The Design and Implementation of an Architecture for Flexible Access to Sensor Networks

Mats Uddenfeldt

Master's Thesis

Supervisor: Richard Gold
Examiner: Arnold Pears

Department of Information Technology
Uppsala University
Sweden

June 19, 2005

Abstract

In the light of the experiences from real-world sensor network deployments we are faced with the problem of accessing sensor networks. Sensor networks are made up of several sensor nodes using wireless communication to communicate. Due to the heterogeneity of sensor network implementations existing access methods are uniquely tailored to fit a particular sensor network deployment. During this thesis work an access architecture named Janus has been designed and implemented. Janus is designed to set a standard for providing uniform access to sensor networks. Janus offers a high degree of flexibility through on-demand dynamic negotiation of services and can be shown to support existing access techniques. Janus also separates the access services from the sensor network implementation allowing for easy upgrades and extensions. A prototype of Janus was also implemented to show the feasibility of the design. The prototype implementation has been successfully tested with a simple command line client connected to a sensor network made up of sensor nodes developed at Freie Universität in Berlin.

Contents

1 Introduction **3**
1.1 Problem description 4
1.2 Goals and methods 4
1.3 Thesis Structure 4

2 Background **5**
2.1 Sensor network scenario 5
2.2 Previous solutions 6
2.2.1 Event-driven access 6
2.2.2 On-demand access 7
2.2.3 Design requirements 8
2.3 SelNet 9
2.3.1 XRP 9
2.3.2 SAPF 10
2.3.3 SelNet in action 11

3 Design of the JANUS architecture **13**
3.1 Description 13
3.1.1 XRP Agent 14
3.1.2 XRP Engine 15
3.1.3 Gateway 16
3.2 Features 17
3.2.1 Flexibility 17
3.2.2 Modularity 18
3.2.3 Extensibility 19

4 Prototype Implementation **21**
4.1 Sensor network 21
4.2 Client 22
4.3 The Janus Architecture 22
4.3.1 XRP Agent 22
4.3.2 XRP Engine 23
4.3.3 Gateway 23
4.3.4 Signalling between Agent and Engine 24

5 Results **27**
5.1 Design 27
5.2 Prototype 28
5.3 Contribution 28

6 Future work **29**

7 Conclusions **30**

A List of XRP commands **32**
A.1 SUBSCRIBE 32
A.2 QUERY 32
A.3 API 32
A.4 DATA REPLY 32

B List of XRP parameters **33**
B.1 XRP parameter structure 33
B.2 REP_SEL 33
B.3 REP_ADDR 34
B.4 EVENT_LVL 34
B.5 DATA 34
B.6 FUNC_ARG 35
B.7 API_HDR 35
B.8 API_AREA 35
B.9 API_FUNC 36

C Example XRP messages **37**
C.1 QUERY for API 37
C.2 DATA REPLY with API 38
C.3 QUERY for function 39
C.4 DATA REPLY with result 39

D Demonstration GUI **40**

Chapter 1

Introduction

Sensor networks are an active field of research and their applications can be found in many different areas, such as environmental, habitat and intrusion monitoring. A sensor network can be defined as a network made up of nodes which can monitor their surroundings (e.g., sound, temperature or motion etc.) using on-board hardware sensors. Typically a sensor network is deployed using small, wireless sensor nodes which communicate using radio. The main advantage of having the sensor nodes form a network is that it is then possible to acquire their sensed data without physically having to access all of the individual nodes. Instead a sensor network relies on a special type of node called the sink. The sink acts as an access point to the sensor network. The most common approach to make that sensed data available is to have the sensor nodes report their data to the sink which in turn delivers it to a passive external service, such as a database. Having a sensor network run in complete isolation severely limits its usefulness.

Real-world implementations of sensor networks enforce this notion as they are often located in harsh or remote locations. This can make it difficult to access and maybe even find the nodes after deployment. Being able to retrieve information from within the sensor network without physical access is particularly useful if the sensors are used to monitor animal habitats [16], trapped under a thick layer of ice during a large portion of the year [17] or used for monitoring intrusion in empty buildings [9]. In the case of habitat monitoring the researchers obviously do not want to interfere more than absolutely necessary since it could potentially ruin their results. An example of this is the Great Duck Island habitat monitoring network [16]. In the words of Mainwaring et al.:

> "Although personnel may be on site for a few months each summer, the goal is zero on-site presence for maintenance and administration during the field season, except for installation and removal of nodes."

In order for a sensor network to be useful, real-world deployments have shown the need of external access mechanisms. After deployment there has to be a way to actively monitor and query the status of the sensor network to collect the data sensed by the network. Providing an external access mechanism to the sensor network is therefore crucial to the operation of the network.

1.1 Problem description

In the light of the experiences from real-world sensor network deployments we are faced with the problem of accessing sensor networks. Sensor networks are typically very low on resources such as memory, processing power, energy and connectivity. This results in a wide range of implementations and communication standards in an attempt to maximize the efficiency of each sensor network. External communication is handled by a sink node, which implements application specific support to interact with an external client. Due to the heterogeneity of sensor networks all existing access methods are to some extent uniquely tailored to fit a particular sensor network deployment. This means that it is difficult to use existing work to facilitate the implementation of new work. For every new type of sensor network we deploy, we would have to come up with a new solution. This can be very time consuming. Our hypothesis is that these methods can be replaced by a general access architecture which is flexible enough to be used together with any existing and future sensor networks.

1.2 Goals and methods

The main goal behind this thesis was to design an architecture to provide general and flexible access to sensor networks. The design should be flexible enough to encompass existing access solutions and extensible enough to work with future ones. To reach this goal the existing solutions had to be studied and evaluated in order to identify the important features required in a new design. Although it is difficult to perform a full evaluation of such a design, a prototype implementation can be used to show the feasibility of the solution. Thus when the evaluation was finished and the design was complete a prototype was implemented to test the design.

The work has been carried out in an experimental fashion. Initial ideas were discussed to give an understanding of their soundness. Those ideas that were found to be sound were implemented and tested to see if they should be investigated further. Successfully implemented ideas were then refined and further discussed.

1.3 Thesis Structure

Chapter 2 describes the technical background of the thesis work. This includes previous solutions to the problem of accessing sensor networks, the introduction of a sensor network scenario to be referenced throughout the thesis and a brief description of the network architecture which inspired our proposed solution. A description of our proposed access architecture and its signalling mechanisms follows in Chapter 3. The prototype implementation is presented in Chapter 4. Chapters 5 and 6 discuss the results of the implementation and possible future work. Chapter 7 concludes the thesis.

Chapter 2

Background

This chapter begins with the introduction of a typical sensor network scenario in which the sensors has been deployed for habitat monitoring. This is to have an example sensor network for the reader to relate to and we will continue to refer to this scenario in Chapter 3. Section 2.2 lists the design requirements which have been collected while evaluating previous solutions to the problem of accessing sensor networks. Our proposed architecture relies upon some of the concepts developed in the SelNet [11] project. We therefore conclude this chapter with Section 2.3 describing what SelNet is.

2.1 Sensor network scenario

A real-world example of a sensor network deployment is the Great Duck Island [16] as mentioned in the introduction. A wireless sensor network was deployed on an island to monitor the microclimates in and around nesting burrows used by a rare colony of birds. Their main goal, in which they succeeded, was to provide non-intrusive and non-disruptive monitoring of sensitive wildlife and habitats. In order to have a common example which we can relate to throughout this thesis we have decided to use a similar habitat monitoring network as our sensor network scenario. We have chosen this scenario because the Great Duck Island is a well-known example of a successful sensor network deployment. It also illustrates some of the questions researchers would like to ask the sensor network.

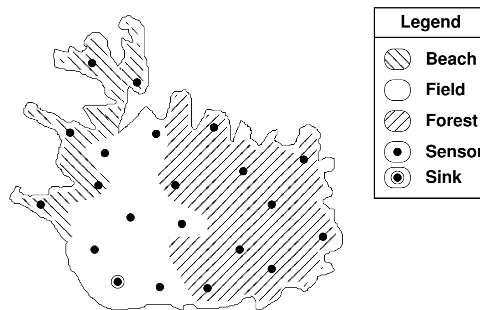


Figure 2.1: Sensors monitoring wildlife on a remote island.

In our scenario we want to monitor a rare colony of birds located on some remote, Atlantic island depicted in Figure 2.1. Our example island has three different terrain types: the beach, the field and the forest. The birds are spread out over the island and inhabit all three terrain types with varying density. Each sensor node has a microcontroller, a low-power radio, memory and is powered by batteries. To monitor their surroundings they have sensors for temperature, humidity, barometric pressure and infrared for motion detection. Researchers would be located outside the island and connect to the sensor network to discover the state of the sensors via a network.

In this habitat monitoring scenario typical questions the researchers would like to be able to ask to the sensor network may be:

- What is the temperature at a particular sensor right now?
- What is the logged barometric pressure of a particular sensor over the last week?
- Which of the sensors on the beach are detecting movement?
- What has been the average level of humidity in the forest for the past month?

The sensor nodes in Figure 2.1 have been placed on the island near the areas where the birds typically nest. One of the nodes acts as a sink and provides the sensor network with external communication capabilities. This node will be used by the researchers to pose questions to the sensor network and by the nodes to report certain events to the researchers. We will leave the example sensor network scenario for now, but we will return to the scenario and these questions when discussing the features of our proposed design in Section 3.2.

2.2 Previous solutions

Faced with this challenge of accessing sensor networks in a generic way we will have to study existing solutions to identify the important features required for the successful design of a general access architecture. In the following section we will present some of these solutions and evaluate them. A sensor network is typically accessed through a special node called the sink. The sink node acts as the access point to the sensor network and a sensor network can have one or more sinks. As mentioned in Chapter 1 the most common approach is to have the sink gather data from the other sensor nodes and report it to an external service. The existing solutions to provide external access to a sensor network can be divided into two categories:

- **Event-driven access** relies on data to be propagated to the outside of the network to be collected when a specific event occurs.
- **On-demand access** relies on a gateway which makes it is possible to access the sensor network directly and send queries to individual nodes.

