Sensor Virtualization with Self-Configuration and Flexible Interactions

Pål Evensen, Hein Meling
Department of Electrical Engineering and Computer Science
University of Stavanger, Norway
paal.evensen@gmail.com, hein.meling@uis.no

ABSTRACT

This paper presents the design and implementation of a simple and elegant middleware architecture providing *virtual sensors* as representatives for any type of physical sensors. With our middleware, external applications can seamlessly discover sensor-hosted services through Zeroconf and it provides a standardized communication interface that applications can use without having to deal with sensor-specific details. The limited capabilities of most types of sensors prevent the inclusion of a full communication stack with IP addressing. Yet, through the use of virtual sensors, a uniform communication interface based on UDP/TCP sockets can be exposed to external applications. This will significantly simplify application development for integrated services involving multiple types of sensors.

For evaluation and testing purposes we present a simple demonstration using SUN SPOT sensor devices connected to a laptop computer through a gateway device. The demonstration shows how the Zeroconf protocol can be used to automatically discover services hosted by a multitude of devices in the home, how to establish networking between the devices, and present the services in a browser window.

1. INTRODUCTION

Wireless communication technologies enables seamless communication between residential network entities such as set-top-boxes, sensors, control units and other devices, and are typically far less costly to install than their wired counterparts due to cabling. These technologies have opened up a whole range of new applications in the utility segment, like automatic meter reading (AMR) of power consumption, remote control of light and heating, security and safety systems, health monitoring and electrical appliances containing tiny embedded systems with networking capabilities in general. These services are beginning to become available to consumers as part of *smart home concepts* [13] offered by certain service providers on top of and integrated with traditional IP-based services such as Voice over IP, IP-based television and Video on Demand.

However, the heterogeneity of communication protocols and the mixture of addressing schemes used by networked devices of different make and model is one of the biggest challenges when developing integrated smart home services. Most smart home systems offered today are based on proprietary all-in-one solutions, where the sensors and actuators might use a proprietary RF protocol over the 868MHz band, while others might use ZigBee, Bluetooth, WiFi or another IEEE 802.x-based protocol over the 2.4GHz band. Furthermore, most devices have their own application-level protocol for communicating control commands and retrieving data. Moreover, due to the limited capabilities of many types of sensors, a full communication stack with IP addressing is simply unfeasible. Yet, it would significantly simplify application development if interaction with the sensors were based on UDP or TCP sockets and IP addressing schemes. Currently, these issues hampers innovation and development of new (possibly third-party) smart home services. Another obstacle to the adoption of smart home technology is the complexity of setting up and managing the networking between devices, deterring most home owners from acquiring such solutions. Hence, it is paramount to the success of networked homes that device configuration is performed automatically.

This paper presents the design and implementation of a simple and elegant middleware architecture providing *virtual sensors* as representatives for any type of heterogeneous physical sensors. A virtual sensor provides transparent discovery of arbitrary sensor devices through the use of Zeroconf protocols [5]. This enables *external applications* to discover sensor-hosted services through Zeroconf and it provides a standardized communication interface that applications can use without having to deal with sensor-specific details. That is, virtual sensors also provides a uniform communication interface to external applications, based on UDP/TCP sockets or even HTTP. This is accomplished by abstracting functionalities common to most sensor models, and writing custom wrappers (drivers) for the specifics of each sensor model. This way, applications need not know anything about the physical or logical communication protocols used by the sensors, making the same network services usable with any sensor model sharing the same basic functionality. For instance, a light-controlling application should be able to operate independently of the actual luminosity sensors used. Note that, the architecture is generic and can be used in a wide range of application areas where

sensors needs to be connected; however, for the sake of illustration, the examples presented here are framed in a smart home setting.

By using virtualized sensors, third-party developers do not need to learn any custom sensor APIs to interact with the sensors, even though the capabilities of the sensors are limited to low-level RF communication. Assuming sensor vendors provide the sensor communication API, third-party developers can supply the necessary custom wrappers for the middleware to use, or vendors can provide such wrappers. Virtual sensors gives flexibility to applications, since replacing sensor devices does not require modifying the implementation of applications using those sensors. This is assuming the basic interaction is the same or similar. Furthermore, with technology innovation, new sensor models may natively support Zeroconf and link-local IP addressing. Applications can then use these with minimal changes, bypassing the virtual sensors.

In previous work, Construct [14] offers a distributed middleware for pervasive systems and provides mechanisms for capturing sensor data and converting them into RDF formated data for storage. Construct employs Zeroconf to locate services, but does not allow discovery of sensor devices as in our middleware. In BOSS [10], a bridging architecture implemented in a base station is used to enable sensor access through the use of UPnP protocols. Our approach to virtualizing sensors based on Zeroconf is more lightweight and allows better application level adaptation.

