directly to the cloud service. XMPP couples different devices and networks together to enable a cross-border communication and collaboration to support post-disaster management stakeholders, to collect data from Chatty Things over different IP-based communication technologies, and to share it with other devices or users. Each device runs an XMPP client through which it can access the XMPP network and publish sensed data. The cloud service manages a data storage for the sensed data and several XMPP domains which facilitate an interconnection and data exchange among different organizations (e.g., rescue teams, government). Data access for third parties / volunteers can be enabled by giving access to the sensor data storage. Querying data from Chatty Things at close range can be done through *XEP-0174*.

To sum up, our design choice fulfills the following system criteria: (1) Use of standard protocols to avoid gateways for linking different protocols and networks which offers the user a transparent access to the various devices, to different networks, and to the sensor data storage; (2) Temporary use of commodity hardware instead of dedicated wireless sensor network hardware; (3) Scalability, reliability, data exchange, expandability, seamless integration, and cost efficiency is realized through the use of XMPP; (4) Simple sensor data access by means of a standard XMPP chat client on the computational devices that ensures a slim and fast implementation for various operating systems on many devices ranging from desktop systems to hand-held devices (cp. [213, Sec. 4.2]).

3.3 Sensor-Specific Grouping Approach

Sensor-specific grouping is an important part of the Chatty Things approach that restricts the advertised services of a smart object to its integrated sensors. Thus, it directly maps the capabilities of a smart object to *XEP-0045 Multi-User Chat (MUC)* rooms to support a slim implementation for filtering and grouping sensor data. The topics of the chat rooms represent the sensor type. During bootstrapping Chatty Things automatically detect the integrated sensors, create/join the chat room with the sensor type as topic, i.e., a sensor-specific grouping, and push the gathered sensor data to the room (see Figure 3.10).

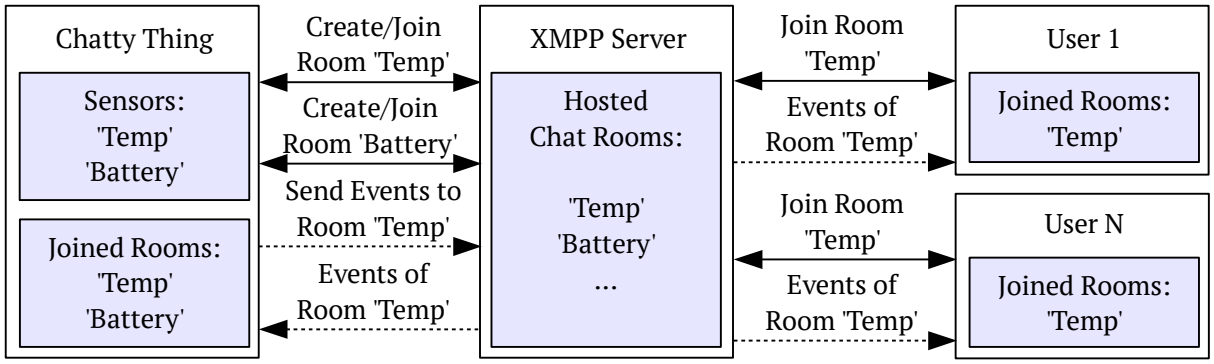

Figure 3.10: Sensor-specific grouping approach and event distribution of Chatty Things using *XEP-0045 Multi-User Chat* rooms

This concept is simple and allows us to reduce the number of XEPs needed for a minimized and modular XMPP stack [215, 218] because the implementation of *XEP-0060 Publish-Subscribe* is avoided for Chatty Things (see Table 3.4). *XEP-0045 Multi-User Chat* exhibits similarities with publish-subscribe: the chat room itself presents the topic, joining a chat room can be seen as 'subscription' to a topic, and the sending of a message to a chat room as 'publication' (cp. [153]). Additionally, *XEP-0045* also supports access control (e.g., roles, affiliations, privileges), presence broadcast, and it allows inter-domain communication.

Table 3.4: Feature comparison of *XEP-0045 Multi-User Chat (MUC)* and *XEP-0060 Publish-Subscribe*

Feature	**XEP-0045**	**XEP-0060**
Event Distribution	Yes	Yes
Discovery	Yes	Yes
Content Filter (Topics)	Yes	Yes
Access Control	Yes	Yes
Presence	Yes	No
Publish-Only	No	Yes

Furthermore, the use of chat rooms also provides a simple and efficient event distribution because many users can simultaneously be informed about an event with a single group chat message. The XMPP server automatically pushes this information to all other entities in the same chat room (see Figure 3.10). The advantage of *XEP-0045* in comparison to *XEP-0060* is that *XEP-0045* is well supported by the majority of all XMPP clients (cp.

Section 3.2.2), since it presents a core XMPP service and it was specified before *XEP-0060* (cp. [153]). A drawback of *XEP-0045* is its lack of a publish-only affiliation. This issue will be addressed later in Section 4.3.

Users can discover chat rooms by sending a service discovery 'disco#items' via *XEP-0030 Service Discovery* to an XMPP server which returns a list of available rooms. Discovering chat rooms of a foreign XMPP domain works the same way and is realized through inter-domain communication (see Figure 3.5 and Figure 3.11). We use these chat rooms as access control and content filter because only registered entities get updates about a topic they have joined. Moreover, redundant data are avoided which further reduces the XMPP message size in smart object networks, since meta data are only transmitted once. The first message after joining a chat room transmits the unit of the sensed data and additional meta data (e.g., geo-location). All other messages only transmit the sensed value. The management of the selected sensor-specific groups and the re/join is handled by the user's XMPP client. The latter, e.g., Pidgin [140], can remember the last used chat rooms and enter them automatically if the user reconnects to the same XMPP domain, so that the user will always be informed about events of nearby Chatty Things. Pidgin is available for a wide range of operating systems and can be used for free if no XMPP client is preinstalled.

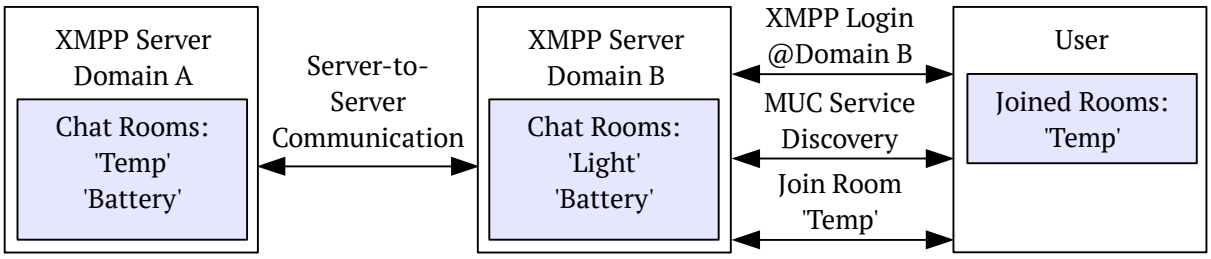

Figure 3.11: Discovering *XEP-0045 Multi-User Chat* rooms of a foreign XMPP domain through inter-domain communication

3.4 Summary

Chatty Things provide the application developers a transparent access to the Internet of Things. User and devices have not to be aware of the given network situation or the network infrastructure. This allows to create IoT-driven devices which can operate out-of-the-box without any user interaction for manual configuration or setup before their first

startup (e.g., Plug'n'Play). Chatty Things meet the service-oriented aspects of the technical challenges of the IoT (cp. Chapter 1). The following service characteristics are provided by Chatty Things at the application layer to enable the development of user-friendly IoT applications for the H2M interaction:

- **Interoperability and (Service) Discovery:** XMPP and mDNS/DNS-SD enable a standardized interaction between different devices and a standardized advertisement of their services via discovery, look up, and name services.
- **Identity Management:** Chatty Things are uniquely identifiable through JIDs and published information can be linked to them (as required by [5, Sec. 2]).
- **Expandability and Flexibility:** XMPP ensures a hardware-independent integration of Chatty Things into the Internet. Chatty Things can easily be extended with new protocol features and accessed through applications that support XMPP.
- **Self-configuration:** The support of different smart object technologies and the integration of a flexible bootstrapping process for any subsequent interaction between different devices in hybrid networks [6] (automatic environment detection) is implemented by means of mDNS/DNS-SD.
- **User Interaction:** Chatty Things provide inexperienced users an easy-to-use interface using standard XMPP client software for accessing sensor data and for setting thresholds.
- **Support for Hybrid Smart Object Networks:** The interaction with Chatty Things works independently of the given smart object network environment. When infrastructure services are failing or are unavailable during bootstrapping the interaction can be established through ad hoc communication.
- **Event Distribution and Information Filtering:** The sensor-specific grouping approach provides users a mechanism to address information or events target-oriented, since only changes in sensed data are transmitted to subscribed users.
- **Data Privacy:** Privacy can simply be achieved through the setup of private XMPP domains which grant access only to users with a registered JID on this XMPP server or remotely through confirmed subscriptions to JIDs of this domain when the corresponding XMPP server is interconnected to a public/foreign XMPP server.

Since smart objects usually have limited memory and computing resources, implementation of a modular, memory-efficient, and extensible XMPP stack which is optimized for low data rates and additionally supports *XEP-0174* and mDNS/DNS-SD have not been realized, yet. The upcoming Chapters 4 and 5 describe our solutions for solving the implementation challenges for the required protocols XMPP and mDNS/DNS-SD on smart objects running the Contiki OS.

4 uXMPP2: XMPP Stack for Smart Objects

Available XMPP libraries and clients have been developed for desktop or mobile systems. They are mostly written in the programming languages C/C++, Python, or Java. In contrast to desktop or mobile systems, smart objects usually have very limited memory and computing resources as well as constrained wireless links (e.g., 127 bytes maximum packet size for IEEE 802.15.4). Thus, these XMPP libraries and clients cannot be used for constrained devices of class 0 and 1 (cp. Section 2.2) because these implementations are too large for such hardware platforms. The first approach running an XMPP client on smart objects was uXMPP [100]. The initial version of which (v0.1) was released in 2009 for the Contiki [32] operating system as an early proof-of-concept. It supports IPv4 and implements only certain XMPP Core functions (e.g., login to a specified XMPP server, send/receive one-to-one chat messages, send presence messages). As the original code of uXMPP v0.1 was in an early developing state, no further controlling options, readily usable API, memory optimizations, or low data rate enhancements for smart objects were implemented. The provided XMPP Core functions are fixed and hard-coded. The work on uXMPP v0.1 has not been continued. We have taken up this approach for the implementation of Chatty Things. This chapter introduces a fundamentally improved, minimized, and modular XMPP stack for smart objects. We call this stack uXMPP2.

4.1 Architectural Solutions

For the implementation of uXMPP2, we chose a building blocks concept of replaceable XMPP features to ensure a predictable memory consumption and to support different use cases. In the Contiki OS a process is implemented either as an application program or as a service. The latter provides functions that can be used by application programs (cp. Section 2.2.1). uXMPP v0.1 was implemented as a single application program. The drawback of this solution is that uXMPP v0.1 works only stand-alone. It provides no API for other processes and no options to dynamically load and unload parts of the

XMPP stack (i.e., application program and XMPP stack cannot be separated from each other). To overcome this problem each component (e.g., XMPP client, XEP-0174 client) of the uXMPP2 stack is implemented as a service of the Contiki OS to enable XMPP API calls for other processes. It allows a simple interconnection between processes and it decouples the application program from the XMPP stack, as depicted in Figure 4.1. Furthermore, the implementation as a service allows the developer to dynamically load and unload XMPP components which do not fit together in the memory at the same time. This feature is very useful if a component is only needed during bootstrapping and others afterwards. uXMPP2 supports IPv6 to manage and connect up to billions of smart objects via XMPP over uIP.

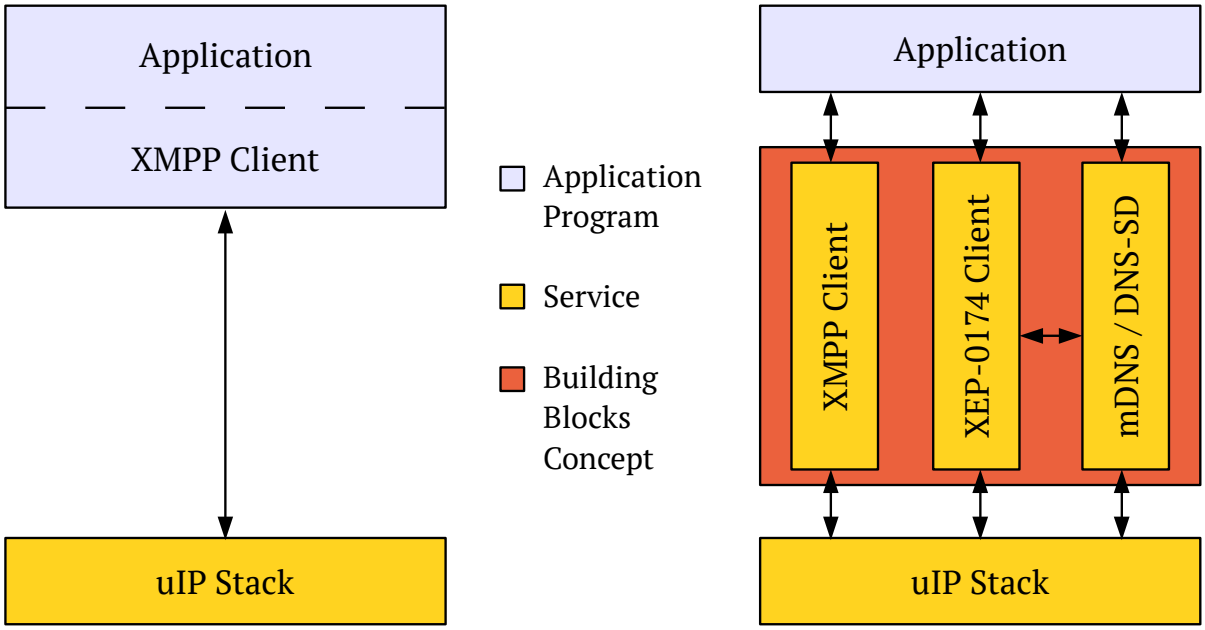

Figure 4.1: uXMPP v0.1 versus uXMPP2

The *XMPP client component* implements the Core/IM module, which is an integral part of uXMPP2 as basis. Additionally required XEPs can be implemented on-top on demand, such as *XEP-0045 Multi-User Chat (MUC)*. uXMPP v0.1 uses a simple, but memory-efficient string comparison for message handling. We added an optional XML parser for enhanced message handling to support more authentication mechanisms. The two implementations are compared in Section 4.2. As login method, *ANONYMOUS* [154] is favored because connecting entities get a randomized and unique JID from the server. This avoids hard-coded or double-assigned JIDs for Chatty Things. They also have not to be pre-configured on an XMPP server. The disadvantage is that an alias for a JID has

a very long length in a chat room. To shorten the displayed name of a Chatty Thing in a multi-user chat and to reduce the size of a group message, we propose *ANONYMOUS JID cutting for MUC*. It uses only a dedicated length of a JID as an alias for a chat room. Version 0.1 of uXMPP sends each XML element of a specified XMPP message in a separate and independent TCP/IP packet which strongly increases the number of sent packets. We avoid this by using the full available TCP/IP payload length (see Table 4.5) and through the reduction of the number of exchanged messages as well as the frequency of the information exchange, as explained in Section 4.3.

The *XEP-0174 client component* implements *XEP-0174 Serverless Messaging*. Since *XEP-0174* requires also a tiny implementation of mDNS/DNS-SD on Contiki-based smart objects, which was still not available, an *XEP-0174* client for Contiki has not been implemented so far. We developed our own tiny *XEP-0174*-based communication stack working with Contiki's integrated IP stack. It consists of our tiny mDNS/DNS-SD implementation, called *uBonjour* (cp. Chapter 5), and an *XEP-0174* client. Compared to the XMPP Core/IM client, it is necessary to implement a TCP listener process for the *XEP-0174* client because users initiate a direct XML stream with a Chatty Thing. The TCP handler manages incoming TCP connection requests via a connection state and accepts the opening of XML streams from other XMPP entities in the network.

Readily Usable API

uXMPP2 supports the application developer in several ways: API to gain simple access to all implemented XMPP functions and to easily implement new features through XEPs. The API provides a basis set of XMPP Core/IM and XEPs for IoT applications (cp. Section 3.2.2) and allows the developer to concentrate on the real problem instead of struggling with constrained resources and new programming paradigms for smart objects. The architecture of uXMPP2 is designed in a way that developers are able to simply implement further XEPs without a deeper knowledge of the Contiki OS because the expandability through XEPs is an important benefit of XMPP. As the focus of this thesis lies on the H2M interaction, only H2M related parts of XMPP Core/IM and corresponding XEPs are implemented in uXMPP2 to ensure a slim code footprint and a low memory usage (see Appendix A.1). All other features can easily be integrated by the application developer. The following Listing 4.1 shows an example implementation of an XMPP client using the uXMPP2 API in line(s): 7 – establish a connection to an XMPP server; 13, 17,

and 21 – receiving connection related events; 15 – joining a chat room; 19 – sending a group chat message. The example skips files for the header and the includes.

```
1  /* ... */
2  uip_ipaddr_t ipaddr;
3
4  PROCESS_THREAD(example_process, ev, data)
5  {
6      PROCESS_BEGIN();
7      xmpp_connect(&ipaddr,"'example.net'","'romeo@example.net'");
8
9      while(1)
10     {
11         PROCESS_WAIT_EVENT();
12
13         if(ev == xmpp_auth_done)
14         {
15             xmpp_join_muc("'room@chat.example.net/romeo'");
16         }
17         else if(ev == xmpp_joinmuc_done)
18         {
19             xmpp_send_muc_msg("Hello Capulets", "'room@chat.example.net'");
20         }
21         else if(ev == xmpp_msg_received_or_send)
22         {
23             if (data != NULL) printf(">Received message: %s\n",(char *) data);
24         }
25         /* ... */
26     }
27     PROCESS_END();
28 }
```

Listing 4.1: uXMPP2 API example

Control and Management: XEP-0050 Inspired Remote Commands

The exchange of control and management commands with Chatty Things is implemented through XMPP *message* stanzas with which threshold values for the sensors can be adjusted, sensor updates can be received, and the battery state can be monitored. The concept was inspired by *XEP-0050 Ad Hoc Commands* that allows users to initiate a command session and to interact with an automated process through an ordinary XMPP client. Since *XEP-0050* is not well supported by current XMPP clients, because it depends on *XEP-0030 Service Discovery* and uses *iq* stanzas, we have not integrated the remote commands as a dedicated module into uXMPP2. Instead, developers can implement their own application-specific commands to offer a simple way for users to adjust thresholds (e.g., for alarming), to view sensed data (e.g., exceeded thresholds, maximum reached

values), or to access recently sensed data. The IoT application announces a command list when an XML stream is opened (*XEP-0174 Serverless Messaging*) or when an XMPP chat message (XMPP IM) with empty or predefined content arrives. Moreover, the use of message stanzas combined with *XEP-0045 Multi-User Chat* allows us to facilitate the control and management of Chatty Things by grouping related devices in a chat room.

Lightweight XML Parsers

The XML parser is responsible for the en-/decoding of XML streams. uXMPP v0.1 does not use an XML parser. Instead, it uses a simple string comparison which checks hard-coded strings against parts of an XML stream, as depicted in the Listing 4.2, to control sequences of the expected XMPP data flow implemented as a state machine. This has the advantage that the implementation is kept simple because it consists of a few lines of code, is fast and memory-efficient. On the other hand, the string comparison is a static filtering approach and not really flexible because only predefined strings are detected. Minimal modifications are not recognized. XMPP servers and clients can differ in the use of the character for delimiters to indicate the start and the end of literals (e.g., the values of attributes) and upper- and lowercase spelling[1] (e.g., the names of attributes) in XML. This breaks a simple string comparison. For instance, simple or double quotes are used by different XMPP implementations as delimiters[2] in XMPP messages. A string comparison drops the whole string if it only looks for a string containing double quotes instead of a string containing single quotes and vice versa.

```
1 if(strncmp(c,"<?xml version='1.0'?>",21) != 0)
2 {
3     printf("xml version not received or wrong");
4 }
```

Listing 4.2: Example of a simple string comparison in C language

In order to overcome this issue we implemented a lightweight XML parser in the C programming language that works more resistant against these failures. We found two promising projects as a starting point for an XML parser integration: *SimpleXML* [155] and *iksemel* [156]. *SimpleXML* is a simple tree-based XML language parser of valid XML 1.0 documents which is small and fast to be included directly into C/C++ applications. When precompiling the header files with msp430-gcc, the SimpleXML takes 15.81 Kbytes

[1] **FROM**='romeo@example.net' versus **from**='romeo@example.net'.
[2] from='romeo@example.net' versus from="romeo@example.net".

of ROM / 1.06 Kbytes of RAM, but this is without defining the needed XMPP messages for the XML parser that requires additional memory. For the integration of SimpleXML in uXMPP2, its code needs to be heavily refactored, as explained in [77, 5]. *iksemel* is another XML parser library designed for XMPP applications. It is highly portable (as suitable for embedded systems). The implementation is small and modular, it provides a *Simple API for XML (SAX)*, *Document Object Model (DOM)* and XMPP parsers, and Simple Authentication and Security Layer (SASL) support. Our memory measurements for the precompilation of its headers show that 12.01 Kbytes of ROM without defining XML messages and 18.23 Kbytes of ROM including XMPP message definitions for the parser are used. The compilation on the MSP430 of the two XML parser implementations with direct access to their provided library functions was not possible because the two libraries exceeded the available memory of the Zolertia Z1. To sum up, these lightweight XML parser implementations cannot run on a smart object without deeper modifications for the XMPP message parsing due to their high memory requirements.

Therefore, we decided to use a modified variant of the simple string comparison as default for uXMPP2 because an efficient memory usage is more important than a flexible, but high memory consuming XML parser. We optimized the parsing of XMPP messages in a way that only characteristic keywords have to match rather than comparing static and predefined strings. This is similar to the manner regular expressions work and very memory-efficient. As an alternative option, we have implemented an XML parser based on the code of iksemel parser library to achieve more robustness and to support more SASL mechanisms. For the integration of iksemel in uXMPP2, its code was refactored and only necessary features were kept, such as the SAX interface, XML parser core functions, the XMPP message parser, SASL, and DIGEST-MD5. The memory size for uXMPP2 with the optimized version of iksemel is 30 Kbytes of ROM, i.e., seven times the size of the uXMPP2 Core/IM using string comparison and no DIGEST-MD5 support for SASL.

On the Use of UDP as Transport Protocol for the Exchange of XML Stanzas

Most developed application protocols for IoT use UDP as transport protocol (e.g., CoAP), but these approaches extend UDP with proprietary reliability and sequence number support to adopt desired characteristics of TCP (cp. Section 2.3.2). The use of (a proprietary) UDP contradicts our main idea of using already established and standardized application protocols to seamlessly integrate smart objects in IP-based networks. This approach is therefore not recommended for uXMPP2. We explicitly want to avoid extensive adaptations

of XMPP clients, libraries, and servers because this would break the compatibility with the current XMPP infrastructure [157]. Therefore, approaches like the TCP Support for Sensor nodes (TSS) [158], Distributed TCP Caching (DTC) [159], and cache and control (cctrl) [160] make TCP applicable to work in low data rate networks. They enable a more energy-efficient use and increase the data throughput of TCP in comparison with UDP.

4.2 XML Compression for the Use on Smart Objects

As the flexibility and the extensibility of XMPP is based on XML, the intensive use of XML in low data rate networks can cause a high network traffic load due to a large message overhead. This overhead can either be reduced through (1) XML compression techniques or through (2) the reduction of the number of exchanged messages and the frequency of the information exchange. An analysis for solutions of (1) is discussed in detail in this section. We analyze the memory usage, the compression ratio, and the implementation effort of current XML compression techniques if they are feasible for the use in uXMPP2. An approach for (2) will be presented in Section 4.3.

4.2.1 XMPP Stream Compression with ZLIB Algorithm

The *XEP-0138 Stream Compression* [161] provides a modular framework that can offer several companding algorithms for negotiating compression modes of XML streams. So the size of the exchanged XMPP messages in the network can be reduced using one of the existing standards available for several XMPP clients, servers, and libraries. In the case of uXMPP2, the number of sent and received IP packets should be minimized if the compression ratio of an algorithm provides good results. For *XEP-0138*, it is mandatory to implement the ZLIB [162] compression algorithm. The following two-step ZLIB checkup gives an overview whether the compression method is usable on smart objects. The first step analyzes the compression ratio[3] (cp. [164, Sec. 3]) of ZLIB for the use in combination with XMPP. The results of our measurement with typical XMPP message types in Table 4.1 and in Table 4.3 show that ZLIB has a best case compression ratio of 80%. Consequently, the space savings of compressing XMPP messages with ZLIB are only 20% (group chat message), whereas it reduces the number of sent IP packets by only one for each XMPP message type (except for presence) compared to the numbers presented in Table 4.5.

[3] Compression ratio $= \left(\frac{\text{compressed size}}{\text{uncompressed size}}\right)$; lower value means better performance [163].

Table 4.1: Sizes of typical XMPP messages compressed with ZLIB (in bytes) and their corresponding number of used TCP/IP packets (theoretically)

Message Type	**Normal Size**	**Compressed Size**	**TCP/IP Packets**
Presence	38 max	35 max (92%)	1
One-to-One Chat	164	140 (85%)	3
Chat Join	101	90 (89%)	2
Group Chat	164	130 (79%)	3

In the second step, we evaluated three C-based libraries to prove whether it is possible to run a ZLIB implementation on Contiki-based smart objects (compiled with msp430-gcc for the Zolertia Z1) with a low memory footprint. We chose the tiny, portable and C-based libraries *easy zlib* [165], *zlib.net* [166], and *miniz* [167] implementing the ZLIB compressed data format specification [162] for this analysis. In particular, *miniz* is written in a single source file providing a high compression speed and an easy integration in current projects. The *easy zlib* library allows an in-memory XML de/compression and the library *zlib.net* provides a typical compression ratio between 2:1 and 5:1. The results of the code size of the three ZLIB libraries are contained in Table 4.2. None of the evaluated libraries fits in the memory of the Zolertia Z1 without any complex modifications or additional overhead because the used internal structure for the de/compression of each library is too large and has to be heavily modified to run properly on Contiki. Moreover, these optimized implementations of the ZLIB standard show that the code is too complex for smart objects and that the code size cannot be shrinked further. In addition, *XEP-0138* requires the ZLIB standard, as it ensures the backward compatibility with existing XMPP libraries and clients for other platforms and their chosen ZLIB implementation.

Table 4.2: Code sizes of available ZLIB libraries (in Kbytes)

ZLIB Library	**Code Size**	**Fits in Z1?**
easy zlib	340	No
zlib.net	114	No
miniz	217 (+33)	No

As the ZLIB compression ratio and the space savings for XML messages are very low compared to the needed complexity of the code and no further results for memory usage

and CPU load on a smart object can be gathered, we decided to look for alternative XML stream compression methods. All in all, the ZLIB-based compression algorithm of *XEP-0138* has not proven to be an advantage for the use in uXMPP2.