2.2.1 Event-driven access

The easiest way to provide access to the sensed data of a sensor network is to have the nodes report their data to an external location. Most deployed wireless sensor networks [3, 9, 16] have implemented this type of event-driven monitoring. The basic idea is that you have a sink inside the sensor network which can be programmed to react

to events inside the network. At a given time interval or when certain preprogrammed events occur, data is propagated through the sensor network and collected at the sink. The sink can either log the data for later processing, or transmit the data to a connected external entity. The external entity would run a service, typically some database interface, which would allow the data to be accessed via a network.

All event-driven access approaches require a decision to be made in regards to which data is interesting and at which intervals they should be reported before the sensor network is deployed. After the point of deployment the external service will passively collect the data generated inside the network. It is possible to improve the performance of the event-driven approaches by having data reported frequently or by programming the nodes with thresholds to immediately report certain events. The major weakness lies in the inability to pose direct queries to interesting nodes or areas within the network.

2.2.2 On-demand access

The on-demand access approaches provide a more advanced interface to the sensor network, since it allows direct access to query the sensor nodes. Using this on-demand approach it is possible to mark certain nodes of the networks as more interesting than others and decide to poll them more frequently.

Plutarch [5] is a network architecture for bridging disparate networking contexts to form a cohesive network. A context is a group of network elements that share naming, addressing, routing and transport protocol. Contexts are bridged by the use of interstitial functions (IFs) which are inserted between contexts to map between them. In this sense Plutarch could be used to bridge an Internet type network with a sensor network, but this has not been implemented yet.

Delay Tolerant Networking (DTN) [10] is a network architecture that is designed for environments with intermittent connectivity, scheduled transmissions, and possibly very long propagation delays. DTN provides an overlying *convergence layer* abstraction which allows DTN to be run on top of both TCP/IP and a sensor network protocol. DTN have been used to connect sensor networks to the Internet [7] by inserting a DTN gateway at the interconnection point.

Both Plutarch and DTN offer a solution in which we can insert queries into the sensor network by constructing packets that adhere to the protocol defined by the sensor network. These packets can then be injected into the sensor network which will respond via the gateway. These solutions require the connecting client to have intimate knowledge of the sensor network implementation and the way the sensor nodes can be accessed. Because of this it can be very time-consuming to provide support for another sensor network type.

Buonadonna et al. [4] have developed the Sensor Network Appliance (SNA). The SNA is a special purpose gateway running Linux. It has a radio module that is used to communicate with the wireless sensors, an Ethernet connection, as well as support for satellite or cellular connectivity. To offer remote access for the sensor network the SNA runs an ODBC-compliant database management system and an HTTP server.

The SNA offers a very useful solution, as it provides multiple services and a way to access the sensor network without the requirement of intimate knowledge of the sensor network implementation. However, by implementing the database management system and the HTTP server at the gateway they have made modifications and upgrades to the interface difficult. The motivation behind upgrades or modifications might be to roll in security patches or to change the way the network is presented by the interface.

Since the gateway is deployed together with the sensor network it suffers from the same limitations in regards to physical access.

SNMP [13] provides a standard interface for accessing the resources of the network. However the expressiveness of SNMP is a concern. It provides a get / set operation for a Management Information Base (MIB). This provides the ability for both event-driven and on-demand types of queries and responses from the sensor network. However, it is not straightforward to send specialized queries into the sensor network from outside the sensor network. With SNMP, we are limited by what information and events that are defined in our MIB at the time of deployment. SNMP could potentially be useful for querying a sensor network if we could make the assumption that the gateway node to the sensor network will never be a sensor network node itself. Since SNMP uses ASN.1 for representation it is not suitable for implementation in severly memory-constrained devices.

Similar concerns about implementation in constrained environments apply to existing RPC-style systems such as SUNRPC [12], Java RMI [18], CORBA [22] and WSDL [21]. The representation systems used such as XDR or XML were not designed for devices which are extremely memory-constrained such as sensor network nodes. We must be mindful of the resources consumption of the tools that we choose.

2.2.3 Design requirements

During the study of the existing solutions the following three features have been identified as important issues when designing an external access architecture:

- **Flexibility**: The signalling has to be flexible in order to encompass both event-driven and on-demand access approaches and support negotiation for offered sensor network capabilities.
- **Modularity**: The architecture needs interchangeable modules to allow several sensor network types and access technologies to be supported.
- **Extensibility**: It should be possible to allow new and upgraded access services after deployment of a sensor network.

All three features are part of addressing the goal to achieve a general and flexible access architecture for sensor networks.

First the signalling mechanism has to be flexible in order to support both event-driven and on-demand access approaches. Event-driven approaches are useful to aggregate data over time, to react to programmed thresholds or events and to collect data at an external point. On-demand approaches make it possible to monitor the sensor network in a direct way, select which nodes are interesting given a certain context, and to direct queries to nodes and areas in the network. Second the signalling mechanism should also allow for dynamic negotiation of the offered sensor network capabilities. By supporting this we gain two things: We can expand the functionality of the sensor network without changing the access method, and we have a way to implement access control allowing for varying degrees of control over the network. Therefore the flexibility of the signalling is imperative for a successful solution to our problem.

The next issue is the modularity of the access architecture. Because of the heterogeneous nature of existing sensor network implementations modularity in the implementation is important if we want to be able to use the same access architecture for all

network types. Having interchangeable modules for sensor network access and functionality makes it possible to add support for new sensor network types and hardware without affecting other parts of the architecture.

In order to extend the longevity and usefulness of a sensor network deployment, we also see the need of extensibility. Upon original design and deployment of a sensor network it might be hard to foresee what the future needs will be. We might want to add support for new access services. Also we might want to upgrade the access services due to be able to roll in security patches or fix bugs.

2.3 SelNet

Our proposed access architecture has its roots in SelNet, an architecture developed in the Communications Research group of Uppsala University, and this section describes SelNet to give an explanation to some of its properties.

SelNet is an architecture that aims to create an abstraction layer that provides basic demultiplexing services over a network. The architecture inserts this extra functionality between the network and link layer in the IP stack. SelNet builds a virtual network on level 2.5 which handles routing in a way invisible to the upper layers. The nodes in SelNet are addressed using labels called *selectors* [1]. An incoming packet is sent to the appropriate packet processing function based on the selector. The selector label is opaque and unique in the local context only, but there is a special selector value which is common to all SelNet nodes. This special selector is defined to receive and process XRP (eXtensible Resolution Protocol) messages. XRP is used for SelNet's control messages and provides an interface to access and configure other nodes. SelNet's packet format is called SAPF (Simple Active Packet Format).

An example of an existing application that uses the concepts of the SelNet architecture is LUNAR (Lightweight Underlay Network Ad-Hoc Routing) [19], which is an ad-hoc routing protocol and uses it to tell neighbors how to route traffic to the destination.

2.3.1 XRP

The eXtensible Resolution Protocol (XRP) is SelNet's format for control messages. These control messages are used for resolution requests and replies, because selectors have to be negotiated before any of the packet processing functions can be accessed. The name has its roots in this mandatory *resolution* step. An example taken from LUNAR is when we want to resolve an IP address to a multihop path through the network. SelNet would then set up the necessary state in the local node and in the network, and return a label in the form of a selector value. In the next step we can use this selector to send packets via the negotiated path.

Figure 2.2 shows the structure of the XRP packet format. The XRP header is four bytes long and begins with an XRP command. In the most simple scenario the XRP command is either a request or a reply. The rest of the header contains the Time To Live (TTL) which is used to limit the spread of messages in the network. The final part of the header is reserved for future use. The XRP command header is followed by one or many XRP command parameters. The parameters can be of variable length and their semantics is given by a *class* field, which names the parameter, and a *class-type*,

[1] Hence the name SelNet from "Selector Network".

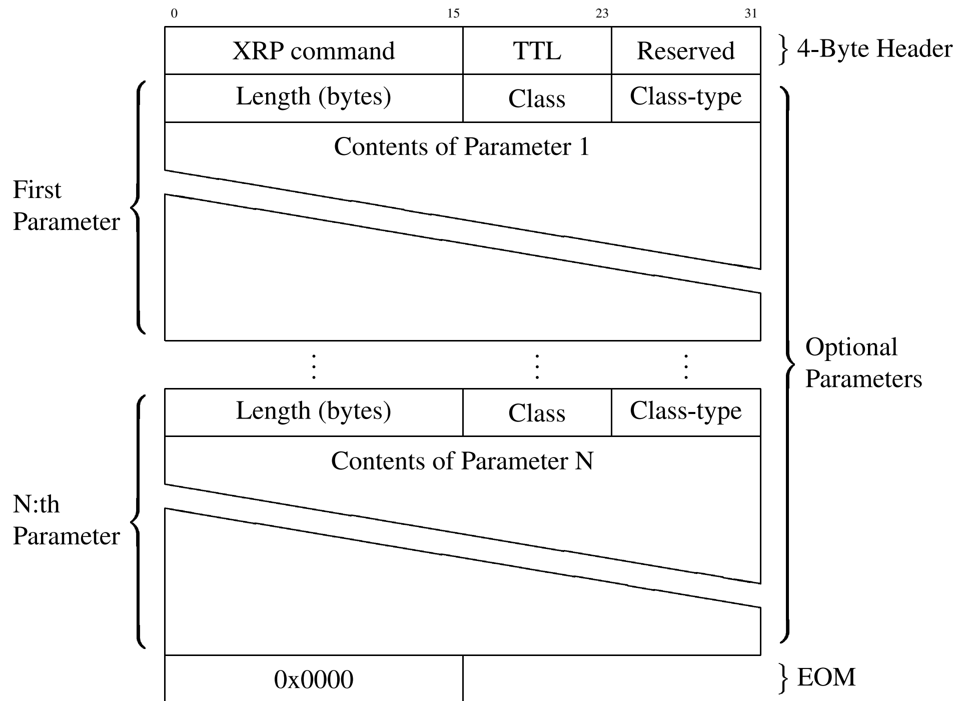


Figure 2.2: The XRP packet format.

which defines the parameter's data type. XRP parameters can appear anywhere inside the parameter block, thus are not bound to a specific position. A null object closes the parameter list, making XRP a self-delimiting packet format.