The rest of this paper is organized as follows: Section 2 presents the background for the paper and state our assumptions. Section 3 presents the architecture of our middleware for virtualized sensors, and Section 4 provide some relevant implementation details. In Section 5 related research is discussed, and Section 6 concludes the paper.

2. BACKGROUND AND ASSUMPTIONS

The sensor virtualization middleware presented in this paper is part of the IS-home project [13], a larger effort aimed at offering an autonomic communication middleware platform to simplify development and deployment of integrated and context-aware services in a smart home environment. The middleware focuses on self-healing, and self-configuration mechanisms as discussed in this paper. It also offers support for developing integrated services, where multiple services can interact to offer synergies across different technologies, e.g. a light-control service could interact with the movement sensors associated with the alarm service to decide wether or not the light should be turned on or off. Context-awareness is currently supported by means of location awareness through both GPS and WiFi signal strength. These features are being used to as a means to activate and deactivate the burglar alarm in the home [16] based on the location of mobile phones associated with the household.

In the context of the IS-home project, we assume a networked device capable of running our middleware; typically this will be a simple *embedded computer* running the Linux operating system. Further, we assume the computer has multiple interconnection interfaces, e.g. ZigBee, Bluetooth, WiFi, GPRS, Ethernet and USB ports for connecting alternative network devices. This computer may run one or more network services, and may act as a gateway between different network applications and devices. A *network service* is a software component that may interact with other network services, to issue commands to sensors and actuators or to simply receive a data stream from sensors.

Zeroconf is an essential component in our middleware architecture; sensors and other network devices will be registered with and discoverable through Zeroconf. Hence, an overview of Zeroconf is given below.

2.1 Zero Configuration Networking

Zero configuration networking is endorsed by the Internet Engineering Task Force (IETF) [11], through various RFCs. There are several implementations of Zeroconf for different platforms, e.g. Bonjour for Mac and Windows and Avahi for Linux. Zeroconf has become a widespread protocol for automatically discovering external devices such as printers, cameras and iPods to communicate over an IP network. The protocol was designed for use in small (less than 1000 clients) local networks. In order to achieve automatic configuration of network devices, Zeroconf automates three core services: IP addressing, name resolution and service discovery [5]. In other words, IP addresses will need to be assigned automatically to each device and coupled with a meaningful name, and services have to be discovered automatically as they enter the network. This is achieved with the following combination of techniques [5]:

- *Link-local addressing* Used to assign IPv4 addresses without relying on a DCHP server present on the network: The device picks an IP address from the reserved local private range of 169.254.x.x at random and sends some ARP requests, asking for the owner. If no reply is received, the device answers its own request, claiming the ownership itself.
- *Multicast DNS (mDNS)* The reason for binding a name to the IP address is that IP addresses are impractical and difficult for humans to relate to, especially when picked randomly and often changing, as is the case with link-local addressing. The principle behind mDNS is the same: Basically, the device sends a few mDNS queries for a self-assigned name, and takes ownership if no other device answers. mDNS is used to provide name binding without a DNS server present.
- *DNS Service Discovery (DNSSD)* Enable users to browse for services without having to know anything about the hosts providing them. It builds on existing standard DNS queries and resource types and provides service discovery without a centralized directory service. Instead each Zeroconf enabled device maintains its own directory of services, as shown in Figure 1.

The philosophy behind the Zeroconf platform is rooted in the assumption that end users are interested in services, not devices. The goal is that users should be able to select services from a list through a graphical user interface. Figure 1 illustrates a Zeroconf-enabled home network, where a laptop running iTunes software, a printer, a webcam, an iPhone and