4.2.2 XMPP Stream Compression with EXI

The *Efficient XML Interchange (EXI)* format [168] is currently suggested as an alternative XML compression method to ZLIB by *XEP-0322* [169]. The EXI specification defines an encoding format that allows an efficient interchange of XML and an effective processor implementation. It should provide a small code footprint for implementations, while being completely compatible with XML [170]. A study comparing different XML encodings (e.g., Binary XML (BXML) [171], Fast Infoset (FI) [172]) of binary XML parsers for embedded systems shows that the most compact representation is achieved using EXI [164, Sec. 3]. The comparison analyzes the encoding efficiency of typical XML contents (e.g., RDF/XML sensor data, Smart Energy (SE) and SensorML [173] example) and the complexity for running each encoding technique on resource-constrained devices using the reference implementation *EXIficient v0.3* [174]. The results (i.e., data in bytes, compression ratio in percent) of the XML encodings comparison in conjunction with the ZLIB compression method are presented in Table 4.3.

Table 4.3: Comparison of efficient XML encodings (adapted from [164, Sec. 3]) extended with our ZLIB compression results

Encoding	**Complexity**	**RDF Test**	**SE Test**	**SensorML Test**
XML	Medium	206	409	300
EXI	Low	6 (3%)	13 (3%)	57 (19%)
BXML	Medium	177 (86%)	210 (51%)	177 (59%)
FI	Medium	143 (69%)	200 (49%)	185 (62%)
ZLIB	High	177 (86%)	210 (51%)	176 (59%)

They clearly show that EXI delivers the best compression ratio of all encodings. Moreover, EXI can encode an XML content in such a manner that the encoding would fit into a single IP packet of uIP (cp. Section 2.2.2). The other approaches of binary encoding and compression methods offer only low compression rates compared to their needed complexity (e.g., medium and high) for the parser implementation and use. This may

be the reason why XEPs (e.g., *XEP-0239*) proclaiming binary XML for XMPP streams were rejected by the XMPP Standards Foundation (XSF) so far. The shortage of the EXIficient parser is that its implementation is written in Java. It takes around 1.9 Mbytes for the Java archive (jar), which includes the parser and all external libraries. Another EXI implementation is the Embeddable EXI implementation in C (EXIP) [175] which promotes a small code and low memory size through using limited external dependencies, but it currently supports only the default EXI encoding and decoding options (so it is not feature complete and the used code and memory size may increase with the next releases). The testing platform for the evaluation of EXIP is presented in [170]. It has been a *Mulle* hardware device (10 MHz Renesas M16C/65 with 47 Kbytes of RAM and 512 Kbytes of ROM) [176]. With Contiki-based smart objects, we are unfortunately far away from these hardware specifications (usually only a fifth of it). Therefore, running these XML encoding implementations on smart objects is not suitable.

The project WS4D-uEXI [177] has the aim to provide a solution for the described problem. WS4D-uEXI is working on a proof-of-concept implementation of EXI for smart objects. It supports all EXI features that are necessary to realize scenarios of all Devices Profile for Web Services (DPWS) stacks of the Web Services for Devices (WS4D) initiative. uEXI is implemented in plain C and supports only a reduced set of encoding modes of the EXI specification because evaluations showed that not all encoding modes are feasible for 6LoWPAN [178]. The parser of uEXI uses only around 1 Kbyte of ROM compiled for the MSP430 microcontroller, but it can only parse incoming messages. It provides functions for handling strings, qualified names, integers, unsigned integers, booleans, dates, and EXI event codes. The needed RAM depends on the parsed message structure. The processing of outgoing messages is realized through the generation of predefined messages and message formats (schemas) using EXIficient during compile time, which are then statically included as EXI grammar for the parser in the application. This causes additional ROM usage and strongly depends on the included schema files. The DPWS stack additionally uses around 3.2 Kbytes of ROM for SOAP, XML, XML schema, and XML namespace.

Ongoing work on the introduced EXI projects has shown that a low memory footprint EXI parser is not easy to implement. The research about EXI on smart objects is not completed, yet, and cannot deliver a generic solution (cp. [179]). Furthermore, *XEP-0322 Efficient XML Interchange (EXI) Format* is still experimental and there is still no official XMPP implementation. Stream compression through EXI is not compliant to available XMPP servers and clients. EXI seems an interesting way to reduce the XML overhead, but EXI implementations for smart objects need to be very slim and memory-efficient.

Instead, approaches are preferred that reduce the number and the frequency of exchanged messages. A solution for this without using any XML compression technique and providing a low memory footprint is presented in the next section.

4.3 Temporary Subscription for Presence (TSP)

The aspect of a many-to-many chat (*XEP-0045*) that is used in our proposed sensor-specific grouping approach (cp. Section 3.3) has several disadvantages for low data rate smart object networks. Group chat messages are broadcasted to every room member. This can cause a high network traffic. Entering a chat room implies that all registered entities can only be seen with their aliases instead with their JIDs. Further (chat) presence information from all room members is automatically available with manual subscription through presence broadcast [133, Sec. 7.2.3] (also known as "presence leak") because the alias of each room member is used to preserve the privacy of each XMPP entity. Unfortunately, this kind of presence announcement has a larger message size (around the size of a chat join, see Table 4.5) than a normal presence message. The reason is that a (chat) presence message includes additional attributes to get handled and forwarded to the chat room by the XMPP server because the JID of each room member has to be replaced by its alias. To solve this problem we propose the *Temporary Subscription for Presence (TSP)* that provides a publish-only affiliation for *XEP-0045* and uses small presence messages to reduce the network traffic of Chatty Things. Since presence notification is an integral part of XMPP Core/IM, it is supported by all XMPP clients. Access control and content filter are provided by *XEP-0045 Multi-User Chat*. The advantages of *TSP* in comparison to other XMPP approaches are summarized in Table 4.4.

Table 4.4: *TSP* in comparison with *XEP-0045*, *XEP-0060*, and presence notification

Feature	**XEP-0045**	**XEP-0060**	**Presence**	**TSP**
Content Filter	Yes	Yes	No	Yes
Access Control	Yes	Yes	Yes	Yes
Message Size	Medium	High	Low	Low
Publish-Only	No	Yes	No	Yes

4.3.1 Lightweight and User-Friendly Event Notification

The *TSP* approach is based on the fact that *presence* messages (38 bytes maximum) fit into a single TCP/IP packet (48 bytes maximum[4]) within IEEE 802.15.4 radio frames (127 bytes maximum). The reason is that presence information (not to confuse with directed presence [124, Sec. 4.6] or presence for entering a chat) is sent from a client without a 'from' or 'to' attribute. Table 4.5 shows the sizes of typical XMPP messages (e.g., presence [124, Sec. 4.4.1], one-to-one chat session [124, Sec. 5.2.1], presence to join a chat room [133, Example 18], group chat message [133, Example 44]), and the number of used IP packets in Contiki for both uXMPP versions.

Table 4.5: Size of typical XMPP messages and sent IP packets as well as the corresponding push of information stage

Message Type	**Size**	**uXMPP v0.1**	**uXMPP2**	**Stage**
Presence	38 bytes max	5 packets	1 packet	I
One-to-One Chat	164 bytes	12 packets	4 packets	II
Chat Join	101 bytes	-	3 packets	-
Group Chat	164 bytes	-	4 packets	II

From the user point of view, there should be a notification when thresholds are exceeded to indicate the need of adjustment or maintenance, instead of continually receiving data from objects as long as everything proceeds normally. Detailed information should only be requested when an intervention by the user is required. Thus, the frequency of information exchanges can be kept low when only status information (e.g., threshold reached) is transmitted. Furthermore, the coordinated subscription to interested/selected events and the avoidance of sending redundant data (i.e., meta data) allow further reductions of the number of exchanged messages. In order to reduce the bandwidth utilization we introduce with *TSP* two stages in *XEP-0045* (cp. Table 4.5) for pushing information:

- **Stage I:** The presence notification is activated through temporary subscriptions to topic-related Chatty Things. It is used to indicate data changes via traffic lights to interested users and objects. In this stage XMPP message types are used that fit into a single IEEE 802.15.4 frame, i.e., stage I uses only presence messages. During bootstrapping a chat join message is used additionally (cp. Section 4.3.3);

[4]Measured for IPv6 packets.

- **Stage II:** Detailed information about the event can be requested via remote commands by users and objects (cp. Section 4.1). This stage applies XMPP message types that require more than one TCP/IP packet for sending XML data.

The separation in two stages has the advantage that small presence messages for the event notification as in stage I and all *XEP-0045*-compliant XMPP chat clients can be used without any modifications because *TSP* has to be implemented only by the Chatty Things (CT) and by the XMPP server (see below). Thus, XML compression techniques, expensive in terms of memory usage and local computation (cp. Section 4.2), are not necessary to implement a low bandwidth usage for the XMPP message exchange. Figure 4.2 depicts the message flow in detail.

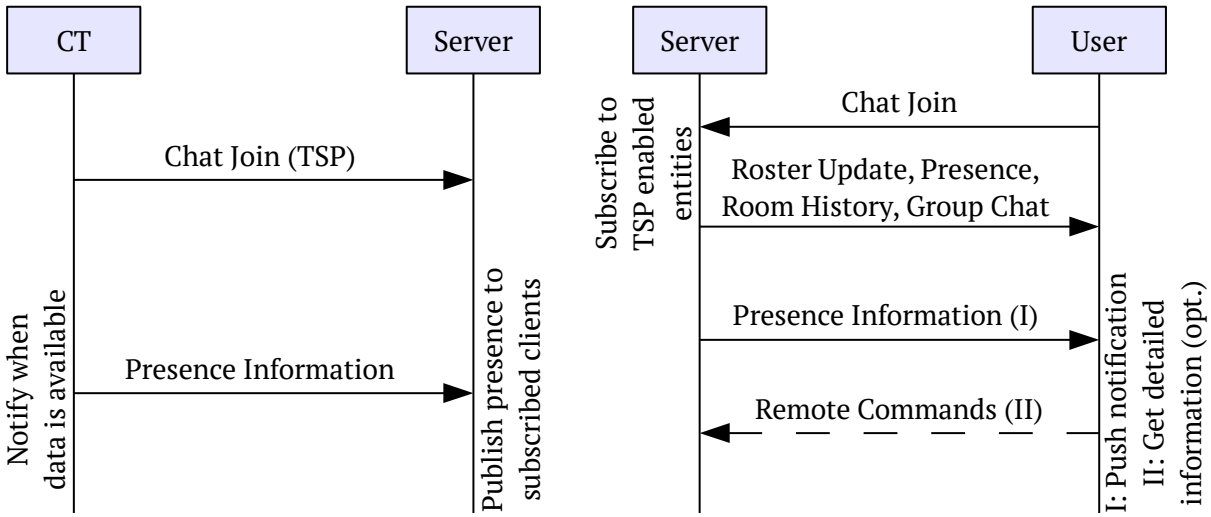

Figure 4.2: *TSP* message flow for Chatty Things (CT) and users

Presence Subscription Dilemma

The dilemma for users that receive small stage I presence messages is that they have to manually subscribe to each sending entity for each joined network. Presence information is private. Therefore, each subscription request to an XMPP entity has to be approved by the requested XMPP entity (see Figure 4.3). As the entities in a network can change (e.g., Chatty Things leave, join) and a user can get in touch with different network environments (e.g., just by walking along), the user's subscriptions to Chatty Things are neither fixed nor stable. The number of publishing Chatty Things can increase considerably. The roster of the user's XMPP client can thus become outdated very fast, as depicted in Figure 4.4.

The XMPP roster depicts each Chatty Thing to which a user has subscribed to as roster item with the name (JID) and the presence information as a status icon. Usually a user is especially interested in topic-related information which (a set of) Chatty Things can provide from the given environment and not directly in a dedicated Chatty Thing.

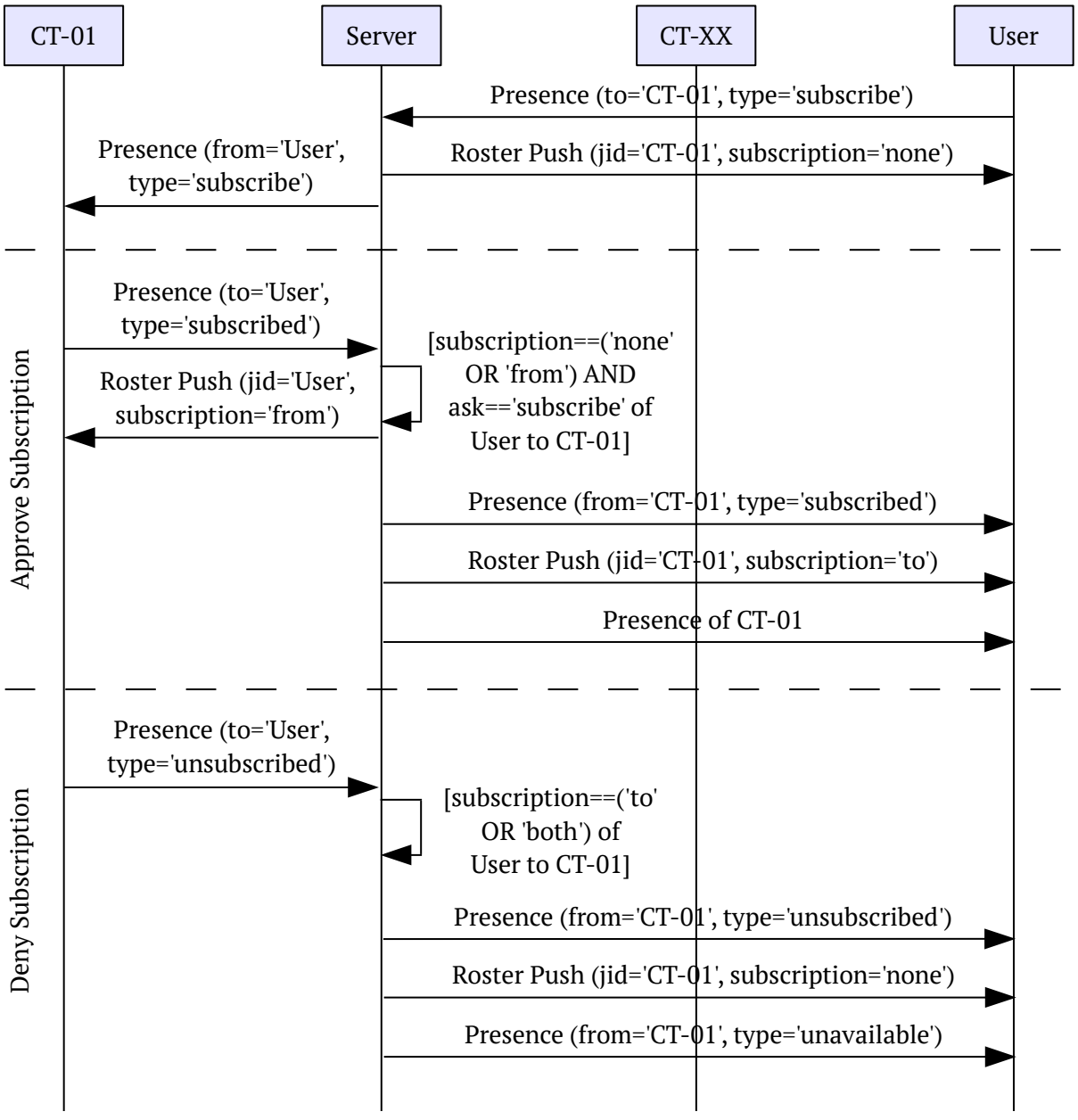

Figure 4.3: XMPP IM presence subscription request

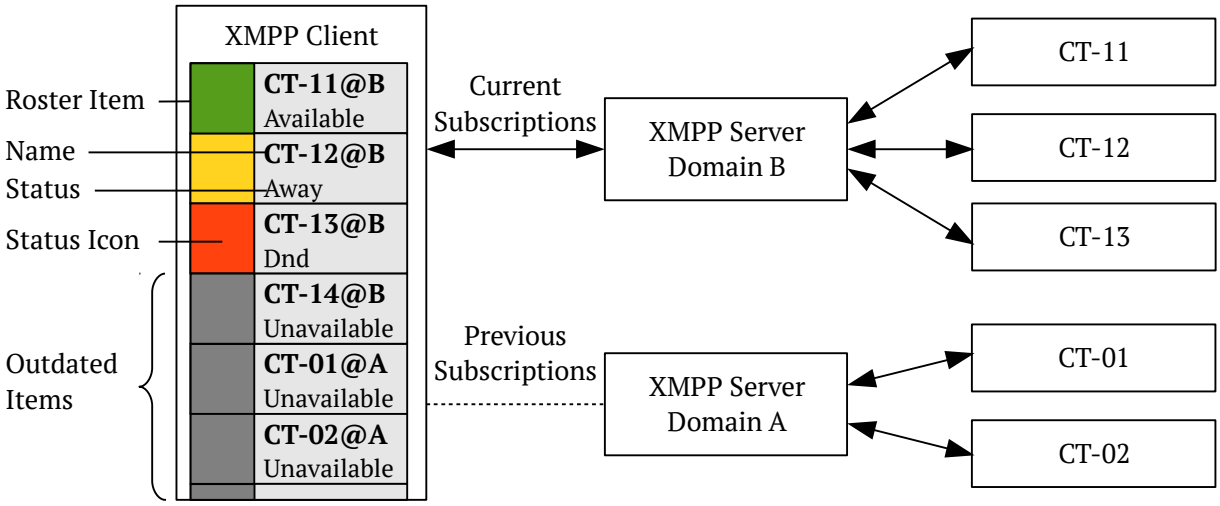

Figure 4.4: XMPP client with roster and outdated roster items

Therefore, an automatic and approved subscription of a user to a topic-related group of Chatty Things (i.e., information filtering) should be provided. For this, we propose a dynamic and up-to-date roster that holds Chatty Things of the current network (e.g., XMPP domain B) and of the user's selected sensor-specific groups only temporary to access their private presence information.

Dynamic Roster and Traffic Lights

TSP enables Chatty Things to approve the temporary subscription to their private presence information by all members of a topic-related chat room (cp. *approve subscription* in Figure 4.3 with *bootstrap* phase in Figure 4.7). This enables the automatic subscription to Chatty Things which joined the same sensor-specific groups of a user and adds them with their JID to the user's roster, as shown in Figure 4.5. If the user leaves the network the temporary Chatty Things is removed automatically from its roster to keep it clean and up-to-date (i.e., temporary subscription, cp. *exit* in Figure 4.7). Thus, the roster enables a sorted and up-to-date view on all interested events of a user. This avoids the use of several IP packets for sending a chat presence or group chat message that do not fit into a single IP packet and use only an alias. With *TSP*, we overcome the issues of the manual grouping of a set of Chatty Things by means of aliases and preserve the *XEP-0045*'s way of easy-to-use and familiar user interaction (e.g., access control and content filter). Users and Chatty Things without enabled *TSP* still act as ordinary chat room members (e.g.,

sending and receiving many-to-many chat messages) as defined in *XEP-0045* with the enhancement of filtering presence information by topics.

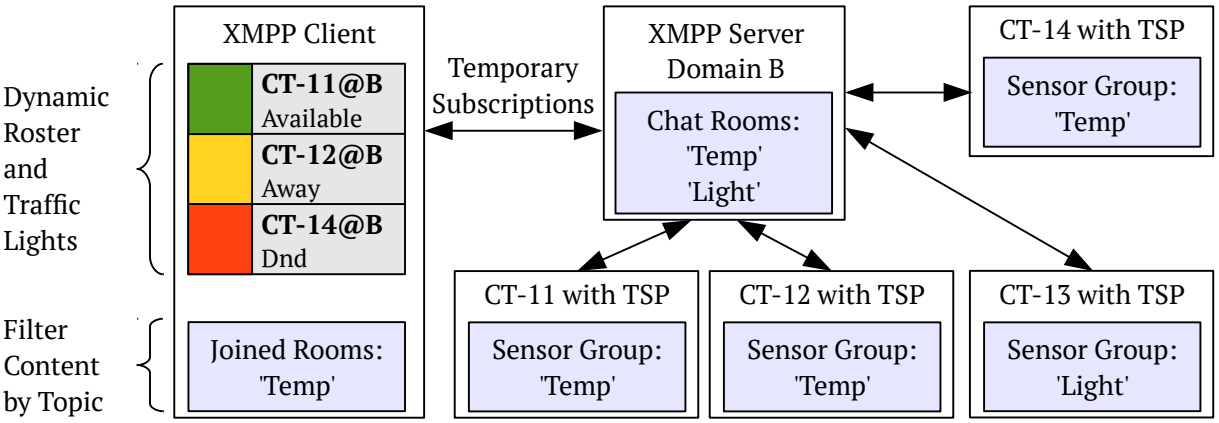

Figure 4.5: XMPP client with the dynamic roster and temporary subscriptions

For Chatty Things, the importance of an event is mapped to the predefined presence status types[5] of the XMPP Core/IM (e.g., 'available', 'away', 'dnd', 'unavailable'). So the status icon in the XMPP roster can be used to represent the condition of a Chatty Thing's sensed data. Presence information is sent to the XMPP network only when a threshold value of a sensor is exceeded. Thus, the status icons can be depicted like traffic lights[6] to simply indicate changes of a monitored environment directly to the user:

- **Green:** The sensed data is below the threshold value (i.e., everything proceeds normally), no presence update is necessary. The presence status type is 'available';
- **Yellow:** The sensed data exceeds the threshold value, a presence message informs all registered users. The presence status type is set to 'away';
- **Red:** The sensed data repeatedly exceeds the threshold value, a presence or a detailed chat message with the actual sensor value(s) is announced to all interested users. The presence status type is set to 'dnd';
- **Grey:** No presence information is reported. The Chatty Thing is either not available in the network or it is broken. The presence status type is 'unavailable'.

TSP allows users to retrieve updates from their local environment (temporary) or events of specific sensors they have subscribed to (XMPP roster) without getting overwhelmed

[5]The used data for the XML element is not meant for presentation to a human user [124, 4.7.2.1].
[6]Most XMPP clients, such as Pidgin, implement this color mapping for presentation of the predefined presence status (types) to a human user.

with information they are not interested in. Stage I dynamically reduces the exchange of presence information to a minimum if everything proceeds normally (e.g., green traffic lights). Detailed information can be requested via remote commands in stage II when an intervention by the user is required (e.g., yellow or red traffic lights).

4.3.2 Publish-Only Affiliation for Sensor-Specific Groups

The behavior of the *XEP-0045 Multi-User Chat* may cause bottlenecks in the message flow of a low data rate network because such a network may consist of a large number of devices each using a part of the limited bandwidth. An XMPP node always takes the role of a subscriber when publishing data. Hence, it gets information about every update just like a normal subscriber because XMPP (chat) presence and chat messages are broadcasted to each room member (see Figure 4.6). This causes a high number of exchanged messages for two reasons. Each message has to be delivered separately and the message does not fit into a single IP packet. This again produces a high number of messages as already mentioned earlier in this section. The reason is that *XEP-0045* misses a publish-only affiliation (i.e., dedicated role of a publisher, cp. Table 4.5). This feature would allow Chatty Things to act only as publisher that monitors the environment and pushes the sensed data in a sensor-specific group without receiving data updates from this group they are not interested in (i.e., less subscriptions and less data traffic).

In order to further reduce the network traffic *TSP* defines a publish-only affiliation, i.e., uninterested entities do not need to be informed about value changes of other sensors. They just collect data and publish them to the subscribers. Chatty Things with enabled *TSP* act only as publishers without getting updates from the XMPP network. So Chatty Things will receive no group chat or presence messages from the XMPP server, as depicted in Figure 4.7. The disadvantage is that the configuration of Chatty Things becomes more complicated. Sending one-to-one chat messages to Chatty Things still works for the user interaction via remote commands. With *TSP*, a definition of interaction roles for the different device classes of the IoT can be enabled. In networks with different classes of devices, each device has a different role and varying duties which have to be taken into account to minimize the network traffic, i.e., there are different priorities and time intervals for receiving updates of sensor data in a network. The results of [180] for the use of so-called *transmit-only nodes* (i.e., a comparable idea to *TSP* enabled publish-only nodes, but implemented at the MAC layer) have proven that the efficiency (e.g., cost and energy) and reliability of smart object networks are improved considerably.

