A Flexible and Scalable Message Broker for Sensor Network Integration

Luis Garcés-Erice, Daniel Bauer, Paolo Scotton
IBM Zurich Research Laboratory
Säumerstrasse 4, 8803 Rüschlikon, Switzerland.
E-mail: {lga,dnb,psc}@zurich.ibm.com

Abstract—Sensor network technologies have experienced an enormous development with the proliferation of sensor devices, custom communication protocols, and methods to exploit the data gathered on such networks. The actual impact of this technology on the enterprise, however, has been quite limited. This is because the community has focused almost exclusively on the sensor devices themselves, while paying little attention to the means to integrate this novel technology into the current enterprise infrastructure. We argue that facilitating this integration would encourage the adoption of sensor networks. Thus we present a scalable message broker for message-oriented middleware that is able to interact with any sensor network by providing a flexible and extensible communication subsystem. This message broker then provides the link to the messaging middleware used in the enterprise network. We describe the architecture of such a broker’s communication subsystem and its implementation in Java™. We present the performance of the broker, demonstrating that it scales to interconnect many sensor networks and is able to cope with potentially very large amounts of data from the sensors’ readings. Finally, we present an overview of the sensor protocols we have developed, and how they are integrated into the broker, enabling for end-to-end interaction with external enterprise applications unaware of the sensor technology.

Index Terms—Sensor networks, Middleware, Messaging, Network protocols, Message broker

I. INTRODUCTION

In the past years, research on wireless sensor networks has been very active, resulting in the development of a plethora of protocols for this type of platform [1]. The devices forming these networks may gather information about the underlying system (sensors) or modify said system following commands (actuators). In this paper, we will follow convention and call both types of devices, by abuse of language, sensors. The MicaZ from Crossbow® [2] is a typical example of low-cost low-power wireless sensor network device featuring a 8 MHz processor with 128 KBytes of programmable flash memory. Hellerstein et al. [3] provide an overview of the range of sensor technologies that are available.

The literature has mostly overlooked the fact that, in most cases, the usefulness of these networks derives from their ability to extract information from the environment, but only to be processed by some external logic to achieve some goal (the same is true for actuator networks, where they allow an external logic to modify the environment). The external logic can be, for example, a business process running in an enterprise network infrastructure as part of some complex application. These enterprise applications are increasingly built around an SOA (Service Oriented Architecture), having different functionality encapsulated in services at the points where the functionality is implemented. At the core of this architecture is an ESB [4] (Enterprise Service Bus), which enables the operation of the applications by allowing the services to communicate. The ESB is, ultimately, implemented by some form of middleware that establishes a lingua franca that services use to exchange information without having to worry about the actual location or invocation requirements of services. This middleware may use a number of mechanisms, such as message queues, publish/subscribe messaging or Web Services (WS). Enabling the connection of sensor networks to this infrastructure is the topic of this paper.

For them to be widely used in real applications in new businesses, sensor networks need to be integrated into the enterprise network allowing 1) monitoring information gathered by the sensor network to be retrieved and processed, and 2) control messages to be sent from an external application to the sensor network. In this paper, the integration is performed by a gateway that connects the sensor and enterprise network. This gateway provides a suitable interface through publish/subscribe middleware for the sensor network functionality to join the ESB. Enterprise applications can thus leverage the capabilities of sensor networks.

Such a gateway must be able to interact with any device independently of the sensor network protocol: the risk exists with this fast-moving technology that any one protocol implemented by the gateway becomes rapidly obsolete, or that the selected vendor stops supporting it. The gateway must thus be flexible enough, so that interacting with future wireless sensor networks merely requires coding the new protocol logic as a component for the gateway. Moreover, following best engineering practices, the gateway must be able to reuse those parts of the protocol that have already been implemented in the past (e.g., the network protocol for one sensor application could be reused for a second one). On the other hand, applications already running on the enterprise network should not need to be modified nor even be aware of working with sensor data, but merely interact with the ESB as explained above. Sensors typically use customized protocols to reduce power consumption during communication. These protocols are often highly optimized. Because of the high customization of these protocols, they are usually cumbersome to program outside of the sensors. Moreover, the sensor networks may be deployed at remote locations where getting
the data delivered to the enterprise network requires using custom communication means. To summarize, the logic in the application should not depend on which sensor network the data came from. Messaging middleware has been a preferred tool for application integration in the enterprise. In a messaging system, senders and receivers do not communicate directly with each other but via a messaging abstraction. This level of indirectness allows senders and receivers to be unaware of each others’ existence and nature, and allows the transmission and reception of a message to be decoupled in time. The exchange of information is usually performed at some central point to which senders and receivers connect: the message broker. For an overview of publish/subscribe messaging systems, see [5].

