RZ 3637 (# 99647) 11/07/2005
Computer Science 12 pages

Research Report

Techniques for Integrating Sensors into the Enterprise Network

Sean Rooney, Daniel Bauer, Paolo Scotton

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
{sro,dnb,psc}@zurich.ibm.com

Techniques for Integrating Sensors into the Enterprise Network

Sean Rooney, Daniel Bauer, Paolo Scotton
IBM Research, Zurich Research Laboratory
Säumerstrasse 4.
8803 Rüschlikon, Switzerland
{sro,dnb,psc}@zurich.ibm.com

***Abstract*— Cheap programmable sensor devices are becoming commercially available. They offer the possibility of transforming existing enterprise applications and enabling entirely new ones. The merging of sensor networks into the enterprise network poses some distinct problems. In particular, information from theses devices must be obtained in a way which minimizes their energy use and must be aggregated and filtered before being sent to the application server to prevent it from being overwhelmed. We describe a range of complementary techniques for integrating sensors into an enterprise network. These comprise new architectural entities within the enterprise network — *edge server* — new means of sharing information within the enterprise network — *messaging binning* — and new protocols for extracting information from the sensor network — *Messo*.**

***Index Terms*— asynchronous messaging, sensor systems, scalability, power efficient protocols**

I. Introduction

A sensor is an entity capable of sensing the state of some underlying systems and transmitting information about that state to some higher-level entity. Hellerstein et al. [5] give an overview of the spectrum of sensor technologies that are, or soon will be, available. Existing work in the sensor field has looked at protocol design for sensor-to-sensor communication [10] and on creating the TinyOS Operating system which runs on the sensors themselves [6].

From a commercial point of view, the use of sensor information within enterprise applications has the potential to enhance existing applications and to offer entirely new ones. For example, RFID readers allow better inventory management systems in warehouses, novel car insurance schemes can be realized using in-vehicle sensors to observe customer driving patterns.

In those applications where there is a small maximum acceptable delay between event occurrence and message reception or where the sensors are too simple to buffer messages, the message rate at the application server is mainly determined by the sum of the peak rates of all the sensors. This can potentially be very large. For example, extrapolating from the retail figures given in [12], a large supermarket chain may handle a number of purchases on the order of 10^8 per day. Assuming a constant purchase rate over a twenty four period, this corresponds to an average of 10^4 purchase events per second.

To reduce the load on the application server, an architectural entity is often placed between sensor and application server. These servers are termed *edge servers* as they are logically located at the edge of the network. This edge server performs application specific processing on the data before forwarding it to the application server. This processing reduces the amount of data the application server has to receive by filtering or aggregating the raw sensor data. For example, an average temperature reading can be calculated for a set of thermal sensors.

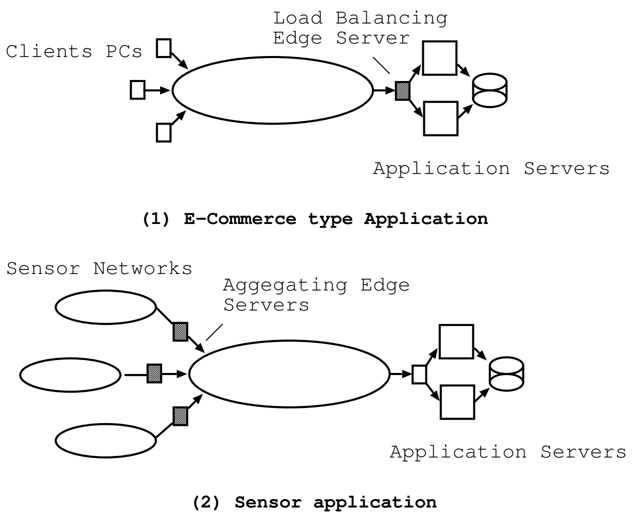


Fig. 1. Integrating sensor networks with enterprise applications

Figure 1 shows the distinction between these edge servers and more traditional ones. Edge servers are necessary for scaling certain sensor applications, but a large number of geographically distributed edge servers means that failure of parts of the system are more the norm than the exception. This problem is exacerbated by the fact that the edge servers often are executing in a more hostile and less protected environment than the application servers, in the sense that edge servers are deployed "in the field" and are not as easily protected as within large centralized server farms. As the logic they execute is application-specific it is important that they are easily extensible and modifiable. We describe an edge server architecture in which tolerance of failure is achieved through the use of a novel form of distributed asynchronous messaging for inter-edge server communication. We motivate its feasibility through a description of its implementation and its use in supporting a real application.