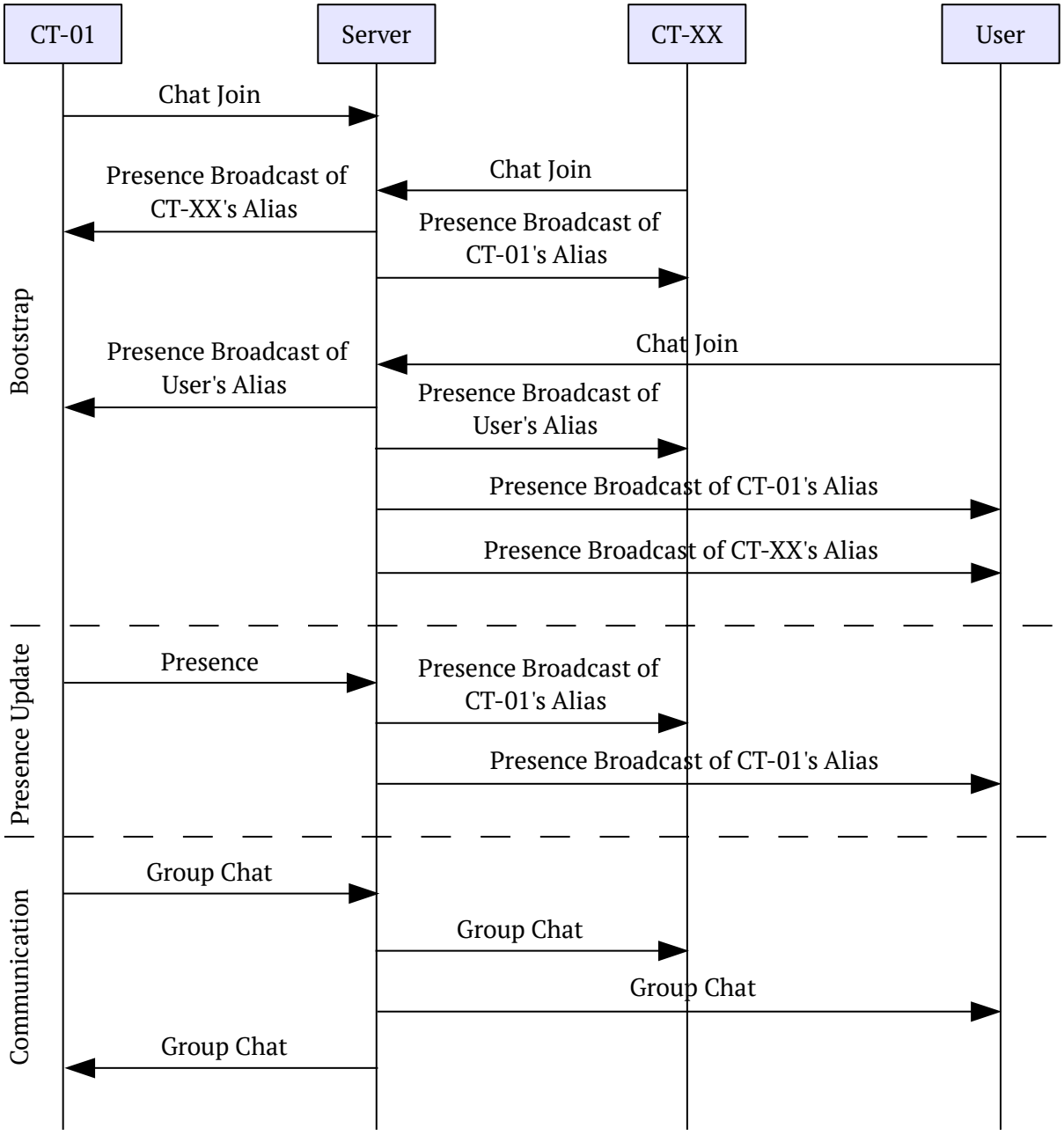

Figure 4.6: XEP-0045 Multi-User Chat (MUC) message flow

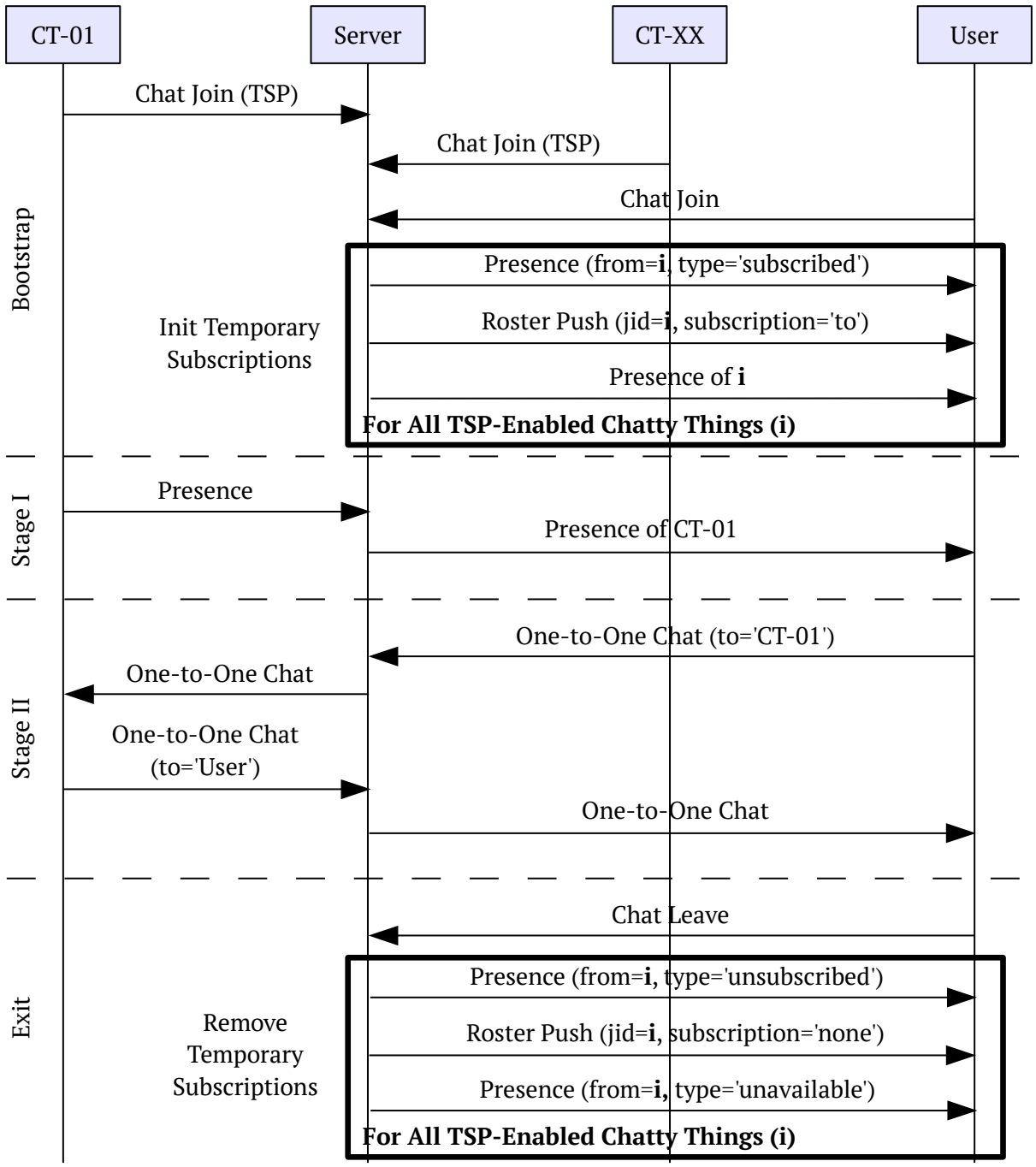

Figure 4.7: Optimized message flow through *TSP* in MUC

4.3.3 Enabling TSP

A Chatty Thing announces *TSP* by adding the optional *<status/>* element to the presence message used to join a chat room. This kind of presence message is sent only once during bootstrapping to initiate *TSP* on the XMPP server side and to set up the respective topics of the Chatty Thing. The *<status/>* element is already defined for this message type in XMPP IM. It specifies a detailed description of the presence status. Unmodified XMPP servers recognize it, but do not react differently (i.e., joining XMPP entity enters the chat room). Backward compatibility is thus ensured. The *<status/>* element is automatically included when the *TSP* module is activated in uXMPP2. The following Listing 4.3 depicts an example (according to [133, Example 20]) of a *TSP* enabled presence message:

```
1 <presence
2         from='hag66@shakespeare.lit/pda'
3         id='n13mt3l'
4         to='coven@chat.shakespeare.lit/thirdwitch'>
5     <status>TSP</status>
6     <x xmlns='http://jabber.org/protocol/muc'/>
7 </presence>
```

Listing 4.3: Example TSP chat join message

The *TSP* flag is only a small adding to the original presence message to join a chat room. It extends the message size only by 20 bytes, as depicted by `<status>TSP</status>` in line 5 in the Listing 4.3.

uXMPP2 Modifications

TSP reduces the needed type of XMPP messages to a minimum set. As pointed out, the join chat message has only be extended with the `<status>TSP</status>` (cp. Figure 4.2 and Listing 4.3), while all other needed message types for sending and receiving presence and group chat messages are already given by the XMPP Core/IM and *XEP-0045* modules of our XMPP stack. In general, *TSP* relies on the XMPP message types presence, chat join, and group chat. When an announcement of *TSP* is needed, the *XEP-0045* module internally sends a join chat message with activated *TSP* flag. The *XEP-0045* module is also used to parse the group chat message. This ensures that the memory usage to generate and parse XML messages is kept very low. So the implementation effort can also be reduced to a minimum which results in a small code footprint of uXMPP2.

XMPP Server Modifications

We have implemented a *TSP* prototype for the XMPP server *Prosody* version 0.8.2 with modifications of its roster manager and its *Multi-User Chat* plug-in. Backward compatibility was tested until version 0.9. Prosody was chosen as XMPP server because it runs on embedded Linux systems. Thus it is the ideal software for the enhanced router (cp. Section 3.2.1). When a *TSP* enabled presence message to enter a chat room is received by the XMPP server, it checks the *<status/>* element for *TSP*. If the element is set the server registers the joining Chatty Thing as a new group member flagged with *TSP* to the room list. The server also adds the Chatty Thing to the roster and forwards its presence message to every non-*TSP* enabled room member. Afterwards a roster update is sent to these XMPP clients and the Chatty Thing appears remotely in the user's roster. Incoming group chat or presence messages are not forwarded to room members with enabled *TSP*. Removing the Chatty Thing from the list is done automatically by the XMPP server when it or the user leaves the chat room by sending a presence of type 'unavailable'. Thus, obsolete Chatty Things are not contained in the user's roster. To be compliant with existing XMPP software, the implementation of *TSP* requires only modifications of the XMPP server. The advantage is that existing XMPP clients do not need to be adapted. Users can benefit from an up-to-date roster and Chatty Things from the reduced network traffic.

To sum up, the proposed *TSP* approach minimizes the network traffic for Chatty Things in *XEP-0045 Multi-User Chat (MUC)* by using small presence messages and introducing a publish-only affiliation. It explicitly avoids the use of XML compression methods to achieve the optimal trade-off between local computation and the XMPP message exchange (cp. [181, Sec. IV.C]). In general, *TSP* reuses a subset of functions and descriptions of existing XMPP Core/IM and XEPs, while extending it with useful functions for the Internet of Things. The introduced *TSP* flag is piggybacked on existing message types. So *TSP* can be treated as a draft of a custom XEP for the IoT (cp. Section 2.4).

5 uBonjour: Minimized Software Stack for the DNS-Based Service Discovery

Self-configuration is mandatory for the Chatty Things approach to automatically bootstrap smart objects independent from the user interaction, the given network environment (e.g., infrastructure or ad hoc), hard-coded start-up parameters (e.g., IP address of an XMPP server), and other pre-configurations (e.g., JIDs for logging in on XMPP servers). Failure-resistant network bootstrapping can be enabled by a service-oriented approach (cp. [182]). It provides transparency with service abstraction for specific device functions and seamless interaction with various device types, while the devices browse their network domain for neighbors and newly published services (cp. [183]). The process in which devices automatically detect joining or leaving devices and available services in a network environment is called *service discovery*. Currently there is no service discovery for the IoT that directly addresses different classes of devices with the same Internet mechanisms [73, 27]. In RFC 6574 [18, Sec. 3.2.1] established service discovery protocols, e.g., Multicast DNS (mDNS) [23] with DNS Service Discovery (DNS-SD) [24], are favored to let different classes of network devices work seamlessly and vendor-independent with each other. This reduces the learning threshold for non-technical users (cp. [184, Sec. 2]). In this chapter the DNS-based service discovery *uBonjour* is introduced. This approach is based on the combination of mDNS and DNS-SD with adjustments for smart objects (e.g., small code footprint, minimized overhead) and allows the service discovery of smart objects with the same Internet mechanisms which are already used. We propose optimizations for mDNS/DNS-SD to effectively reduce the number of exchanged IP packets to meet the requirements of low data rate smart object networks, while ensuring compatibility with DNS.

5.1 Service Discovery in the IoT

Existing approaches for service discovery in the Internet of Things can be divided in service-oriented architectures and specific implementations of 6LoWPAN approaches. Service-

oriented IoT architectures use either a residential gateway [185] or a complex middleware [6]. Current developments of smart object-specific application protocols reinvent similar approaches to DNS-SD for 6LoWPAN. *Resource Directory (RD)* [186] or registry-based *TRENDY* [187] use CoAP as underlying communication protocol and provide service discovery only in a smart object network with a dedicated feature set and a description of available CoRE resources. The discovery of smart objects from ordinary computational devices, however, requires in both cases a gateway, e.g., an HTTP/CoAP translator (cp. [73]) or a directory agent as a proxy (cp. [187]). Moreover, TRENDY needs to expand the role of a 6LoWPAN border router to additionally act as a directory agent because it operates at lower layers. A seamless service discovery that directly addresses different classes of devices with the same Internet mechanisms (cp. [73, 27]) cannot be set up. Compared to service discovery implementations at lower layers, implementations at the application layer, such as DNS-SD and RD, have the advantage that no additional mapping mechanisms onto a high-level service discovery are required [7, Sec. 7].

With mDNS/DNS-SD, a user-friendly and seamless discovery of smart objects can be implemented. *Multicast DNS (mDNS)* is part of a group of standards that is used to automatically enable computers to look for or find other devices and to share their services with each other in network environments without manual configuration by the user. The task of mDNS is to resolve domain names without the help of any unicast DNS server by delivering messages to the reserved multicast addresses `224.0.0.251` (IPv4) and `ff02::fb` (IPv6) via UDP port `5353`. Devices inquire network addresses with requests to a multicast group. The respective device responds with its list of DNS resource records. mDNS is often implemented together with *DNS Service Discovery (DNS-SD)*. The two are available for various platforms, e.g., for Mac OS, iOS and Windows with *Bonjour* [188], and for Linux, BSD, OpenWRT, and Android with *Avahi* [128]. DNS-SD is another part of the standards used to discover devices and share their services. It is combined with mDNS and also supported by Bonjour and Avahi. DNS-SD enables the location and announcement of services of entities in a network domain. DNS resource records are again used to provide information about services. A device usually offers its service by propagating the following DNS resource records:

- **SRV Resource Record:** Defines the service (e.g., service type [189], protocol, domain name), the port, and the hostname of the service offering device in a domain;
- **A / AAAA Resource Record:** Used to map the hostname of the service offering device to an IPv4 / IPv6 address;

- **PTR Resource Record:** Used to resolve devices hostnames in a domain through their network addresses. Also used in mDNS to assign service instances to a service;
- **TXT Resource Record:** Used to propagate user-defined text, e.g., distribute presence in XMPP *Serverless Messaging* (cp. Section 3.2.1).

In contrast to other service discovery protocols (e.g., Service Location Protocol (SLP) [190], Universal Plug and Play (UPnP) [191]), DNS-SD is simple and it has a small protocol overhead, which makes it especially suitable for smart objects. Furthermore, no dedicated peers or centralized instances are necessary (i.e., in use with mDNS), such as directory agents in the case of SLP (cp. [7]). Examples for practical mDNS and DNS-SD implementations are Bonjour [188] and Avahi [128], both widely used on desktop and mobile systems. They are open source and written in C/C++, but too big to fit in the memory of a smart object. A smaller implementation is *Liaison* [192], which has around 100 Kbytes code size and is written in C++. The porting of one of these three implementations onto smart objects would be an extensive and time-consuming task because a complex refactoring with subsequent restructuring of the design is necessary to adapt the implementations to the requirements of these devices. In [16], a mDNS implementation into Contiki with a memory footprint of only 1.0 Kbytes ROM and 0.5 Kbytes RAM has been reported, but there is no code proof available. A direct integration of mDNS for Contiki can be found online [193]. It offers an advanced version of the uIP hostname resolver function and supports IPv4 and IPv6, but not DNS-SD. The most promising mDNS and DNS-SD implementation with only 14 Kbytes code size is *Ethernet Bonjour* [194] for the Arduino platform [195]. It was written in C++ for the WIZnet chipset on the Ethernet shield by Georg Kaindl and supports only IPv4 for Ethernet frames. Thus, there is still no mDNS and DNS-SD implementation available for smart objects that uses the uIP stack and supports IPv6.

5.2 Architectural Solutions and Limitations

We propose here a DNS-based service discovery for smart objects on top of uIP that we call *uBonjour* [219]. It enables a standardized service discovery of smart objects with the same Internet mechanisms which are already used for ordinary computational devices. Interoperability is guaranteed by the established Domain Name System (DNS) for the IoT at the application layer. Thus, there is no need to use specialized protocol gateways nor to run smart object-specific code on computational devices.

uBonjour is a minimized implementation of mDNS and DNS-SD [219] on smart objects to discover and address devices and available services in IP-based network environments. It is based on the Ethernet Bonjour project (cp. Section 5.1). For uBonjour, the Ethernet Bonjour code was extended and reimplemented in an optimized way for the Contiki OS with IPv6 support. The interoperability was tested to work with the mDNS/DNS-SD implementation *Avahi*, as it will be described in Section 6.3. We reused the parser/message generator and its function stub. C++ parts were ported to C and the WIZnet chipset related code was rewritten to use the uIP stack of the Contiki OS. These measures ensure that uBonjour runs properly on Contiki with a minimized memory consumption. The general features of uBonjour for application protocols and developers are the resolving of hostnames, the discovering, registration, removal, and updating of services. The implemented mode of operation is described in detail in Appendix A.2. Applications can register and announce their availability as services in the network and discover other devices that support the same application protocol to establish a direct communication.

uBonjour supports an application-, device-, and vendor-independent announcement for smart objects that want to offer their services in a network domain. It implements the standardized behavior of mDNS according to [23] and DNS-SD [24] to ensure a compliant message exchange with different kinds of computer systems using either Bonjour or Avahi. This enables computational devices to discover and address smart objects and their advertised services in an easy-to-use and transparent way without using application protocol gateways. uBonjour supports self-configuration instead of hard-coded addresses, so that smart objects can scan their network environment and share results without the need to know the exact network topology. Adding a new smart object is performed through requesting information from surrounding devices, while a coordinated exit is enabled using "service unavailable" messages. Bootstrapping of smart object networks is simplified because services can be found and accessed autonomously by all network devices without any manual configuration or user intervention (cp. [27, 196]). Therefore, uBonjour is an essential part of the Chatty Things approach. It implements a parameter-less bootstrapping in hybrid networks and provides the basis for the *XEP-0174 Serverless Messaging* when XMPP servers are unavailable during bootstrapping and at run-time.

5.2.1 Parameter-Less Bootstrapping of Chatty Things

Adding a Chatty Thing to a network is possible by requesting services (e.g., XMPP server) or by receiving information from surrounding devices. The respective device responds

with its list of DNS resource records that contains its host/domain name(s), IP address, port, and the assigned service(s). Due to the constrained resources of smart objects, not all available components of the uXMPP2 stack (cp. Section 4.1) can be activated during bootstrapping and at run-time. The IPv6 stack memory footprint (see Appendix A.4) and the dynamic memory use are the limiting factors. An intelligent handler process for the two XMPP clients (e.g., Core/IM, *XEP-0174*) and uBonjour is therefore very important to ensure memory-efficiency and flexibility. We use multi-level bootstrapping to reduce the memory consumption of the XMPP stack by stepwise enabling its components depending on the given network environment:

I: uBonjour is activated to discover an XMPP server in the given network environment. If an XMPP server is found step **II** follows, otherwise step **III**;

II: Infrastructure mode – Deactivate the uBonjour client, activate the XMPP client, and connect to the XMPP server;

III: Ad hoc mode – Activate the *XEP-0174* client.

During run-time an automatic switching between infrastructure and ad hoc mode is triggered depending on incoming DNS resource records (e.g., XMPP server is un-/available) or the connection state to the XMPP server (e.g., connected, lost). The switching process follows these rules:

IV: Infrastructure mode – If the connection to the server gets lost and a predefined number of reconnections attempts fails then deactivate the XMPP client, goto step **I**;

V: Ad hoc mode – If an XMPP server joins the network deactivate the *XEP-0174* client, goto step **II**.

The incremental activation of components is a resource-efficient way for the discovery, the self-configuration, and the seamless integration of Chatty Things in IP-based networks without the need for any user interaction or manual pre-configuration. Chatty Things automatically discover their given network environment and react autonomously on topology changes and un/available services. An extension of this discovery functionality beyond local network boundaries is possible using Wide-Area DNS-SD [197] with the same DNS-SD APIs. This can ultimately boost the integration of smart objects into the current Internet infrastructure [198] and support service discovery in wired and wireless networks. The extensions for Scalable DNS Service Discovery (dnssd) proposed by the IETF working group [199] focuses on this aspect. In conjuction with Wide-Area DNS-SD, a solution for a

general bootstrapping of different network devices can be implemented that overcomes the problems of a dedicated server-based rendezvous point, as explained in [220]. This has the advantage that devices can be found via registered services in the Internet using standardized DNS messages. As mDNS and DNS-SD rely on DNS, which can be seen as the backbone of the Internet, the DNS-based service discovery is widely deployed.

5.2.2 Limitations for the Use of DNS in Smart Object Networks

As mDNS and DNS-SD were initially designed for computational devices in home networks with nearly no limit of bandwidth, memory, and processor resources, the two protocols lack optimizations for low data rate smart object networks [199, Sec. 2.3]. Therefore, the following DNS mechanisms require optimizations for their use in low data rate smart object networks and in uBonjour.

DNS Message Compression

A DNS message is composed of a DNS header and at least one resource record [200, Sec. 4.1]. The DNS header takes 12 bytes of a DNS message. The number of embedded resource records in a DNS message is stored in fields of the DNS header. A DNS record contains fields for the (domain) name (e.g., owner name, hostname, service name), the resource type, the class code, the Time-To-Live (TTL), the length of the resource data, and the resource data. Figure 5.1 depicts the DNS resource record format.

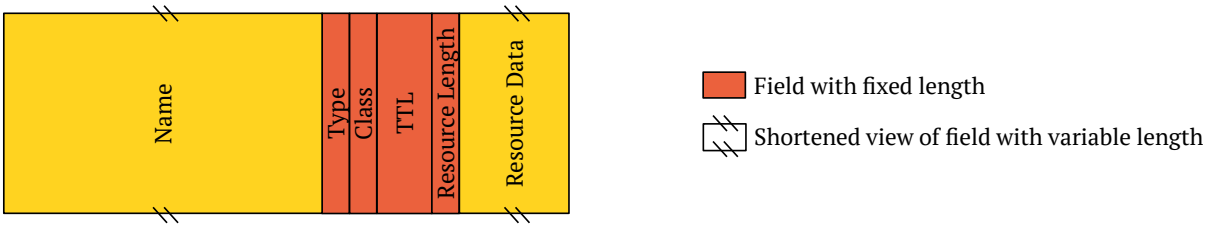

Figure 5.1: DNS resource record format, as defined in RFC 1035

Embedding a set of resource records in a single DNS message saves 12 bytes for avoiding an additional DNS header for each resource record. Furthermore, DNS message compression allows one to shorten names by using pointers to a prior occurrence of the same name [200, Sec. 4.1.4]. The length of a pointer takes only 2 bytes because it is represented by a two octet sequence containing the pointer flag (the first two bits are set to one) and the

offset from the start of the prior occurrence of the same name. Figure 5.2 gives an example of the DNS message compression for a DNS message containing the two resource records `SRV` and `PTR`. The DNS name pointer can be used within a record (i.e., the pointer of the `SRV` record uses the prior occurrence of the name `local`) or to a prior record if two records are sent within the same DNS message (i.e., `PTR` name and resource data point to a prior occurrence of the same name or a substring located in the `SRV` record).

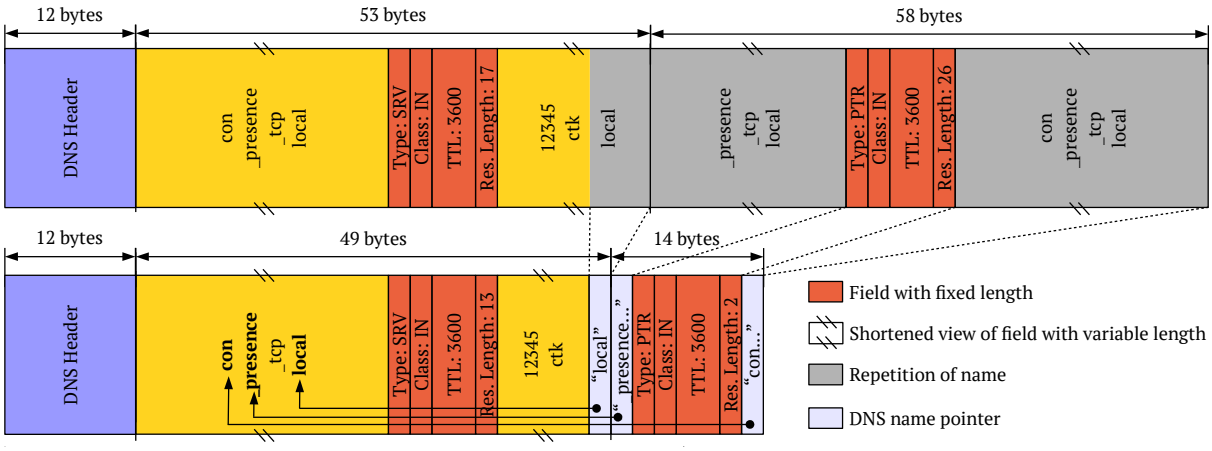

Figure 5.2: Example DNS message compression

A DNS record has fields with fixed lengths (e.g., type, class, TTL, resource length) and with variable lengths (e.g., name, resource data). The use of DNS name pointers preferentially concentrates on the fields with variable lengths to decrease the DNS message size. This requires only a minimal larger code size and no additional buffers for the generation of name pointers and the calculation of their offsets. The highest DNS message compression ratio is achieved if all four required DNS records fit into a single IP packet, since only then every occurrence of a name can be replaced by a pointer and only one DNS header is used. The minimal DNS message compression ratio requires the integration of at least two DNS resource records and the use of one DNS header, as shown in Figure 5.2. This avoids the exchange of redundant data.

As the available IP payload sizes vary for different smart object hardware platforms (with disabled reassembly, cp. Section 2.2.2), not all required DNS resource records can fit into a single IP packet without adaption. Thus, a strategy is required to efficiently implement DNS message compression for uBonjour in respect to available IP payload sizes of different smart object hardware platforms for Contiki. For this, we propose as solution the *Adjustable DNS Message Compression (ADMC)* which will be described in Section 5.3.2. To achieve

the highest compression ratio for all available IP payload sizes further enhancements are required. They are proposed as *ADMC Enhanced* in Section 5.3.3.

Availability Time of a Service

The TTL field in each DNS resource record needs to be set to a time value in seconds to specify how long a published service will be available. A normal value in this case is 120 seconds, which may be increased for further optimization. The use of larger values minimizes the number of sent messages between devices. This setting does not interfere with the joining of new devices because these can explicitly ask for available services in the network. The default value is set to 3600 seconds in uBonjour, i.e., each smart object has only to re-announce its service every hour.

mDNS Traffic Reduction

mDNS specifies optimizations [23, Sec. 7] to keep the data traffic to a minimum, such as the Known-Answer Suppression or the Duplicate Question Suppression method. The *Known-Answer Suppression* method reduces the total number of answers. If a device wants to send a query and it has some cached answers to it these answers are added to its own query. For this, each device needs to cache the published service offerings in the network and wait a randomly chosen time before it answers a request. If other devices recognize answers in a query matching their own they refrain from sending their own answers. Thus, the number of necessary responses to gather information about the network is reduced. The *Duplicate Question Suppression* method, in contrast, reduces the total number of requests. A device does not send a request when it notices that another device sends a request that matches its own. This prevents the sending of redundant DNS responses because less `PTR` query messages are sent. Again, each device has to wait a random period before it can send its request.

Up to now, we have refrained implementing these two message suppression methods for uBonjour because the two optimizations require to store a bundle of message related data for their proper functionality. This increases the needed buffer and code size, thus increasing the use of RAM and ROM for the discovery service. To avoid this we developed another optimization approach for uBonjour, which will be introduced in Section 5.3.1.

5.3 Optimization Approaches

This section introduces optimizations for the implementation of mDNS/DNS-SD on typical smart object hardware platforms with the focus on low memory consumption and efficient DNS message transport. To enable a high compatible and device-independent DNS-based service discovery uBonjour strictly avoids the need to run smart object-specific code or data representations on computational devices. Instead, uBonjour's message transfer optimizations focus on increasing the ratio of maximum DNS payload size to DNS header length for all available IP payload sizes in conjunction with known and newly introduced DNS compression methods. This enables the efficient integration of all required DNS resource records into a single IP packet.