There is a challenge in building a gateway that is powerful and flexible enough to integrate a wide variety of sensor network protocols. Given the requirements above, we propose in this paper to implement the gateway connecting the enterprise applications and the wireless sensor networks with a flexible message broker. A key component of this broker is the communication subsystem, because it needs to be extensible (to cope with new protocols and sensor types), modular (to be able to reuse existing software components), and scalable (to be able to gather data simultaneously from multiple sensor networks). Scalability is an important issue, as we expect to have each of these gateways connected to a significant number of deployed sensor networks. The reasons for this are the ease of management of having at most a few gateways as well as optimization of the utilization of the resources (one gateway per sensor network would be wasteful). Thus, even if the data stream generated by each network is not very important, the aggregated throughput may be considerable.

The main contribution of this paper is the description of the design and implementation of a message broker for publish/subscribe-based sensor network protocols. We describe how we extend the publish/subscribe (referred to as pub/sub hereafter) system into the sensor network using a configurable message broker as an application-level bridge: first we review in Section II some efforts with similar aims, and we describe the typical network protocols that are in use with a network of sensors; we emphasize the need to cope with the diversity of protocol stacks for different sensor applications. Then in Section III we detail our implementation of a messaging broker which allows multiple dynamically configurable protocol stacks to be used for communication with a core pub/sub engine, and we show how this broker acts as the gateway between the sensor and the enterprise networks. In Section IV, we provide experimental results of the broker’s scalability. Finally, we argue in Section V that pub/sub is a suitable abstraction for a variety of commercial sensor networks. We provide an overview of our implementation of protocols that support the pub/sub abstraction over a sensor network and have been integrated into the broker.

II. RELATED WORK

The authors of [6] clearly identify, from an IT infrastructure perspective, the need for a middleware platform that integrates sensor networks with back-end applications.

The authors propose a taxonomy separating concerns of the sensor network, gateway and back-end infrastructure, while identifying functional requirements that need to be considered through the infrastructure, like distributed programming layer and management of the sensor network. We consider our approach an example of such an architecture, choosing pub/sub middleware as our distributed programming abstraction and the message broker as the gateway integrating sensors in the realm of enterprise applications instead of a generic IT back-end. A proposal from some of these authors for an end-to-end middleware platform is available in [7]. The need for middleware to adapt to various types of heterogeneous devices is also explored in the divergent Grid work [8] using deep middleware. This middleware interacts with the network level to configure itself to the environment where it runs. The authors study different middleware mechanisms, including pub/sub.

A. Flexible Multi-Protocol Gateways

There has been extensive research in the networking community on how to build protocol stacks from individual, reusable components. The x-kernel [9] is a communication architecture in which protocol stacks can be assembled from protocol modules. Many research project have used the x-kernel for their communication needs because of its adaptable and efficient design. Although the x-kernel modular protocol stack has been taken as starting point of our work, our approach is different for two reasons. First, in our approach the granularity of the protocol implementation is rather "coarse". In the broker, only application-specific protocols are implemented, while in general we rely on the operating system or specific libraries for the instantiation of the network-layer communication protocols (e.g. TCP/IP, UDP, TinyOS, ZigBee). These application-specific protocols are implemented as a single module in the modular stack. Therefore we do not need the very fine-grained approach proposed by x-kernel, where specific functions inside a given protocol layer can be instantiated as a modular element. Secondly, our approach exploits the parallelism across modules and stacks to improve performances in multi-core architectures.

In the sensor network space, to the best of our knowledge, the only effort similar to ours is the XServe gateway offered by CrossBow [2], which allows client applications to access sensor data using an XML API. This offering is severely limited compared with what we aim to achieve. XServe can only interact with motes running the XMesh(TM)protocol, and is in fact part of an entire end-to-end package (from sensors to applications). Thus it cannot be extended to include other sensor types. Moreover, the XML API offered to the applications is not likely to be compatible with already deployed applications.