To send an XRP command to a remote host the sender must first create a reply-channel. This is done by installing a reply function and labeling it with a new selector. This selector value is appended to the XRP command to be transmitted and used by the receiving host to direct the reply. When an XRP command is received it is interpreted and executed on the receiving host. It can therefore be used to trigger remote functionality through the use of specialized packet processing functions.

2.3.2 SAPF

The Simple Active Packet Format (SAPF) is SelNet's packet format. It is used independent of which kind of packet we want to send through the network.

The packet format relates to SelNet's simple demultiplexing philosophy and only carries a selector identify its destination and a data payload. The data payload could either be any XRP command (eg. requests, replies) or pure data traffic. As can be seen in Figure 2.3 the SAPF selector has a fixed length of 64 bits and the data payload has variable length. This means that SAPF packets have to be encapsulated in a protocol which contains information about packet lenght, such as IP or Ethernet. All selectors are located in a local SAPF hash table so they can be looked up quickly. The selectors in the SAPF table can be redirected to another selector or a callback packet processing function along with the data payload as argument.

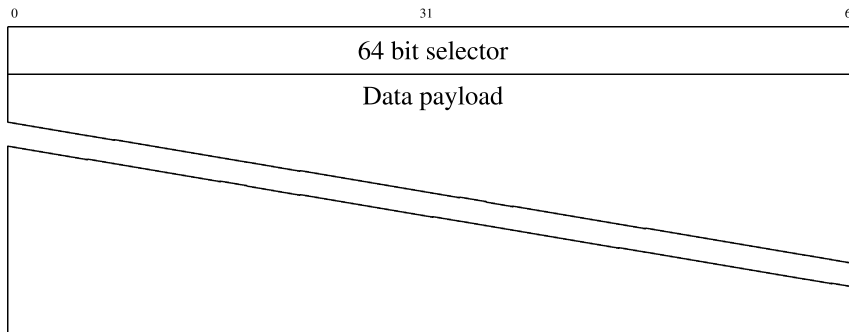


Figure 2.3: The SAPF packet format.

2.3.3 SelNet in action

In a simple example of packet forwarding XRP would be used to resolve the address of the destination end point. When the resolution process completes, there is a SAPF path between the two end points. This path can then be used for forwarding packets beneath the IP layer. One of the key design decisions of the SelNet approach is to separate the control and data planes of the IP header so that the control plane can be controlled and manipulated separately to forwarding. The functionality from the IP header (e.g., identifying communicating end points) are transfered into the XRP resolution protocol where it can be extended and managed. This leaves the task of generic packet forwarding to simple label-switching.

The simplest form of resolution that SelNet can perform is in the style of ARP where a node wishes to resolve an IP address to an Ethernet address. We can see a diagram of this event in Figure 2.4. The diagram also shows the position of SelNet as a virtualized link layer in relation to the other layers of the protocol stack. When SelNet Node A wants to start sending packages to Node B it starts by looking up the receiving host via an XRP request similar to a DNS lookup. This request specifies:

1. The type of address that node A wishes to resolve (e.g., an IP address)

2. What it wishes the address to be resolved into (e.g., an Ethernet address)

Before Node A sends out it's lookup request, it installs a function to handle the reply. This function is identified by a unique selector and will be used by Node B to direct the reply. The lookup request from Node A is propagated through the network while leaving a trail of reply-functions on each and every host it passes. Each node keeps track of these reply-functions using a hash table containing SAPF selectors associated with a forwarding function. When Node B receives this request for resolution it will

1. Check if the requested address is the same as its own

2. If it is the same, check if the resolution is permitted

3. If it is permitted, send back a resolution reply (RREP) to Node A

The resolution reply is sent back to Node A using this chain of reply-functions to pass the XRP reply back to the originating Node A. It contains a **network pointer** [20] which is a tuple consisting of a selector and a link layer address. The selector in this case represents the IP stack of Node B. Node A can now use this network pointer to transmit data which will reach this packet processing functions on Node B. In this case the packet processing function triggers an IP delivery of the data it receives, but it could also forward data to another node or in reality represent any computing function on the node. We will later show how we use this feature to provide access to a connected sensor network. This can be made possible by designing specialized packet processing functions, which can be invoked using XRP commands.

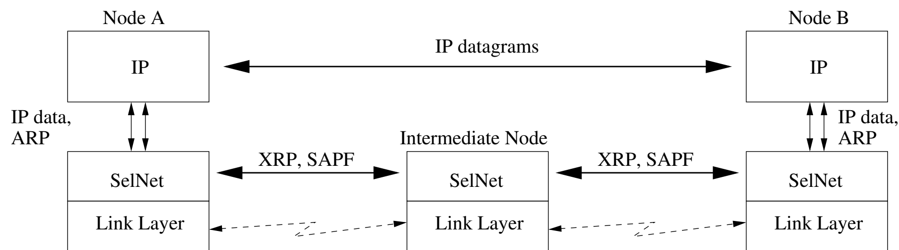


Figure 2.4: SelNet network overview.

Chapter 3

Design of the JANUS architecture

The following chapter presents the Janus architecture as a proposed solution to the problem of accessing sensor networks. Section 3.1 describes the different parts of the Janus architecture: the XRP agent, the XRP engine and the gateway. We will describe their role in the Janus architecture and try to give some insights to the design decisions as they are introduced. The design was intended to be flexible, modular and extensible and we will address how Janus meets these requirements in Sections 3.2.1, 3.2.2 and 3.2.3 by referring to the example scenario from Section 2.1.

3.1 Description

The Janus architecture is a middleware system which is placed between the sensor network and the external access network. The architecture itself is divided into two separate subsystems: one that faces the sensor network and one that faces the access network. This duality in the architecture led to adopting the name Janus from the two-faced Roman god of doorways. An overview of the base architecture can be seen in Figure 3.1.

The **XRP Agent** exchanges XRP messages with the XRP Engine in order to set up an interface to the functionality offered. Before the initial negotiation the XRP Agent has no knowledge of the sensor network except the address of the gateway. The XRP Agent also provides the external connectivity of the Janus architecture. It can either be implemented as an integral part of a connecting client or as a type of proxy which can be used to gain access to the sensor network. A client can then use the negotiated interface to receive event reports and to query the sensor network.

The **XRP Engine** provides the access network with an interface to reach the sensor network resources. This interface is reminiscent of the RPC (Remote Procedure Call) client-server model. Requests to invoke local functions comes in over the network in the form of XRP messages from the XRP Agent. These commands are then interpreted as queries for the sensor network resources.

The **Gateway** is located between the external access network and the sensor network. It acts as a typical sensor network gateway in the way that it is the only ingress / egress point between the access network and the sensor network. This means that it handles both incoming connections from the access network as well as the reports

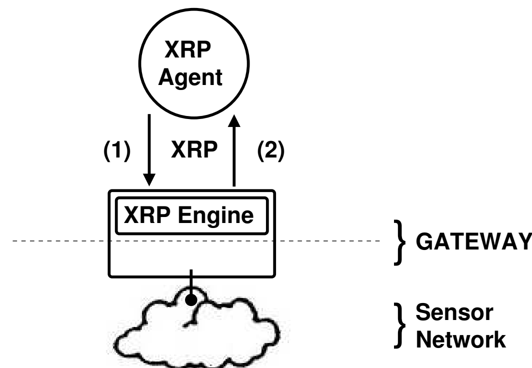


Figure 3.1: Base architecture overview: (1) XRP requests sent by the agent (2) XRP replies sent by the engine.

and events originating from within the sensor network. In addition to this the Janus gateway contains an XRP Engine which processes incoming XRP commands.

To make this thesis easier to read we will refer to the XRP Agent as the *agent* and the XRP Engine as the *engine* from this point and forward.

The engine and the agent communicates using the eXtended Resolution Protocol (XRP) which was introduced in Section 2.3.1. The XRP protocol offers very flexible semantics. An XRP message begins with a command and is followed by an optional number of parameters. In the Janus architecture we use the XRP protocol to provide a dynamic RPC-style interface to the sensor network.

The signalling in the Janus architecture takes places between the two XRP entities: the engine running on the gateway of the sensor network, and the agent that communicates with the engine and provides a way for external clients to connect to the architecture. One of the main reasons to split the architecture into two separate entities was that it allows us to separate the access services from the sensor network implementation. The gateway provides the architecture with all the necessary sensor network specific knowledge. The signalling between the agent and engine is not dependent on the way that the gateway accesses the sensor network or the way that an external client connects to the architecture. The way in which we use the XRP protocol and the messages exchanged are explained in more depth in Section 4.3.4. A full list of the XRP commands, parameters and messages used in our implementation can be found in the appendix section of this thesis.

3.1.1 XRP Agent

The role of the XRP agent is to exchange XRP messages with the engine to query the resources of the sensor network. The agent exchanges XRP messages with the server in order to dynamically discover available sensor network resources, to send queries to the gateway concerning the state of the sensor network and to receive information from the gateway about the events in the sensor network. Figure 3.2 shows a client that has connected to an agent in order to send queries to the sensor network. The agent acts as the translation point between the client and the engine to provide access to the sensor network resources.

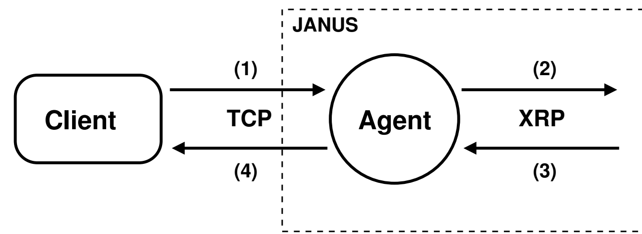


Figure 3.2: The client sends queries to the agent **(1)** which will translate these into XRP commands to an engine **(2)**. When the reply has been acquired from the sensor network the engine will reply **(3)** with an XRP message that is translated and sent to the client **(4)**.