Jara et al. [73] analyzed mDNS/DNS-SD and published theoretical optimizations to reduce the message overhead from another point of view: the avoidance of any authority and additional record and the reduction of the number of used IEEE 802.15.4 frames per sent DNS resource record by redefining the format of the `TXT` record entries. Here, multiple `TXT` entries are summarized to a unique entry by using the CoRE link format [84] description and data compression techniques (e.g., Lempel-Ziv 1977 (LZ77) algorithm). This introduces smart object-specific data representations to mDNS/DNS-SD. On the other hand, it annuls the backward compatibility because current DNS implementations expect only one information per `TXT` record. Therefore, a further indication for this type of data representation is needed, but it was not considered. The approach strongly concentrated on shrinking each DNS resource record to fit into a single IEEE 802.15.4 frame, but not on re-using redundant information of lower layers. This introduces a message overhead of 36 bytes for using a set of dedicated DNS headers because each DNS resource record is sent in a separate DNS message (cp. Section 5.2.2). Since these ideas are not backward compatible to the DNS standard and introduce smart object-specific code to computational devices, they were not implemented in uBonjour. In addition, the recommended compression mechanisms rely on external algorithm, which are not optimized for memory-constrained systems and produce a high memory usage on smart objects (cp. Section 4.2).

5.3.1 Memory and Traffic Reduction

uBonjour is supposed to assist smart objects in finding available services and being discovered by other classes of computational devices inside a network. Therefore, the implementation must be as slim as possible to allow other applications to reside in the

smart object's limited memory as well. A large quantity (about 60%) of uBonjour's source code size is consumed by the handling of received DNS records and by the generation of DNS responses. Therefore, we optimized the memory management for this code part to minimize the memory consumption. The buffer size of the parser could be reduced because the handling is now done directly inside the uIP buffer. The generation of DNS responses requires only a small buffer of the size of a DNS header, while the rest of the message generator directly uses the uIP buffer. This *in-place processing strategy* facilitates a memory-efficient service discovery for the Contiki OS [68, Sec. III-H].

One-Way Traffic (OWT)

Since the *Known-Answer Suppression* and the *Duplicate Question Suppression* method would consume too much memory (cp. Section 5.2.2), we propose a traffic reduction for mDNS and DNS-SD in smart object networks, called *One-Way Traffic (OWT)* [219]. The *OWT* optimization is a built-in function and can be activated during the compilation of uBonjour. This optimization puts a smart object into a passive mode in which the device only publishes its services periodically and responds only to incoming name and service requests. Passive mode disables the active resolving of hostnames and the ability to parse service query responses. Thus, *OWT* explicitly enables queries for advertised services in one direction: from outside to inside the smart object network. Service query responses are targeted only to ordinary computational devices. The activation of *OWT* and the subsequent disabling of hostname resolving and service query response parsing significantly reduces the used code size and can also save energy because message parsing and network traffic are minimized overall (cp. Section 6.3).

Furthermore, the *OWT* optimization also reduces the needed of lines of code, since the parser handling for incoming service query responses can be skipped. These are about 400 lines of code. We do not lose much of the core functionality of uBonjour because smart objects are still able to actively register services and to react to requested services from outside the smart object network. Overall, this behavior facilitates the lightweight aspect of the discovery service by coupling ordinary computational devices with smart objects. Computational devices can scan their environment for smart objects with a preinstalled mDNS and DNS-SD service, while nearby smart objects can directly answer to them with DNS records without the need of installing additional protocols or using application protocol gateways. This establishes an easy-to-use and well-known discovery mechanism for consumers and offers a simple integration strategy for system administrators.

Memory Optimization Stages

OWT enables different memory optimization stages for uBonjour that can be chosen during compilation depending on the use case:

- **Stage I:** No optimization - all implemented mDNS/DNS-SD features are enabled. This causes the highest memory consumption.
- **Stage II:** Low optimization - *One-Way Traffic (OWT)* extended with additional `A` and `AAAA` record detection. It deactivates the active resolving of hostnames and the parsing of `SRV` and `TXT` query responses. The state offers a reduced function set of uBonjour, but the memory usage can be decreased significantly. This stage is used for the parameter-less bootstrapping (cp. Section 5.2.1).
- **Stage III:** High optimization - *OWT* is fully enabled. It deactivates the active resolving of hostnames and the parsing of service query responses (e.g., `SRV`, `TXT`, `A`, `AAAA`). This stage offers the lowest memory consumption and a minimal feature set.

With each stage, a loss of features in favor of memory savings is accompanied. Beside memory optimizations (e.g., parsing and generating DNS responses) and traffic (i.e., *OWT*) reduction, we also reduce the needed number of exchanged DNS messages in smart object networks by implementing DNS compression for uBonjour.

5.3.2 Adjustable DNS Message Compression (ADMC)

The aim of the *Adjustable DNS Message Compression (ADMC)* [219] is to automatically adjusts the number of DNS resource records that can be sent in an IP packet to minimize the DNS message overhead independent from the used smart object hardware platform. The optimal integration strategy for the correct numbers of DNS records depends on the minimal length of each DNS resource record and their combinations. The minimal length of a DNS record is the sum of fields with fixed lengths, the length of additional information (e.g., port, IP address, user-defined text), and the sum of the lengths of all name pointers which completely replace each name. Table 5.1 shows examples of all required DNS records and their lengths to announce the availability of *XEP-0174* (cp. Section 3.2.1). The column for the length holds three values: the first value shows the full length of the DNS record including the service name (e.g., `_presence._tcp.local` for *XEP-0174*) without the use of name pointers, the second value shows the length of a DNS record if name pointers are only used within the record (i.e., each record has to be sent by a separate DNS message

or the record is included as the first one), and the third value shows the minimal length (for details see Appendix A.3). It is important to note that the second value does not include the length of the service type (e.g., `_presence`) because this is an individual value and its length cannot be predetermined (cp. [189]). The lengths of the unique owner name and the hostname are limited for simplicity to 3 bytes[1] in each case. Furthermore, Table 5.1 depicts the repetition of DNS names. The name field of the `SRV` record is included in the name field of `TXT`, in the resource data field and as a substring in the name field of `PTR`; the name field of `A` and `AAAA` occurs in the resource data field of `SRV`; the last substring of all names is repeated in every mentioned field and a prior occurrence can be replaced with a pointer within a single DNS resource record.

Table 5.1: Examples of used DNS resource records for *XEP-0174* and their length (full/use of name pointers only within a record/minimal)

Type	**Example (name, type, class, TTL, resource {length, data})**	**Length**
`SRV`	con._presence._tcp.local, `SRV`, `IN`, 3600, 17, 12345 ctk.local	53/39/20
`PTR`	*_presence._tcp.local*, `PTR`, `IN`, 3600, 26, *con._presence._tcp.local*	58/29/14
`TXT`	*con._presence._tcp.local*, `TXT`, `IN`, 3600, 13, status=avail	49/40/25
`A`	*ctk.local*, `A`, `IN`, 3600, 4, 172.16.150.0	25/25/16
`AAAA`	*ctk.local*, `AAAA`, `IN`, 3600, 16, aaaa::212:7402:2:202	37/37/28

Enabling all pointer possibilities is only feasible if all required DNS resource records are included in a single DNS message because only then the beginning `SRV` record holds a sequence of labels for all required names. The other DNS resource records can then use name pointers to this prior occurrence. Thus, these DNS resource records can be integrated with minimal length. In this case, the minimal needed DNS payload size is 94 bytes for IPv4 and 106 bytes for IPv6. Unfortunately, some hardware platforms only support a very low IP payload size when running Contiki for usable application data (with disabled reassembly, cp. Section 2.2.2). This prevents that the combination of all required DNS resource records can fit into a single DNS message. The Zolertia Z1 hardware platform has one of the lowest IPv4/v6 payload sizes which allows an available DNS payload size of 56 bytes for IPv4 and 68 bytes for IPv6. It is, therefore, a good reference to calculate the necessary boundaries for the optimal number of DNS resource records fitting into a single

[1] A string of the length of three offers more than 2 million possibilities (e.g., 128^3) in the case of using *American Standard Code for Information Interchange* (ASCII) to create enough unique names for a number of smart objects in a local domain.

IP packet of the uIP stack. The minimal DNS message compression ratio (cp. Section 5.2.2) can only be achieved for the Z1 if the combination of the two DNS resource records is chosen in a way that all name pointer possibilities for the second record can be enabled, as demonstrated in Figure 5.3 (for details see Appendix A.3). In contrast to the Z1, most hardware platforms (e.g., Tmote Sky, AVR Raven, Redbee Econotag) can handle a DNS payload size of (or larger than) 168 bytes for IPv6.

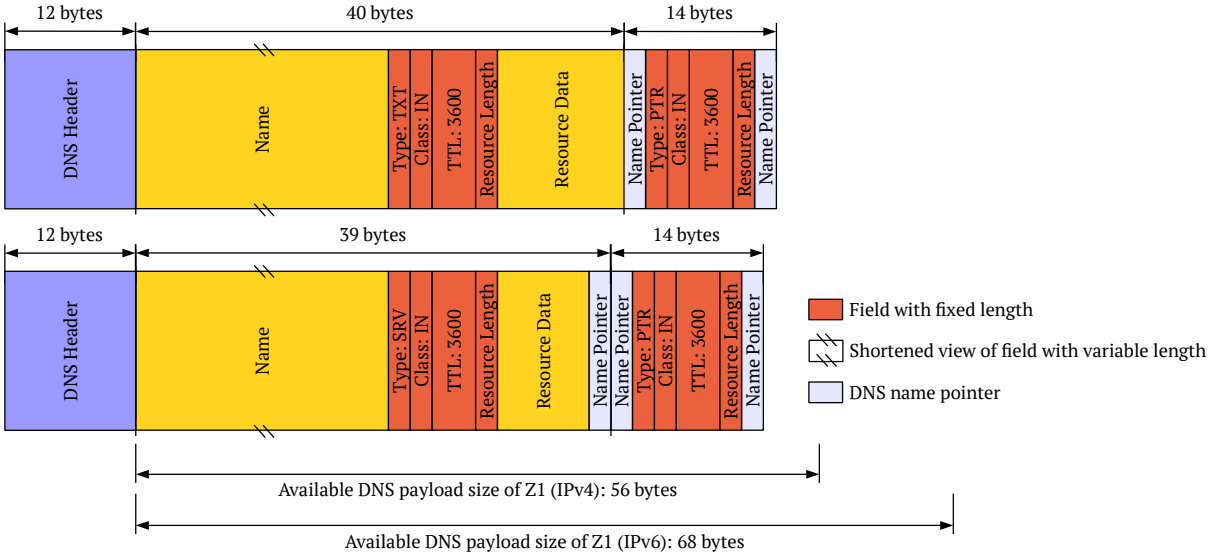

Figure 5.3: Example combinations of the integration of two DNS resource records

For this, *ADMC* uses a decision matrix (see Table 5.2) to summarize the boundaries of the available IP payload size[2] with the number of DNS records that can be integrated into a single IP packet depending on the available DNS payload size. The integration of two DNS records into a single DNS message is possible for a DNS payload size of 56 bytes (i.e., a `PTR` and an `A` record) if service types [189] with a string length shorter than 8 bytes are used. For larger strings (e.g., `_presence` or `_xmpp_client`), further optimizations are necessary (cp. Section 5.3.3). Thus, each DNS resource record has to be sent by a separate DNS message. Four IP packets are needed for this. Possible combinations for the integration of two DNS resource records into a single DNS message in case of 68 bytes payload size are: a `PTR` record with a `TXT` (54 bytes; leaving 14 bytes for the service type) or with a `SRV` record (53 bytes; leaving 15 bytes for the service type), as depicted in Figure 5.3. So only three DNS messages need to be sent.

[2]The IP payload size does abstract from the different IP header sizes of IPv4 and IPv6.

Table 5.2: Decision matrix to integrate the correct number of DNS resource records into a single IP packet (in bytes)

IP Payload Size (x)	**Min. DNS Payload Size**	**IP Packet(s)**	**Free Bytes**
$x < 80$	56 (IPv4)	4 (3)	> 10 (7)
$80 \leq x \leq 140$	68 (IPv6)	3 (2)	> 10 (4)
$x > 140$	94/106 (IPv4/v6)	1	> 10

ADMC implements these calculated boundaries shown in Table 5.2 in uBonjour. This ensures the efficient use of DNS name compression for each supported hardware platform of Contiki through the integration of the correct number of DNS records into a single DNS message depending on the available IP payload size of a device. *ADMC* is totally compatible with existing DNS standards. It uses the default configuration parameters (e.g., 6LoWPAN fragmentation, uIP buffer sizes, no header compression) of Contiki for each supported hardware platform.

5.3.3 Enhanced Optimizations: The Way to *ADMC Enhanced*

Hardware platforms with a very low IP payload size cannot fully be optimized with *ADMC* for low data rate networks because the sending of a service announcement still requires four IPv4 or three IPv6 separate DNS messages, respectively. This section proposes further optimizations to reduce the overhead and the response time for the efficient exchange of DNS messages in smart object networks. The ultimate goal is to enable standard-compliant compression approaches to integrate all four DNS resource records into a single IP packet and to simultaneously support most hardware platforms of Contiki. We propose here some approaches for the enhanced compression of further fields and additional information of DNS resource records to achieve the maximum compression ratio which avoids any repetitions within the DNS message. Each enhanced optimization (e.g., compression methods) is compared in terms of the implementation effort, the number of used IP packets, and the level of compatibility with the RFC 1035 [200, 4.1.4].

Using 6LoWPAN Compression

The reason for the small available IP payload sizes for application data is that the IP and UDP headers take most of each sent IP packet (48 bytes maximum), as explained in Section 2.2.2. To overcome this restriction a compression format for IPv6 datagrams over IEEE 802.15.4-based networks was defined in RFC 6282 [201], called IPv6 Header Compression (IPHC) and Next Header Compression (NHC). For the communication with global addresses, the IP and UDP headers are compressed down to 10 bytes [202]. This releases 38 bytes of the two headers and can theoretically enlarge the DNS payload size of the Zolertia Z1 to 106 bytes. As the minimal length for all four DNS resource records is 106 bytes, a single DNS message cannot be sent without further optimizations because there is no space left for the string of the service type. In contrast to *ADMC*, the number of sent IP packets can be reduced from three to two for the Zolertia Z1 hardware platform leaving enough space for the service type. Overall, the 6LoWPAN compression is very helpful for hardware platforms with very low IP payload sizes. Since the compression works at the lower layers, no changes of the DNS standard and of implementations at the application layer are necessary. Drawbacks are that not all required DNS resource records can be integrated into a single DNS message (e.g., single IP packet) and that IPv4 cannot profit from this kind of compression because it is only available for IPv6.

Class Code and TTL Field Compression

The standardized DNS message compression does not consider repetitions in the fields with fixed lengths. For this reason, a size of 10 bytes is unusable in each DNS record, although a repetition of the class code or the TTL value can occur when multiple DNS resource records are sent in a single DNS message, as shown in Table 5.1. In this case, all DNS records have to set the same values for the class code and the TTL field. The two fields take a length of 6 bytes (e.g., 2 bytes for class code, 4 bytes for TTL), while the use of a pointer to a prior occurrence requires only 2 bytes. We propose the use of such a pointer for the two fields to reduce the unusable amount to 6 bytes (leaving the type, the resource length fields, and a pointer). In the best case, 12 bytes can be saved if all four DNS resource records are sent in a single DNS message (see Figure 5.6) or 4 bytes in the worst case if only two DNS resource records fit into a single DNS message (see Figure 5.4). The second case would lead to a length of 18 bytes for the service type for the combination of a `PTR` and a `TXT` record or to a length of 19 bytes for the service

type when combining a PTR and a SRV record in the case of 68 bytes (cp. *ADMC* in Section 5.3.2). Furthermore, the following combinations are possible for IPv6: a SRV with a TXT record (60 bytes; leaving 8 bytes for the service type) and a PTR with an AAAA record (57 bytes; leaving 11 bytes for the service type). Thus, only two DNS messages for IPv6 need to be sent. For a DNS payload size of 56 bytes (IPv4), the second case allows the combination of a PTR and an A record and leads to a length of 11 bytes for the service type instead of 7 bytes, i.e., only three DNS messages need to be sent.

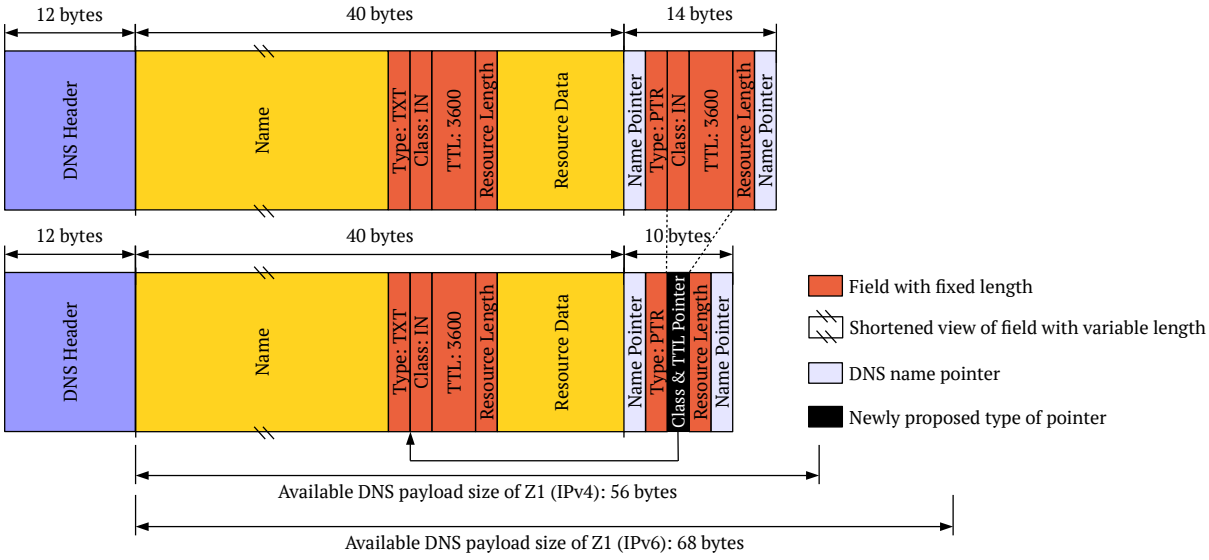

Figure 5.4: Example of the class code and the TTL field compression

As the proposed compression scheme of the class code and the TTL field differs from the compression of (domain) names, another pointer flag is required to indicate this optimization. Thus, the first two bits cannot start with two zero bits (indicates a label) or two ones (indicates a pointer to a name). All other combinations (e.g., 10 and 01) are reserved by RFC 1035 (for future use) and can be used to indicate a pointer for the proposed class code and the TTL compression. This optimization, however, saves only a few bytes, but its implementation requires only few lines of code and provides a high compatibility level with the current DNS standard.

Redundant Information Filtering

Sending four DNS resource records to announce a service in the network contains a lot of redundant information (e.g., names, IP addresses). From our perspective, these data

can be reconstructed by retaining redundant information to reduce the network traffic. The SRV record holds information to build a whole PTR record and the beginnings of TXT and A (AAA) records. The rest of the information (i.e., IP address, user-defined text, and their respective resource lengths as data delimiters) can easily be appended to the resource data of the corresponding SRV record. Thus, the minimal length of a single DNS message containing all information for the four DNS resource records is reduced from 94 bytes to 60 bytes for IPv4 and from 106 bytes to 72 bytes for IPv6. The latter is depicted in Figure 5.5.

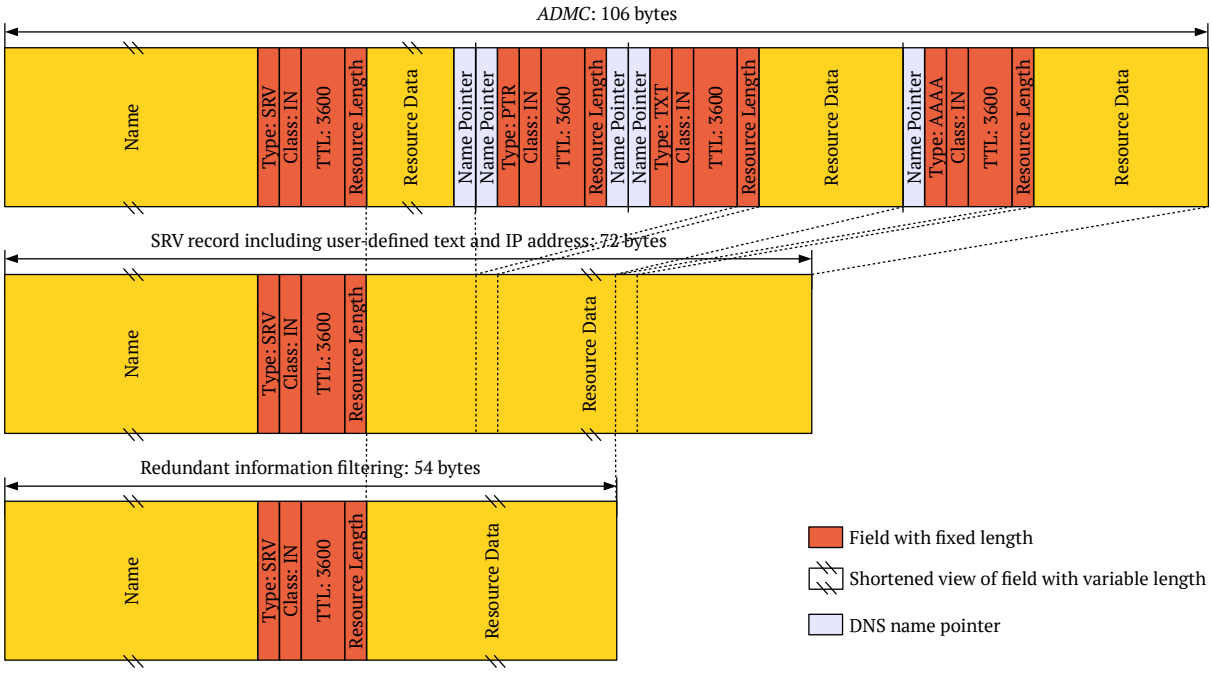

Figure 5.5: Example of the redundant information filtering for IPv6

Unfortunately, this is not small enough for the available DNS payload sizes of the Zolertia Z1 hardware platform, but we can divide the needed number of DNS messages in half[3] and send only a SRV record with the appended IP address and an additional TXT record. The IP addresses of the A and AAAA records are another redundant information which may be reconstructed from the lower layers using the source address of the IP header. This principle has been inspired by RFC 2464 [203] that defines the reconstruction of 128 bit IPv6 addresses from 64 bit MAC addresses. An implementation can be either realized via

[3]It uses two instead of four separate DNS messages, cp. Table 5.2.

a pointer flag[4] in the DNS resource record or by directly extracting[5] the source address from the IP packet. With the latter in mind, the user-defined text of a `TXT` record can be appended to a `SRV` record. This reduces the minimal length of a single DNS message simulating all four DNS resource records to 54 bytes for IPv4 and IPv6, as depicted in Figure 5.5. Thus, a single DNS message for the Zolertia Z1 platform can be used in the case of IPv6 (e.g., 68 bytes maximum available for the DNS payload size per IP packet) which leaves 14 bytes for the service name.

The main disadvantage of our optimization is that it cannot be understood from mDNS- and DNS-SD-compliant implementations because they expect all four DNS resource records each represented by a dedicated record. To overcome this issue and to enable backward compatibility a filter mechanism (comparable to a filter rule used in firewalls) is needed. Such a filter mechanism can be integrated in the Avahi daemon, which already acts as a DNS message repeater between different network interfaces (e.g., IEEE 802.15.4, IEEE 802.11 radio links), as explained in Section 3.2.1. Our filter mechanism works in bidirectional order: the `TXT`, `PTR` and `A(AAA)` records sent to the IEEE 802.15.4 network interface are blocked and their information is appended to the corresponding `SRV` records; the `SRV` records sent from the IEEE 802.15.4 network interface are used to generate the necessary `TXT`, `PTR` and `A(AAA)` records. This information is removed from the respective `SRV` records before forwarding them to the other network interface. This optimization needs a high implementation effort to provide backward compatibility to existing implementations (e.g., Avahi, Bonjour). On the other hand, it significantly reduces the minimal needed length of a single DNS message (in combination with IP address reconstruction) for hardware platforms supporting only a very low IP payload size. For IPv6, it provides the minimal needed length to integrate all four DNS resource records into a single IP packet and for IPv4, it can save up to 50% of the sent IP packets.

Reasonable Trade-Off: *ADMC Enhanced*

Regarding the aforementioned compression approaches, the largest compression rate can be achieved with redundant information filtering (especially for IPv4). However, the implementation effort is very high. It introduces new dependencies at the application layer. The rest of the optimizations are lightweight and can ensure a high compatibility

[4]This will work without heavy modifications when no IP header compression method (e.g., HC1, HC2, IPHC/NHC) is enabled in the 6LoWPAN.

[5]The lower layer (IP stack) has to provide such functionality via API calls.

level to existing mDNS and DNS-SD implementations, as their modifications can easily be adopted. Table 5.3 summarizes the advantages and disadvantages. Furthermore, the required number of DNS messages (e.g., sent IP packets) for the Zolertia Z1 hardware platform are listed to compare each enhanced optimization with *ADMC*.

Table 5.3: Comparison of the enhanced DNS message compression methods with *ADMC*

Optimization	**Implementation Effort**	**Compatibility Level to DNS**	**IPv4/v6 Packet(s)**	**Free Bytes**
ADMC	Low	Highest	4 / 3	> 10
6LoWPAN Compression	Lowest	Highest	4 / 2	> 10
Class Code and TTL Field Compression	Low	High	3 / 2	11 / 8
Redundant Information Filtering	Highest	Low	2 / 1	11 / 14
ADMC Enhanced	Low	High	3 / 1	13 / 26

As a high compatibility level[6] is very important to comply with current DNS standards, we favor optimizations that can seamlessly be integrated. The reserved pointer flags (e.g., `10` and `01`) of RFC 1035 can indicate the use of the enhanced compression methods for the class code / TTL field and the IP address reconstruction. In contrast, the proposed redundant information filtering requires a lot more adaptions to DNS and to the application layer to guarantee compatibility with existing implementations. Therefore, the requirement of using four DNS resource records to announce services with DNS-SD should not be changed. Combining the class code and TTL field compression with the IP address reconstruction allows a more optimized message flow of DNS in smart object networks, while preserving the defined formats of the four required DNS resource records. We call this approach *ADMC Enhanced* [221]. Figure 5.6 depicts this structural difference between *ADMC Enhanced* and the redundant information filtering as well as the compatibility of *ADMC Enhanced* (i.e., use of pointers) to DNS message compression (e.g., *ADMC*). Compared to *ADMC*, the `PTR` and `A` records for IPv4 and the `TXT` and `AAAA` records for IPv6 can still be integrated into a single DNS message. The needed number of IP packets is reduced to three ones for IPv4 and to only two ones for IPv6. In the case of IPv4, the two enhanced compression methods save 6 bytes and provide a larger available length for the service type of 13 bytes

[6]Highest: ensures backward compatibility; High: reuse of already defined methods (e.g., pointer) and the message format; Low: introduces new methods or changes the message format.

instead of 7 bytes, for IPv6, 14 bytes are still available for the service type. The class code and TTL field compression save 4 bytes and the IP address reconstruction additionally 14 bytes. The activation of the 6LoWPAN compression (IPHC/NHC) allows us to send all four DNS resource records in a single IP packet because the best compression ratio of 12 bytes for the class code and TTL field compression can be used within a single DNS message. The minimal length of all four DNS resource records is reduced to only 80 bytes, whereas the maximum available DNS payload size is extended to 106 bytes when communicating with global addresses. For this configuration, the string of the service type can have a maximum length of 26 bytes.