Although not directly related to sensor networks, an interesting example of integration of heterogeneous environments and devices is the context bridges proposed in [10]. Context-aware applications cannot get information from different context management systems, thus the need for gateways that bridge these different domains offering a common shared abstraction.
These allow applications to adapt to any computing environment (office, ad-hoc network, home, etc.) and still be able to communicate with applications in other environments. The role of the bridges is similar to that of the message broker, an integration solution where a uniform common software platform is not feasible.

B. Pub/Sub Sensors

Mires [11] is a messaging pub/sub middleware for sensor networks. This approach differs significantly from ours in that Mires is conceived as a way for the sensors to communicate with each other, not as a means to integrate sensor networks into a larger infrastructure. In any case, the authors provide little information on the implementation details and instead give high-level UML diagrams of how communication would take place.

Directed diffusion [12] is a form of pub/sub in which a spanning tree is formed from publishers to subscribers. The tree is obtained by flooding the subscription requests through the network and assigning weights to each network edge as a measure of how appropriate it is as a return path from publishers to subscribers. Those weights are reinforced if the edge can handle traffic from multiple publishers. Here again, pub/sub is not used as a tool for sensor network integration.

The usefulness of the pub/sub mechanism in sensor networks is recognized for example in [13]. The authors propose a probabilistic dissemination protocol where an application in a sensor broadcasts its subscription interests, and these get propagated to a predefined number of hops called horizon. Each event is propagated through a predefined fraction of the available neighbors. Having no links in a broadcast medium, this translates into a sensor rebroadcasting an event every time if it also shares the subscription interest, or with probability equal to the fraction of links to be used if it does not. Subscriptions are maintained through a soft-state protocol through leases, where the interest of application in the sensor is refreshed or explicitly revoked. Integrating such an approach by means of a broker should not be difficult: it is very similar to our proposal described in [14], which has been integrated in the broker as detailed in Section V-A.

C. Sensor Technologies to Support

A gateway for wireless sensor networks has to support many potentially diverse sensor communication technologies.

TinyOS is an Operating system for sensors [15]. It is not the only such operating environment, but is widely used both commercially and in academia. TinyOS offers a set of libraries that allow applications to read from a range of sensor types and send/receive data over a radio interface.

The ZigBee consortium [16] has designed a standard MAC and network layer to run over the IEEE 802.15.4 radio interface, specially designed for sensors. ZigBee offers quite a rich set of functionality, but this comes at the cost of increased complexity and the impossibility of customization to particular needs. Our feeling is that ZigBee will play a role when sensors from different vendors have to interoperate, for example among consumer appliances, but when there is a sensor network dedicated to one specific application, then the advantages of standardization are less appealing.

Bluetooth has been proposed as a radio layer for sensor networks [17], but its large memory footprint, short range, power usage, and long connection establishment times mean that it is likely to play only a marginal role in this space.

In this section, we have given an overview of some of the network layer technologies available, and we have no reason to believe that there will be a consolidation with time: power will always be a scarce resource in such networks, and tailored solutions will always do better than one-fits-all ones. Except in very specialized areas, we see only few advantages to sensor network interoperability. In fact, as explained before, commercial deployments will very likely be based on a single network technology. In the case where communication between different sensor networks is needed, this can be achieved by higher-level abstractions, such as the one presented in this paper. In summary, we contend that there is a clear need for an entity residing in the enterprise network and acting as an integration point of the various sensor network technologies.

III. DESCRIPTION OF A MESSAGE MIDDLEWARE BROKER FOR SENSOR NETWORKS

A. Overview

A broker can be envisaged as a set of First-In First-Out (FIFO) queues into which messages are placed and from which messages are forwarded. Each message and each queue are associated with a topic. Publishers on a topic place their messages in the corresponding queue, and the broker send these messages to all subscribers registered on that topic. The core broker functions are implemented by an entity called the pub/sub engine. The pub/sub engine is largely independent of the means by which messages are sent from publishers and forwarded to subscribers. It is only necessary that publishers, subscribers, and the broker can all coherently determine which message belongs to which topic. This paper focuses on topic-based pub/sub. We consider that its simplicity, compared to content-based pub/sub, makes it more suitable to the limited resources available in sensor devices (a topic being in the end just a type or a tag for some data). The only element of the broker that actually depends on the type of pub/sub is the pub/sub engine, so the architecture described here would be largely unchanged if a content-based pub/sub engine and content-based routing protocols were to be used instead.