Low end sensors are typically battery powered. Much work in the literature has focused on how communications with such devices can be performed in a power efficient way. The protocols that connect the sensors to the enterprise network must also be frugal in their power usage. We explain the protocols we have developed for allowing the edge server to communicate with the sensor network, building on previous work in the literature but extending it such that it integrates easily into enterprise middleware. We describe our implementation of the *Messo* protocol which runs on the TinyOS operating system.

Some of the work has been described in previous papers [15], [16]. Those papers assumed a complex sensor such as a telematics device in a car or an RFID reader in a supermarket. What is novel here is the extension of the architecture to take into account much simpler sensors.

II. Inter-Edge Server Communication

A. Asynchronous Messaging

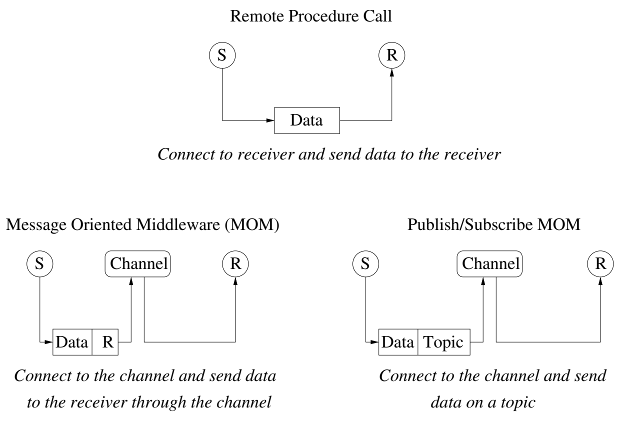

Remote Procedure Call
S
R
Data
Connect to receiver and send data to the receiver
Message Oriented Middleware (MOM)
S
Channel
R
Data
R
Connect to the channel and send data to the receiver through the channel
Publish/Subscribe MOM
S
Channel
R
Data
Topic
Connect to the channel and send data on a topic

Fig. 2. Remote procedure call and messaging operations

The use of asynchronous messaging reduces the degree of coupling between system components. To understand why this is so, first consider the transmission of data between a sender and receiver using a Remote Procedure Call. First a transport-layer connection is established between the sender and the receiver. Then the sender sends data to the receiver. The degree of coupling is quite tight because the party that initiates the connection must know the address of the other, and both parties must be active simultaneously for data to be exchanged. In a messaging system, senders and receivers do not communicate directly with each other but via a messaging abstraction. This additional indirection allows senders and receivers to be unaware of each others' existence and allows the transmission and reception of a message to be decoupled in time. In consequence neither senders nor receivers need to maintain state about each other, allowing the addition and removal of, for example, edge servers and sensors without requiring the reconfiguration of others. The abstraction to which producers send messages and from which consumers receive them has many names in the literature, but we shall refer to it as a *channel*. Figure 2 illustrates these principles. For a good overview of publish/subscribe messaging systems, see [4].

An important distinction between messaging systems is whether a message is retained within the channel until it is explicitly removed by a consumer (*consume* semantics) or whether it is dispatched to all subscribed entities and automatically removed from the system (*multicast* semantics). The former is typically provided by message queues and is most useful when a message should be consumed by exactly one consumer, whereas the latter is useful for the dissemination of the same message to many consumers and is typically provided by message brokers. Sometimes an additional distinction is made with channels that support multicast semantics between *channel-based*, *subject-based* and *content-based systems* distinguished by whether the consumer receives all messages produced on a channel, only those on a certain subject, or those that match consumer-defined patterns on the message header and content.

The capacity of a channel can be measured in terms of both the volume of messages and the number of publishers/subscribers that the channel can support. For sufficiently high volumes of messages or numbers of publisher/subscribers, a single channel will become a bottleneck. In order for the messaging system to scale, it must be distributable across multiple channels.