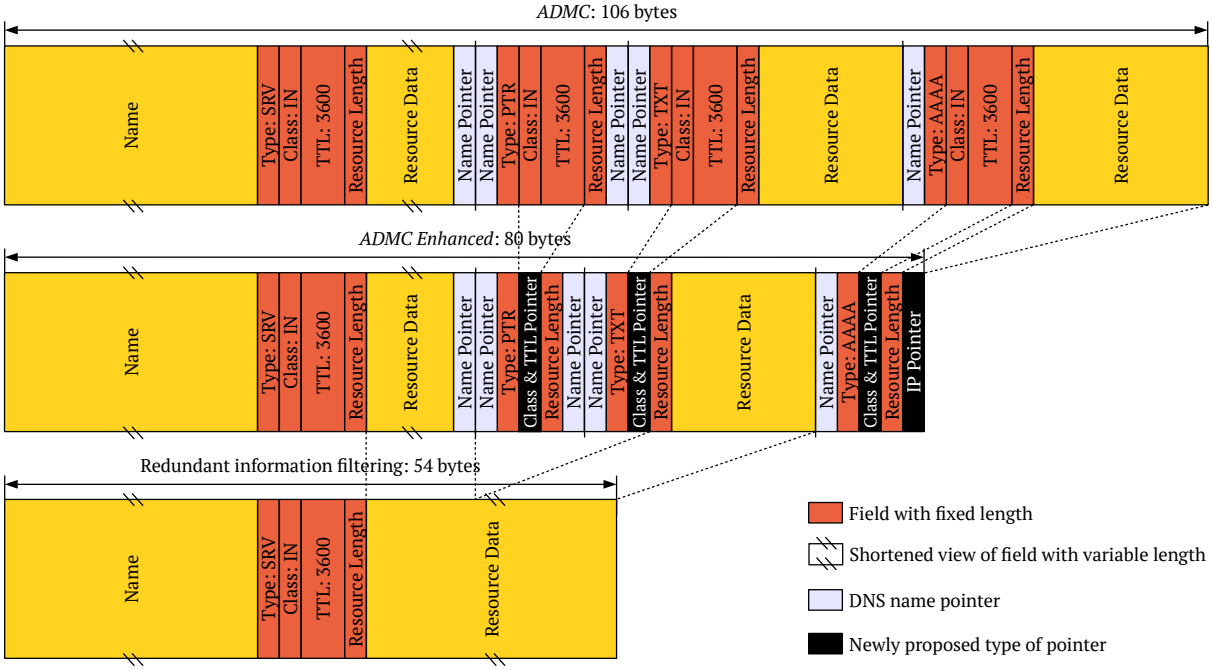

Figure 5.6: Comparison of the enhanced DNS compression methods for IPv6

To sum up, the implementation of *ADMC Enhanced* in uBonjour enables an efficient and standardized DNS message transport in low data rate networks without introducing smart object-specific code to DNS. A high compatibility can be ensured by using the reserved pointer flags to indicate the two compression methods of *ADMC Enhanced*.

6 Evaluation

In this chapter we present the results of the evaluation of the main components of the Chatty Things approach, uXMPP2 and uBonjour. The objective of this evaluation has been to measure the performance of the two components in order to show that XMPP and mDNS/DNS-SD can efficiently be implemented on smart objects. Furthermore, we want to prove that the XMPP layer for the IoT allows the seamless integration of smart objects and an efficient event distribution. The evaluation starts with a description of the used test environment, followed by the evaluation of uXMPP2 and uBonjour regarding the memory footprint and the response time. Finally this chapter presents results of the integration of Chatty Things into a real world IoT testbed and the performance of the XMPP layer.

6.1 Experimental Setup

The evaluation testbed corresponds to a typical IoT scenario in which various computational devices, such as smartphones, netbooks, and smart objects can be connected via enhanced routers over IP links, as described with the smart home use case in Section 3.2.1 and illustrated in Figure 3.6. The enhanced router provides physical links between different network access technologies (General Purpose Access Point (GPAP) [125]) and supports additional software packages, such as the XMPP server *Prosody* [127] and the mDNS/DNS-SD implementation *Avahi* [128]. The setup is depicted in Figure 6.1. It requires a dedicated smart object that runs Contiki's implementation of a 6LoWPAN border router to bridge the IEEE 802.15.4 radio technology over a USB connection to a computational device, i.e., Linux PC. The border router converts 802.15.4 / 6LoWPAN frames to Ethernet / IPv6 frames. The smart object network and the computational network are interconnected via the Serial Line Internet Protocol (SLIP) [204]. For administrative simplicity, the Linux PC runs the enhanced router, and the two XMPP clients *Pidgin* [140] and *Empathy* [205] to verify the user interaction with Chatty Things. The forwarding of mDNS/DNS-SD messages to the Ethernet interface is automatically handled by Avahi (i.e., pre-configured parameters

were `enable-reflector=yes` and `allow-point-to-point=yes`), while XMPP messages are directly forwarded to the XMPP server.

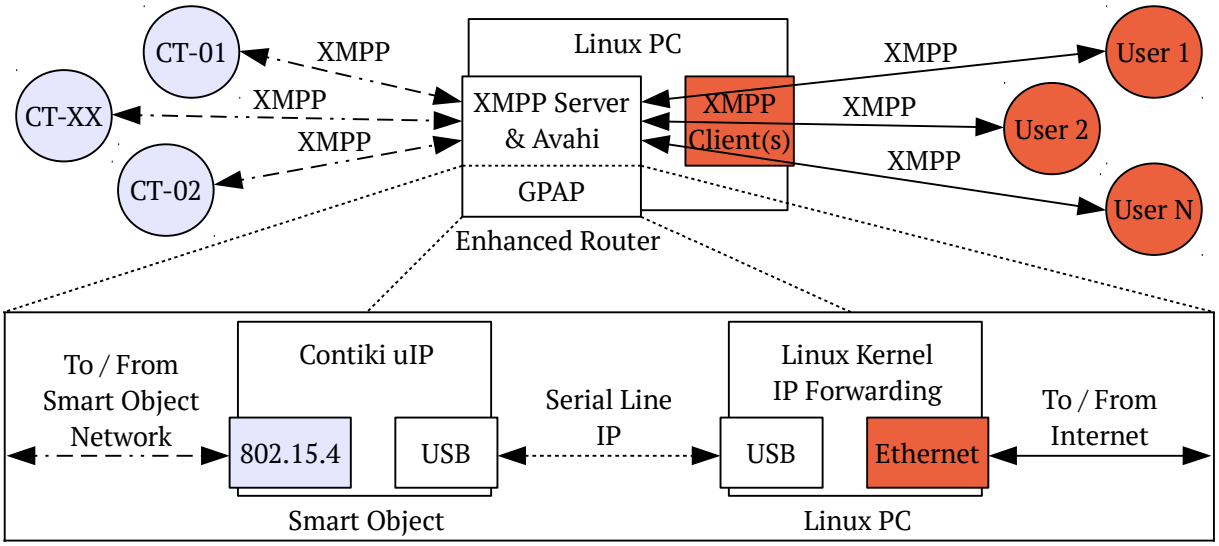

Figure 6.1: Connecting a smart object network to the Internet (adapted from [206])

For larger test setups, the COOJA network simulator (included in Contiki, cp. Section 2.2.1) was used. The advantage is that the behavior of the developed applications for MSP430-based hardware platforms can be directly simulated with different network topologies and without programming hundreds of smart objects because the native firmware image of these hardware platforms can be run in the COOJA simulator without any modifications on the source code [51, 207]. Simulation results can easily be reproduced with a wide range of network topologies and any number of nodes. In addition, we were able to use COOJA's integrated data logger for a detailed analysis of our experiments. To enable a direct interconnection from the COOJA simulator to the real world a simulation node had to be programmed with a 6LoWPAN border router that forwards all packets from COOJA via SLIP to the network of the host computer (e.g., Linux PC, cp. Figure 6.1).

All experiments were performed with Contiki on Zolertia Z1 and Tmote Sky hardware platforms that feature an IEEE 802.15.4-compliant Chipcon 2420 RF transceiver (cp. Section 2.2.3). The compiled firmware images for these devices differ only in the size of the Contiki operating system and not in the size of the compiled applications. This is caused by their used hardware components (e.g., integrated sensors, memory size) which need a different driver abstraction for each hardware platform. The respective firmware images of the applications were built with msp430-gcc (GCC) 4.4.5 [208]. If

not mentioned separately, ContikiMAC [49] is used for all smart objects, i.e., the default MAC layer in Contiki OS. The number and configuration of the used smart objects is described separately in the respective section of each tested component. Presented results are averaged over 100 runs.

6.2 uXMPP2 Evaluation

The evaluation of uXMPP2 consisted of two steps. First we determined the memory footprint of the XMPP stack for Contiki. Next we investigated the message optimization achieved with the *Temporary Subscription for Presence (TSP)*.

6.2.1 Memory Footprint

The memory footprint is very important for a lightweight and memory-efficient implementation of the protocol stack (see Appendix A.4). As the code footprint of a firmware image increases with additional function definitions and external calls (cp. [206]), we use only one single file for uXMPP2 to avoid external calls and to minimize function definitions as much as possible. Furthermore, we use several compiler flags[1] to reduce the firmware size of uXMPP2. The sizes of the implemented modules of uXMPP2 are listed in Table 6.1.

Table 6.1: Memory footprint of uXMPP2 for MSP430 (in Kbytes)

Component / Module	**ROM**	**RAM**
uXMPP2 Core/IM	4.14	0.18
uXMPP2 *XEP-0045* module	1.19	0
uXMPP2 *TSP* module	0.01	0
uXMPP2 *XEP-0174*	2.96	0.14
uBonjour (*OWT* enabled)	3.56	0.3
uXMPP2 Total	11.85	0.62

The uXMPP2 Core/IM takes only 4.14 Kbytes of ROM / 0.18 Kbytes of RAM. In comparison with the uXMPP v0.1 stack, which needs 12.42 Kbytes of ROM / 0.65 Kbytes

[1] Reducing Contiki OS' Firmware Size [Online] `http://www.sics.se/contiki/wiki/index.php/Reducing_Contiki_OS'_Firmware_Size`.

of RAM with less XMPP features, the memory footprint of the uXMPP2 Core/IM uses only a third of this size. This shows that the proposed architectural changes and optimizations of uXMPP2 allow a more efficient implementation. The memory use of the *XEP-0174*-based communication stack (including uBonjour) is 6.51 Kbytes of ROM / 0.44 Kbytes of RAM. Enabling all implemented features of uXMPP2 takes only 11.85 Kbytes of ROM and 0.62 Kbytes of RAM. Compared to implementations of REST-based approaches, such as uDPWS or CoAP (cp. Section 2.3), uXMPP2 is in the same range regarding the memory footprint. The current uDPWS implementation uses 10.03 Kbytes of ROM / 3.07 Kbytes of RAM on a Tmote Sky hardware platform, but this implementation misses some features of DPWS (cp. [92, 209]). The open source library *libcoap* requires about 12 Kbytes of ROM on the Contiki OS to implement the two basic CoAP operations *PUT* and *GET*, whereas the Contiki's CoAP example implementation *rest-server-example* uses from about 8.5 to 26 Kbytes of ROM without debug code and resource-specific handlers (cp. [68, 210]).

6.2.2 TSP Message Optimization

TSP (cp. Section 4.3) has been proposed to minimize the network traffic for Chatty Things in *XEP-0045 Multi-User Chat (MUC)* by using small presence messages and a publish-only affiliation. In order to prove this we measured the bootstrap time for a set of Chatty Things that join a chat room. In doing so the Chatty Things simultaneously exchange several XMPP messages, so that the effect of *TSP* can be measured by the reduction of the number of exchanged packets and the bootstrap time. The evaluation of *TSP* was performed with the COOJA network simulator. The setup consisted of simulation nodes with the uXMPP2 stack and a simulation node programmed with Contiki's 6LoWPAN border router implementation running in COOJA. The test runs were done with 2, 4, 6, 8, and 10 nodes booted at the same time. During each test run every node connected to the modified XMPP server Prosody (cp. Section 4.3.3) and joined the test room to sent a group chat message afterwards. Three different test runs were performed: *MUC* (no node ran with *TSP*), *TSP* (all nodes ran with *TSP*), and *Mixed* (50% of the nodes ran *TSP*, whereas the rest switched *TSP* off).

Number of Sent Packets during Bootstrap

It can be seen in Figure 6.2 that *TSP* significantly reduces the network traffic of Chatty Things during the bootstrap phase. In the scenario with 10 nodes the number of sent

packets reduces by 24.86% for the mixed setup and by 54.04% when *TSP* runs on all nodes. As reference, the number of exchanged messages in uXMPP v0.1 is indicated. uXMPP v0.1 requires that a separate IP packet is sent for each XML element (cp. Table 4.5). Since each

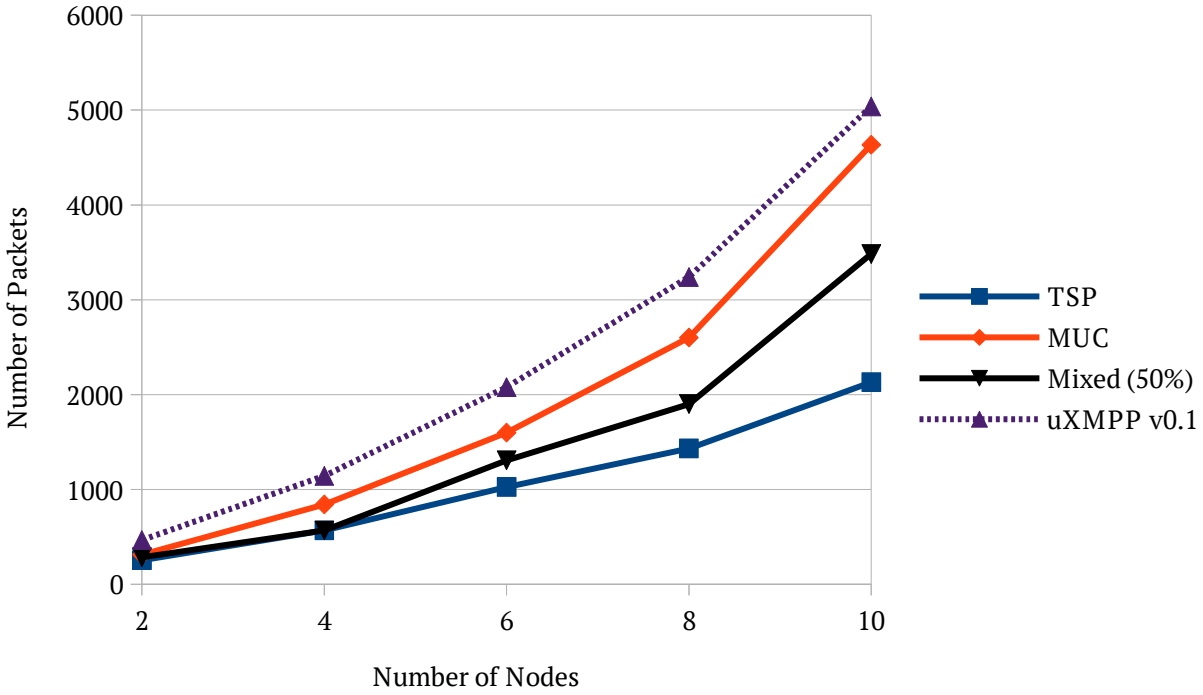

Figure 6.2: Network traffic comparison of *TSP*, *MUC*, and *Mixed* (in packets)

XMPP login requires a series of messages (at least 9 XMPP messages for *ANONYMOUS*), a large number of packets is exchanged during bootstrapping. This includes the XMPP acknowledge messages to the XMPP server and vice versa as well as all involved IP packets needed to establish and handle the TCP connections. After this point the network traffic for smart objects with enabled *TSP* will not increase further because these nodes not receive messages any more which are related to the topic (joined chat room).

Bootstrap Time

Figure 6.3 shows that the bootstrap time for nodes with enabled *TSP* is reduced because less packets need to be exchanged between the node and the XMPP server. In the 10 node scenario *TSP* reduces the bootstrap time by 20.07% for the mixed setup and by 46.73% for the *TSP* setup. The bootstrap time increases with the number of joining nodes because more packets are sent.

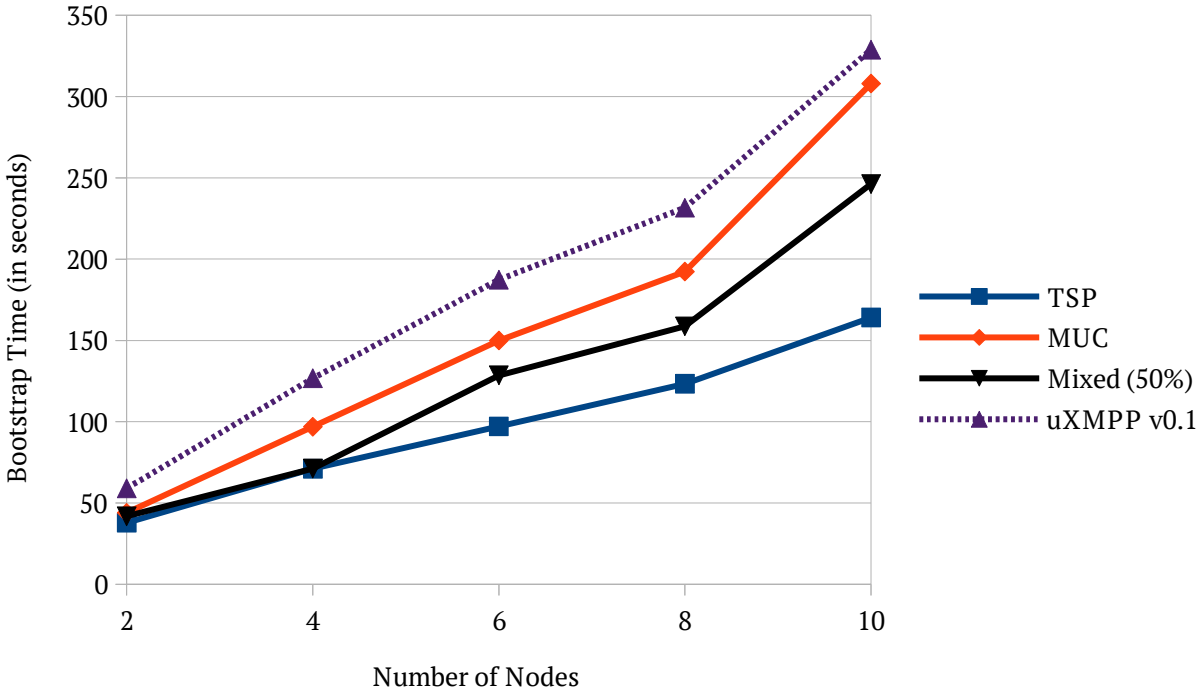

Figure 6.3: Bootstrap time for the *TSP*, *MUC*, and *Mixed* setup (in seconds)

The results show that *TSP* lowers the network traffic for Chatty Things without the need for XML compression techniques, while being standard-compliant to existing XMPP chat clients. The evaluation of the uXMPP2 stack with Contiki OS shows good results in terms of memory footprint, feature coverage, and bandwidth usage.

6.3 uBonjour Evaluation

In this section we evaluate the compatibility and the performance of uBonjour regarding memory footprint and response time for both IPv4 and IPv6.

6.3.1 Memory Footprint

uBonjour consists only of 1450 lines of code. Table 6.2 shows the detailed memory footprint. uBonjour including one service registration requires 3.5 Kbytes of ROM / 0.3 Kbytes of RAM for IPv4, and 3.56 Kbytes of ROM / 0.3 Kbytes of RAM for IPv6. As mentioned in Section 5.3.1, the *One-Way Traffic (OWT)* optimization significantly reduces the amount of used memory. It is cut into halves and the lines of code are reduced to around 1050.

Each additional service registration for uBonjour costs around 0.13 Kbytes (IPv4) and 0.14 Kbytes (IPv6) of ROM. These two values were both measured including the total sum of freely selectable parameters for the DNS records (cp. Section 5.3.2) and the memory for providing the needed storage structure.

Table 6.2: Memory footprint of all implemented stages of uBonjour with/without uIP stack for MSP430 (in Kbytes)

uBonjour	**ROM**	**RAM**
IPv4 / IPv6 Stage I	5.67 / 5.69	0.4
IPv4 / IPv6 Stage II	3.93 / 3.95	0.3
IPv4 / IPv6 Stage III (OWT)	3.5 / 3.56	0.3
IPv4 / IPv6 Stage I with uIP stack	15.46 / 25.24	1.6 / 3.26
IPv4 / IPv6 Stage II with uIP stack	13.71 / 23.5	1.55 / 3.21
IPv4 / IPv6 Stage III (OWT) with uIP stack	13.28 / 23.11	1.45 / 3.21

The difference in the memory footprint of uBonjour between IPv4 and IPv6 is small because the two variations just differ in the IP address lengths (16 bytes for IPv6 versus 4 bytes for IPv4 addresses). Minimal larger buffers for sending and storing registered services are therefore needed with IPv6. This is not supported though by the uIP stack in general: the uIPv6 stack is two times larger in RAM and twice as large in ROM consumption compared to its IPv4 counterpart. This means that for the Zolertia Z1 nearly half and for the Tmote Sky around 85% of ROM is allocated by the uIPv6 stack and the Contiki OS alone (see Figure A.1). A slim and memory-efficient implementation of uBonjour is therefore even more important for IPv6 than for IPv4. In general, uBonjour runs on Contiki-based smart object hardware platforms with a ROM size of at least 48 Kbytes (e.g., Tmote Sky, cp. Section 2.2.3) for both IPv4 and IPv6.

6.3.2 Response Time

The test setup for IPv6 uses a one-hop network with static routes (cp. Section 6.1). Two additional smart objects (i.e., hops two and three) running uBonjour complete the IPv6 setup, as depicted in Figure 6.4. The IPv4 test setup consists of a smart object running uBonjour that is directly connected via SLIP to a computer. We monitored incoming DNS

packets with Wireshark[2] for the two cases to verify the correctness of the DNS records generated by the smart object, and to monitor the interaction between the computer (i.e., Linux PC) and the smart objects.

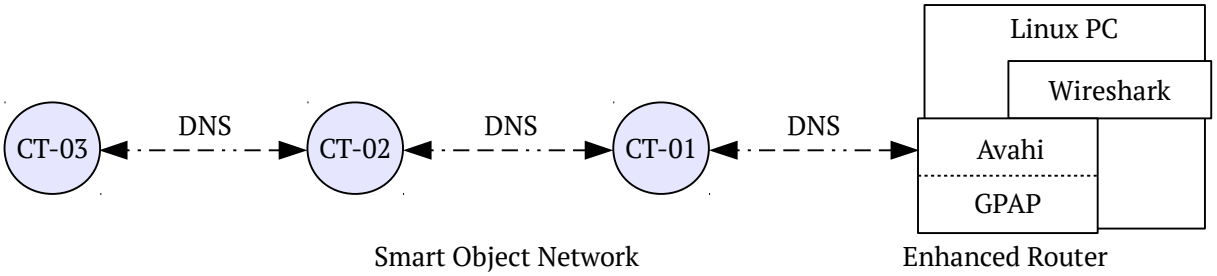

Figure 6.4: IPv6 multi-hop scenario of uBonjour

The response time was measured by sending a `PTR` record to the multicast group. It is the time between the sending of the record and the reception of all DNS records of the respective smart objects. uBonjour does not support specific optimizations and an active forwarding of DNS messages. Multi-hop routing is handled by Contiki's IP stack and depends on the performance of its used lower layers [211]. The average response times for a set of multi-hop scenarios with enabled *Adjustable DNS Message Compression (ADMC) (Enhanced)* and disabled *ADMC (Enhanced)* are listed in Table 6.3.

Table 6.3: Response time (in ms) of uBonjour

IP Packet(s)	**IPv4 SLIP**	**IPv6 1-Hop**	**IPv6 2-Hops**	**IPv6 3-Hops**
1 (*ADMC Enh.*)	-	346 ms (72%)	781 ms (60%)	1226 ms (47%)
1 (*ADMC*)	-	600 ms (51%)	1008 ms (48%)	1488 ms (36%)
3 (*ADMC*)	-	776 ms (37%)	1292 ms (34%)	1706 ms (27%)
4 (*ADMC*)	43 ms (39%)	1028 ms (17%)	1420 ms (27%)	2196 ms (6%)
4 (No *ADMC*)	71 ms	1233 ms	1954 ms	2324 ms

Service discovery with uBonjour takes 43 ms with and 71 ms without DNS name compression for directly connected smart objects over SLIP for IPv4. In IPv6 scenarios a 6LoWPAN border router is mandatory, hence packets are always delayed by one-hop. The measured response times for multi-hop with en-/disabled *ADMC (Enhanced)* are: 346 ms to 1233 ms for a one-hop scenario, 781 ms to 1954 ms for a two-hop one, and 1226 ms to 2324 ms

[2]Wireshark network protocol analyzer [Online] `http://www.wireshark.org/`.

for three-hop scenarios. The outcome shows that *ADMC* and especially *ADMC Enhanced* significantly reduce the response time in all scenarios (IPv4 / IPv6). The highest reduction of the response time (e.g., the average hop savings are around 59% for *ADMC Enhanced* and around 45% for *ADMC*) is reached when all required DNS records can be integrated into a single DNS message. Less IP packets and thus less IEEE 802.15.4 radio frames need to be sent if all repetitions can be replaced within a DNS message. Furthermore, using only DNS name compression in a DNS resource record reduces the response time and lowers the network traffic for low data rate smart object networks as well (in comparison to four IP packets with *ADMC*) because less data is transmitted per IP packet. Comparing the response times of IPv6 with IPv4 over SLIP shows that the 6LoWPAN border router is responsible for a certain amount of latency for IPv6 which cannot be avoided. Figure 6.5 visualizes all results of the measured values for the response time of uBonjour.