Below we argue in Section V-A that many commercial sensor applications can fit into a pub/sub abstraction. Considering the sensor network as one or more publishers or subscribers has the advantage that a single uniform abstraction can be used in both the sensor network and the enterprise network. The broker then acts as a bridge between these two domains, effectively integrating the former into the latter. Section II has outlined the diversity of protocols used in sensor networks and argued that this is unlikely to change. Moreover, as devices are evolving rapidly (and improving), the addition of new sensor networks in the enterprise should not have to be conditioned by the technology used in earlier deployments. As a consequence,
a broker that communicates with sensor networks must be capable of supporting multiple application-specific protocol stacks. The introduction of new sensor applications should require only a minimal effort to update and reconfigure the broker.

Using this flexibility exclusively to integrate sensor networks would not exploit the broker’s full potential. As described before, there are a number of middleware technologies that can be used by an ESB to interconnect the enterprise applications components. The flexibility of the broker can also be leveraged to interface to the enterprise network independently of changes in the middleware technology used to implement the ESB. In the implementation described in this paper, we have focused on message-oriented middleware, but nothing in the design precludes the use of other mechanisms, provided that the appropriate mediation and transformation functions are implemented in the corresponding stacks. We focus on the sensor network side of the integration.

We have also focused our efforts on achieving a tighter integration of other communication means into the enterprise network. For example, one can imagine that the sensor networks may be deployed at locations where a wired network infrastructure is not available. The broker would probably need to support point-to-point radio communication, or possibly an Internet connection over a constrained mobile-telephony channel. In the case of communication across the Internet, the protocol decomposition is quite shallow; we have a particular pub/sub protocol running over a layer that encapsulates the sockets. For example, when TCP is used, then the pub/sub protocol operates directly over the socket layer. When communicating over a reliable wireless network, customized reliable transport protocols running over UDP can take advantage of their knowledge about the MAC and application layers to optimize delay and link usage. For example, in a 802.11 wireless LAN, the communication stack can be optimized such that it relies on the retransmission of lost packets to the base station and transmission delay does not trigger useless transport-level retransmissions. In situations where the enterprise network has to be accessed through VPNs over GPRS, a similar approach can be used to deal with connectivity to mobile phone antennas. Moreover, as traffic at the GPRS provider network is usually tunneled toward the Internet using TCP, TCP-over-TCP clocking can be avoided. Although several algorithms have been designed to cope with these or similar situations, our approach makes the system independent of the availability of a given feature in an OS stack. The optimization of the stack can focus on the most relevant parameter for the typical scenario in the enterprise network, coded as a module. Reusability of modules in other stacks enables tuning to adapt to other scenarios. Existing protocols can be extended by providing additional functionality on top; new protocols can be implemented from scratch, all the way to the physical layer, provided the broker has access to raw sockets (the functionality is available in Java\(^1\) to develop native libraries for raw sockets in any supported system; we have implemented libraries for Linux\(^2\) and Win32\(^3\) systems.)

The pub/sub engine can use any format for messages and topics; we opt for the one proposed by SCADA MQTT [18] as it is simple and designed for low-end devices. Protocol stacks running MQTT as their pub/sub protocol can then access the pub/sub engine directly, whereas others using different mechanisms need to ensure that the messaging format conforms to that specified by MQTT. MQTT is supported in many products implementing an ESB in the enterprise network.

JMS [19] is an API for message exchange in Java. JMS offers the possibility of sending and receiving messages from queues, as well as a publish/subscribe interface to send and receive messages on topics. No wire protocol is defined, so JMS implementations do not inter-operate in general. JMS is extensively used in the enterprise, and is the standard for J2EE applications message exchange, with many applications and developers using this API. However, many of the rich set of features offered by JMS do not apply in a sensor network. MQTT supports a JMS API on top, which allows for trivial integration into many ESB products and applications.

B. Design and Implementation

The communication system of the broker is a runtime environment for the execution of protocol stacks. Its main design goal is a flexible way to support arbitrary pub/sub protocols. Protocols may be added to or removed from the broker; they may be deployed or reconfigured.