Janus uses dynamic discovery of sensor network resources by letting the agent negotiate available services with the engine on the gateway. This is particularily usefulin situations where the agent does not have explicit knowledge of the exact nature of a sensor network. It enables the agent to find out which resources are available to it via the engine. One of the main advantages of the Janus architecture is that the same agent can be used to access completely different sensor networks without being forced to implement this explicit knowledge. After the API has been negotiated between the agent and the gateway the agent can issue queries to the sensor network through the gateway in a way similar to the RPC client-server model. This is done by sending a request XRP messages over the network targeting one of the selectors exported with the API. Examples of these requests involve querying the current temperature, humidity levels and motion sensing. Through the interface offered by the gateway it is also possible for an agent to register an interest for a particular type of event within the sensor network. The agent will then be notified when this specific event occurs. Examples of these events include when motion is detected anywhere inside the network or when there is a significant change in temperature in a given region. The agent can be invoked by separate access servers to provide user interaction and present data to legacy clients.

3.1.2 XRP Engine

The engine is in charge of the agent interaction on the gateway. This interaction takes the form of XRP messages sent back and forth between the agent and the engine. When an agent has connected to the gateway it begins to issue XRP commands. These commands are interpreted by the engine. The agent can discover the available sensor network resources in the gateway by sending a request for the API to the function module on the API. The engine processes this request and composes an XRP message with the API represented as XRP parameters. This XRP message is then transmitted back to the agent. This messages contains a list of the selectors bound to the individual functions of the function module on the gateway.

Functions are invoked on the gateway when the engine processes an XRP message targeting one of these selectors in a way similar to the RPC-style invocation. When a local function is triggered it will communicate directly with the sensor network to retrieve the answer to the posed question, build a reply message containing this answer and transmit it back to the agent.

3.1.3 Gateway

The gateway sits between the sensor network and the access network and contains an engine as seen in Figure 3.3. Its role is to handle incoming connections from agents in the access network and to listen to incoming events from the sensor network. When an agent connects to the gateway it can begin to issue XRP commands, which are interpreted by the engine at the gateway.

The gateway consists of two modules: one that implements the methods in which the sensor network can be accessed and one that implements the functionality of the actual sensor nodes in the network. The access module is concerned with how to write and interpret packets destined for and arriving from the sensor network and exerts control on this traffic. The functionality module has knowledge about which basic functionality has been implemented on the nodes and how their sensors can be queried over the network. It is also possible to add more advanced composed functions to give answers to complex queries (e.g., about areas or selected nodes in the network). We return to the discussion of the modularity of the gateway in Section 3.2.2.

For every sensor network accessed with the Janus architecture there would have to be at least one gateway. This is to provide the sensor network with an ingress / egress point in which it can be accessed. In our examples scenario from Section 2.1 the gateway would be in direct contact with the sink node of the sensor network. If the need arises to create a gateway for a previously unsupported sensor network only the access and function modules of the gateway needs to be coded. In other words it would be necessary to implement an access method to be able to communicate with the network and functions which can be used to query the sensor hardware on the actual sensor nodes. Due to the dynamic nature of the Janus architecture adding a new sensor network type like this would not warrant any changes outside the gateway. If we decide to upgrade our nodes with a new type of sensors, we would only have to change the functionality module and could leave the access module intact.

The functionality module contains all the functions which can be used to query the sensor network. These functions can be exported using XRP to allow an agent to access and query the sensor network. Simplified, the API is represented as a list of the available functions and the corresponding selectors used to invoke them. The list is implemented using the special API-style XRP messages and is described in more detail in Section 4.3.4.

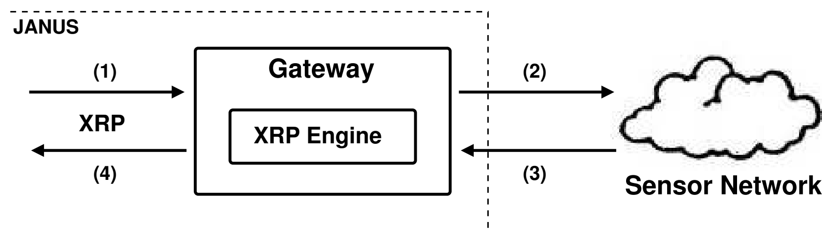


Figure 3.3: An XRP message comes in from an agent in the Janus network **(1)** and is received by the gateway which hands it over to the engine for processing. The engine invokes a packet processing function which queries the sensor network **(2)**. The reply comes back from the sensor network **(3)** and is passed on to the agent **(4)**.

3.2 Features

The design was intended to show flexibility, modularity and extensibility. In this section we discuss these issues and some details of the design that are relevant to them. Before addressing these, we first review our basic design.

Janus is organized as a middleware system in which an external client can access the resources of a sensor network. In the basic design a gateway sits at the edge of the sensor network and implements access to functionality on the nodes. In our example scenario researchers are using a client to connect to Janus in order to query the status of one or more of the sensors in the bird colony they are monitoring. This could be a simple question as: *What is the temperature at a particular sensor right now?*. The agent translates this query into an XRP command and sends it to the gateway. When the gateway receives an XRP command, this command is handed to and interpreted by the engine. This command could for instance be the invocation of a function, in this case the function to query the temperature of a single node, which is subsequently executed. The answer from the sensor network (i.e., the temperature of the node) is reported back to the agent. This process continues until the researchers disconnects from the agent.

The performance and usefulness of a sensor network depends greatly on the nature of the access architecture. This is in part because of the limited resources in regards to memory, processing power, bandwidth and energy on the nodes. In Section 2.2.3 we identified the need of an architecture which exhibits:

- Flexible signalling to encompass existing access solutions and support on-demand negotiation for sensor network resources.
- Interchangeable modules to allow several sensor network types to be supported.
- The possibility to extend and add access services after deployment of the sensor network.

3.2.1 Flexibility

In the list of requirements from Section 2.2.3 we put flexibility in the signalling mechanism as the most important feature of all. The reason behind this was that it is only with flexible signalling we achieve support for both event-driven and on-demand access approaches and the ability to provide on-demand negotiation for the sensor network capabilities. It is also important to be able to encompass future access methods.

We have achieved this flexibility by using the XRP protocol to exchange messages between the agent and the engine. The XRP protocol has several properties which make it a very good candidate for the Janus architecture. Due to the nature of XRP as an interpreted messaging protocol it is very expressive. An XRP message is a command that carries arguments in the form of parameters. By using XRP to place a query for some information or data in the sensor network, this query has to be processed by the engine in order to return the appropriate the result. This is where the interpretation takes place and where we can see the expressiveness of the XRP protocol. In contrast to using a static packet format, XRP performs signalling to set up functions on the engine to perform the tasks the agent has requested. The reasoning behind this design choice is that it is easier for a signalling protocol to express a new piece of functionality through its instruction set than a packet header format to do the same.

In the passive approach the agent would register the interest for a particular event at the engine. This subscription could either be limited to a special type of event or use

a more general setting for the level of verbosity. In the scenario from Section 2.1 this could be represented by the question: *Are any of the sensors on the beach detecting movement?* When this event occurs inside the sensor network it will be propagated through the nodes to the gateway. The gateway will then consult its subscription list and send an XRP message to the interested agents informing them and the connected researchers of the event.

In the active approach the agent acts in a way similar to how an RPC client would, by sending requests for function invocation at the engine. This function would be implemented to give answers to questions about the registered sensor data of a particular node in the network (e.g., *What is the logged barometric pressure of a particular sensor over the last week?*). By invoking a function the tasks that the agent has requested are performed and the result is returned to the agent.

The RPC-style interface used by an agent to invoke functions is negotiated through the use of XRP messages to request and describe the API of the gateway. This on-demand negotiation gives us the option of adding access control on the gateway, but more importantly it gives us an extra degree of freedom when it comes to changing and expanding the functionality of the sensor network. It also allows the agent to access multiple different gateways as long as they follow the Janus architecture.

3.2.2 Modularity

Due to the heterogeneity of sensor networks it is important to have modularity in the access architecture. This is particularly true if we want to be able to use the architecture as a general replacement of existing approaches. Janus achieves this modularity by having the sensor network access method and functions as separate modules in the gateway. This is where the interface to either access collected data or to query the sensor network is placed. The three main methods used in real-world sensor network deployments today are: directed data diffusion [**?**], database abstractions [15] and event-driven monitoring. Event-driven monitoring has already been discussed in Section 3.2.1, but we will turn to the other two and see how each of these can be used within the Janus architecture. Dunkels et al. describes the following examples [8].

Directed diffusion [14] is a data-centric routing protocol for sensor networks. The protocol is based on a model in which a sink node first registers a *data interest* with the network. When the interest has been propagated through the network, the sensor network begins sending data in the reverse direction of the interest propagation. Directed diffusion fits well with the Janus architecture and only requires a properly implemented access module in the gateway. The initial XRP query from the agent to the gateway causes a data interest to be registered with the network. The gateway will wait for the reply from the sensor network and report the answer back to the agent when it is available. In Chapter 6 we discuss the possibility of injecting XRP queries all the way into the sensor network using directed diffusion.

TinyDB [15] is a widely used sensor network database abstraction. TinyDB makes the sensor network appear as a database that can be queried for data from the outside. Data queries are made using an SQL-like syntax, and data is transported through the sensor network back to the gateway. Data aggregation is used to reduce the amount of communication within the network. The database abstraction model also works well with Janus. The XRP query from the agent will be translated into a specific SQL query which will be delivered to the network. After this the gateway will wait for the reply and report the answer back to the agent in standard fashion.

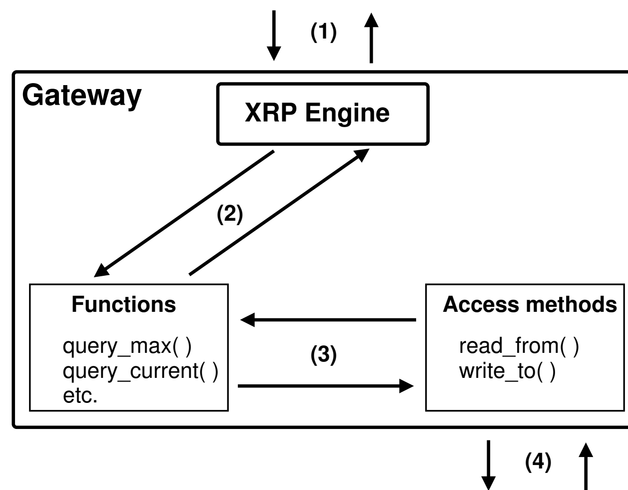


Figure 3.4: Interchangeable modules in the gateway. Incoming XRP commands **(1)** are interpreted by the engine and invoke the corresponding functions **(2)** from the functionality module. The access module is invoked using standardized calls **(3)** which implements the specific communication requirements of the sensor network **(4)**.