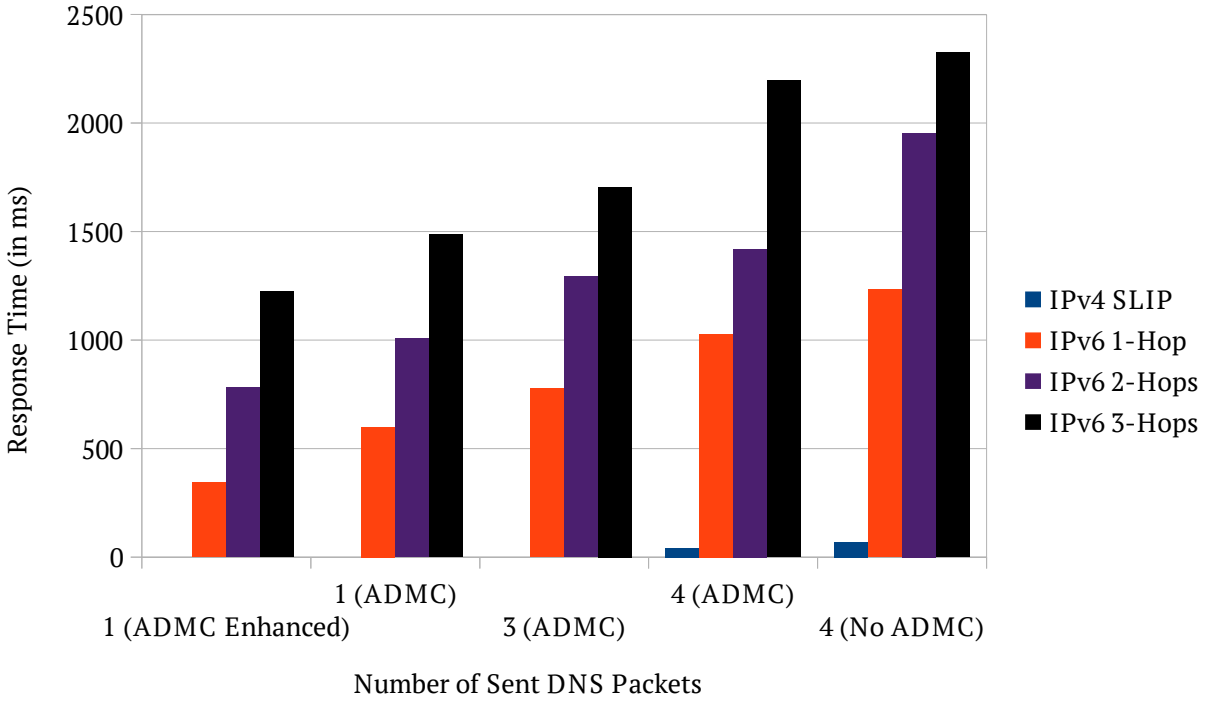

Figure 6.5: Comparison of the response time (in ms) of all IP scenarios

The evaluation of *ADMC* and *ADMC Enhanced* shows that the proposed DNS message compression methods are efficient enough for reducing the overhead of DNS messages in smart object networks (e.g., 6LoWPAN). Furthermore, the implementation of *ADMC (Enhanced)* can be used for a variety of smart object hardware platforms because *ADMC (Enhanced)* always combines several compression methods for uBonjour depending on the available IP payload size, e.g., DNS name compression, class code and TTL field

compression, and IP address reconstruction. As the class code / TTL field compression and the IP address reconstruction are not restricted to smart objects, these optimizations can easily be integrated into the current DNS standards.

6.4 XMPP Layer Performance

Finally we prove the seamless integration of Chatty Things into a real world IoT testbed and evaluate the performance of the XMPP layer regarding the distribution time of events. The latter allows a direct performance comparison between different application protocols. Furthermore, we can show XMPP's efficiency and scalability for the push of information which is used by the sensor-specific grouping approach. The test consists of three parts. The first part introduces the testbed and evaluates the seamless integration of Chatty Things into current Internet infrastructures. The second one measures how much time is needed to distribute an event (e.g., change of a sensor value) from a smart object to a user (i.e., into a computational network). The third part measures how much time and resources are needed to distribute an event from the computational network (e.g., XMPP server) to a set of concurrent users.

6.4.1 Seamless Integration of Chatty Things

The seamless integration of Chatty Things into the Internet is evaluated by means of the *ACDSense* scenario introduced in [218]. ACDSense is an interorganizational network scenario for sensor data exchange spanning the three universities RWTH Aachen University, TU Dresden, and BTU Cottbus–Senftenberg. Each site contributed heterogeneous sensor implementations and developed different types of client applications for accessing and controlling sensors, ranging from standard general-purpose XMPP clients over custom mobile apps (i.e., Dresden) to web-based widget dashboards (i.e., Aachen). The used sensor implementations were commodity hardware with a USB-connected thermometer at Aachen, a sensor data generator using historical weather data at Dresden, and Chatty Things at Cottbus. We used the setup described in Section 6.1 to seamlessly integrate Chatty Things into the ACDSense scenario. The three sites are connected over the Internet through their hosted XMPP servers using inter-domain communication, as depicted in Figure 6.6. The pushing of sensor data in ACDSense follows the proposed sensor-specific grouping approach: events are grouped by *XEP-0045 Multi-User Chat (MUC)* rooms and

are pushed simultaneously to many users (cp. Section 3.3). Thus, access to the sensor data is possible with any XMPP client using any XMPP server as rendezvous point of the ACDSense scenario.

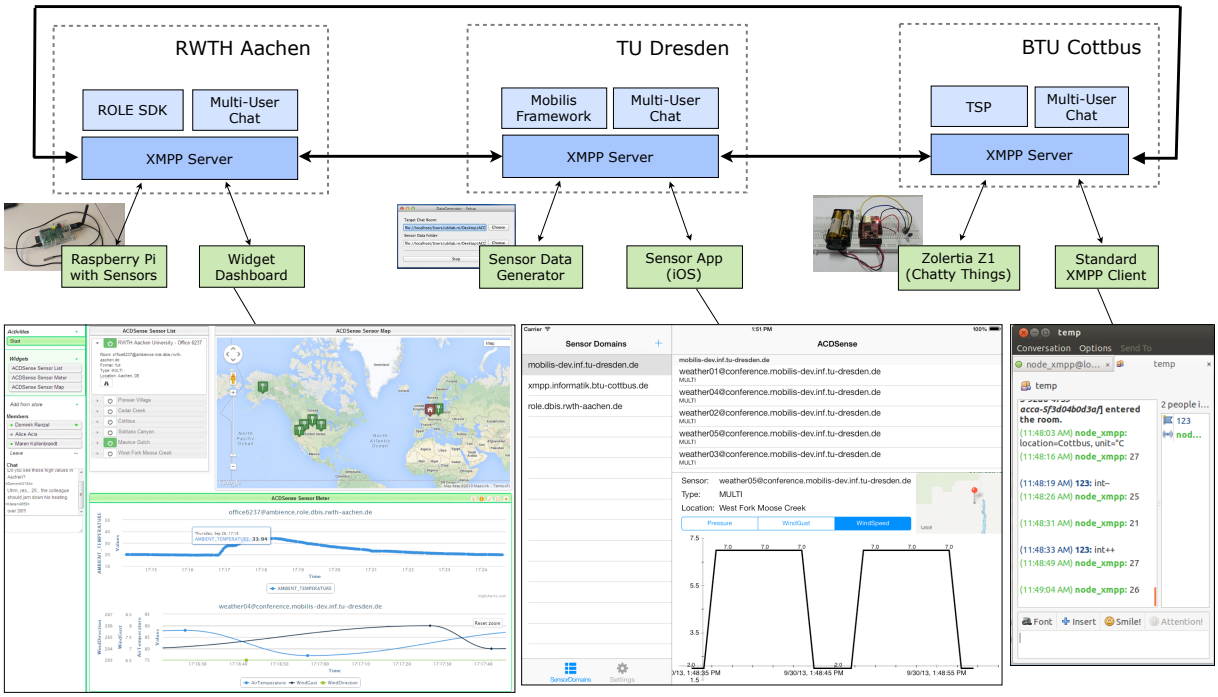

Figure 6.6: Architecture of the ACDSense scenario and the integration of Chatty Things ([218])

The whole scenario was set up within four weeks and then tested and evaluated over several weeks. Most work of the setup related to administrative challenges for the inter-domain communication (for details see to [218, Sec. VI.D]). To sum up, the ACDSense scenario proves that Chatty Things can seamlessly be integrated into XMPP infrastructures. Furthermore, we proved that the sensor-specific grouping approach as part of the XMPP layer can easily be implemented and efficiently be used by different XMPP client and server implementations.

6.4.2 Completion Time

A typical benchmark for the comparison of the performance of application protocols is the completion time which measures the time between requesting a service and getting the sensed data (e.g., temperature) in a response message (cp. [42, Sec. 6.3]). As reference systems for uXMPP2, we chose two REST-based approaches: the RESTful web services

of [42] and uDPWS [209], as introduced in Section 2.3. The three approaches have in common that a (single) standardized protocol at the application layer (i.e., HTTP or XMPP) is used to interconnect different classes of devices ranging from ordinary computers to smart objects. Differences exist in the notification of the events: HTTP uses the request-response mechanism, XMPP the publish-subscribe paradigm. For uXMPP2, this means that updates of the sensed data are directly pushed to the interested devices without the need of dedicated request messages. Additionally, CoAP is used as a reference protocol because of its proclaimed very low message overhead and reduced response times for M2M communication in smart object networks, but it is not treated as an equivalent reference system (i.e., it requires HTTP for a full integration into the Internet, cp. Section 2.3.2).

In order to compare the completion time of uXMPP2 with that of the other approaches we used the built-in temperature sensor of the Zolertia Z1 to generate a comparable message data payload and service characteristic (e.g., temperature reception). In addition, we reconstructed the test scenarios of uDPWS [209] and RESTful web services [42, Sec. 6.3] to be compliant: one-hop network (IPv6), while one Zolertia Z1 runs the 6LoWPAN border router and is connected via SLIP to the computer (cp. Section 6.1). The Chatty Thing pushes its sensed temperature data to the chat room with the sensor-specific topic *'temp'*. The results of the performance test are compared with the REST-based approaches in Table 6.4. The completion time of uXMPP2 was measured for the presence status, the one-to-one chat, and the group chat message exchanges to give a general performance overview of the most used XMPP XML message stanzas. The exchange of the group chat message requires 3, the one-to-one chat message requires 2, and the presence message requires 1 TCP/uIP packet(s). For the tests with CoAP, only requests of a CoAP client[3] outside the smart object network to smart objects running CoAP server instances[4] were used (i.e., delay times of the CoAP/HTTP gateway are not included). Note that the message sizes of uDPWS include TCP and lower layer header sizes, while its completion time was measured under Contiki OS version 2.3. The code of uDPWS is not compatible with newer Contiki OS versions and thus the completion time for uDPWS cannot be measured with currently available MAC layers, such as ContikiMAC [49]. All measurements for uXMPP2 and CoAP were done with a disabled MAC layer (i.e., *nullmac*[5]) to measure only the raw performance of the application protocol without the influence of the lower layers and with ContikiMAC (i.e., values in brackets for completion time in Table 6.4).

[3] Copper (Cu) CoAP user-agent for Firefox.
[4] Contiki REST example with enabled CoAP.
[5] „A MAC protocol that does not do anything.", nullmac.c,v 1.15 2010/06/14 19:19:17 Adam Dunkels.

Table 6.4: Comparison of the XMPP layer for the IoT performance with REST-based approaches (publish-subscribe versus request-response)

Approach	**Request Size**	**Response Size**	**Completion Time**
uDPWS	650 bytes	751 bytes	343 ms
RESTful Web Services	85 bytes	141 bytes	440 ms
CoAP	15 bytes	19 bytes	54 ms (211 ms)
uXMPP2 (Group Chat)	-	144 bytes	200 ms (540 ms)
uXMPP2 (One-to-One Chat)	-	96 bytes	122 ms (362 ms)
uXMPP2 (Presence)	-	35 bytes	38 ms (121 ms)

The results of Table 6.4 show that the uXMPP2 stack is very competitive in comparison to the considered reference systems. On the one hand, the use of publish-subscribe in uXMPP2 saves data packets for avoiding the request message and time for its transfer between two entities. On the other hand, XMPP messages of the size of the presence message allow a very low completion time which is compared to the CoAP completion time[6] (i.e., for the integration into existing Internet infrastructure the delay time for a CoAP/HTTP gateway needs to be added) a really good value. This means for XMPP if a stream compression method can be used which is able to compress XML stanzas to the available payload size of a single TCP/uIP packet then XMPP messages can in general be exchanged with a very low completion time in smart object networks. Such a compression method for the efficient interchange of XML stanzas (i.e., EXI) is currently being developed as uEXI [177, 179] for smart objects. Therefore, the use of EXI in Chatty Things might significantly decrease the number of exchanged TCP/uIP packets and the completion time.

6.4.3 Efficiency of Information Distribution

The event distribution of the sensor-specific grouping approach depends on the performance of the XMPP server because the server is responsible to push a group chat message to

[6]CoAP *observe* (server push of notifications) can further reduce the completion time between CoAP instances. This was not considered, since it is a unique feature of CoAP and server push is not available for HTTP/1.1 (cp. [80]) or compatible to other workarounds (cp. Section 2.3.1).

all XMPP clients that joined the same chat room. This test evaluated the effect of the number of joined XMPP clients per chat room (i.e., 10, 100, and 1000) in terms of message delivery time and message throughput. The XMPP server (i.e., Openfire [137]) hosting the chat room ran on a dedicated physical machine (i.e., Intel i7 Quadcore). A second machine emulated the data sender and a large number of XMPP clients (i.e., Smack XMPP library [144]). The exchanged sensor data consisted of weather data from [212] and the average message size was 385 bytes. The test results are listed in Table 6.5.

Table 6.5: Efficiency of distributing information via chat rooms ([218])

Measurement	**10 Clients**	**100 Clients**	**1000 Clients**
Message Delivery Time	2.5 ms	12.1 ms	103.9 ms
Delivered Messages per Second	116	1171	11644

The increase in the message delivery time is proportional to the total number of messages sent per second. For the transport, an impressive workload of 42 Mbit/s upstream and 4.1 Mbit/s downstream was measured in the case of 1000 joined clients. The CPU load of the XMPP server ran at 12-18% in this test. This proves the high scalability of the sensor-specific grouping approach because chat rooms can be used for the efficient distribution of events from one XMPP entity to many entities.

7 Conclusions and Future Work

This thesis has presented an approach for the seamless integration of smart objects into the Internet under Human-to-Machine (H2M) communication aspects, called Chatty Things, that uses established and standardized application protocols to make the Internet of Things (IoT) readily deployable. The IoT describes a technological progress in which smart objects connect the real world with the Internet to provide services that can be used by human beings to directly interact with their physical environment. Current application protocols for the IoT focus on the Machine-to-Machine (M2M) communication and introduce dedicated protocols for smart objects requiring specialized protocol gateways, smart object-specific code or data representations that hinder a seamless integration. The Chatty Things approach uses as a standardized application layer XMPP and mDNS/DNS-SD for IP-based smart objects and their integration into the Internet without the need of application protocol gateways [213, 214, 215, 218, 219, 221, 222, 223]. This allows the use of established Internet mechanisms and software human beings are familiar with for the H2M interaction.

The implementation of the XMPP layer for smart objects requires a modular and minimized XMPP stack. For this, we have designed the optimized XMPP stack for low data rate networks uXMPP2. The characterizing features of this stack are: a readily usable API, an essential set of XMPP Extension Protocols (XEPs) (i.e., sensor-specific groups), a proposal for lightweight and user-friendly event notification (i.e., *TSP*), a parameter-less bootstrapping, and a seamless fallback mechanism for ad hoc use cases when infrastructure services are failing for Chatty Things. The evaluation of the uXMPP2 stack with Contiki OS shows good results regarding a low memory footprint and a high compatibility to existing XMPP software and infrastructures. The uXMPP2 Core/IM takes only 4.14 Kbytes of ROM and 0.18 Kbytes of RAM. Our experiences with uXMPP2 result in an open source implementation for the Contiki OS version 2.6. It reimplements the uXMPP2 API methods (cp. Appendix A.1) besides the *XEP-0174* client component. The source code is available online at [224].

The proposed sensor-specific grouping approach restricts the advertised services of a smart object to its integrated sensors and directly maps its capabilities to *XEP-0045 Multi-User Chat* rooms to support a slim implementation for filtering and grouping sensor data. To further reduce the network traffic for Chatty Things in chat rooms we have developed and evaluated the *Temporary Subscription for Presence (TSP)* approach. *TSP* introduces a publish-only affiliation for *XEP-0045* and uses small presence messages that fit into one single IP packet to signalize topic-related status updates and events of smart objects to subscribed XMPP entities in the whole network through temporary presence subscription. It provides an XMPP-compliant message reduction without using complex and costly techniques like XML compression. Thus, *TSP* introduces two stages in *XEP-0045* for pushing information. Stage I uses only presence messages and dynamically reduces the exchange of messages to a minimum if everything proceeds normally. Stage II allows users and objects to request detailed information about the event via remote commands.

In order to support the self-configuration of smart objects we further have developed a discovery service for the IoT based on mDNS and DNS-SD, called uBonjour [219, 221]. It allows one to autonomously integrate smart objects into a given network environment without requiring hard-coded bootstrap parameters. Thus, smart objects can more precisely react on topology changes and to joining or leaving network devices. Finding an XMPP server using mDNS/DNS-SD was developed as part of this thesis under the term parameter-less bootstrapping for Chatty Things. The research results of the parameter-less bootstrapping were proposed in Section 3.3.2 of *XEP-0347 Internet of Things - Discovery* [223]. The aim of *XEP-0347* is to standardize and simplify the installation, the configuration, the discovery, and the connection of XMPP-driven smart objects. If no XMPP server can be discovered in the given network during bootstrapping, Chatty Things use *XEP-0174 Serverless Messaging* as fallback mechanism. Therefore, we have implemented the first *XEP-0174* compatible XMPP client on smart objects to provide a Peer-to-Peer (P2P) communication between a computational device and a nearby Chatty Thing for ad hoc use cases. The *XEP-0174* client component of uXMPP2 takes 2.96 Kbytes of ROM / 0.14 Kbytes of RAM and uBonjour uses 3.56 Kbytes of ROM / 0.3 Kbytes of RAM. Our experiences with uBonjour in implementing DNS-SD on IP-based smart objects resulted in a rewrite of the DNS-SD core functionalities for the Contiki OS version 2.6. It uses the uIP hostname resolver function of the merged mDNS code of [193]. The source code can be found online at [225].

Moreover, we have proposed an enhanced DNS message compression method for the efficient use of mDNS/DNS-SD-based service discovery in low data rate smart object

networks. In the first step, we implemented the standardized DNS message compression for uBonjour as *Adjustable DNS Message Compression (ADMC)* that significantly reduces the response time by integrating all four DNS resource records into a single IP packet for most smart object hardware platforms (with support of an IP payload size larger than 140 bytes). In the second step, further optimizations were investigated to reduce the length of a single DNS message including all required DNS resource records for smart object hardware platforms with a very small IP payload size but without breaking standards. In this way, we have introduced two enhanced message compression methods for DNS (i.e., the class code / TTL field compression, and the IP address reconstruction) and implemented both as *ADMC Enhanced* for uBonjour. The two enhanced approaches ensure a high compatibility level to existing DNS implementations and can simply be adopted by RFC 1035 because these methods are indicated by pointer flags reserved for future use.

To sum up, we have proved the seamless integration of smart objects into the Internet using XMPP and mDNS/DNS-SD. The Chatty Things approach enables a transparent (H2M) interaction and service discovery for the IoT that allows a standardized integration with low effort into the current Internet infrastructure. Inexperienced users can access Chatty Things through ordinary notebooks or computers with any familiar and available XMPP software at the application layer. Smart objects can thus be simply deployed and handled comparable to consumer electronics. Through the implementation and evaluation of the minimized XMPP and mDNS/DNS-SD stacks for Contiki, we showed that these protocols run on smart object hardware platforms with a ROM size of at least 48 Kbytes. *TSP* and the enhanced DNS message compression methods demonstrate that existing standards can economically be used in low data rate networks without introducing smart object-specific code or data representations, while providing (a high level of) compatibility (i.e., they are not restricted to smart objects).

The results presented in this thesis have shown that established standards at the application layer can boost the handling/interaction with smart objects and their integration into the Internet. In order to support encrypted communication for the uXMPP2 stack future work should focus on the provision of efficient implementations of security standards, such as TLS, on all classes of constrained smart object hardware. Using security requires additional memory and code size. Ongoing work of implementing TLS on smart objects show that the required memory footprint exceeds the available memory size of class 0 devices (e.g., 48 Kbytes of ROM for Tmote Sky, cp. [39, 115, 116]). Therefore, it has to further be investigated how and which functionality of (established) security mechanisms can efficiently be used in low data rate smart object networks and how their respective

implementations can take only a very low memory footprint. Another important focus for the seamless integration of smart objects into the Internet should be to enable a standardized service discovery that operates over existing networks and that directly addresses different classes of devices with the same Internet mechanisms. DNS-SD may be a solution for the IoT as the results of this thesis have demonstrated. Drawback of a DNS-based service discovery for the IoT is that its current design misses features of handling smart objects that are offline for certain periods (e.g., these devices fail to respond to queries) or to provide the discovery functionality beyond local network boundaries. The extensions for Scalable DNS Service Discovery proposed by the Internet Engineering Task Force (IETF) working group will focus on these aspects (cp. [199]).

List of Figures

List of Tables

Listings

A Appendix

A.1 Implemented uXMPP2 API Methods

XMPP Core/IM. Implements most parts of an XMPP client, such as client-to-server XML streams [22, Sec. 4, 8.1 and 8.2], TLS and SASL negotiation [22, Sec. 5 and 6], exchanging presence and messages [124, Sec. 4.2.1, 4.4.1, 4.5.1, and 5].

XEP-0045 Multi-User Chat. Supports the 'occupant use cases' entering a chat room via groupchat 1.0 protocol [133, Sec. 7.2.1] and presence broadcast [133, Sec. 7.2.3], sending a group chat message to all room members [133, Sec. 7.4] and leaving a chat room [133, Sec. 7.14].

XEP-0050 Inspired Remote Commands. Does not implement the standardized protocol flow (i.e., uses iq stanzas), instead it provides the sending of a set of determined commands (i.e., specified by the application developer) to Chatty Things through the use of message stanzas.

XEP-0174 Serverless Messaging. Implements discovering other XMPP entities [25, Sec. 4], exchanging presence [25, Sec. 5] and stanzas [25, Sec. 7], XML stream initiation [25, Sec. 6] and closing [25, Sec. 8], going offline [25, Sec. 9].

A.2 Implemented uBonjour API Methods

Resolving Hostnames. To discover the address of another device in the network a device needs to send a DNS query for the domain name to the multicast group of the network. The device with the corresponding domain name replies with an `A` record including its network address. If the hostname can be resolved uBonjour broadcasts an event to all listening processes with the IP address, otherwise a timeout for the query is triggered. Only one name can be resolved at the same time. Sending a new query at the same time stops the ongoing search for a hostname and starts a new search with the currently submitted hostname.

Discovering Services. The service discovery works analogously to the resolving of a hostname. A device initiates the service discovery by sending a `PTR` record to the multicast group containing the name of the searched service. If a service query is resolved uBonjour posts an event to all processes with `PROCESS_BROADCAST` containing the resolved IP address and the port as data.

Registration, Removal, and Update of Services. To publish an available service a smart object has to send four DNS records as described in Section 5.1. Each application running on a device has to register a service with its service name, IP address (provided by Contiki), and port, if it wants to be found in the network by other devices. If a `PTR` query arrives, the corresponding device replies with one `SRV`, `TXT`, `A` or `AAAA`, and `PTR` record. To remove a service from a network the device needs to send a `PTR` record with the Time-To-Live (TTL) set to zero. uBonjour API also supports updating an already published service by resending the four DNS records with changed data. uBonjour can handle up to eight service registrations per device by default. This value can be adjusted to the memory size of the specific device.

A.3 Byte Counting of the DNS Record Length

In DNS a name has to be split into a sequence of label lengths and then labels. A label length (LL) counts 1 byte, while each label is taking 1 byte per character. The length of a pointer (P) takes only 2 bytes. Pointers are distinguished from a label by setting the first two bits to ones (pointer flag), whereas labels must start with two zero bits (cp. [200, Sec. 4]). The fields with fixed variable lengths (FVL) (e.g., type, class, TTL, resource length) take each 10 bytes. In a `SRV` record the prio/weight/port (PWT) field takes 6 bytes. The length of a byte sequence is calculated with the following function d, whereas x is the input of a sequence (e.g., string) and N the length of x: d(x) = N (in bytes).