Some attempts [1], [3], [14], [17] have been made to scale publish/subscribe systems to large scale networks. The typical application involves the distribution of information from a small number of authoritative sources, for example, the distribution of stock quotes is the canonical application in the literature. Existing

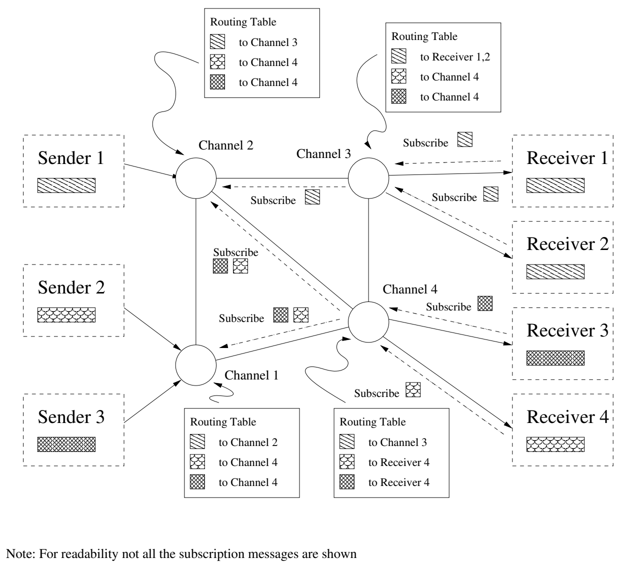


Fig. 3. Simple example of message routing

solutions are information disseminating rather than information gathering. Scalability in such systems refers to the fact that there may be a very large number of geographically distributed consumers. All of the projects use a message routing approach in which messages are routed between channels. Message routing reduces the number of consumers that each channel has to handle. The projects differ in the details of the format and means by which the association between channels is established and how the routing information is disseminated. In Siena [3], an application advertises its ability to produce messages and its desire to subscribe to them to an ingress channel, which propagates the information in aggregated form to other connected channels. A message routing tree can then be formed such that channels know which other channels to forward messages to. Figure 3 shows a very simple example of message routing.

While the message routing approach is appropriate for information distribution, it is unsuitable for gathering information from sensors because the data produced by one sensor is typically consumed by only one edge server. Data may be forwarded to more than one edge server in order to enhance fault tolerance or if several different applications are making use of the same data, but even in these cases the number of consumers is small. This observation leads us to propose an alternative means of distributing the messaging system.

B. Message Bins and Meta-Channels

A message bin is a message channel with an associated controller. The channel retains messages until they are consumed. The message bin's controller announces

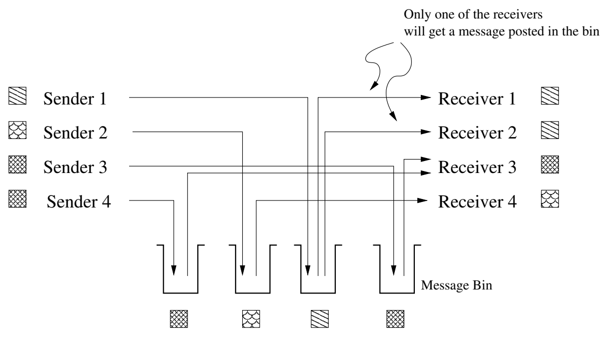


Fig. 4. Simple example of message bins

its willingness to receive messages on a given topic. A message producer selects a bin among those that have announced their willingness to receive messages on the topic that the producer wants to produce. An entity wishing to subscribe to a topic chooses some subset of the message bins that are willing to accept messages on that topic. By configuring the system such that not all message bins accept all topics, we reduce the number of connections that publishers and subscribers have to maintain.

The message bins make announcements, called *ChannelDescriptions*, using a meta-channel. The meta-channel, unlike a message bin, supports multicast semantics. There is nothing special about the meta-channel other than the information it carries, and that it can be implemented using any messaging system that supports multicast semantics. In our current implementation we have used both the CORBA [13] Notification system and a broker based on the Message-Queue Telemetry Transport (MQTT) [7]. Message producers and consumers both subscribe to the meta-channel in order to learn about the availability of bins. The message bins publish the *ChannelDescriptions* periodically, so that a subscriber to the meta-channel will learn about the existence of all message bins sometime after subscribing to the meta-channel. All receivers of *ChannelDescriptions* must treat them as soft state, i.e. information that must be refreshed regularly or otherwise expires.