An important design choice is the granularity at which the configuration is done. At one extreme, entire protocol stacks can be configured. Such a coarse-grained solution may allow very efficient implementations and simple configuration, but it does not allow the re-use of protocol components in different stacks. In addition, the runtime environment cannot exploit parallelism within a stack, as each stack is a monolithic entity. At the other extreme is a very fine-grained configuration based on individual protocol functions such as checksum computation. This is essentially the approach taken by the x-kernel. It requires a rather complex configuration. A protocol stack is described by an acyclic graph that describes the dependencies and order of execution of individual protocol functions.

The communication subsystem in our proposed broker uses an approach that is in between these two extremes. In the broker, protocol stacks are composed of individual protocol layers. These layers form a simple stack as shown in Figure 1. Each layer is implemented by a protocol module. Experience indicates that this approach is a good compromise for application-specific protocols. It offers high flexibility owing to the re-use of protocol modules and allows an efficient concurrent execution environment.

Each protocol stack has an entry and an exit point. The bottom-most module interfaces to a network or I/O service that is provided by the operating system. The top-most module interfaces to the pub/sub engine that forms the core of the broker. Otherwise, there is no restriction to what a protocol module may implement as long as it implements the protocol module’s programming interface correctly. Figure 2

---

\(^1\)Java is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

\(^2\)Linux\(^T^M\) and Win32\(^R^T^M\) systems.)
Fig. 1. Example communication protocol stacks.

shows the data interface of a module. The handleSend and handleReceive methods are invoked by the runtime environment to process a packet (protocol data unit) in the sending and receiving direction, respectively. The module invokes dispatchSend and dispatchReceive to hand off a packet to the runtime environment, where it is passed on to the next module in the stack. Modules also implement a handler for time-outs and a control interface for starting and stopping.

![Protocol Module Interface](image)

Fig. 2. Protocol Module Interface

The common end-point of all protocol stacks is the pub/sub engine. The pub/sub engine implements a simple topic-based publish-subscribe model. Messages in this model are data objects without any visible structure; the pub/sub engine treats them as opaque data types. Messages are published to topics, where a topic is a path name in a hierarchical topic space. Topics in the topic space follow a similar structure to path names in a file system: a list of topic names that are separated by a '/' character. For example, "ABuilding/2ndFloor/Office123/Temperature" could be used as a topic where temperature information of a particular office is made available. Subscribers may use wildcards to subscribe to an entire set of topics. Two types of wildcards exist. The '+' character stands for an arbitrary topic name within a hierarchy. For example, "ABuilding/2ndFloor/+/Temperature" designates all temperature topics of the 2nd floor of building A, independent of the office. The second wildcard character is the hash sign '#', which stands for an arbitrary path in the hierarchy. The hash sign can only be used at the end, for example "ABuilding/2ndFloor/#" indicates all topics that start with the "ABuilding/2ndFloor".

The pub/sub engine provides a programming interface for publishing messages on a topic and for subscribing to a topic or topic set (if wildcards are used). Publishers and subscribers register a handler with the pub/sub engine, which is then used for message exchanges.

1) Configuration and Instantiation: Protocol stacks are managed by the Communication Manager. It is able to manage multiple stacks in parallel. At the core of each protocol stack is a protocol anchor. The anchor is responsible for instantiating and administering stacks. If offers a life-cycle management to start and stop individual stacks.

Protocol stacks are configured by listing the names of the modules composing the stack. This stack description identifies the anchor module as well as all the other protocol modules in the stack, i.e., a stack description defines the type of the stack. In addition, each stack is further specified by a set of stack parameters. These parameters define, among other things, network and transport addresses. By separating stack description and stack parameters, it is possible to instantiate several stacks of the same type that differ in their parameters, for example a TCP-based stack type could be started several times on different TCP ports.

When the communication manager instantiates a new protocol stack, either at start-up time of the broker or during runtime by an administrator, the implementations of each module are dynamically loaded. A dry-run test of the stack verifies that the configuration is sound. Then, the protocol anchor is activated. It listens for incoming client connections and instantiates a new protocol stack for each client that connects. Figure 3 shows this hierarchical management scheme.