A.3.1 Example Calculation of the Minimal DNS Record Length

This subsection describes the calculation of the third value of the length values (in bytes) summarized in Table 5.1.

d(SRV) = d(P) + d(FVL) + d(PWT) + d(P) = 20
d(PTR) = d(P) + d(FVL) + d(P) = 14
d(TXT) = d(P) + d(FVL) + 13 = 25
d(A) = d(P) + d(FVL) + 4 = 16
d(AAAA) = d(P) + d(FVL) + 16 = 28

A.3.2 Example Calculation of the DNS Record Length Using Name Pointers Only Within a Record

This subsection describes the calculation of the second value of the length values (in bytes) summarized in Table 5.1. The free space for the length of the service type is named in the following *FS4ST* and takes 0 bytes.

d(SRV) = d(LL) + d(*con*) + d(LL) + d(FS4ST) + d(LL) + d(*_tcp*) + d(LL) + d(*local*) + d(LL) + d(FVL) + d(PWT) + d(LL) + d(*ctk*) + d(P) = 39
d(PTR) = d(LL) + d(FS4ST) + d(LL) + d(*_tcp*) + d(LL) + d(*local*) + d(LL) + d(FVL) + d(LL) + d(*con*) + d(P) = 29
d(TXT) = d(LL) + d(*con*) + d(LL) + d(FS4ST) + d(LL) + d(*_tcp*) + d(LL) + d(*local*) + d(LL) + d(FVL) + 13 = 40

d(A) = d(LL) + d(*ctk*) + d(LL) + d(*local*) + d(LL) + d(FVL) + 4 = 25
d(AAAA) = d(LL) + d(*ctk*) + d(LL) + d(*local*) + d(LL) + d(FVL) + 16 = 37

If we assume that the substring `local` in the `A` and `AAAA` records can be replaced by a name pointer:

d(A) = d(LL) + d(*ctk*) + d(P) + d(FVL) + 4 = 20
d(AAAA) = d(LL) + d(*ctk*) + d(P) + d(FVL) + 16 = 32

A.3.3 Example Calculation of the Highest DNS Message Compression Ratio

This calculation uses the precalculated values of Sections A.3.1 and A.3.2. The first record that needs to be integrated uses its precalculated value of Section A.3.1, whereas the rest of the records use the precalculated values of Section A.3.2 because each occurrence of name can be replaced completely by a name pointer. This is shown in the following using the `SRV` record as the beginning record:

d(IPv4) = d(SRV) + d(PTR) + d(TXT) + d(A) = 39 + 14 + 25 + 16 = 94
d(IPv6) = d(SRV) + d(PTR) + d(TXT) + d(AAAA) = 39 + 14 + 25 + 28 = 106

A.3.4 Example Calculation of the Minimal DNS Message Compression Ratio

This calculation uses the precalculated values of Sections A.3.1 and A.3.2. The first record that needs to be integrated uses its precalculated value of Section A.3.1, whereas the second record uses the precalculated values of Section A.3.2. Cases if all occurrences of names can be replaced completely by a name pointer for the second record:

d(SRV) + d(PTR) = 39 + 14 = 53
d(SRV) + d(TXT) = 39 + 25 = 64
d(SRV) + d(A) = 39 + 16 = 55
d(SRV) + d(AAAA) = 39 + 28 = 67
d(PTR) + d(TXT) = 29 + 25 = 54

Cases if only occurrences of a substring (e.g., `local`) can be replaced by a name pointer for the second record:

d(PTR) + d(A) = 29 + 20 = 49
d(PTR) + d(AAAA) = 29 + 32 = 61
d(TXT) + d(A) = 40 + 20 = 60
d(TXT) + d(AAAA) = 40 + 32 = 72

A.3.5 Example Calculation of the Redundant Information Filtering

The minimal length of a single DNS message for appending redundant information (e.g., IP address, user-defined text) to the resource data of the `SRV` record (including a delimiter like the resource length field, i.e., 2 bytes):

d(IPv4) = 39 + 2 + 13 + 2 + 4 = 60
d(IPv6) = 39 + 2 + 13 + 2 + 16 = 72

Send only a `SRV` record with the appended IP address:

d(IPv4) = 39 + 2 + 4 = 45
d(IPv6) = 39 + 2 + 16 = 57

Appending the user-defined text of the `TXT` record to the `SRV` record:

d(IPv4/6) = 39 + 2 + 13 = 54

A.3.6 Example Calculation of ADMC Enhanced

The highest DNS message compression ratio of IPv6 (cp. Section A.3.3) is reduced by using three times the class code and TTL pointer (i.e., saves 4 bytes each time) and the IP pointer (i.e., saves 14 bytes):

d(IPv6) = 106 - 3 ∗ 4 - 14 = 80

A.4 IPv6 Stack Memory Footprint

The uXMPP2 implementation consists of different modules which again use different parts of Contiki's uIP stack. XMPP Core/IM uses TCP to establish connections, the *XEP-0174* client uses TCP to listen for incoming connection requests, and uBonjour uses UDP to send and receive multicast DNS messages. This results in a diverse memory use for the uIP stack (shown in Table A.1) because each requirement enables different functions of the uIP stack. Enabling all components of uXMPP2 on Contiki results in a total memory consumption of 21.18 Kbytes of ROM / 3.76 Kbytes of RAM for the IPv6 stack, i.e., 1.63 Kbytes of ROM and 0.87 Kbytes of RAM are thus additionally used for TCP functions when compared to running only an UDP/IP stack (e.g., uBonjour). For the parameter-less bootstrapping of Chatty Things this means that the IPv6 stack memory footprint increases and leaves less memory for the uXMPP2 stack at run-time because the functions of uIP can only be enabled during compile time.

Table A.1: Memory footprint of the IPv6 stack in Contiki (in Kbytes)

Connection Type / Component	**ROM**	**RAM**
UDP / uBonjour	19.55	2.89
TCP (Listen) / uXMPP2 *XEP-0174*	19.50	3.58
TCP (Connect) / uXMPP2 Core/IM	20.40	3.08
All	21.18	3.76

In general, UDP consumes less memory compared to TCP, although the difference between using TCP listeners and TCP connectors are small (0.9 Kbytes of ROM / 0.5 Kbytes of RAM). The reason lies in the different buffer management for sending and receiving messages. A UDP-based communication is handled efficiently and dynamically through the IP stack, whereas a TCP connection needs additional static and application-related buffers, i.e., 150 bytes for uXMPP2.

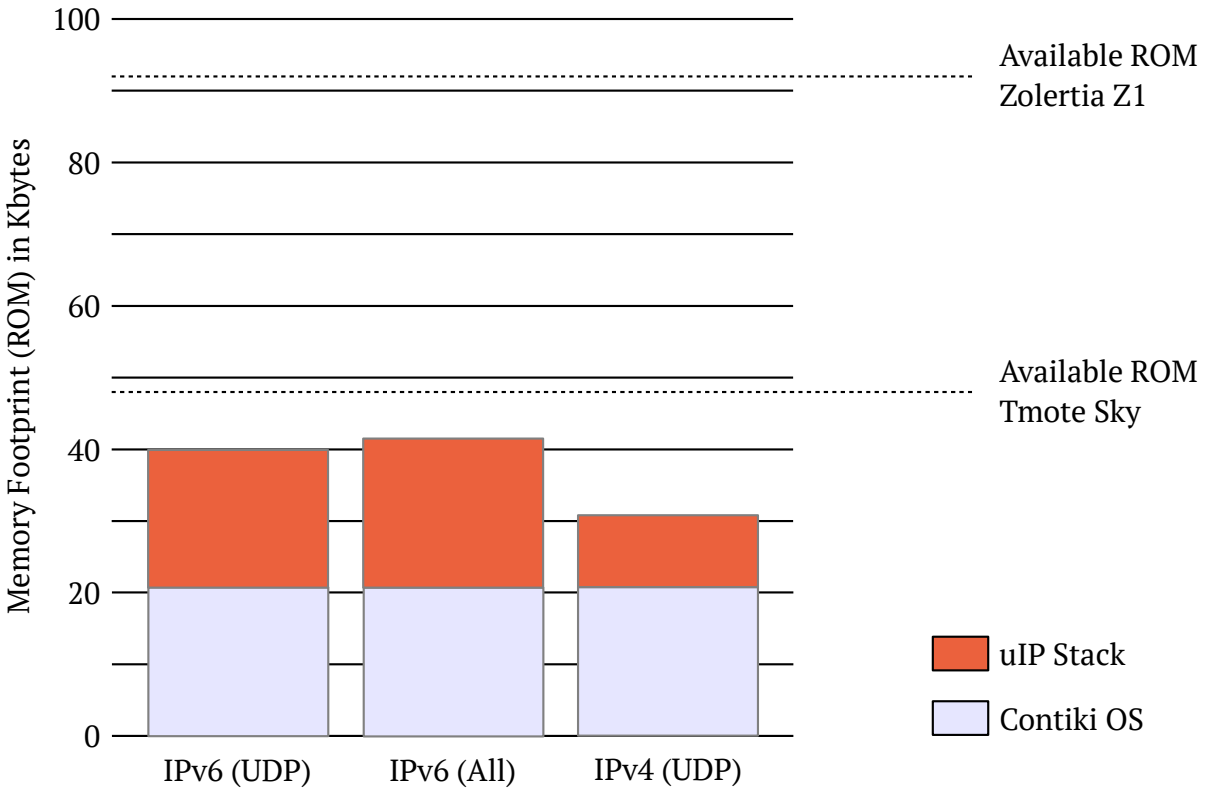

Figure A.1: ROM usage of uIP running with different configurations (in Kbytes)

Acronyms

6LoWPAN	IPv6 over Low power Wireless Personal Area Network
A	Address record for IPv4
AAAA	Address record for IPv6
ADMC	Adjustable DNS Message Compression
API	Application Programming Interface
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interchange
AVR	Microcontroller family from Atmel
BXML	Binary XML
BSD	Berkeley Software Distribution
CoAP	Constrained Application Protocol
CoRE	Constrained RESTful Environments
CPU	Central Processing Unit
CRAM	Challenge Response Authentication Mechanism
CT	Chatty Thing
DPWS	Devices Profile for Web Services
DNS	Domain Name System
DNS-SD	DNS Service Discovery
DOM	Document Object Model
DTC	Distributed TCP Caching
DTLS	Datagram Transport Layer Security
EXI	Efficient XML Interchange
EXIP	Embeddable EXI implementation in C
FI	Fast Infoset
GCC	GNU Compiler Collection
GPAP	General Purpose Access Point
H2H	Human-to-Human
H2M	Human-to-Machine
HTTP	Hypertext Transfer Protocol

IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IM	Instant Messaging
IoT	Internet of Things
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
IPHC	IPv6 Header Compression
IPSec	Internet Protocol Security
JID	Jabber Identifier
LL	Link Level
LZ77	Lempel-Ziv 1977
M2M	Machine-to-Machine
MAC	Media Access Control
MD	Message-Digest Algorithm
mDNS	Multicast Domain Name System
MQTT	Message Queuing Telemetry Transport
MQTT-S	Message Queuing Telemetry Transport for Sensors
MSP430	Microcontroller family from Texas Instruments
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
MUC	Multi-User Chat
NAT	Network Address Translation
NHC	Next Header Compression
OGC	Open Geospatial Consortium
OS	Operating System
OSGi	Open Services Gateway initiative
OWT	One-Way Traffic
P2P	Peer-to-Peer
PC	Personal Computer
PEP	Personal Eventing Protocol
PTR	Pointer record
QoS	Quality of Service
RAM	Random Access Memory
RD	Resource Directory
RDF	Resource Description Framework
RELOAD	REsource LOcation And Discovery
REST	Representational State Transfer
RF	Radio Frequency
RFC	Request For Comment

ROM	Read-Only Memory
RTT	Round Trip Time
SASL	Simple Authentication and Security Layer
SAX	Simple API for XML
SCRAM	Salted Challenge Response Authentication Mechanism
SE	Smart Energy
SHA	Secure Hash Algorithm
SNMP	Simple Network Management Protocol
SLIP	Serial Line IP
SLP	Service Location Protocol
SOAP	Simple Object Access Protocol
SOX	Sensor Over XMPP
SRV	Service record
SWE	Sensor Web Enablement
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TSP	Temporary Subscription for Presence
TSS	TCP Support for Sensor nodes
TTL	Time To Live
TXT	Text record
UDP	User Datagram Protocol
uDPWS	micro DPWS for embedded devices
uIP	micro IP stack for embedded devices
UPnP	Universal Plug and Play
URI	Uniform Resource Identifier
USB	Universal Serial Bus
uXMPP	micro XMPP stack for embedded devices
VoIP	Voice over IP
WLAN	Wireless Local Area Network
WS4D	Web Services for Devices
WSN	Wireless Sensor Networks
XEP	XMPP Extension Protocol
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XSF	XMPP Standards Foundation
ZLIB	Data compression algorithm

Bibliography

[1] P. Patel, A. Pathak, T. Teixeira, and V. Issarny, "Towards Application Development for the Internet of Things", in *Proceedings of the 8th Middleware Doctoral Symposium*, Lisbon, Portugal, 2011, MDS '11, pp. 5:1–5:6, ACM.

[2] A. Bassi and G. Horn, "Internet of Things in 2020 - ROADMAP FOR THE FUTURE", in *Workshop on RFID / Internet-of-Things*, Brussels, Belgium, May 2008, INFSO D.4NETWORKED ENTERPRISE & RFID INFSO G.2MICRO & NANOSYSTEMS in co-operation with the WORKING GROUP RFID OF THE ETP EPOSS.

[3] K. Ashton, "That 'Internet of Things' Thing", *RFID Journal*, Jul. 2009.

[4] C. Bormann, M. Ersue, and A. Keranen, "Terminology for Constrained-Node Networks", Request for Comment 7228, IETF, May 2014.

[5] F. Mattern and C. Floerkemeier, "From the Internet of Computers to the Internet of Things", in *From Active Data Management to Event-Based Systems and More*, K. Sachs, I. Petrov, and P. Guerrero, Eds., vol. 6462 of *Lecture Notes in Computer Science*, pp. 242–259. Springer Berlin / Heidelberg, 2010.

[6] T. Teixeira, S. Hachem, V. Issarny, and N. Georgantas, "Service Oriented Middleware for the Internet of Things: A Perspective", in *Towards a Service-Based Internet*, W. Abramowicz, I. Llorente, M. Surridge, A. Zisman, and J. Vayssiére, Eds., vol. 6994 of *Lecture Notes in Computer Science*, pp. 220–229. Springer Berlin / Heidelberg, 2011.

[7] J. Vasseur and A. Dunkels, *Interconnecting Smart Objects with IP: The Next Internet*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[8] M. Ersue and J. Korhonen, "Position Paper on "In-Network Object Cloud" Architecture and Design Goals", in *25th Workshop of Interconnecting Smart Objects with Internet*, Prague, Czech Republic, Mar. 2011, The Internet Architecture Board.

[9] A. Dunkels, "Towards TCP/IP for Wireless Sensor Networks", Licentiate thesis, Mar. 2005.

[10] A. Dunkels, T. Voigt, J. Alonso, H. Ritter, and J. Schiller, "Connecting Wireless Sensornets with TCP/IP Networks", in *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*, Frankfurt (Oder), Germany, Feb. 2004.

[11] D. Barisic and A. Pfefferseder, "Unified Device Networking Protocols for Smart Objects", in *25th Workshop of Interconnecting Smart Objects with Internet*, Prague, Czech Republic, Mar. 2011, The Internet Architecture Board.

[12] G. Mulligan, "The 6LoWPAN Architecture", in *Proceedings of the 4th workshop on Embedded networked sensors*, Cork, Ireland, 2007, EmNets '07, pp. 78–82, ACM.

[13] A. Dunkels and J. Vasseur, "IP for Smart Objects", Sep. 2008, IPSO Alliance White Paper #1.

[14] F. Khattak, P. Ginzboorg, V. Niemi, and J. Ekberg, "Role of Border Router in 6LoWPAN Security", in *Workshop on Smart Object Security*, Paris, France, Mar. 2012.

[15] M. Isomura, C. Decker, and M. Beigl, "Generic Communication Structure to Integrate Widely Distributed Wireless Sensor Nodes by P2P Technology", in *Proceedings of the 7th International Conference on Ubiquitous Computing*, Tokyo, Japan, Sep. 2005.

[16] J. Schönwälder, T. Tsou, and B. Sarikaya, "Protocol Profiles for Constrained Devices", [Online]. Available: `http://www.iab.org/wp-content/IAB-uploads/2011/03/Schoenwaelder.pdf`, 2012.

[17] M. Brachmann, O. Garcia-Morchon, and M. Kirsche, "Security for Practical CoAP Applications: Issues and Solution Approaches", in *Proceedings of the 10th GI/ITG KuVS Fachgespraech Sensornetze (FGSN11)*, H. Frey, Ed., Paderborn, Germany, Sep. 2011, University of Paderborn.

[18] H. Tschofenig and J. Arkko, "Report from the Smart Object Workshop", Request for Comment 6574, IETF, Apr. 2012.

[19] R.-C. Wang, Y.-C. Chang, and R.-S. Chang, "Design Issues of Semantic Service Discovery for Ubiquitous Computing", in *Proceedings of the International Conference on Multimedia and Ubiquitous Engineering*, Seoul, Korea, 2007, pp. 880–885, IEEE Computer Society.

[20] S. Giordano and D. Puccinelli, "The Human Element as the Key Enabler of Pervasiveness", in *Proceedings of the 10th IFIP Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, Favignana Island, Italy, Jun. 2011, pp. 150–156, IEEE Computer Society.

[21] V. Trifa, S. Wieland, D. Guinard, and T. M. Bohnert, "Design and Implementation of a Gateway for Web-based Interaction and Management of Embedded Devices", in *Proceedings of the 2nd International Workshop on Sensor Network Engineering (IWSNE'09)*, Marina del Rey, CA, USA, Jun. 2009.

[22] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Core", Request for Comment 6120, IETF, Mar. 2011.

[23] S. Cheshire and M. Krochmal, "Multicast DNS", Request for Comment 6762, IETF, Feb. 2013.

[24] S. Cheshire and M. Krochmal, "DNS-Based Service Discovery", Request for Comment 6763, IETF, Feb. 2013.

[25] P. Saint-Andre, "XEP-0174: Serverless Messaging", Standards track, XMPP Standards Foundation, Nov. 2008, Version 2.0.

[26] A. Hornsby and R. Walsh, "From Instant Messaging to Cloud Computing, an XMPP Review", in *Consumer Electronics (ISCE), 2010 IEEE 14th International Symposium on*, Braunschweig, Germany, 2010, IEEE, pp. 1–6.

[27] D. Guinard, V. Trifa, and E. Wilde, "Architecting a Mashable Open World Wide Web of Things", Technical Report 663, Department of Computer Science, ETH Zurich, Feb. 2010.

[28] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", Request for Comment 7230, IETF, Jun. 2014.

[29] G. F. Hurlburt, J. Voas, and K. W. Miller, "The Internet of Things: A Reality Check", *IT Professional*, vol. 14, pp. 56–59, 2012.

[30] M. Almeida and A. Matos, "Bridging the Devices with the Web Cloud: a RESTful Management Architecture over XMPP", in *Proceedings of the 6th International Mobile Multimedia Communications Conference*, Lisbon, Portugal, 2010, vol. 10 of *Mobimedia '10*.

[31] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol", Request for Comment 5246, IETF, Aug. 2008.

[32] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors", in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN 2004)*, Tampa, FL, USA, Nov. 2004, pp. 455–462, IEEE Computer Society.

[33] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An Operating System for Sensor Networks", in *Ambient Intelligence*, W. Weber, J.M. Rabaey, and E. Aarts, Eds., pp. 115–148. Springer Berlin Heidelberg, 2005.

[34] S. Hachem, T. Teixeira, and V. Issarny, "Ontologies for the Internet of Things", in *Proceedings of the 8th Middleware Doctoral Symposium*, Lisbon, Portugal, 2011, MDS '11, pp. 3:1–3:6, ACM.

[35] 802.15.4 Task Group, "IEEE 802.15.4", [Online] `http://www.ieee802.org/15/pub/TG4.html`, 2006.

[36] J. W. Hui and D. E. Culler, "IP is Dead, Long Live IP for Wireless Sensor Networks", in *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, Raleigh, NC, USA, 2008, SenSys '08, pp. 15–28, ACM.

[37] S. Duquennoy, F. Österlind, and A. Dunkels, "Lossy Links, Low Power, High Throughput", in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, Seattle, Washington, USA, 2011, SenSys '11, pp. 12–25, ACM.

[38] T. Sánchez López, D. C. Ranasinghe, M. Harrison, and D. Mcfarlane, "Adding sense to the Internet of Things", *Personal Ubiquitous Comput.*, vol. 16, no. 3, pp. 291–308, Mar. 2012.

[39] S. Kuryla and J. Schönwälder, "Evaluation of the Resource Requirements of SNMP Agents on Constrained Devices", in *Managing the Dynamics of Networks and Services*, I. Chrisment, A. Couch, R. Badonnel, and M. Waldburger, Eds., vol. 6734 of *Lecture Notes in Computer Science*, pp. 100–111. Springer Berlin / Heidelberg, 2011.

[40] N. Kushalnagar, G. Montenegro, and C. Schumacher, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals", Request for Comments 4919, IETF, Aug. 2007.

[41] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", Request for Comment 4944, IETF, Sep. 2007.

[42] D. Yazar and A. Dunkels, "Efficient Application Integration in IP-based Sensor Networks", in *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, Berkeley, California, USA, 2009, BuildSys '09, pp. 43–48, ACM.

[43] The Operating System for the Internet of Things, "IPSN 2010 Demo: An IP-based Sensor Network with Augmented Reality", `http://www.contiki-os.org/2011/08/ipsn-2010-demo-ip-based-sensor-network.html`, 2011.

[44] K. Fall, "A Delay-Tolerant Network Architecture for Challenged Internets", in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, Karlsruhe, Germany, 2003, SIGCOMM '03, pp. 27–34, ACM.

[45] M. Durvy, J. Abeille, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels, "Making Sensor Networks IPv6 Ready", in *Proceedings of the 6th ACM Conference on Networked Embedded Sensor Systems (SenSys)*, Raleigh, NC, USA, Nov. 2008.

[46] A. Dunkels, T. Voigt, and J. Alonso, "Making TCP/IP Viable for Wireless Sensor Networks", in *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN 2004), work-in-progress session*, Berlin, Germany, Jan. 2004.

[47] D. Barnett and A. J. Massa, "Dr Dobbs Journal - Inside the uIP Stack", [Online] `http://www.drdobbs.com/embedded-systems/184405971`, Feb. 2005.

[48] B. Ostermaier, M. Kovatsch, and S. Santini, "Connecting Things to the Web using Programmable Low-power WiFi Modules", in *Proceedings of the Second International Workshop on Web of Things*, San Francisco, California, USA, 2011, WoT '11, pp. 2:1–2:6, ACM.

[49] A. Dunkels, "The ContikiMAC Radio Duty Cycling Protocol", Tech. Rep. T2011:13, Swedish Institute of Computer Science, Dec. 2011.

[50] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He, "Software-based On-line Energy Estimation for Sensor Nodes", in *Proceedings of the Fourth IEEE Workshop on Embedded Networked Sensors (Emnets IV)*, Cork, Ireland, Jun. 2007.

[51] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-Level Sensor Network Simulation with COOJA", in *Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, Nov. 2006.

[52] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt, "MSPsim – an Extensible Simulator for MSP430-equipped Sensor Boards", in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, Jan. 2007.

[53] J. H. Davies, *MSP430 Microcontroller Basics*, Newnes, Newton, MA, USA, 2008.

[54] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors", *SIGPLAN Not.*, vol. 35, no. 11, pp. 93–104, Nov. 2000.

[55] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón, "Towards Interoperability Testing for Wireless Sensor Networks with COOJA/MSPSim", in *Proceedings of the 6th European Conference on Wireless Sensor Networks, EWSN 2009*, Cork, Ireland, Feb. 2009.

[56] A. Dunkels, "Full TCP/IP for 8 Bit Architectures", in *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, San Francisco, May 2003, Usenix.

[57] R. Braden, "Requirements for Internet Hosts – Communication Layers", Request for Comment 1122, IETF, Oct. 1989.

[58] A. Dunkels, "Rime — A Lightweight Layered Communication Stack for Sensor Networks", in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, Jan. 2007.

[59] Atmel Corporation, "AVR2016: RZRAVEN Hardware User's Guide", [Online] `http://www.atmel.com/Images/doc8117.pdf`, May 2012.

[60] Atmel Corporation, "ZigBit 2.4 GHz Wireless Modules", [Online] `http://www.atmel.com/Images/doc8226.pdf`, May 2012.

[61] Computer Systems & Telematics Freie Universität Berlin, Germany, "First Steps for ScatterWeb - A platform for teaching & prototyping wireless sensor networks", [Online] `http://www.emmanuelbaccelli.org/sensors/First_Steps_MSB.pdf`.

[62] Memsic Corporation, "IRIS", [Online] `http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=135%3Airis`, 2010.

[63] STMicroelectronics, "STM32-SK/HIT data brief", [Online] `http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATA_BRIEF/CD00083685.pdf`, 2012.

[64] Memsic Corporation, "MICAz", [Online] `http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=148%3Amicaz`, 2010.

[65] M. Alvira, "Using the Freescale MC1322x Series ARM7 Processor with integrated 802.15.4", [Online] `http://mc1322x.devl.org/`.

[66] Memsic Corporation, "TelosB", [Online] `http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=152%3Atelosb`, 2010.

[67] Zolertia, "Z1", [Online] `http://zolertia.sourceforge.net/wiki/images/e/e8/Z1_RevC_Datasheet.pdf`, Jan. 2012.

[68] M. Kovatsch, S. Duquennoy, and A. Dunkels, "A Low-Power CoAP for Contiki", in *Proceedings of the 2011 Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*, Valencia, Spain, Oct. 2011, pp. 855–860, IEEE Computer Society.

[69] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", Request for Comment 2460, IETF, Dec. 1998.

[70] Zolertia, "List of Z1 compatible sensors", [Online] `http://zolertia.sourceforge.net/wiki/index.php/Z1_Sensors`, Jan. 2012.

[71] Texas Instruments, "Chipcon 2420 - 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver", [Online] `http://www.ti.com/lit/ds/symlink/cc2420.pdf`, Mar. 2014.

[72] M. Hauswirth, D. Pfisterer, and S. Decker, "Making Internet-Connected Objects readily useful", in *25th Workshop of Interconnecting Smart Objects with Internet*, Prague, Czech Republic, Mar. 2011, The Internet Architecture Board.

[73] A. J. Jara, P. Martinez-Julia, and A. Skarmeta, "Light-weight multicast DNS and DNS-SD (lmDNS-SD): IPv6-based resource and service discovery for the Web of Things", in *International Workshop on Extending Seamlessly to the Internet of Things (esIoT 2012)*, Palermo, Italia, Jul. 2012.

[74] D. Antoniades, I. Polakis, G. Kontaxis, E. Athanasopoulos, S. Ioannidis, E. P. Markatos, and T. Karagiannis, "we.b: the Web of Short URLs", in *Proceedings of the 20th international conference on World wide web*, Hyderabad, India, 2011, WWW '11, pp. 715–724, ACM.

[75] S. Chhabra, A. Aggarwal, F. Benevenuto, and P. Kumaraguru, "Phi.sh/$oCiaL: the Phishing Landscape through Short URLs", in *Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*, Perth, WA, 2011, CEAS '11, pp. 92–101, ACM.

[76] F. Klien and M. Strohmaier, "Short Links under Attack: Geographical Analysis of Spam in a URL Shortener Network", in *Proceedings of the 23rd ACM conference on Hypertext and social media*, Milwaukee, Wisconsin, USA, 2012, HT '12, pp. 83–88, ACM.

[77] D. Yazar, "RESTful Wireless Sensor Networks", Master Thesis, Uppsala Universitet, Oct. 2009.

[78] Server-Sent Events, "W3C Working Draft", [Online] `http://www.w3.org/TR/eventsource/`, Apr. 2012.

[79] B. Fitzpatrick, B. Slatkin, and M. Atkins, "PubSubHubbub Core 0.3 – Working Draft", [Online] `http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html`, Feb. 2010.

[80] M. Kovatsch, M. Lanter, and Z. Shelby, "Californium: Scalable cloud services for the internet of things with coap", in *Proceedings of the 4th International Conference on the Internet of Things (IoT 2014)*, Cambridge, MA, USA, October 2014.

[81] C. Bormann, A. P. Castellani, and Z. Shelby, "CoAP: An Application Protocol for Billions of Tiny Internet Nodes", *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, 2012.

[82] B. C. Villaverde, D. Pesch, R. De Paz Alberola, S. Fedor, and M. Boubekeur, "Constrained Application Protocol for Low Power Embedded Networks: A Survey", in *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, Los Alamitos, CA, USA, 2012, vol. 0, pp. 702–707, IEEE Computer Society.

[83] Z. Shelby, K. Hartke, and C. Bormann, "Constrained Application Protocol (CoAP)", Request for Comment 7252, IETF, Jun. 2014.

[84] Z. Shelby, "Constrained RESTful Environments (CoRE) Link Format", Request for Comment 6690, IETF, Aug. 2012.

[85] G. Moritz, "DPWS for 6LoWPAN", Internet-draft, IETF, Jun. 2010.

[86] R. Housley, "Outgoing IETF Chair: Governments Should Embrace Internet Standards", *Internet Society*, Mar. 2013.

[87] K. Hartke, "Observing Resources in CoAP", Internet-draft, CoRE Working Group, Dec. 2014.

[88] S. Lemay, V. Solorzano Barboza, and H. Tschofenig, "A TCP and TLS Transport for the Constrained Application Protocol (CoAP)", Internet-draft, CoRE Working Group, Sep. 2014.