In addition to the list of accepted topics, the *ChannelDescriptions* contain the following information:

- the technology used to support the message bin, e.g. tuplespace;
- the information needed to connect to the bin, e.g. address, port;
- the number of publishers and the number of subscribers;
- an indication of the current load on the message bin;
- the time period between successive publications of *ChannelDescriptions* in seconds, i.e. the refresh

rate.

This information is sent in the form of an XML document, meaning the same format can be used regardless of the publish/subscribe system supporting the meta-channel. Additional information may be added to the message in order to aid filtering in specific technologies. For example, the message types supported by the channel are added to the variable header part of the CORBA event or into the property field with the Java Message Service (JMS) message. This allows consumers to specify that they are only interested in information about channels carrying messages of a given type.

Failure of a channel may be inferred from non-reception of a channel description in a time period proportional to the time period stated in the last *ChannelDescription*, for example twice the announced refresh period. It is recommended that refresh periods be on the order of many seconds in order to reduce the effect of queuing and propagation delays. The more frequent the refresh rate, the more reactive the system is in response to change but the higher the control overhead. Receivers may make a local decision to ignore the alleged failure of a channel due to non-reception of a *ChannelDescription* if they are successfully sending messages to or receiving messages from that channel. However, they must remove all stored state about that channel and not try to reuse it after their current sessions have finished.

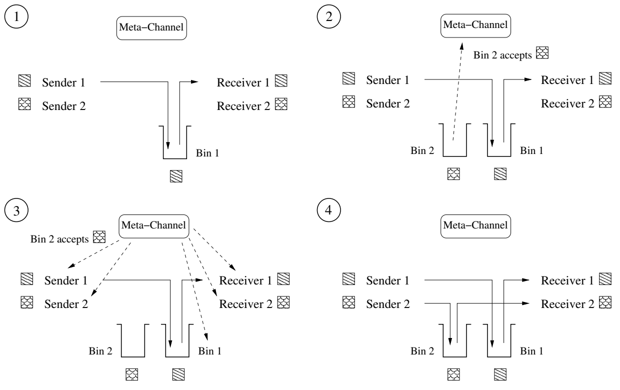


Fig. 5. Example of dynamic configuration

Figure 5 shows an example of dynamic configuration using the meta-channel. In Subfigure 2, when Bin 2 is activated, it communicates to the meta-channel that it is willing to accept a given type of messages. The meta-channel broadcasts this information to all the components of the system (Sub-figure 3) and in particular to Sender 2 and Receiver 2, which deal with messages of the same type as those processed by Bin 2. Therefore Sender 2 and Receiver 2 can connect to Bin 2 as depicted in Sub-Figure 4.

In the message binning approach, all entities must agree on the means of communicating with the meta-channel; but as message bins never communicate with each other, several different technologies can simultaneously be used to support the message bins. So, for example, within the edge server architecture, both lightweight and complex queuing systems can coexist. This is important as the architecture deals with the boundary between very simple sensors on one side and more sophisticated application servers on the other.

The means by which the channel controller extracts information from the channels is technology-specific. For example, in the tuple server implementation we have used [9], there is a special control tuple space which contains information about the others; in the MQTT microbroker [7], there is a control topic which serves a similar purpose. The CORBA notification service [13] does not offer the necessary information directly but can be extended to so so. The controller may reside within the same process as the channel or within an entirely different process. A given process may contain controllers for many different channels; however controllers must always reside on the same machine as the channel they control in order to reduce the possibility of inconsistency within the system.

Soppera et al. [20] describe an approach broadly similar to our own, but using IP multicast as the technology for the meta-channel. This removes the need for a distinct meta-channel but requires coordination between the middleware and network layers.

In summary, the meta-channel offers a means for disseminating information about channels within the system. This allows entities to adapt to changes in the system as load fluctuates or channels fail. Clients learn from the meta-channel where data messages may be published to and where they can be received from. Data messages are never forwarded between channels allowing heterogeneous channel types to coexist within the system. A system entity may identify and react to failure independently of other system entities. The need for a separate management system is lessened, but not entirely removed. For example, it may be the case that all channels are overloaded, because the offered load is too high for the available number of edge servers. In this case the meta-channel can be used for the publication of alarm messages to which management systems can subscribe.