2) Execution Environment: The execution environment consists of an event dispatcher with a thread pool and a timer. At the heart of the dispatcher is an event queue. An event has a header, containing the destination protocol module to which the event should be delivered, and data. A typical event would be the arrival of a packet from the network. The dispatcher continuously chooses a free thread from the thread pool, the next event to process, and the protocol module that should handle the event. The protocol modules themselves write events back into the queue using the dispatchSend and dispatchReceive calls, and this is their only interaction with the rest of the protocol stack. Each module in the protocol stack keeps a reference to the module above and below it. A protocol module forwards a message up the stack by dispatching it to the protocol module above it; messages are sent down the stack by dispatching packets to the protocol module below. The journey of a message through the stack can be seen as a series of standardized interactions between the protocol modules and the dispatcher.

The effect of the above is to create a type of assembly line or pipeline in which multiple messages in the same flow (but in different modules in the stack at any given time) can be
processed simultaneously by different threads in the thread pool. The dispatcher guarantees that only a single event is processed by a given protocol module at a given moment. This in turn guarantees in-order delivery.

The timer generates one-shot or periodic timer events on behalf of the protocol modules. These events are also added to the dispatcher’s event queue, from where they are delivered to the destination protocol module. Timer events have higher priority than data events. This is taken into account in the dispatcher by having distinct event queues with different priorities and ensuring that all higher-priority queues are empty before taking an event from a lower-priority queue.

A key problem is finding the optimal number of threads that should be executed concurrently. A general solution cannot be given because it depends on the nature and depth of the protocol stacks, the ratio of I/O to CPU-related tasks, and other activities running on the machine. If too few threads are run, then the system is underperforming, whereas if too many are run a penalty is paid for the context switching between many threads.

We have designed a mechanism to adjust the number of threads running by monitoring the ratio of the time they are busy to the time they are idle. If the aggregate ratio is too high then additional threads are added to the system, up to some maximum. If the ratio is too low then threads are removed from the system, assuming that there is at least one thread left. The length of time across which the threads are observed is called an epoch, and the thread manager makes the adjustment decision at the end of every epoch. If the ratio of busy to idle at which the system is considered overloaded is too close to the ratio at which it is considered underloaded, then the system will oscillate. Optimal values for the epoch and the overloaded and underloaded ratios can either be obtained from analysis or testing the expected range of operation of the system. We supply default settings of 60 seconds for the epoch and overloaded = 0.9 and underloaded = 0.1.

Figure 4 shows an overview of the main components of the runtime environment. The dispatcher processes events from one or several event queues. Events are assigned to active threads, which then execute the designated protocol modules on the communication stack. Multiple protocol modules can be executed in parallel, even within the same communication stack. During execution, protocol modules may generate additional events that are put into the dispatcher’s event queue, for example, to invoke the next protocol module down or up the stack.

Figure 4. Dynamic dispatching of events to modules in the stack.

IV. BROKER SCALABILITY

As pointed out, scalability is a key feature of the broker. We expect the broker to act as a gateway for several sensor networks simultaneously, having to process a potentially high aggregate throughput coming from many different sources. To check the capabilities of our implementation, we run the following test: because we do not have enough sensor devices to perform the experiment with a significant number of stacks, we instead run a stack implementing a reliability protocol on top of UDP. Note that in general the communication with the sensor network is also datagram-based, with some logic on top that enables reliable communication with the sensor network. Thus the setup is arguably similar to that used when contacting a real sensor network. Recall that our aim is to test the scalability of the broker, not that of a particular sensor network protocol or its implementation.

The broker runs on an IBM® BladeCenter® with two In-
We test the performance of the system in an actuation scenario. We have one publisher application sending messages to the sensor networks. The message payload is 512 bytes. Note that this is quite a large payload for sensors, given that, for example, TinyOS uses a maximum message size of 49 bytes. So each message we send translates to roughly 10 packets on the sensor network. The results we show in Figure 5 are calculated as the average of the receiving rates over all subscribers. We run tests for various numbers of subscribers (i.e., sensor networks), increasing the sending rate of the publisher until it can no longer be sustained or an error appears.

Note that for 10,000 subscribers, more than 20,000 messages per second are pushed out to the clients, so we achieve more than 2 messages per second simultaneously on 10,000 sensor networks. For large subscriber populations, the amortized Java virtual machine memory usage of the Communication Subsystem per client is around 5 KB (this figure does not take into account the memory used by the operating system for sockets, though).

Having 10,000 sensor networks attached to one gateway is probably not realistic, but it shows the scalability of our approach. Also consider that normally the broker would not be deployed on such powerful hardware; but given these results, we are confident that the broker can support up to hundreds of sensor networks on much more modest equipment.