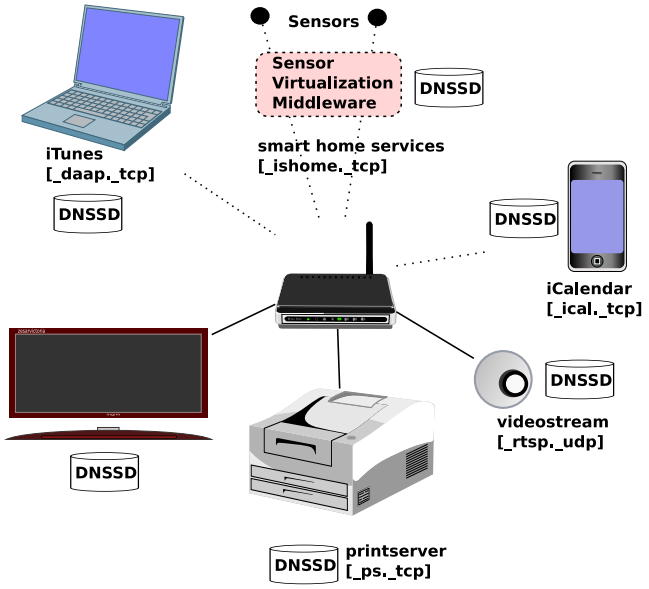

Figure 1: Zeroconf-enabled home network

sensors can access each others services. Services are advertised in the format <*Name*> <*Type*> <*Domain*> <*Port*>, where <*Name*> is the user-friendly name of the service, and <*Type*> is the service type. The webcam, for instance, will advertise its service as

```
"videostream" _rtsp._tcp local 554
```

indicating that it offers a video stream over the RTSP protocol on TCP port 554. Each device has its own DNSSD instance, keeping a list of available services. In the illustration, our middleware resides between the sensors and the rest of the network, as most sensor nodes does not have sufficient resources available to run their own DNSSD instances.

The list of services is kept up to date in a distributed and thought-out manner, using a combination of the following techniques to keep track of available services present on the network:

- Clients refreshes their local lists at irregular intervals, often as infrequent as once an hour, to keep network strain low.
- At startup, new services sends a few Multicast DNS packets, notifying all clients on the network of their presence.
- When services leave gracefully, they send a Multicast DNS goodbye message or use DNS Dynamic Update to remove its information from the Unicast server.
- If a service crashes, loses its network connection or in some other way leaves without being able to inform the network, the service stays in the clients lists until the next time a client refreshes its list, or tries to access the service, in which case the client removes the service from the list and informs the other clients.

Combined, these methods prevents the network from being flooded with control traffic.

3. ARCHITECTURE OVERVIEW

The middleware architecture is organized into multiple layers of abstraction to provide sensor-based services to external applications. That is, physical sensors appears to behave as if they provide Zeroconf-like services, denoted *virtual services* herein. Hence, the services provided to applications become independent of the sensor hardware used. Figure 2 shows a simplified overview of the platform, and its contextual place in the service architecture of a smart home.

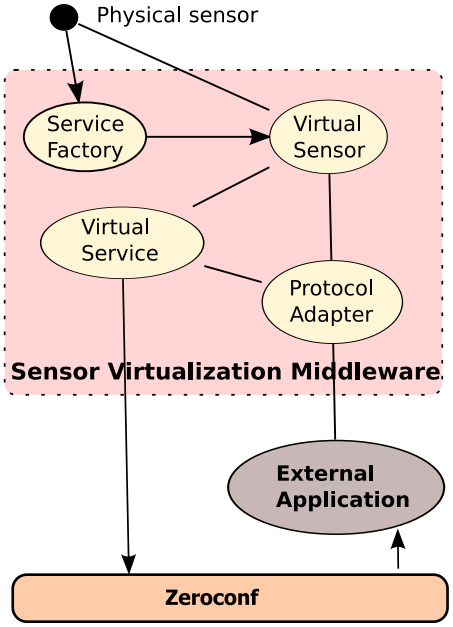

Figure 2: Middleware architecture

The middleware takes advantage of standardized Zeroconf protocols to provide automatic network configuration of sensors and service discovery to external applications, which makes the sensor services available to any Zeroconf-enabled application on the same network.

The main components of the middleware are:

- The **Service Factory** listens on the network for new sensor devices, and creates virtual representations of these.
- The **Virtual Sensor** communicates directly with the sensor nodes, and keeps track of connectivity. It translates application commands received through the protocol adapter and forwards it to the physical sensor, using the native communication protocol of the sensor.
- A **Virtual Service** represent a service provided by a sensor. It registers the communication endpoint (hostname and port number) of the service with Zeroconf and listens for connection requests from external applications. Upon receiving a connection request, the virtual service creates a protocol adapter to handle the communication with the application.

- Applications communicate with sensors through **Protocol Adapters**. They provide a standardized communications interface independent of the kind of sensor involved in the communication, and are generic for all virtual services. Once the application has established a connection with the protocol adapter, the adapter communicates directly with the virtual sensor. Communication from a sensor to external applications is done in the exact same manner, but in the reverse order.

The *Service Factory* and *Virtual Sensor* are the only components in our architecture that needs customization. That is, they are both comprised of a generic part, and a custom part that needs to be tailored specifically for each supported sensor type.