The support of a sensor network type in Janus is defined by the access and functionality modules of the gateway. Changing these does not change the requirements on the agent or client side. If the researchers in our example scenario upgrade their sensor nodes to include a microphone to monitor sound levels, it would be a trivial task to add the corresponding functions in the gateway.

3.2.3 Extensibility

The importance of the extensibility is simple to explain. Being able to extend and upgrade the access services we use to reach the sensor network can greatly improve on the longevity of a deployed sensor network. If an error or a need arises that was not conceived upon original design and deployment of a sensor network we want to be able to make a correction or an addition to the access services. Janus separates the access services from the sensor network implementation making it easy to perform these upgrades and extensions. A view of the Janus architecture connected to multiple access services and multiple sensor networks is depicted in Figure 3.5. It is important to note that the Janus architecture does not rely on a single agent to function. Instead there would typically be a number of agents, some implemented as an integral part of the clients and some acting as proxies, active at the same time.

The reason we want to use several application-specific access techniques is to enable common access services such as HTTP, SQL or an SMS gateway to process the information gathered by the sensor network. In our example sensor network scenario described in Section 2.1 the beach, the field and the forest could in fact be three different sensor networks monitoring different aspects of the environment. In this case we may want to access the sensor network resources in different ways. We could be instantly notified about special events on the beach by using an SMS gateway, log humidity and temperature from the forest using an SQL database or get an updated

view of the entire island network using a web server. Janus provides this application-specific access by letting the services connect to the network using an agent as a proxy. By taking this approach we can offer application-specific services without sacrificing the flexibility Janus provides us with. It would also be possible to perform service composition by dynamically combining functions and even different sensor networks together. The discussion on future possibilities like this continues in Chapter 6, Future Work.

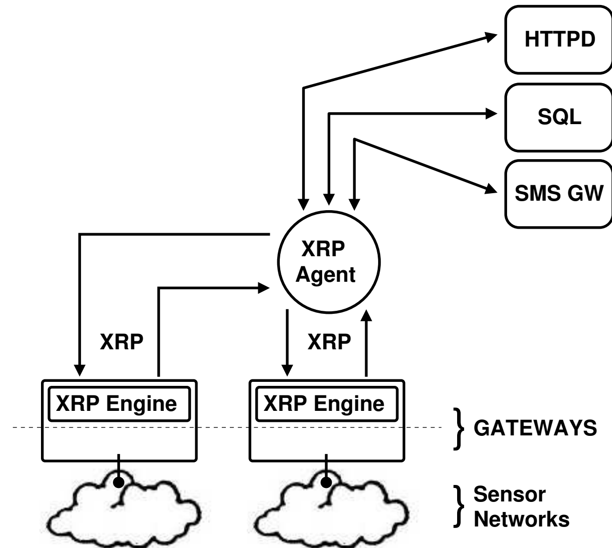


Figure 3.5: Extended architecture with access servers connected to multiple sensor networks via an XRP agent.

Chapter 4

Prototype Implementation

The Janus architecture consists of three parts: the agent, the engine and the gateway. However for the purpose of testing a prototype implementation that is not enough. We also need a sensor network to connect to and a client which can access the agent.

This section will begin with a short description of the sensor network implementation used together with the prototype implementation and the simple command line client which was used to connect to the agent. The focus of the chapter lies in Section 4.3 on the description of how the Janus architecture was implemented and an in-depth explanation of the signalling between the engine and agent.

4.1 Sensor network

The sensor network used in the prototype implementation was built on standard equipment sensor boards developed at Freie Universität in Berlin. Technical details regarding the sensor boards can be found at the Scatterweb ESB home page [2]. The sensors can detect motion, light, temperature, vibration and sound. The sensors nodes run Contiki [6], a small operating system for tiny devices which allowed us to implement the desired functionality in C. In relation to the example sensor network scenario presented in Section 2.1 we have implemented a network where nodes store sensed sound data, aggregate it and deliver it to the edge of the network in response to individual queries. The sensors also notice if they are moved and this special event is then automatically reported. Connectivity is supported by attaching a USB serial cable to the sink node. This work was done by Sergio Angel Marti as part of the Winternet project [8]. The functions implemented at the gateway are:

- **Active queries** : Requests can be sent to the nodes to retrieve the answer to an active query. In our network implementation the following queries can be made:
  - **Query current** : When the sink node asks a specific node about the current sound level in one of its sensors, the node checks the current sensor state and sends back the result.
  - **Query log** : Nodes consult their sound sensor periodically, compute a mean of the sound data every 5 seconds and store in their memory to form a set of measurements over the last minute. When a node receives a *query_log* message from the sink, it will reply with this set.

- **Event notification** : Critical and/or interesting events in the network are reported to the sink for notification outside the network. In our implementation we have the following events:
  - **Node detection** : When a new node joins the network, it reports this event to the sink. Node access control is carried out by the sink node, which makes sure that only allowed nodes can be part of the sensor network.
  - **Node selection** : When a button is pressed on the node it enters the selected state. This is to simulate one of the sensors reaching a threshold value on one of its sensors (eg. loud sound, movement detected, ...).
  - **Node position** : All nodes in our implementation are dynamically assigned an identifier, which is not bound to the node hardware, but to its geographic position. When a node changes position and adopts a new different identifier, it reports this event to the sink.

4.2 Client

The client used to connect to the agent in the prototype implementation was designed with a simple text-based interface. Even though it is very simple it features on-demand negotiation of sensor network functionality though the agent and can query both single nodes and areas in the sensor network. A view of the text interface can be seen in Figure 4.1.

```
$ ./democlient
DEMOCLIENT waiting for commands [1..n] or 'quit'
1 == request API
2 == subscribe to events
3 == is sink ok?
4 == query current value
5 == query logged values
6 == query which node has max in area
```

Figure 4.1: The text-based client used to connect to the prototype implementation.

4.3 The Janus Architecture

We have implemented a subset of the Janus architecture presented in Chapter 3 in C++ under Linux and connected it to a sensor network using serial USB attached to a sink node. We have run tests with the agent and the gateway on separate machines on a LAN and with a sensor network of up to 16 sensor nodes.

4.3.1 XRP Agent

The role of the agent is to exchange XRP messages with an engine to query the resources of the sensor network and to receive events generated inside the sensor network. The agent acts as the negotiator or translator between the connecting client and the Janus architecture. In our implementation a client connects to the agent using Berkeley TCP sockets. To be able to offer the client the ability to actively query

the sensor network the agent must first request and parse the API of the gateway in a way explained in Section 4.3.2. The agent can also register an interest to receive event notification from the sensor network by sending an XRP SUBSCRIBE message to the gateway. An agent can either subscribe to events of a given verbosity (e.g., silent, critical, verbose, very verbose) or to specific events (e.g., motion in a given area). The engine will then add the agent to its list of subscribers and inform it when an event takes place.

In our implementation the location (i.e., IP address) of the gateway is statically defined, but in the Janus architecture there would ideally be some kind of service discovery to locate gateways in your vicinity. This could be done using broadcasted XRP resolution requests in a way similar to the one described in [11].

4.3.2 XRP Engine

The engine is in charge of the agent interaction on the gateway. Figure 4.2 captures this interaction and shows the XRP messages sent back and forth between the agent and the engine. The agent can discover the available sensor network resources on this gateway by sending an XRP message containing a QUERY for the API to the gateway. When our implemented engine receives this request it will respond with a pre-built XRP message containing a DATA REPLY with the API. After this initial handshaking the agent will be able to invoke functions at the engine by sending QUERY style messages for the selectors listed in the API. When this happens the QUERY is demultiplexed in the engine and a local function invoked with the data payload as a function argument. In our implementation the function argument is either the address of a node (position 1-16) or the alias of an area (e.g., beach, field, forest and selected) inside the network. The local function communicates with the sensor network and returns a result. An XRP DATA REPLY message is then built and sent back to the agent. Appendix C contains examples of a few of these messages.

4.3.3 Gateway

The gateway sits between the sensor network and the access network and contains an XRP engine. It handles incoming connections from XRP agents and listens to incoming events from the sensor network. In order to communicate with the sensor network an access module using the USB serial port for sending and retrieving messages had to be coded. The semantics to communicate with the sensor network were very simple which turned out to be a limiting factor in composing more advanced functions. Despite this limitation we successfully managed to implement the following functions in the function module at the gateway:

- **Query node** : We allow direct access to the nodes by using a node identifier. In our implementation the node identifier maps to a geographic position.
  - **Query current** : Retrieves the current sensor value (i.e., sound level).
  - **Query log** : Retrieves a log of the sensor values recorded over time.
- **Query area** : We can define an area inside the sensor network, by grouping several nodes at the gateway. An area can be queried by as a single entity. In response to an area query, the gateway issues individual queries to the desired nodes.

  - **Query max** : Returns the node identifier of the node with the highest sensor value in a given area.
  - **Query mean** : Returns the average sensor value of the nodes in a given area.

- **Event notification** : Certain events in the network are reported by the sink to the gateway. These events can be reported to an agent using event notification. We support the event of a new node being added to the network, of a node being selected by pressing a button, and of the geographic position of a node being changed, i.e., it is moved.

Areas within the sensor network were statically defined in the gateway as arrays of the nodes geographic positions. This relates to the beach, field and forest of the sensor network scenario in Section 2.1. However we decided to use the special event of a node being selected to define dynamic areas within the network. This makes it possible to direct questions to the *selected* nodes within the network. A real-world example would be if all the nodes who have detected motion within the last 5 seconds enter the selected state. The client can then poll these nodes as a group to find out more information about what is going on in active parts of the network.