V. INTEGRATING SENSOR NETWORKS INTO THE ENTERPRISE

In their survey on network abstractions for TinyOS, Levis et al. [20] classify tree-based routing as a general abstraction that OS's for sensors should support directly. The sensor network technologies outlined in Section II all use a routing tree as the basic communication infrastructure. It is thus safe to assume that in general, sensor networks communicate with the outside world through one or more gateways located at the root of the routing tree. This is the location of our broker. We describe how sensor networks can be subsumed into the pub/sub paradigm to interact with the messaging broker, considering two possibilities: 1) the sensor network as a whole is treated like a publisher or subscriber, 2) each sensor is seen as an individual publisher or subscriber by the broker. The correct approach depends largely on the type of application running on the sensor networks and outside in the enterprise network: if the external application is only concerned by the sensor network as a whole or by parts of it, then the first possibility is more convenient. Examples of such applications are those gathering data aggregates from a sensor deployment, like average temperatures in an area of the network, or controlling actuators in coordinated groups, like sensors with orientable panels forming an antenna array; or having lights being switched on or off in parts of a building. On the other hand, applications addressing independent operations to individual sensors may be easier to design with the second possibility. An example of such an application would be sensors controlling price displays on the shelves of a store: each price needs to be controlled independently, according to stock or demand (although it may be interesting to drop or rise the price of entire lines of products, which would again be a case for the first possibility).

For each case, we present an example of sensor network protocol developed in our laboratory. Both protocols were implemented in the TinyOS platform, using the NesC language on CrossBow® MicaZ motes. Similar solutions could be implemented on sensors using a ZigBee stack, for example. Ideally, we would only need to change the module dealing with the network layer in the broker.

A. TREATING SENSOR NETWORKS AS PUBLISHERS AND SUBSCRIBERS

Applications in the fixed network need to interact with the sensor network either to obtain the information that sensors produce and/or to change the control settings of the devices. Having the application interact directly with the sensors would make the applications more complicated, requiring them to keep track of individual sensors and maintain the means to communicate with them. Often the identity of individual sensors is unimportant to the application, e.g., the application wants to know the temperature at a location, but does not care how the set of sensors present at that location combine to supply the information. The authors of [3] call this a data-centric approach to networking as opposed to the address-centric approach, which underpins the Internet.

One approach to shield the applications from the complexity of the sensor network is TinyDB [21]. TinyDB treats the

---

2 Intel Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

3 Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
entire sensor network as a database that can be queried using SQL-type requests. Each sensor receives the requests from the parent and propagates them to the children in the routing tree, whereas replies are returned back up the tree. The root of the tree is the connection to the fixed network from which requests originate. There are a number of issues why TinyDB is not suitable for our purposes: the application needs to reorganize itself when the routing tree changes; the results of a query can only be sent to one gateway to the outside, so we cannot do load balancing between brokers, and the database paradigm does not fit reliable actuation.

Alternatively, the sensor network can be treated as a publisher or subscriber on one or more topics within a pub/sub framework. Pub/sub does not allow the gathering of arbitrary data from the sensors, only those for which it was previously configured. However, for a wide class of applications this is adequate. It is more flexible in the way data is distributed because the structure of the data is not tightly bound to the routing strategy. Well-known sensor-network techniques, such as data aggregation, routing trees, snooping, and synchronized wake/sleep periods between neighbors, can also be used in this context.

A message broker supporting multiple different protocol stacks acts as a bridge between the fixed and the sensor networks. Note that by hiding the sensors from the broker, the latter does not need to dedicate resources to each individual sensor. The utilization of the resources is improved by pooling them, so that they are used by the sensor network as a whole. This enables better scalability, as we have seen in Section IV, by making the internals of the broker simpler. Applications within the enterprise network can connect to the broker, allowing them to send/receive data from the sensors using existing commercial messaging systems. Figure 6 shows the general schema of the interaction with the broker. We now show an example of sensor network integration using a pub/sub system.