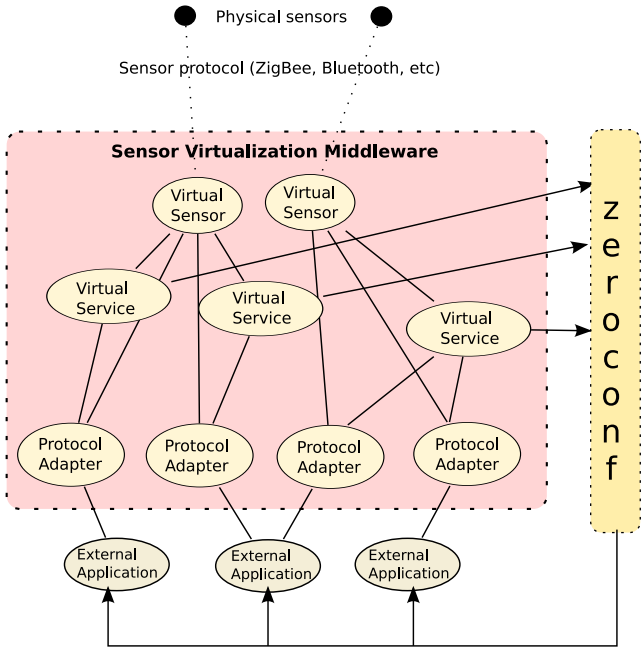

Figure 3: Detailed middleware architecture

Keeping in line with the service-oriented philosophy of Zeroconf, our middleware separates the services from the sensors. This appears to be the most flexible solution because it allows the system to support more than one service per sensor, e.g. a single sensor may provide both temperature and humidity services.

An alternative approach could be to have multiple services associated with a single *Virtual Sensor*, but that would clutter the architecture, making the virtual services less generic and requiring more logic to be added to the virtual sensor.

Another alternative approach could be to create one virtual sensor per service, making the system act as multiple services hosted by a single sensor are all hosted by different nodes. However, this solution appears somewhat messy. It would also require added logic to the *Virtual Sensor* as well as to the *Service Factory*, and it would not be a true representation of the physical reality, potentially causing problems with application logic further down the chain of development.

Having the virtual services separated from the virtual sensors allows the virtual service component to be generic for all supported sensor types. In addition, this approach is a good match with the Zeroconf APIs, as the methods provided by these are geared towards services instead of devices.

Figure 3 show a more detailed view of the system. Each physical sensor is represented by a corresponding virtual sensor. Furthermore, each service offered by the sensor is represented with a virtual service. A virtual sensor can have many services, e.g. if the same physical sensor device is a multi-sensor device, the different sensor readings can be offered to applications through distinct virtual services. A virtual service can also have many connections through different protocol adapters. For example, multiple services for the same sensor can be registered with Zeroconf at the same time, one accessible over TCP and another over SOAP.

External applications use Zeroconf to identify and locate virtual services provided by sensors, and communicates with them through the protocol adapter. An application can be composed of one or more services, but only needs one socket per service.

3.1 Adding New Sensor Types

Adding support for new types of sensors involves developing device-specific versions of the *Service Factory* and *Virtual Sensor*. In order to simplify development, the middleware comes with abstract versions of these components, allowing implementations to reuse common functionality, effectively giving developers a blueprint of the required classes.

Essentially, the custom part of the service factory needs code for detecting connection requests from the physical sensors and for creating the appropriate virtual sensor. Obviously, the virtual sensor must also be able to communicate natively with the physical sensors.

3.2 Adding New Communication Protocols

The protocol adapter is a generic communication interface through which external applications connect. Different applications might require different communication protocols, and the middleware supports adding new protocol adapters. Currently, a TCP-based protocol adapter is supported, yet support for UDP, HTTP, SOAP and RMI can easily be added, as shown in Figure 4. Once an adapter has been developed, it can be reused without modification for all sensor types supported by the middleware. In addition to making the middleware flexible, this ensures future compatibility with new protocols as they emerge.

4. IMPLEMENTATION DETAILS

Our sensor virtualization middleware is written in Java, with the core components represented in the classes ServiceFactory, Sensor, Service, TCPSocketAdapter and RegisterDNSSD.