C. Distributed Meta-Channels

In a sufficiently large system, a single meta-channel would become the bottleneck and a single point of failure. In consequence, the distribution of control information itself needs to be distributed. However, unlike data messages, control messages are not consumed. In that sense the distributed control plane is similar to the message routing systems such as Siena [3]. It would be possible to use Siena-type message routing systems for the dissemination of control information. However in our initial implementation we chose not to as the sensor systems, although large, are not as large as the Internet. For example, assuming a uniform channel technology capable of supporting as few as 100 publishers or subscribers, then each meta-channel could support 100 channel controllers, the channel of each capable of supporting 100 sensors. Ten meta-channels would be required for a system containing 100,000 sensors. So even a large number of sensors corresponds to only a small number of meta-channels.

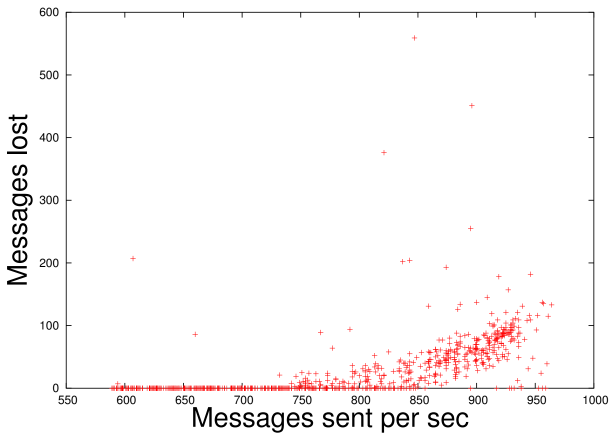
Instead of using a message routing approach, we flood control information amongst meta-channels. The entity which bridges control information available on a given meta-channel to others is called a meta-channel controller. The meta-channel controller entity creates a digest of all the channel descriptions on the meta-channel it controls and publishes it on all meta-channels (including its own). The digest is a list of *ChannelDescriptions* of the normal channels as well as a *ChannelDescription* of the meta-channel. Data message producers and consumers need only to subscribe to these control digests as they contain all the information required. The channel controllers use the digest as confirmation that their control information is actually being read, and use the refresh period in the *ChannelDescription* as a minimum for their own control messages. The meta-channel controller can therefore apply back pressure to the channel controllers to reduce their sending rate, allowing the system to adapt its responsiveness as a function of the load on the meta-channels.

The system needs to bootstrap itself so one meta-channel is well known. Channel controllers subscribe to this meta-channel to learn about other, possibly more appropriate, meta-channels for the publication of their controller information. Meta-channel controllers use it to learn about all the other meta-channels in the system. Initially, the configuration of a bootstrap meta-channel is done statically.

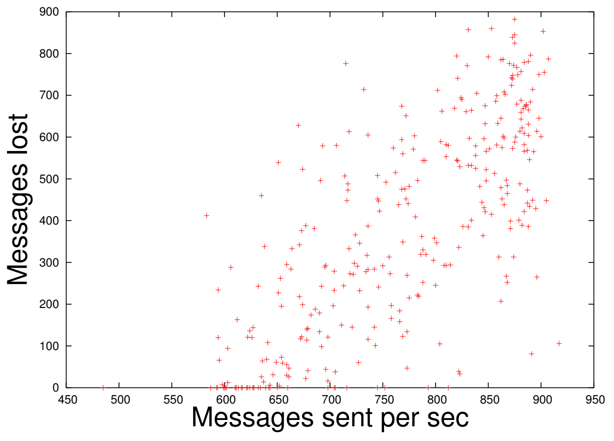
Flooding works well for a small number of meta-channels, but for a large system we would use message routing for disseminating the *ChannelDescription* messages.

D. Load-Balancing

As described in Section II-B, message bins announce the topics they are willing to receive using the meta-channel and their current load. Message producers use this information to determine where to actually place a message. The message producers make this decision



(a) One Subscriber



(b) Two Subscribers

Fig. 6. Crosstalk between consumers on the TAO notification server

entirely locally. There is no attempt to create a global schedule, not only because the schedule would need to be recalculated and redistributed upon failure, but also because estimating the service rate of a message bin requires exact knowledge of the software stack and the resources required by the executing software for a given message arrival rate. This is not easily achievable because resource consumption is non-linear with offered load; non-trivial computer systems have many thresholds across which behavior changes in a step-like fashion. Moreover crosstalk between software processes when executed together causes them to behave in ways that are difficult to predict, even when their behavior in isolation is known. For example, look at Figure 6(a) which shows the sending rate against the loss rate for the TAO implementation [19] of the CORBA notification server with one publisher and one subscriber, all components running on different machines, and then compare it with Figure 6(b), where the only difference is that there is an additional subscriber on a fourth machine[1]. The reasons for the effect need not concern us here; what is important is that it is difficult to extrapolate the behavior of the notification server with two subscribers, even if the behavior with one subscriber is well known.