To be able to demonstrate what was going on inside the sensor network we added support for a graphical presentation interface in the gateway. The GUI was developed in Java and acts as a graphical sniffer that visualizes queries coming in for nodes and areas in the sensor network, individual nodes responding to these queries and the results reported back to the client. A screenshot of the demonstration GUI can be found in Appendix D.

4.3.4 Signalling between Agent and Engine

We will now give an example of the signalling that takes place between the agent and the engine. Before we go into the details of our signalling example we will look at the features of the XRP protocol, which makes this possible. See Section 2.3 for a more detailed description of the XRP packet format and [19] for its possible uses.

XRP permits us to specify a extensible number of parameter fields for each packet. This makes XRP expressive enough to meet our needs. The XRP packet begins with an XRP command. In a sense the XRP protocol is so expressive that this could be compared to a normal natural language command. In our prototype implementation we have limited the support for commands to either a SUBSCRIBE request, a QUERY request to invoke a function or the delivery of an API, or a DATA REPLY. The XRP command parameters are of variable length can appear anywhere inside the parameter block. This gives us a very high degree of freedom when designing commands and their mandatory parameters. A full listing of the XRP commands and parameters used in our implementation can be found in Appendix A and B.

The XRP commands are essential to how Janus operates and we will therefore describe how we implemented the signalling between the agent and engine in detail. A good example which shows the versatility of the XRP protocol is how we build the DATA REPLY containing the API of the gateway and how a function is subsequently invoked using said API.

All XRP messages in our prototype implementation are transmitted between the agent and the engine using the UDP socket interface. All the functions to interact with the sensor network are implemented in the access and functionality modules at

the gateway. We provide both low-level functions and high-level functions. Low-level functions can be used to query individual nodes and high-level functions allow for more advanced queries, like the average sensor value inside a given area of the network.

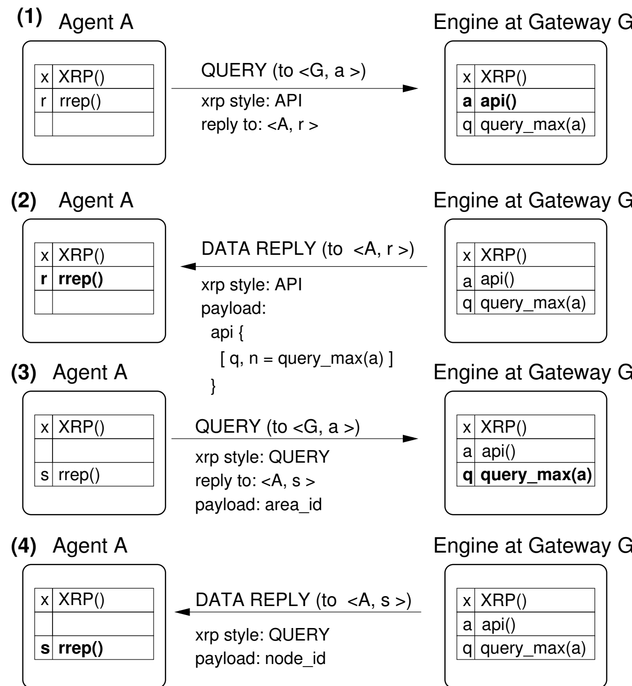


Figure 4.2: Signalling: The left column in the table represents function addresses and the right table column represents function names. (1,2) Exporting the API of the sensor network (3,4) Issuing an RPC by addressing a selector bound to a fun ction together with the packet payload as an argument.

The gateway listens to events that originate inside the sensor network and waits on incoming connections to reach its XRP engine. Once an agent connects, it negotiates with the gateway to retrieve an API of functions to reach the sensor network. After the API has been exported to the agent, it can issue requests by invoking functions at the gateway. The functions are accessed through a hash table containing SAPF selectors associated with different functionality. The signalling is achieved using a combination of XRP request and DATA REPLY messages as shown in Figure 4.2.

In addition to this Figure 4.2 also shows a simplified view of the function table within each node in order to describe the inner workings of the signalling. The XRP packets are sent between the agent and the engine at the gateway. A selector to handle the reply is always installed prior to sending a request and we will refer to this selector below as the *reply selector*. Each request message contains the reply selector along with the address of the sending node. This information is used by the gateway to target its reply. We will take a closer look at two cases of signalling: first the way the agent behaves to negotiate and retrieve the API of the gateway and second the invocation of a function at the gateway in order to query the status of the sensor network.

When an agent wishes to retrieve the API from a gateway, it will installs a reply

selector *r* and issues **(1)** a request for the API to the gateway. When the engine receives this request it will:

1. Check if the request is permitted
2. If it is permitted, respond with a DATA REPLY containing the API

The engine running at the gateway processes the request and responds **(2)** with a DATA REPLY to the *r* selector on the agent. This would be an ideal place to implement some type of access control for security reasons. If the agent presents itself in an incorrect manner, the gateway will simply refuse to release its API or release a severely restricted version. The reply to the API request contains the API represented as XRP parameters inside the packet. Full examples of such a reply message and other types of messages can be found in the Appendix section. In Figure 4.2 we show the *query_max()* function along with the selector which identifies it. The *query_max()* function takes an area as the argument and returns the node with the maximum sensor value.

After the API has been exported, the agent can invoke functions at the gateway by addressing selectors on the gateway with a QUERY style request message. Again the agent installs a reply selector *s* to issue **(3)** a QUERY with the function argument as the payload, in this case the requested area inside the network. When the engine receives this request it will:

1. Check if this selector is present in its function has table
2. If it is present, invoke the corresponding local function
3. Wait for the function to terminate with a result
4. Respond with a DATA REPLY containing the result

If the selector is not present in the function has table, the QUERY will simply be ignored. In our example case the corresponding local function is the *query_max()* function. This function will issue individual requests for the sensors within the given area in order to identify the node with the highest sensor value. When the function has terminated with this information, it proceeds by building a DATA REPLY containing the returned value as an XRP parameter and sends **(4)** this to the reply selector at the agent. In this way any type of functionality can be addressed through QUERY messages targeted for other selectors from the API which would be bound to other local functions.

Chapter 5

Results

In this chapter we will present a summary of the results and experiences of this thesis work. The first section takes a look at the design of the Janus architecture and presents a summary of its features. In the final section we will turn to the experiences from the prototype implementation.

5.1 Design

We have succeeded in designing an access structure to give flexible access to sensor networks. During the evaluation of previous solutions we identified three important properties for the design to be successful. The design had to show flexibility, modularity and extensibility. The Janus architecture has been designed with these features in mind. Thanks to the inherent properties of the XRP protocol Janus offers maximum flexibility through on-demand dynamic negotiation of services. Through this flexible signalling it can support both existing event-driven and on-demand access techniques. Modularity is achieved by using the gateway as a single ingress / egress point to access the sensor network. Changes to the way the gateway accesses the sensor network and its functionality can be done without interfering with any other part of the design. It is easy to compose new modules in order to use the Janus architecture together with a new type of sensor network. Extensibility is achieved in Janus by using the middleware approach of inserting a flexible architecture between the two disparate networks. This makes it possible for both networks to fundamentally change as long as they adhere to the Janus architecture. Alternative access applications or servers may be deployed on-demand to meet new system requirements. It is important to remember that different types of sensor networks are useful for different scenarios. It is not our intention to impose any limits on how the sensor networks are implemented. Instead we suggest using the Janus architecture to allow all differnt sensor network types to be accessed in a clear and consistent manner. The design relies on the XRP protocol to provide a consistent interface to reach the functionality in the sensor network.

The usefulness of these properties in the design can be explored through the discussion of *invariants* [1] in network architectures as suggested in [8]. Invariants are pieces of an architecture that cannot be changed without stopping an architecture from functioning. Ahlgren et al. [1] claim that all existing network architectures contain invariants and we agree with their conclusions. Examples of such invariants are the IPv4 addresses in the Internet and the SIM card in mobile phones. During the design

of Janus we realized that the access application and the sensor network are potentially quite disparate and variable. Because of this we do not think that either is particularly suited to be the invariant of an access architecture for sensor networks. Instead we insert the Janus architecture between the access applications and the sensor network and propose that its flexible XRP signalling scheme should be the explicit invariant of our system [8]. This means that Janus remains useful for as long as the expressiveness of XRP is enough to capture the requirements of the systems it is supposed to support.

5.2 Prototype

The proof-of-concept prototype of Janus was implemented in C++ under Linux to successfully show the feasibility of the design of the Janus access architecture. The prototype implementation was tested with a simple text-based client connected to a sensor network made up of standard equipment sensor nodes. By implementing this in the prototype we showed that it was possible to combine basic functionality on the actual sensor nodes to implement reactive data operations for chosen regions of the network. The prototype was tested running on a LAN connected to a sensor network made up of standard equipment sensor nodes. A simple command line client was used to connect to the prototype and use it to interact with the sensor network.

The running prototype shows that it is possible to mold the passive and active approaches used in existing access solutions into a single system. It also shows that dynamic negotiation of sensor network functionality can be achieved by exporting an API over the network. Finally it shows that XRP has the required properties to provide flexible access to sensor networks through the use of an RPC-style interface.

The only implicit requirements put on the sensor network is that it supports a protocol in which messages can be tracked using an enumerator in the header. This is crucial to the gateway as it lets it keep track of which replies belong to which requests.

5.3 Contribution

The major contribution of this thesis work is the design and implementation of the Janus architecture. The design has its roots in current access techniques and can be shown to be compatible with all of them. The reasoning behind the design was a collaborative effort in which all the members of the Winternet projet team took place. The major work of the author of this thesis was to implement a functioning prototype adhering to the features of the design. This prototype implementation has shown not only that it is possible to implement the design but that the design behaves as expected. We were able to demonstrate the prototype at the SITI/SSF day during the spring of 2005.

Our results provide a solid foundation to build on which could eventually lead to the architecture being implemented in full and deployed. New projects have already been started in order to further develop the ideas surrounding the Janus architecture. A road map for future development is presented in Chapter 6.