The ServiceFactory maintains a list of sensors that the middleware is capable of communicating with. As the sequence diagram in Figure 5 illustrates, the ServiceFactory listens for

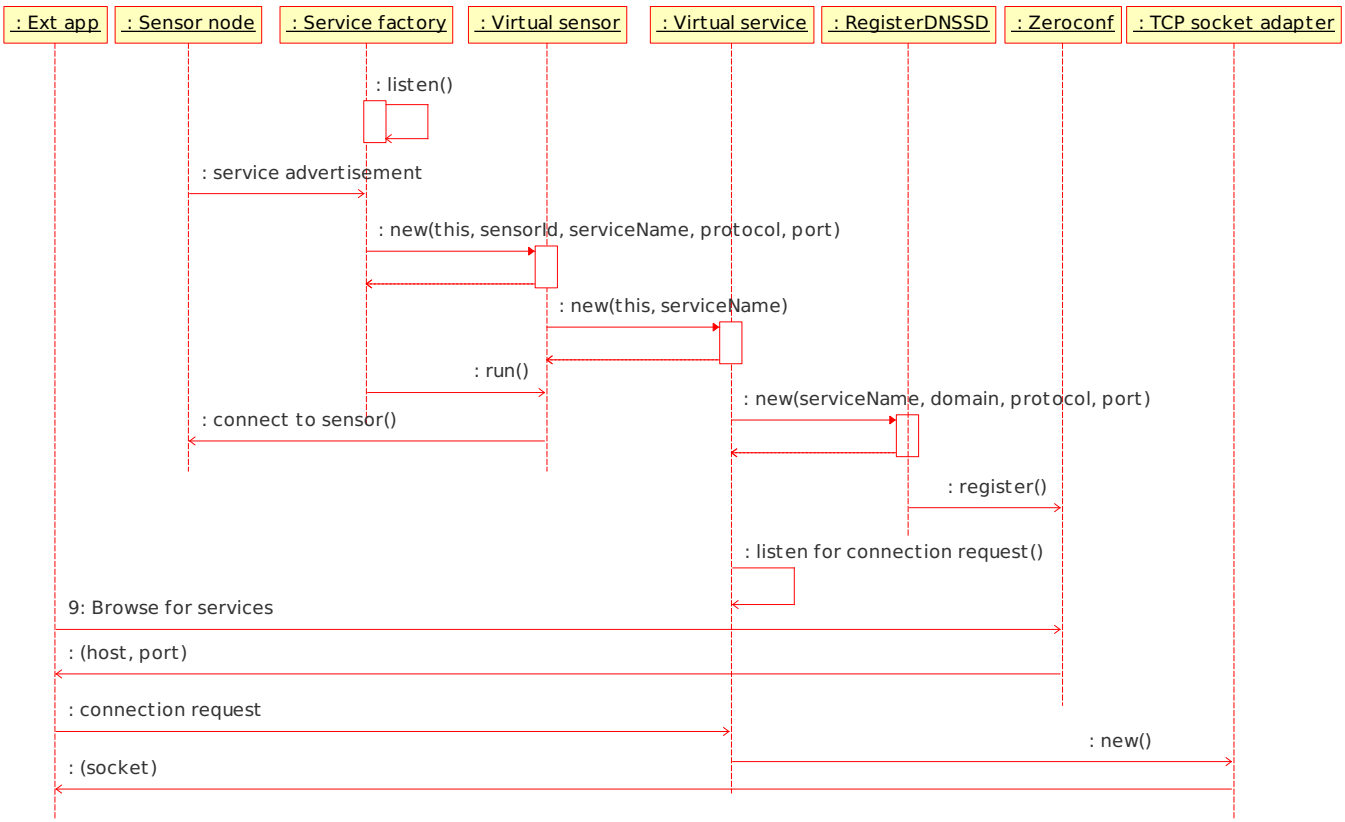

Figure 5: Service discovery and connection establishment

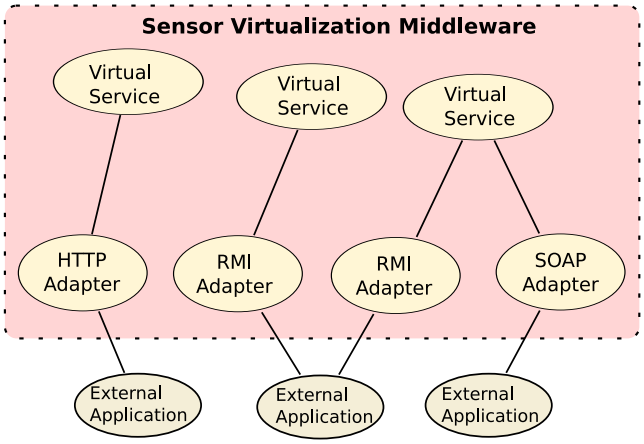

Figure 4: Protocol adapter

service advertisements broadcast by sensors in the network. When a broadcast is received, the ServiceFactory first checks if it already has the source sensor in its list. uses its "create new service"-method, giving the sensor a new Service in its list of available services. If it does not recognize the sensor that sent the advertisement, it creates a new Sensor, using a constructor that takes service name as argument. The Sensor constructor sets up one-to-one communication with the physical sensor and then creates a service. The VirtualService registers itself with Zeroconf. This is done using the RegisterDNSSD class, which implements classes from Apple's Bonjour API.