The three factors that determine the distribution of load in the system are: (1) how the topics are distributed between message bins, (2) how message producers determine where to publish messages, and (3) how subscribers determine where to subscribe. Message producers and subscribers connect to a maximum of *N* message bins.

Message producers will not connect to a message bin that is announcing itself as overloaded; if they are already connected, they will not publish a message to that message bin. Message producers spread their load across all lightly loaded message bins to which they are connected. They may disconnect from one message bin and connect to another if the first bin is consistently overloaded. If a message producer cannot deliver a message on a given topic, it sends an alarm message on the meta-channel. Another unloaded message bin may then choose to start accepting messages on that topic. If the number of alarm messages sent per time unit exceeds a threshold, a human operator is notified, who then may add additional edge servers to the system or restart failed ones.

The disadvantage of a self-adjusting system is that it may become oscillatory if not properly designed. For example, a set of publishers might all decide to move their publications away from an overloaded message bin to a less loaded one. This causes the less loaded one to become congested, causing them all to switch back; the system never stabilizes although the resources, if correctly shared, would suffice to satisfy all requests. To avoid this oscillatory behavior, we introduce damping into the system. When a message publisher receives a *ChannelDescription* from the meta-channel, thereby indicating that a message bin to which it is connected is overloaded, it only decides to change message bin with some probability. This probability is inversely proportional to the current number of publishers announced in the *ChannelDescription*. For example, if there are *M* publishers on the overloaded channel, then each chooses a random number in the range 1..*M* and only changes location if the value is equal to *M*. The expected outcome is that only one publisher changes location for every *ChannelDescription* received, but any number may actually do so. Using this system, the probability that *M* publishers all switch to a new channel is $1/(M\,(M-1))$, and the probability they all switch back is the square of this.

Similarly a message bin only agrees to accept a new topic when an alarm is raised with a probability that is inversely proportional to the number of other message bins not offering that topic. When an alarm is sent on the meta-channel, message bins remove all topics for which they currently have no publishers, i.e. the alarm acts as a sort of garbage collection to free resources.

Note that indication of load is binary, i.e. a channel is either overloaded or it is not. An alternative would be to introduce a measure of load, for example instantaneous queue occupancy or number of messages received per second in a previous time window. More subtle load balancing decisions might then be made than that described here. In practice, we found that it was difficult to find simple measures of load that would allow a meaningful comparison between different channels running on computers with different resources.

After a step change in load, the system converges to a stable state without entities directly coordinating with one another. The speed of convergence is controlled by the exact probability function used and by the frequency at which *ChannelDescription* messages are sent.

III. Edge Server/Sensor Network Communication

A. Data-Centric Sensor Networks

When considering the communication between edge server and sensors it is important to distinguish between different sensors types. When a sensor is externally powered and has enough processing capabilities to support TCP/IP, it can be treated as an edge server. It may use 'lighter weight' middleware tailored for lower end devices such as MQTT, but communications between the sensor and the channel follow the conventional publish/subscribe semantics. As far as the channel is concerned it is an addressable entity from which data is received and to which data can be sent. An example of such a sensor would be a position reporting device on a vehicle.

When a sensor is not externally powered then all operations must be performed so as to minimize the energy usage. In particular, as radio power varies with the square of distance such sensors will typically form a network in which messages from sensors are forwarded across other sensors to reach a destination. The dominant

[1]The default settings for the TAO 1.3.1 notification server were used except that multi-threading was enabled. Publishers, subscribers and the notification channel ran on different machines, each running a Linux 2.4 kernel. The *attempted* sending rate was increased by 10 messages per second for each test starting at 600 per second up to 1000. The word *attempted* is used because of the synchronous nature of the communication between publisher and notification server: publishers slow down as the server slows down. The tests were repeated 20 times.

power draw within a sensor is typically the radio even when it is idling. Consequently, energy efficient network and MAC layer protocol are designed to allow the radio to be put to sleep as often as possible.