Chapter 6

Future work

As the work on the Janus architecture continues there are several directions in which to turn. This chapter will describe the next natural steps we perceive should be taken to improve the behavior of the prototype and introduces some more advanced ideas for future work.

The existing prototype does not allow for several clients accessing the agent nor several agents accessing the gateway. Attention should be spent on this since it offers the possibility of testing the prototype and thus the feasibility of the design in bigger network scenarios. Allowing for more advanced client interaction, such as being able to manually define an area in the network should definitely be investigated. Also more advanced composed functionality should be considered and implemented. An example of such a future function could be one that answers the question: *Which are the regions that have a temperature above 30 degrees Celsius?*

More energy could definitely be spent on improving the semantics of the XRP messages and parameters used in the prototype implementation. Composing agent requests through natural language expressions could turn out to be very interesting.

Adding support for legacy servers to attach themselves to the agent would show that it is possible to successfully use existing servers to interact with the sensor network through the Janus architecture. A web server appears to be most likely as the first step as it would be very easy to show that it is possible to have a totally dynamic client setup complete with a graphical interface. Other interesting services could be a simple database interface and some means of active notification, perhaps through the use of an SMTP server. Similarly adding support for other sensor network types, by writing access and function models for a new gateway is a very logical step for future work. Having a running prototype system where several access services connect to several sensor networks would truly show the possibilities the Janus architecture offers. One of these possibilities would be to combine several sensor networks at the agent and to have these accessed transparently from a client by using *virtual* sensor network addressing.

Another promising direction for future work is to completely integrate Janus with the Contiki Operating System to provide an active networking platform for sensor networks [8]. We could use the dynamic program loading functionality of Contiki to load programs shipped in XRP messages. Code could be dynamically deployed into the sensor network and subsequently queried with XRP activation messages. In other words XRP requests can be sent into the sensor network which will then invoke code that has been dynamically deployed at run-time.

Chapter 7

Conclusions

During this thesis work an access architecture named Janus have been designed and implemented. We saw the need of a general access architecture, which should exhibit three important features:

- **Flexibility**: The signalling has to be flexible in order to encompass both event-driven and on-demand access approaches and support negotiation for offered sensor network capabilities.
- **Modularity**: The architecture needs interchangeable modules to allow several sensor network types and access technologies to be supported.
- **Extensibility**: It should be possible to allow new and upgraded access services after deployment of a sensor network.

Janus is designed to offer maximum flexiblity in the signalling by using the XRP protocol to exchange messages between the agent and the engine. Due to the nature of XRP as an interpreted messaging protocol it is very expressive. In contrast to using a static packet format, XRP performs signalling to set up functions on the engine to perform the tasks the agent has requested. The signalling in Janus allows for on-demand negotiation of services and can be shown to support existing access techniques.

Due to the wide range of sensor network implementations it is important to have modularity in a general access architecture. Janus achieves this modularity by using the gateway as a single ingress / egress point to access the sensor network. Changing the way the gateway accesses the sensor network and its functionality does not interfer with any other part of the design. It is also easy to compose new modules in order to use Janus with a new type of sensor network.

Being able to extend and upgrade the access services we use to reach the sensor network can greatly improve on the longevity of a deployed sensor network. Janus separates the access services from the sensor network implementation by taking the middleware approach. By inserting a flexible architecture between the two disparate networks, it is possible for both networks to fundamentally change as long as they follow the Janus architecture. This makes it easy to perform important upgrades and extensions even to already deployed sensor networks.

The proof-of-concept prototype of Janus was successfully implemented in C++ under Linux. Basic functions in the sensor network were combined to implement reactive data operations for regions of the network. The prototype implementation has been

tested on a LAN connected to a sensor network made up of standard equipment sensor nodes and was demonstrated during the SITI/SSF day of the spring 2005.

The results of the initial design and prototype implementation are very promising and work to improve upon and extend the prototype have already started.

Acknowledgments

The Winternet project group working on Janus were (excluding the author of this thesis): Adam Dunkels, Richard Gold, Arnold Pears, Sergio Angel Marti and Henrik Olsson. They deserve recognition for the collaborative effort on the design and implementation of Janus and the testing and presentation of the prototype. This thesis was partly financed by the Winternet project.

Appendix A

List of XRP commands

A.1 SUBSCRIBE

Used to subscribe to passive event notification from the sensor network. When the XRP engine processes a SUBSCRIBE request it will add the REP_SEL and REP_ADDR to its subscription list. The EVENT_LVL defines the verbosity in the default scenario, but can also refer to a specific event.

XRP command : SUBSCRIBE (0x0004)

Parameters : REP_SEL, REP_ADDR, EVENT_LVL

A.2 QUERY

Used to send a query for a selector. A valid function selector will be invoked with the parameter FUNC_ARG as the argument.

XRP command : QUERY (0x0005)

Parameters : REP_SEL, REP_ADDR, FUNC_ARG

A.3 API

Used by the agent to send a query for the API.

XRP command : API (0x0006)

Parameters : REP_SEL, REP_ADDR

A.4 DATA REPLY

Used to send data back to the agent. The only special case is when we send an API which uses a special DATA parameter described in Appendix B.

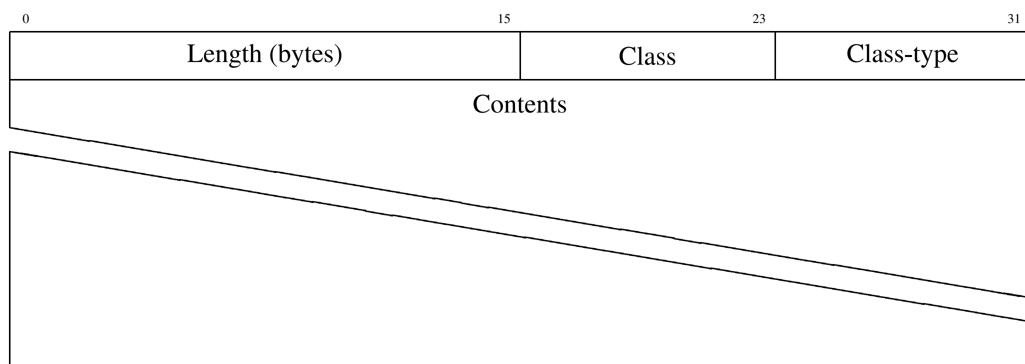
XRP command : DATA REPLY (0x0007)

Parameters : DATA or API_HDR

Appendix B

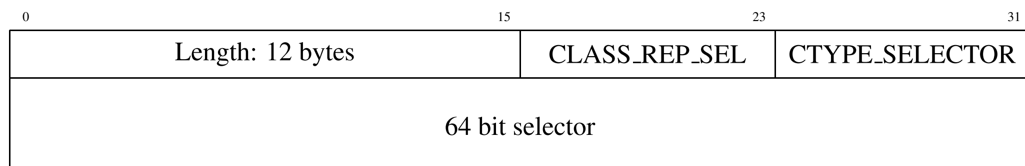
List of XRP parameters

B.1 XRP parameter structure



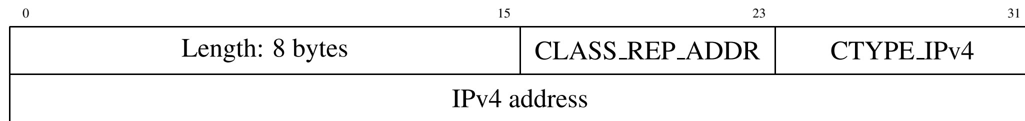
B.2 REP_SEL

Used to specify the reply selector.



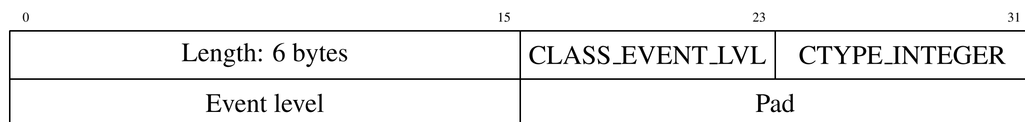
B.3 REP_ADDR

Used to specify the reply (IPv4) address.



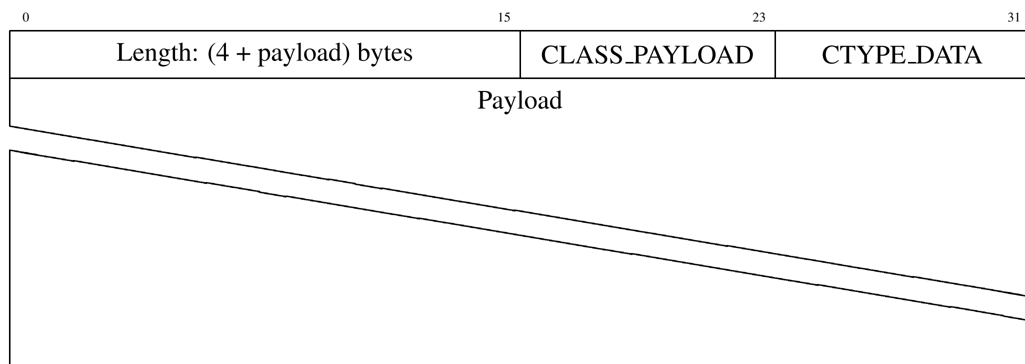
B.4 EVENT_LVL

Used by the SUBSCRIBE command to specify the desired event level or type.



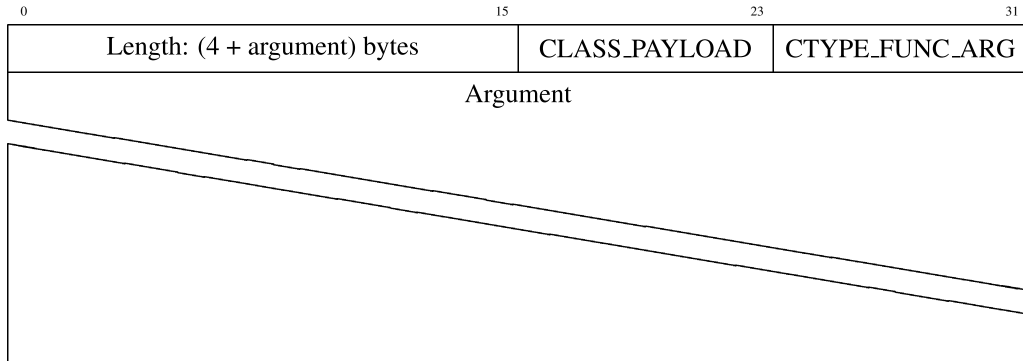
B.5 DATA

Used to send pure data. Padded to the 32-bit boundary.