After the service has been registered with Zeroconf, it listens for socket requests on the corresponding TCP port, and spawns a TCPSocketAdapter, thread for each connection request.

External applications can multicast a DNSSD request for available services that resides on the same network and the Zeroconf framework will reply with the name of the host on which the virtual service is running, and the port number to connect to. An application can then use this information to send a connection request that will be handled by the virtual service, and get a TCP socket in return. Commands received by the TCP socket adapter is forwarded to the virtual sensor, which translates these into the appropriate sensor-specific command, which, in turn, is transmitted to the physical sensor, using the device's native communication protocol.

Each virtual sensor keeps track of the state of its associated physical sensor. A sensor is considered to have failed if an IOException is caught, e.g. due to a communications failure, or the ping timeout has expired. A random ping timeout is used with an interval $t_{ping} \in [10, 60]$ seconds. If a sensor fails, the virtual sensor is responsible for *unregistering* the service from Zeroconf, removing itself from the list of sensors maintained by the ServiceFactory, and terminate. Similarly, if an IOException is caught when external applications are trying to access the sensor, the virtual service will be unreg-

istered and terminated.

To demonstrate the capabilities of our middleware, we have developed a simple temperature reading application. The application uses programmable sensors from Sun Microsystems, called SPOTs (Small Programmable Object Technology). These are Java ME programmable sensors that comes with built in temperature detectors and accelerometers. We have implemented the SPOTs as virtual sensors and their temperature detectors as virtual services.

Communication is done with the wireless IEEE 802.15.4 standard, which uses the 2.4GHz band and is the standard the ZigBee protocol is built on top of. Since these particular sensors are programmable, they can be made to respond to any command we choose. In this case, an interface taken from the demonstration applications that came with the SPOT software development kit was modified to suit our application. The interface simply maps enums to bytes, and is implemented by both the client class running on the physical sensor and the virtual sensor class running within the middleware. The main reason for mapping enums to bytes in this manner is to improve the readability of the application code, and to conserve power when transmitting data (byte values are much smaller to transmit and easier to process than strings).

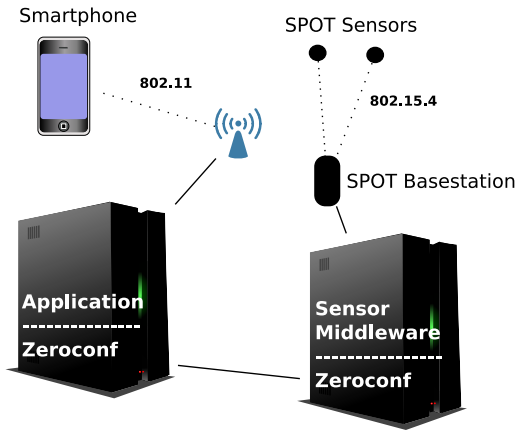

Figure 6: Demonstration implementation

In addition to the middleware, the implementation consists of the following applications:

- The **Simple Sensor Client** runs on the SPOTs and communicates wirelessly with the middleware over IEEE 802.15.4 using a base station that is connected to the host computer via USB. It is a multithreaded application that supports both broadcast and unicast over the SPOT specific Datagram protocol. As mentioned, it implements the PacketTypes interface, so that when receiving a byte value of 11 for instance, it knows that this is a temperature request.

- Running on an Apache Tomcat J2EE application server, the **Service Browser** application allows the user to browse for services and request values from these through a web browser. It uses the provided functions of Apples Bonjour API to browse and resolve services without requiring the user to perform any network configuration. The Apache Struts presentation framework is used to generate the web pages.

Using a class that implements the BrowseListener interface from the Zeroconf Java API allows the service browser application to find services on the network with very little code. The constructor:

```
new BrowseDNSSD("_ishome._tcp");
```

starts a thread that finds services of type `_ishome` that speaks TCP and keeps the application updated with any changes. When the user clicks on a service, displayed as a link on the webpage, the following method call is made:

```
DNSSD.resolve(0, DNSSD.ALL_INTERFACES, name,
"_ishome._udp", domain, this);
```

The call returns service name, host name and port, and using this information, the application can create a TCP socket to the virtual service, which is listening on the advertised port. Once connection is established, the application can issue commands and get value readouts from the physical sensor.

Although we have used a J2EE application server to do the service browsing in our demonstrator, it could just as well be performed by an applet running on a smartphone. While Figure 6 only shows one service browsing application, a real-world smart home scenario is likely to have a number of applications browsing for services using the Zeroconf protocol.