[89] Eurotech, International Business Machines Corporation (IBM), "MQTT v3.1 Protocol Specification", [Online] `http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/MQTT_V3.1_Protocol_Specific.pdf`, 2010.

[90] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-S - A Publish/Subscribe Protocol For Wireless Sensor Networks", in *Proceedings of 2nd Workshop on Information Assurance for Middleware Communications "IAMCOM'08"*, Bangalore, India, Jan. 2008.

[91] Eurotech, International Business Machines Corporation (IBM), "MQTT For Sensor Networks (MQTTs) v1.0 Protocol Specification", [Online] `http://mqtt.org/MQTTs_Specification_V1.0.pdf`, Oct. 2007.

[92] E. Zeeb, G. Moritz, D. Timmermann, and F. Golatowski, "WS4D: Toolkits for Networked Embedded Systems Based on the Devices Profile for Web Services", in *Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW 2010)*, San Diego, California, USA, Sep. 2010, pp. 1–8, IEEE Computer Society.

[93] A. Broering, J. Echterhoff, S. Jirka, I. Simonis, T. Everding, C. Stasch, S. Liang, and R. Lemmens, "New Generation Sensor Web Enablement", *Sensors*, vol. 11, no. 3, pp. 2652–2699, 2011.

[94] N. Giang, M. Ha, and D. Kim, "SCoAP: An integration of CoAP protocol with web-based application", in *GLOBECOM*, Atlanta, GA, USA, 2013, pp. 2648–2653.

[95] J. Cheng and T. Kunz, "A Survey on Smart Home Networking", Tech. Rep. SCE-09-10, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, Sep. 2009.

[96] "openhab - empowering the smart home", [Online] `http://code.google.com/p/openhab/`, 2012.

[97] "XMPP Extensions", [Online] `http://xmpp.org/xmpp-protocols/xmpp-extensions/`, 2012.

[98] A. Rowe, M. E. Berges, G. Bhatia, E. Goldman, R. Rajkumar, J. H. Garrett, J. M. F. Moura, and L. Soibelman, "Sensor Andrew: Large-Scale Campus-Wide Sensing and Actuation", *IBM Journal of Research and Development*, vol. 55, no. 1.2, pp. 6:1 –6:14, Jan.-Mar. 2011.

[99] N. Pin and J.K. Nurminen, "Integrate WSN to the Web of Things by using XMPP", in *Proceedings of the 3rd International Conference on Sensor Systems and Software (S-Cube 2012)*, Lisbon, Portugal, Jun. 2012.

[100] A. Hornsby and E. Bail, "uXMPP: Lightweight Implementation for Low Power Operating System Contiki", in *Proceedings of the International Conference on Ultra Modern Telecommunications and Workshops (ICUMT 2009)*, St. Petersburg, Russia, Oct. 2009, pp. 1–5, IEEE.

[101] M. Schulz, "mbed Cookbook - XMPPClient", Jan. 2011.

[102] R. Ostinelli, "Internet Of Things: Vision, Prerequisites and OpenSpime", 2009.

[103] P. Saint-Andre, "Jabber Security", [Online] `http://www.saint-andre.com/jabber/Security.pdf`, Jul. 2007.

[104] P. Saint-Andre, K. Smith, and R. Tronon, *XMPP: The Definitive Guide Building Real-Time Applications with Jabber Technologies*, O'Reilly Media, Inc., 2009.

[105] G. Weis and A. Lewis, "Using XMPP for Ad-hoc Grid Computing - an Application Example using Parallel Ant Colony Optimization", *IPDPS '09 Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, 2009.

[106] D. Bernstein and D. Vij, "Intercloud Directory and Exchange Protocol Detail using XMPP and RDF", *IEEE 6th World Congress on Services*, 2010.

[107] M. E. Gregori, J. P. Cámara, and G. A. Bada, "A Jabber-based Multi-agent System Platform", in *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, New York, NY, USA, 2006, AAMAS '06, pp. 1282–1284, ACM.

[108] R. Lübke, D. Schuster, and A. Schill, "A Framework for the Development of Mobile Social Software on Android", in *Mobile Computing, Applications, and Services*, J. Zhang, J. Wilkiewicz, and A. Nahapetian, Eds., vol. 95 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 207–225. Springer Berlin Heidelberg, 2012.

[109] "Movim – Kick as social network", [Online] `https://movim.eu/`, 2014.

[110] "Cisco Jabber Features", [Online] `http://www.cisco.com/web/products/voice/jabber_features.html`, 2012.

[111] "Microsoft Lync - Unified Communications Solution", [Online] `http://lync.microsoft.com/`, 2012.

[112] "XMPP Software", [Online] `http://xmpp.org/xmpp-software/`, 2012.

[113] P. Costa, G.P. Picco, and S. Rossetto, "Publish-Subscribe on Sensor Networks: a Semi-Probabilistic Approach", in *Proceedings of the IEEE International Conference on Mobile Adhoc and Sensor Systems Conference*, Washington, DC, USA, 2005, pp. 323–332, IEEE Computer Society.

[114] Th. Eugster, A. Felber, R. Guerraoui, and A-M. Kermarrec, "The many faces of publish/subscribe", *ACM Comput. Surv. 35 (2)*, 2003.

[115] W. Jung, S. Hong, M. Ha, Y.-J. Kim, and D. Kim, "SSL-Based Lightweight Security of IP-Based Wireless Sensor Networks", in *Advanced Information Networking and Applications Workshops, International Conference on*, Los Alamitos, CA, USA, 2009, vol. 0, pp. 1112–1117, IEEE Computer Society.

[116] V. Perelman, "Security in IPv6-enabled Wireless Sensor Networks: An Implementation of TLS/DTLS for the Contiki Operating System", Master Thesis, Jacobs University, Jun. 2012.

[117] A. Melnikov and K. Zeilenga, "Simple Authentication and Security Layer (SASL)", Request for Comment 4422, IETF, Jun. 2006.

[118] K. Zeilenga, "The PLAIN Simple Authentication and Security Layer (SASL) Mechanism", Request for Comment 4616, IETF, Aug. 2006.

[119] J. Klensin, R. Catoe, and P. Krumviede, "IMAP/POP AUTHorize Extension for Simple Challenge/Response", Request for Comment 2195, IETF, Sep. 1997.

[120] P. Leach and C. Newman, "Using Digest Authentication as a SASL Mechanism", Request for Comment 2831, IETF, May 2000.

[121] C. Newman, A. Menon-Sen, A. Melnikov, and N. Williams, "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms", Request for Comment 5802, IETF, Jul. 2010.

[122] K. Zeilenga, "Anonymous Simple Authentication and Security Layer (SASL) Mechanism", Request for Comment 4505, IETF, Jun. 2006.

[123] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Address Format and Presence", Request for Comment 6122, IETF, Mar. 2011.

[124] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence", Request for Comment 6121, IETF, Mar. 2011.

[125] E. Dressler and D. Tavangarian, "Heterogeneous Communication in Smart Ensembles", in *Intelligent Interactive Assistance and Mobile Multimedia Computing*, D. Tavangarian, T. Kirste, D. Timmermann, U. Lucke, and D. Versick, Eds., vol. 53 of *Communications in Computer and Information Science*, pp. 155–166. Springer Berlin, 2009.

[126] "OpenWrt - Wireless Freedom", [Online] `https://openwrt.org/`, 2012.

[127] "Prosody IM", [Online] `http://developer.pidgin.im/wiki/SupportedXEPs`, 2012.

[128] The Avahi Team, "More About Avahi - Details about mDNS, DS-DNS and Zeroconf", [Online] `http://avahi.org/wiki/AboutAvahi`, 2012.

[129] S. Fedor and M. Collier, "On the Problem of Energy Efficiency of Multi-Hop vs One-Hop Routing in Wireless Sensor Networks", in *Advanced Information Networking and Applications Workshops, International Conference on*, Los Alamitos, CA, USA, 2007, vol. 2, pp. 380–385, IEEE Computer Society.

[130] L. Casone, "Improving Many-to-One Traffic flowing in Multi-Hop 802.15.4 WSNs using a MAC-level Fair Scheduling", in *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference*, Los Alamitos, CA, USA, 2007, vol. 0, pp. 1–7, IEEE Computer Society.

[131] W. Keith Edwards, "Discovery Systems in Ubiquitous Computing", *IEEE Pervasive Computing*, vol. 5, pp. 70–77, 2006.

[132] J. Hildebrand, P. Millard, R. Eatmon, and P. Saint-Andre, "XEP-0030: Service Discovery", Standards track, XMPP Standards Foundation, Jun. 2008, Version 2.4.

[133] P. Saint-Andre, "XEP-0045: Multi-User Chat", Standards track, XMPP Standards Foundation, Feb. 2012, Version 1.25.

[134] M. Miller, "XEP-0050: Ad-Hoc Commands", Standards track, XMPP Standards Foundation, Jun. 2005, Version 1.2.

[135] P. Millard, P. Saint-Andre, and R. Meijer, "XEP-0060: Publish-Subscribe", Standards track, XMPP Standards Foundation, Jul. 2010, Version 1.13.

[136] "ejabberd Community Site - Protocols Implementation", [Online] http://www.ejabberd.im/protocols, 2012.

[137] "Openfire", [Online] http://www.igniterealtime.org/builds/openfire/docs/latest/documentation/protocol-support.html, 2012.

[138] "Tigase", [Online] http://www.tigase.org/tigase-server-features, 2012.

[139] "Using Apple iChat with Wide-Area DNS Service Discovery (DNS-SD)", [Online] http://www.dns-sd.org/iChatWideArea.html, 2012.

[140] "Pidgin Development Wiki - Supported XEPs", [Online] http://developer.pidgin.im/wiki/SupportedXEPs, 2012.

[141] "Psi - The cross-platform XMPP client for power users", [Online] http://psi-im.org/wiki/Supported_Protocols, Oct. 2012.

[142] "XMPPFramework for Objective C", [Online] https://github.com/robbiehanson/XMPPFramework/tree/master/Extensions, 2013.

[143] "Multi-account jabber-client", [Online] http://www.xabber.com/, 2013.

[144] "Smack API", [Online] http://www.igniterealtime.org/builds/smack/docs/latest/documentation/extensions/index.html, 2013.

[145] "XEP-0174 (Link-local) support in Smack", [Online] http://community.igniterealtime.org/thread/36186, 2013.

[146] "OSGi Alliance - The Dynamic Module System for Java", [Online] http://www.osgi.org/, 2012.

[147] Marvell Technology Group Ltd., "PlugComputer Community", http://www.plugcomputer.org/, 2012.

[148] "Systemteq - Smart room control systems for hotels", [Online] http://www.systemteq.com/, 2012.

[149] "TechSpot - Smart hotel rooms", [Online] http://www.techspot.com/news/19439-smart-hotel-rooms.html, 2005.

[150] V. Rajesh, J. M. Gnanasekar, R. S. Ponmagal, and P. Anbalagan, "Integration of Wireless Sensor Network with Cloud", in *International Conference on Recent Trends in Information, Telecommunication and Computing*, Kochi, Kerala, India, Mar. 2010.

[151] L. Shu, M. Hauswirth, L. Cheng, J. Ma, V. Reynolds, and L. Zhang, "Sharing Worldwide Sensor Network", *International Symposium on Applications and the Internet*, 2008.

[152] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan, "Building Efficient Wireless Sensor Networks with Low-Level Naming", *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Oct. 2001.

[153] Isode Ltd, "XMPP PubSub", *Whitepapers*, Jul. 2011.

[154] P. Saint-Andre, "XEP-0175: Best Practices for Use of SASL ANONYMOUS", Informational, XMPP Standards Foundation, Sep. 2009, Version 1.2.

[155] B. Ecker, "SimpleXML", [Online] `http://www.omegadb.net/simplexml/`.

[156] iksemel, "Fast and portable XML parser and Jabber protocol library", [Online] `http://code.google.com/p/iksemel/`.

[157] T. Marston, "Breaking Backwards Compatibility is EVIL", [Online] `http://www.tonymarston.net/php-mysql/breaking-bc-is-evil.html`, Jan. 2006.

[158] T. Braun, T. Voigt, and A. Dunkels, "TCP Support for Sensor Networks", in *IEEE/IFIP WONS 2007*, Obergurgl, Austria, Jan. 2007.

[159] A. Dunkels, T. Voigt, J. Alonso, and H. Ritter, "Distributed TCP Caching for Wireless Sensor Networks", in *Proceedings of the Third Annual Mediterranean Ad Hoc Networking Workshop (MedHocNet 2004)*, Bodrum, Turkey, Jun. 2004.

[160] U. Bürgi, "Performance Optimization for TCP-based Wireless Sensor Networks", Master Thesis, Bern University, 2011.

[161] J. Hildebrand and P. Saint-Andre, "XEP-0138: Stream Compression", Standards track, XMPP Standards Foundation, May 2009, Version 2.0.

[162] P. Deutsch and J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", Request for Comment 1950, Network Working Group, May 2006.

[163] D. Salomon, G. Motta, and D. Bryant, *Data Compression: The Complete Reference*, Springer, 4. edition, Dec. 2006.

[164] Z. Shelby, M. Luimula, and D. Peintner, "Efficient XML Encoding and 6LowApp", Internet-draft, 6lowapp, Oct. 2009.

[165] First Objective Software, Inc., "easy zlib C/C++ compress and XML compression", [Online] `http://www.firstobject.com/easy-zlib-c++-xml-compression.htm`.

[166] zlib, "A Massively Spiffy Yet Delicately Unobtrusive Compression Library", [Online] `http://zlib.net/`, May 2012.

[167] miniz, "Single C source file Deflate/Inflate compression library with zlib-compatible API, ZIP archive reading/writing, PNG writing", [Online] `http://code.google.com/p/miniz/`.

[168] W3C, "Efficient XML Interchange (EXI) Format 1.0", [Online] `http://www.w3.org/TR/2011/REC-exi-20110310/`, Mar. 2012.

[169] P. Waher and Y. DOI, "XEP-0322: Efficient XML Interchange (EXI) Format", Standards track, XMPP Standards Foundation, Mar. 2014, Version 0.4.

[170] R. Kyusakov, J. Eliasson, and J. Delsing, "Efficient Structured Data Processing for Web Service Enabled Shop Floor Devices", in *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE)*. Jun. 2011, pp. 1716–1721, IEEE Computer Society.

[171] Open Geospatial Consortium, "Binary Extensible Markup Language (BXML) Encoding Specification, Version 0.0.8", [Online] `http://portal.opengeospatial.org/files/?artifact_id=13636`, Jan. 2006.

[172] International Telecommunications Union, "ITU-T X.891 | ISO/IEC 24824-1 (Fast Infoset)", [Online] `http://www.itu.int/ITU-T/asn1/xml/finf.htm`, May 2007.

[173] Open Geospatial Consortium, "OpenGIS SensorML Encoding Standard v 1.0 Schema Corregendum 1 (1.01)", [Online] `http://portal.opengeospatial.org/files/?artifact_id=24757`, Jun. 2007.

[174] D. Peintner, "EXIficient - XML becomes efficient ...", [Online] `http://exificient.sourceforge.net`, Oct. 2011.

[175] Embedded Internet Systems Laboratory (EISLAB), "EXIP - Embeddable EXI implementation in C", [Online] `http://exip.sourceforge.net/`, Dec. 2011.

[176] J. Johansson, M. Voelker, J. Eliasson, A. Oestmark, P. Lindgren, and J. Delsing, "Mulle: A minimal sensor networking device - implementation and manufacturing challenges", in *Proceedings of IMAPS Nordic*, Jun. 2004, pp. 265–271.

[177] University of Rostock, "WS4D-uEXI - EXI for WSNs and 6LoWPAN", [Online] `http://code.google.com/p/ws4d-uexi/`, May 2012.

[178] G. Moritz, D. Timmermann, R. Stoll, and F. Golatowski, "Encoding and Compression for the Devices Profile for Web Services", in *Proceedings of the 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, Perth, WA, Apr. 2010, pp. 514–519.

[179] Y. Doi, Y. Sato, M. Ishiyama, Y. Ohba, and K. Teramoto, "XML-less EXI with Code Generation for Integration of Embedded Devices in Web based Systems", in *Internet of Things (IOT), 2012 3rd International Conference on the*, Wuxi, China, Oct. 2012, pp. 76–83.

[180] J. Zhao, C. Qiao, R. S. Sudhaakar, and S. Yoon, "Improve Efficiency and Reliability in Single-Hop WSNs with Transmit-Only Nodes", *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 3, pp. 520–534, 2013.

[181] A. Reinhardt, D. Christin, and R. Steinmetz, "Pre-Allocating Code Mappings for Energy-Efficient Data Encoding in Wireless Sensor Networks", in *9th IEEE International Workshop on Sensor Networks and Systems for Pervasive Computing 2013 (PerSeNS 2013)*, San Diego, California, USA, Mar. 2013, pp. 578–583.

[182] M. Hammoudeh, S. Mount, O. Aldabbas, and M. Stanton, "Clinic: A Service Oriented Approach for Fault Tolerance in Wireless Sensor Networks", in *Proceedings of the IEEE International Conference on Sensor Technologies and Applications (SENSORCOMM 2010)*, Venice/Mestre, Italy, Jul. 2010, pp. 625–631, IEEE Computer Society.

[183] D.-K. Chen, "Systematic Review of Applying Service Oriented Architecture in Networking", in *Proceedings of the 2010 Sixth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, Washington, DC, USA, 2010, IIH-MSP '10, pp. 167–170, IEEE Computer Society.

[184] D. Yazar, N. Tsiftes, F. Österlind, N. Finne, J. Eriksson, and A. Dunkels, "Augmenting Reality with IP-based Sensor Networks", in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, New York, NY, USA, Apr. 2010, IPSN '10, pp. 440–441, ACM.

[185] J. Bardin, P. Lalanda, and C. Escoffier, "Towards an Automatic Integration of Heterogeneous Services and Devices", in *Proceedings of the 2010 IEEE Asia-Pacific Services Computing Conference*, Washington, DC, USA, 2010, APSCC '10, pp. 171–178, IEEE Computer Society.

[186] Z. Shelby, S. Krco, and C. Bormann, "CoRE Resource Directory", Internet-draft, IETF, Jul. 2012.

[187] T. Ashraf Butt, I. Phillips, L. Guan, and G. Oikonomou, "TRENDY: an Adaptive and Context-Aware Service Discovery Protocol for 6LoWPANs", in *Proceedings of the Third International Workshop on the Web of Things*, Newcastle, United Kingdom, 2012, WOT '12, pp. 2:1–2:6, ACM.

[188] Apple Inc., "mDNSResponder", [Online] `http://www.opensource.apple.com/tarballs/mDNSResponder/`, 2012.

[189] "DNS SRV (RFC 2782) Service Types", [Online] `http://www.dns-sd.org/ServiceTypes.html`, 2012.

[190] E. Guttman, C. Perkins, J. Veizades, and M. Day, "Service Location Protocol, Version 2", Request for Comment 2608, IETF, Jun. 1999.

[191] "UPnP Forum - UPnP Specifications", [Online] `http://upnp.org/sdcps-and-certification/standards/`, 2012.

[192] "Liaison - because with Liaison, the client finds you", [Online] `http://www.acm.uiuc.edu/signet/liaison/`, 2003.

[193] "Contiki Branch of darkdeep", [Online] `http://svn.deepdarc.com/code/contiki/trunk`, 2012.

[194] "Bonjour/Zeroconf with Arduino", [Online] `http://gkaindl.com/software/arduino-ethernet/bonjour`, 2012.

[195] "Arduino - Homepage", [Online] `http://arduino.cc/en/`, 2012.

[196] S. Cheshire, "Service Discovery in Zero Configuration Networks", in *25th Workshop of Interconnecting Smart Objects with Internet*, Prague, Czech Republic, Mar. 2011, The Internet Architecture Board.

[197] "Setting up DNS to Allow Clients to Advertise their own Wide-Area Services", [Online] `http://www.dns-sd.org/#WA`, 2012.

[198] S. Pöhlsen, C. Buschmann, and C. Werner, "Integrating a Decentralized Web Service Discovery System into the Internet Infrastructure", in *Proceedings of the 2008 Sixth European Conference on Web Services*, Washington, DC, USA, 2008, ECOWS '08, pp. 13–20, IEEE Computer Society.

[199] K. Lynn, S. Cheshire, M. Blanchet, and D. Migault, "Requirements for Scalable DNS-SD/mDNS Extensions", Internet-draft, DNS-SD/mDNS Extensions, Oct. 2014.

[200] P. Mockapetris, "Domain Names - Implementation and Specification", Request for Comment 1035, IETF, Nov. 1987.

[201] J. Hui and P. Thubert, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks", Request for Comment 6282, IETF, Sep. 2011.

[202] J. Hui and D. Culler, "6LoWPAN: Incorporating IEEE 802.15.4 into the IP Architecture", Tech. Rep., Internet Protocol for Smart Objects (IPSO) Alliance, White paper #3, Jan. 2009.

[203] M. Crawford, "Transmission of IPv6 Packets over Ethernet Networks", Request for Comment 2464, IETF, Dec. 1998.

[204] J. Romkey, "A NONSTANDARD FOR TRANSMISSION OF IP DATAGRAMS OVER SERIAL LINES: SLIP", Request for Comment 1055, IETF, Jun. 1998.

[205] G. Campagna, "Empathy", [Online] `http://live.gnome.org/Empathy`, Mar. 2013.

[206] G. Oikonomou and I. Phillips, "Experiences from Porting the Contiki Operating System to a Popular Hardware Platform", in *Proceedings of the International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*, Barcelona, Spain, Jun. 2011, IEEE Computer Society.

[207] T. Voigt, J. Eriksson, F. Österlind, R. Sauter, N. Aschenbruck, P. J. Marrón, V. Reynolds, L. Shu, O. Visser, A. Koubaa, and A. Köpke, "Towards Comparable Simulations of Cooperating Objects and Wireless Sensor Networks", in *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, Pisa, Italy, 2009, VALUETOOLS '09, pp. 77:1–77:10.

[208] "MSP430-GCC-4-4.5 Tarball", [Online] `https://sourceforge.net/projects/zolertia/files/contiki-enviroment/msp430-gcc-4.4.5.tar.gz/download`, 2012.

[209] G. Moritz, "uDPWS - The Devices Profile for Web Services (DPWS) for deeply embedded devices", [Online] `http://code.google.com/p/udpws/wiki/Introduction`, Aug. 2010.

[210] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg, "Implementation of CoAP and its Application in Transport Logistics", *Proceedings of the Workshop on Extending the Internet to Low power and Lossy Networks (IP+SN 2011), Chicago, IL, USA*, 2011.

[211] J. Silva, T. Camilo, P. Pinto, R. Ruivo, A. Rodrigues, F. Gaudêncio, and F. Boavida, "Multicast and IP Multicast Support in Wireless Sensor Networks", *Journal of Networks (JNW)*, vol. 3, no. 3, pp. 19–26, Mar. 2008.

[212] Kno.e.sis Project Consortium, "Linked Sensor Data", `http://wiki.knoesis.org/index.php/LinkedSensorData`, 2010.

Publications in the Context of this Thesis

[213] R. Klauck and M. Kirsche, "Combining Mobile XMPP Entities and Cloud Services for Collaborative Post-Disaster Management in Hybrid Network Environments", *The Journal of Mobile Networks and Applications*, vol. 18, no. 2, pp. 253–270, 2013.

[214] R. Klauck and M. Kirsche, "XMPP to the Rescue: Enhancing Post Disaster Management and Joint Task Force Work", in *Proceedings of the 2nd International Workshop on Pervasive Networks for Emergency Management (PerNEM), co-located with the 10th IEEE Conference on Pervasive Computing and Communication (PerCom 2012)*, Lugano, Switzerland, Mar. 2012.

[215] R. Klauck and M. Kirsche, "Chatty Things - Making the Internet of Things Readily Usable for the Masses with XMPP", in *Proceedings of the 8th International Conference on Collaborative Computing: Networking, Applications & Worksharing (CollaborateCom 2012)*, Pittsburgh, USA, Nov. 2012, IEEE.

[216] J. Gäbler, R. Klauck, M. Kirsche, and H. König, "uBeeMe - a Platform to Enable Mobile Collaborative Applications", in *Proceedings of the 9th International Conference on Collaborative Computing: Networking, Applications & Worksharing (CollaborateCom 2013)*, Austin, USA, Oct. 2013, IEEE.

[217] S. Mehner, R. Klauck, and H. König, "Location-Independent Fall Detection with Smartphone", in *Proceedings of the 6th International Conference on PErvasive Technologies Related to Assistive Environments*, Rhodes, Greece, May 2013, PETRA '13, pp. 11:1–11:8, ACM.

[218] D. Schuster, P. Grubitzsch, D. Renzel, I. Koren, R. Klauck, and M. Kirsche, "Global-scale Federated Access to Smart Objects Using XMPP", in *Proceedings of the IEEE International Conference on Internet of Things 2014 (iThings 2014)*, Taipai, Taiwan, Sep. 2014, IEEE.

[219] R. Klauck and M. Kirsche, "Bonjour Contiki: A Case Study of a DNS-based Discovery Service for the Internet of Things", in *Proceedings of the 11th International IEEE Conference on Ad-Hoc Networks and Wireless (ADHOC-NOW 2012)*, X. Li, S. Papavassiliou, and S. Ruehrup, Eds., Belgrade, Serbia, Jul. 2012, vol. 7363 of *Lecture Notes in Computer Science (LNCS)*, pp. 317–330, Springer Berlin.

[220] R. Klauck and M. Kirsche, "Integrating P2PSIP into Collaborative P2P Applications: A Case Study with the P2P Videoconferencing System BRAVIS", in *Proceedings of the*

5th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2009), Washington, DC, USA, Nov. 2009, IEEE.

[221] R. Klauck and M. Kirsche, "Enhanced DNS Message Compression - Optimizing mDNS/DNS-SD for the Use in 6LoWPANs", in *Proceedings of the 9th IEEE International Workshop on Sensor Networks and Systems for Pervasive Computing 2013 (PerSeNS 2013), co-located with the 11th IEEE Conference on Pervasive Computing and Communication (PerCom 2013)*, San Diego, California, USA, Mar. 2013, pp. 590–595.

[222] R. Klauck, M. Kirsche, J. Gäbler, and S. Schöpke, "Mobile XMPP and Cloud Service Collaboration: An Alliance for Flexible Disaster Management", in *Proceedings of the 7th International Conference on Collaborative Computing: Networking, Applications & Worksharing (CollaborateCom 2011)*, Orlando, USA, Nov. 2011, IEEE.

[223] P. Waher and R. Klauck, "XEP-0347: Internet of Things - Discovery", Standards track, XMPP Standards Foundation, Nov. 2014, Version 0.4.

[224] R. Klauck, "A Minimized XMPP Stack for the Contiki OS – Contiki Branch of rklauck", [Online] `https://github.com/rklauck/contiki/blob/xmpp/apps/xmpp`, 2015.

[225] R. Klauck, "A DNS-SD Implementation for the Contiki OS – Contiki Branch of rklauck", [Online] `https://github.com/rklauck/contiki/blob/master/core/net/resolv.c`, 2013.