Messo/Preso: Messo and Preso are two independent sensor network protocols for sensing and actuation, respectively, on the TinyOS platform. The reader interested in the internal details of the protocols is referred to [14], [22]. Both protocols follow the principles of pub/sub abstraction described above: to gather the sensor data, the external application subscribes to a topic where the sensor network publishes. Conversely, to actuate on the network, an application publishes to a topic to which the sensor network has subscribed. The sensor network may publish on multiple topics as the different sensors may be taking different measurements. Each topic forms a soft-state overlay among the sensors interested in that topic in the routing tree. In this way, the data transmitted on a given topic is not a concern for sensors on other topics. The overlays are maintained by the sensors periodically announcing the topics they are interested in. These announcements are aggregated along the path and routed to the top of the tree, where the broker manages subscriptions or publications on the topics of interest. Because the announcements follow the underlying routing tree, this setup is robust against topology changes.

In our broker, we create a stack for the network and higher layers of the sensor device. Figure 7 shows the relation between the broker and the root of a wireless sensor network. The Device Messaging Layer module on the broker performs some basic health checks on messages received and adds/removes the diverse headers associated with the specific routing protocol the sensors are using. For example, when the TOS routing protocol is used, the module removes the TOS and multi-hop headers when receiving packets and adds the broadcast header when sending them. Then, on top, we have the module implementing the sensor network pub/sub protocol, e.g., Messo or Presso, which interfaces with the pub/sub engine by conforming to SCADA MQTT mechanisms.

The Crossbow devices have a UART interface which allows them to be connected via a serial line to a PC. TinyOS can be programmed such that every active message received on the radio interface is sent to the serial line and vice versa. Communication with the PC is achieved through a PPP-like framing protocol implemented in both the TinyOS programming language NesC for the sensor and in Java for the PC.
Communication with the sensors through a serial line is an artifact of having the radio chip and the broker running on different platforms. The use of a dedicated gateway platform with a radio interface would allow a tighter integration, removing the need for serial communication, and makes the broker appear as an element of the sensor network. Note that, thanks to the flexibility of the broker, this only requires exchanging the serial line module by one encapsulating the OS support of the radio interface.

B. Individual Sensors as Publishers and Subscribers

Although we have just seen that considering the sensor network as a whole has advantages that simplify the integration of the sensors into the pub/sub system, there may be cases where this is not the most adequate approach. When each device is essential to the task performed by the sensor network, the assumption that individual sensors may fail and the network must continue to operate normally does not hold. Sensors may then not be of the inexpensive, disposable type. We expect the number of sensors in those cases to be rather small. Then each individual sensor may act as a publisher, and each individual actuator as a subscriber.

The flexible broker can also accommodate sensor networks following this approach. Moreover, given the scalability shown in Section IV, we expect no problems for the broker to cope with large numbers of sensors. This does not preclude scalability problems within the sensor network itself, but there is nothing the broker can do about that. We now provide an example of such a protocol integrating individual sensors into the pub/sub system.

MQTT-S: MQTT-S [23] is an implementation of MQTT for sensor devices. MQTT-S has also been adapted to be used with our broker. The sensors connect to the broker through a gateway that communicates with the corresponding protocol stack. The gateway appears to the broker as a single MQTT publisher or subscriber, hiding the complexity of multiple individual sensor connections. This way, a large number of sensors can be addressed individually, without the need for an individual connection per sensor in the broker. In this sense, the gateway acts as a concentrator.

Most of the protocol functionality is implemented in the broker and the gateway associated with the sensor network, leading to sensor clients with a small memory footprint. Implementations exist for both ZigBee and TinyOS communication stacks. Although unable to interoperate with each other directly, both types of sensor networks can communicate through the broker by having two stacks with different network protocol modules.

VI. CONCLUSION

Widespread acceptance and usage of sensor networks in the enterprise can only come if this technology can be integrated into the environment in which enterprise applications are developed and run. Integrating sensor networks into the enterprise network involves merging two different communication worlds into a coherent whole. Existing work has shown how to hide the sensor networks complexity behind a higher layer of abstraction by presenting the sensor network as a database to external applications. We have proposed a different approach with a middleware infrastructure using the pub/sub abstraction, focusing on the need to have the sensor network information processed and used outside the sensor network itself. We have described our implementation of a configurable and scalable broker that can deal with multiple different customized protocols, integrating sensor networks into the enterprise applications by means of message-oriented middleware. The performance of this broker is sufficient to communicate with a large number of sensor networks attached. We have also provided an overview of the protocols implemented on both the sensor network and the broker, supporting the pub/sub messaging abstraction, showing how an entire sensor network can be viewed as a publisher or a subscriber on a small set of topics.

REFERENCES