5. RELATED WORK

Zeroconf is not the only protocol providing service discovery and automatic network configuration: Some protocols, like Jini, are solutions to specific problems, while others, such as Construct [14, 6] provides a complete platform for developing pervasive applications.

Service Location Protocol (SLP) is an IETF proposed standard, and is supported by some of the largest industry actors, including Hewlett Packard, IBM, Sun Microsystems and Apple [2]. Apple did, however, replace SLP with DNSSD and mDNS as the preferred zeroconf protocol between Mac OS X 10.1 and 10.2, which makes the technology somewhat obsolete.

Jini is SUN's take on service handling, and as such, it is Java-based. Theoretically, any communication protocol that supports serialization of objects could be used, but since Jini is built on top of Remote Method Invocation (RMI), it is not practical to use other protocols. Jini systems are divided into Service, Lookup Service and Client components. Although SUN maintains that it is platform independent, only Java is used in practice [3]. It needs to run within a Java Virtual Machine (JVM), and it is rather heavyweight. Even though it supports the Java 2 Micro Edition (J2ME) virtual

machine, clients need to be able to dynamically download and execute Java classes, and small devices running J2ME typically does not have the processing power and resources to do this [7]. This can be worked around by including a proxy that executes the code and presents the data to the client through a servlet [15], but it nonetheless complicates matters.

Universal Plug and Play (UPnP) have many of the same objectives as Zeroconf, but while Zeroconf is a three layered foundation for automatic device configuration, UPnP is an organization, maintaining an open-ended collection of device specific protocols [4]. Whenever a new device type appears on the market, the UPnP forum creates a working group to develop a protocol for that particular type of device. The application protocols is built on top of standard internet protocols such as IP, TCP, UDP, SOAP/XML and HTTP to ensure platform independence.

UPnP offers automatic addressing, service discovery, and comes with protocols for controlling sensors and actuators.

A key difference between UPnP and Zeroconf is that while the UPnP organization is focusing mainly on application protocols without paying very much attention to the underlying layers, Zeroconf provides the underlying communication layers, but leaves it up to the developer to decide how the application protocol for a specific device is going to be implemented.

IP addressing is achieved in exactly the same manner as Zeroconf, using IPv4 link-local addressing. Unlike Zeroconf which have mDNS, UPnP does not handle name resolution and thus requires a DNS server present on the network to provide this. According to the UPnP Device Architecture definition [9], most often UPnP-enabled devices only provide URLs using numeric IP addresses.

UPnP-enabled components are either devices, hosting services, or control points, controlling devices.

For use in smart home applications UPnP does have some disadvantages: For one, UPnP use heavyweight SOAP XML objects over HTTP for communication, requiring an XML parser on both ends and at the same time increasing processing and bandwidth usage. Zerconf, on the other hand, uses standard DNS packets to advertise services, which are much smaller in comparison. Another problem with the UPnP protocol is its inherent chattiness; Its Simple Service Discovery Protocol (SSDP) was built on an IEFT draft which was abandoned in 1999, partly because the working committee recognized that the network would become flooded with control traffic in a setting with more than ten SSDP devices communicating. Another obstacle is that UPnP does not include support for prolonged periods of the network link being down, which is a likely occurrence in the noisy environment of a sensor-driven smart home.

Construct has many similarities with our middleware in that it is a distributed middleware for pervasive systems that provides an abstraction from the various protocols and data formats of sensors, actuators and other information devices. Like our middleware it is mainly written in Java and employs Zeroconf to locate services. Construct provides the following five core services: *Discovery*, *Management*, *Sensing*, *Actuation* and *Distribution*. Data is exchanged between system components in Resource Description Framework (RDF) format, which is the World Wide Web Consortium (W3C) metadata language that the Web Ontology Language (OWL) [17] language is built upon.

While our focus is on finding a standardized way for applications to communicate with sensors, the main focus of Construct appears to be on data capture and processing of information. In Construct, captured data are converted into RDF format and put in a distributed data store, accessible to other applications via the SPARQL query language or RDF over a TCP socket or via HTTP. Jena, a semantic web framework for Java is used to process the information.

Bridge Of the SensorS (BOSS) [10] is a middleware for UPnP enabling sensors that does not support the UPnP protocols. It achieves this by implementing a bridge in the base station that resides between the sensors and the applications that translates UPnP commands into the sensors native format. This approach have some similarities with our own, but instead of virtualizing the sensors, it provides a translation of commands.