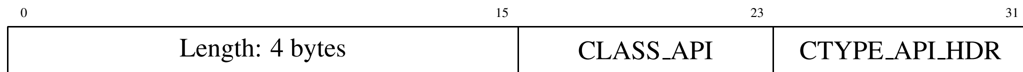
B.6 FUNC_ARG

Used to send the arguments when invoking function selectors. Padded to the 32-bit boundary.



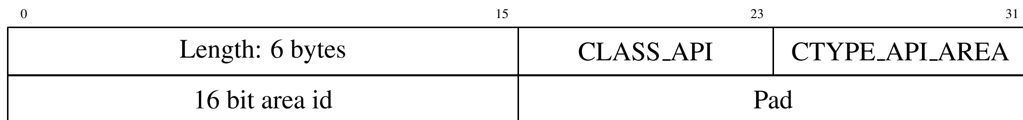
B.7 API_HDR

Used as the header of an APIi reply. The API header is followed by one or several API_AREA and API_FUNC parameters.



B.8 API_AREA

Used to notify the agent of a pre-defined area which can be accessed using the provided area id.



B.9 API_FUNC

Used to notify the agent of an existing function along with its return and argument type (e.g., integer, char, ...).

0 ... 15	23	31
Length: 16 bytes	CLASS_API	CTYPE_API_FUNC
64 bit function descriptor		
16 bit function return type	16 bit function argument type	

Appendix C

Example XRP messages

None of these messages contain the information about the target selector since that 64 bit identifier is located in the encapsulating SAPF packet.

C.1 QUERY for API

First we need to send a QUERY to retrieve the API of the gateway.

0 … 15	23	31
API (0x0006)	TTL: unused	Reserved: null
Length: 12 bytes	CLASS_REP_SEL	CTYPE_SELECTOR
Reply sel: 0x0000000000000020		
Length: 8 bytes	CLASS_REP_ADDR	CTYPE_IPv4
IP: 192.168.0.42		
0x0000		

C.2 DATA REPLY with API

A DATA REPLY with the API is sent back to the agent. This example contains the identifiers of two predefined areas and two functions: *query_current* which takes a node identifier as argument and returns an integer representing the current sensor value, and *query_max* which takes an area identifier as argument and returns the node identifier of the sensor node that holds the highest value.

0 15	23	31
DATA REPLY (0x0007)	TTL: unused	Reserved: null
Length: 4 bytes	CLASS_API	CTYPE_API_HDR
Length: 6 bytes	CLASS_API	CTYPE_API_AREA
Area id: 1	Pad	
Length: 6 bytes	CLASS_API	CTYPE_API_AREA
Area id: 2	Pad	
Length: 16 bytes	CLASS_API	CTYPE_API_FUNC
Descriptor: query_current		
Returns: integer	Argument: node identifier	
Length: 16 bytes	CLASS_API	CTYPE_API_FUNC
Descriptor: query_max		
Returns: node identifier	Argument: area identifier	
0x0000		

C.3 QUERY for function

This is a simple QUERY to find out which node holds the maximum sensor value in a given area. The area identifier is provided as argument to the function.

0 15	23	31
QUERY (0x0005)	TTL: unused	Reserved: null
Length: 12 bytes	CLASS_REP_SEL	CTYPE_SELECTOR
Reply to selector: 0x0000000000000021		
Length: 8 bytes	CLASS_REP_ADDR	CTYPE_IPv4
Reply to IP: 192.168.0.42		
Length: (4 + 2) = 6 bytes	CLASS_PAYLOAD	CTYPE_FUNC_ARG
Argument: 1 (area identifier)		
0x0000		

C.4 DATA REPLY with result

The resulting node identifier is reported back. In this case node 6 held the highest sensor value in the given area.

0 15	23	31
DATA REPLY (0x0007)	TTL: unused	Reserved: null
Length: (4 + 2) = 6 bytes	CLASS_PAYLOAD	CTYPE_DATA
Payload: 6 (node identifier)		
0x0000		

Appendix D

Demonstration GUI

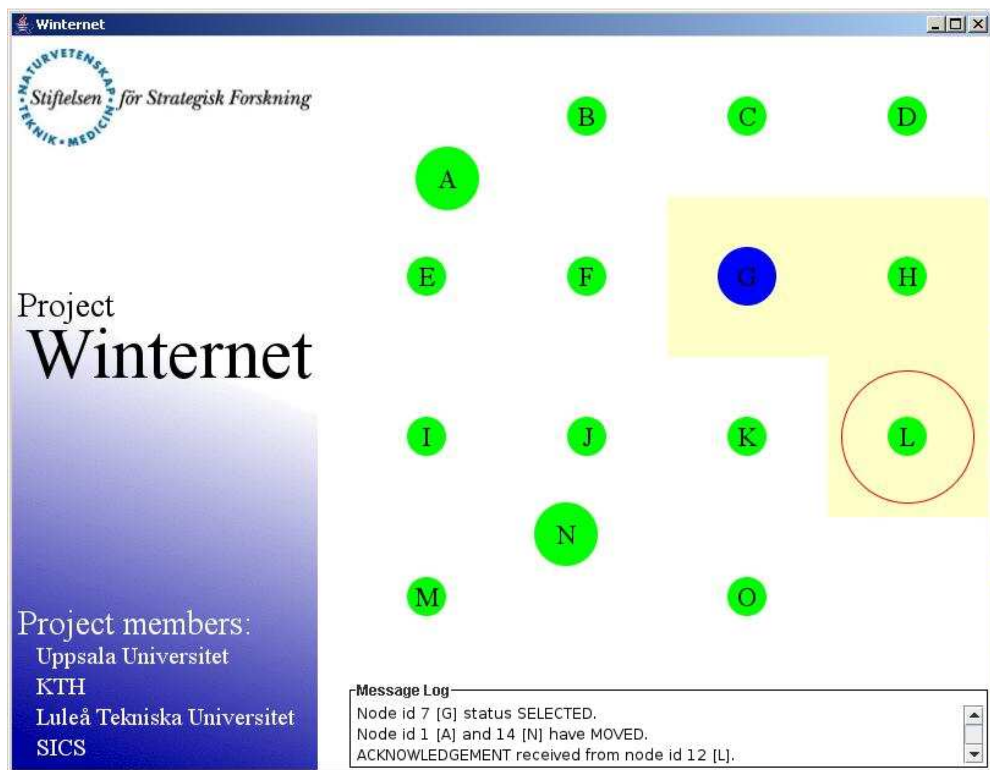


Figure D.1: Graphical interface used in the Winternet demonstration session.

Bibliography

[1] Bengt Ahlgren, Marcus Brunner, Lars Eggert, Robert Hancock, and Stefan Schmid. Invariants: A new design methodology for network architectures. In *SIGCOMM FDNA*, 2004.

[2] CST Group at FU Berlin. Scatterweb Embedded Sensor Board. Web page. 2003-10-21.

[3] R. Beckwith, D. Teibel, and P. Bowen. Report from the field: Results from an agricultural wireless sensor network. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.

[4] P. Buonadonna, D. Gay, J. Hellerstein W. Hong, and S. Madden. TASK: Sensor Network in a Box. In *Proceedings of the Second European Workshop on Sensor Networks*, 2005.

[5] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: an argument for network pluralism. In *FDNA '03: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 258–266. ACM Press, 2003.

[6] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.

[7] 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, February 2004.

[8] Adam Dunkels, Richard Gold, Sergio Angel Marti, Arnold Pears, and Mats Uddenfeldt. Janus: An Architecture for Flexible Access to Sensor Networks . In *First International ACM Workshop on Dynamic Interconnection of Networks*, 2005.

[9] Adam Dunkels, Thiemo Voigt, Niclas Bergman, and Mats Jönsson. The Design and Implementation of an IP-based Sensor Network for Intrusion Monitoring. In *Swedish National Computer Networking Workshop*, Karlstad, Sweden, November 2004.

[10] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the SIGCOMM'2003 conference*, 2003.

[11] Richard Gold, Per Gunningberg, and Christian Tschudin. A virtualized link layer with support for indirection. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, 2004. http://user.it.uu.se/∼rmg/pub/fdna05-gold.pdf.

[12] Network Working Group. Request for Comments (RFC): 1831 Sun Microsystems Standards Track August 1995 RPC: Remote Procedure Call Protocol Specification Version 2.

[13] D. Harrington, R. Presuhn, and B. Wijnen. STD 62: Simple network management protocol version 3 (SNMPv3), December 2002.

[14] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[15] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, 2003. ACM Press.

[16] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *First ACM Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, GA, USA, September 2002.

[17] P. Padhy, K. Martinez, A. Riddoch, H.L.R. Ong, and J.K. Hart. Glacial environment monitoring using sensor networks. In *Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[18] Java Distributed Systems Home Page. *Java RMI specifications*. http://www.sun.com/rmi?

[19] C. Tschudin, R. Gold, O. Rensfelt, and O. Wibling. LUNAR: a Lightweight Underlay Network Ad-hoc Routing Protocol and Implementation. In *Proceedings of The Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN'04)*, 2004.

[20] Christian Tschudin and Richard Gold. Network Pointers. In *ACM Workshop on Hot Topics in Networking (HotNets-I)*, 2002.

[21] W3C. Web Services Description Language (WSDL) 1.1, 2003. http://www.w3.org/TR/wsdl.

[22] Zhonghua Yang and Keith Duddy. CORBA: A platform for distributed object computing (a state-of-the-art report on OMG/CORBA). *Operating Systems Review*, 30(2):4–31, 1996.