Open Services Gateway initiative (OSGi) [1] provides a gateway for connecting different devices and services together through a central point, allowing applications to be composed from different, reusable service modules [8]. The framework is module based and written in Java. It only specifies the application programming interface, not the underlying implementation, leaving it up to the developers to handle the actual communication with the devices of a smart home.

SOCAM (acronym for Service-Oriented Context-Aware Middleware) is built on top of OSGi, and brings context-awareness to the table. The authors of the project has developed a context model that allows contexts occurring in a physical space to be converted into semantic data using OWL to represent their context ontologies.

6. CONCLUSIONS AND FUTURE WORK

By virtualizing the physical sensors in smart homes, we can provide external applications with a uniform communication interface. For a smart home to appear as seamless as possible to its inhabitants, it is important that the configuration of services and devices are handled in a manner that requires minimal user interaction. We have demonstrated how the important task of automating the discovery of services and devices as well as the networking between applications can be solved using Zeroconf.

While the middleware presented here makes the communication protocol between sensors and application generic, the application protocol is not. By implementing an ontology built with OWL, the application protocol could be made generic and platform independent as well. We also intend to expand our application to include support for other types of sensors beyond the Sun SPOTs supported in the current implementation.

Some utility services such as alarm and health services would greatly benefit from the middleware having autonomic computing [12] attributes such as self-healing and self-protection. These features could ensure the safety of heart patients under remote surveillance from the hospital where the successful transmission of monitored data is of great importance. Self-protecting and self-healing attributes of the middleware could also protect alarm systems from Denial of Service (DoS) attacks like network flooding or jamming.

Enabling remote access to the services in the home over wide area networks such as the Internet or GPRS can be useful for tasks like adjusting the heat before coming home, or turning off the alarm to let someone in when one is away. Remote accessibility brings up some security and privacy concerns that needs to be addressed by the middleware as well as the applications at some point. The drawback of exposing services on the network to enable synergies between applications is that they may be visible to malevolent outsiders. End users needs to be confident that their privacy and security is not compromised by the openness of their smart home.

7. REFERENCES

[1] OSGi Alliance. The osgi architecture. Accessed 15.02.2009.

[2] Ilkka Karvinen Bilhanan Silverajan, Jaakko Kalliosalo. Using ietf service discovery methods in ipv6 and middleware platforms and implementing slpv2 for ipv6. In *EUNICE 2003*. 2003.

[3] Blerta Bishaj. Comparison of service discovery protocols, 2007.

[4] Stuart Chesire. How does zeroconf compare with viiv/dlna/dhwg/upnp?

[5] Stuart Chesire and Daniel H. Steinberg. *Zero Configuration Networking - The Definitive Guide*. O'Reilly, 2006.

[6] Lorcan Coyle, Steve Neely, Graeme Stevenson, Mark Sullivan, Simon Dobson, and Paddy Nixon. Sensor fusion-based middleware for smart homes. pages 53–60. International Journal of Assistive Robotics and Mechatronics (IJARM), 2007.

[7] Ron Dearing. J2me clients with jini services, 2003.

[8] Pavlin Dobrev, David Famolari, Christian Kurzke, and Brent A Miller. Device and service discovery in home networks with osgi. In *IEEE Communications Magazine • August 2002*, pages 86–92. IEEE, 2002.

[9] UPnP Forum. Upnp device architecture 1.0, 2008.

[10] Kangwoo Lee Jongwoo Sung Hyungjoo Song, Daeyoung Kim. Upnp-based sensor network management architecture. In *Second International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2005)*. 2005.

[11] IETF. Internet engineering taskforce. http://www.ietf.org/.

[12] Richard Murch. *Autonomic Computing*. On Demand Series. IBM Press, 2004.

[13] Chunming Rong, Hein Meling, and Dagfinn Wåge. Towards integrated services for health monitoring. In *First International Workshop on Smart Homes for Tele-Health*, Niagara Falls, Canada, May 2007.

[14] Lorcan Coyle Steve Neely Graeme Stevenson Simon Dobson, Paddy Nixon and Graham Williamson. Construct: An open source pervasive systems platform. In *Consumer Communications and Networking Conference, 2007. CCNC 2007. 2007 4th IEEE*, pages 1203–1204. IEEE, 2007.

[15] Inc. SYS-CON Media. J2me clients with jini services. http://www2.sys-con.com/itsg/virtualcd/Java/archives/0806/patil/index.html, 2004.

[16] Jan Magne Tjensvold. Mobile Control System for Location Based Alarm Activation. Master's thesis, Dept. of Electrical Engineering and Computer Science, University of Stavanger, June 2008.

[17] W3C. Web ontology language. http://www.w3.org/TR/owl-features/.