An additional observation is that it is often the case that the identity of simple sensors is not important, e.g. if we wish to know the temperature at a given location, which sensor from the set resident at that location actually reports is irrelevant. When this is the case then in-band processing can be performed by the sensor network itself, to aggregate data, e.g. throwing away duplicate readings. This reduces the amount of communication within the sensor network and hence saves power. Intanagowiwat et al [10] terms this a data-centric, as opposed to an address-centric, approach. Individual sensors are anonymous from the point of the view of the enterprise network allowing sensors to be added/removed from the network without requiring external configuration.

TinyDB [11] describes a means by which data can be extracted from a sensor network in a data-centric way. TinyDB treats the entire sensor network as a database that can be queried using SQL type requests. In TinyDB the sensors organize themselves into a tree such that requests are received from the parent and propagated to the children, while replies are returned back up the tree. Nodes within the tree combine data received both from their children and from themselves in order to reduce the total amount of data that needs to be transmitted. The root of the tree is the connection to the fixed network from which requests originate. The children and parents radio duty cycles are synchronized such that they the times that they wake up overlap.

B. Messo Protocol

Messo is a protocol that allows data-centric sensors to publish into a channel. Subscribers to that channel receive publications using the normal subscription protocol provided by the channel. We use the MQTT broker as the channel implementation, but Messo is independent of MQTT.

As in TinyDB there is a single location to which all data flows. We use the TinyDB tree approach to structure the sensor network such that the root of the tree is the edge server running the broker. Messo nodes are capable of publishing on a topic, topics are typically mapped onto sensor types: light, temperature, pressure etc, and Messo publications are one or more readings from these sensors. Figure 7 shows the Messo sensor network tree.

Messo nodes publish on a topic by sending a Messo publish message to their parent. Messo messages have a maximum length determined by the underlying sensor technology being used. Data is not segmented across

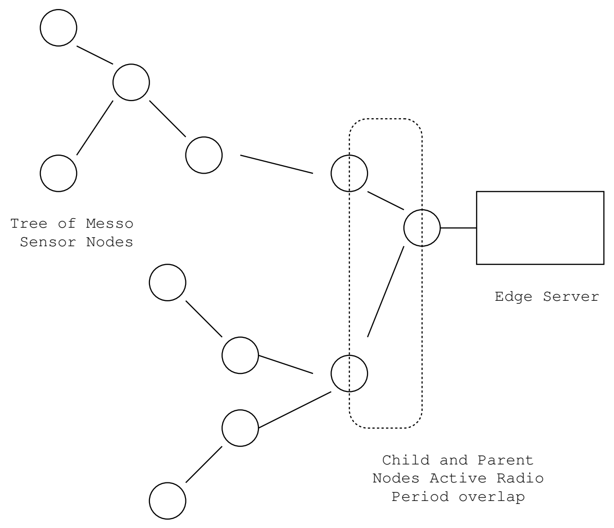

Tree of Messo Sensor Nodes
Edge Server
Child and Parent Nodes Active Radio Period overlap

Fig. 7. Overview of the Messo Tree

messages, if a Messo node wishes to send more data than fits into one message then it sends multiple independent messages. This means that a sensor reading must fit into one Messo message. A Messo node receiving a publication from a child forwards it on to its parent adding any data it may have on the same topic if there is enough free space in the payload. When the Messo publication is received at the edge server, it is mapped into one or more equivalent MQTT messages and published into the broker.

A Messo node will only publish on a topic if it can communicate with the broker at the root node and that broker has at least one subscription for that topic. In this way messages are not needlessly forwarded to the broker, and in consequence power squandered, unless there is a recipent for the message. The Messo node periodical sends the set of topics that it can publish-on to the root node. Just as for data messages, forwarding nodes may add additional topics into the set as it is forwarded up the tree, but in addition parent nodes remember the topics that their sub-tree can publish on, this set is called the publication set.

The channel periodically sends the current subscription set to the root node. Each Messo node only forwards this set to the sub-tree in which the intersection between the subscription set and the announced publication set is non empty. A Messo node will only publish on a topic for some period after it has received a subscription set message containing that topic, i.e. a softstate approach is used in which the subscription set message acts as a refresh.

Figure 8 shows an exanple of control message exchange between the channel and the Messo nodes. Each Messo node sends a publish set message stating which

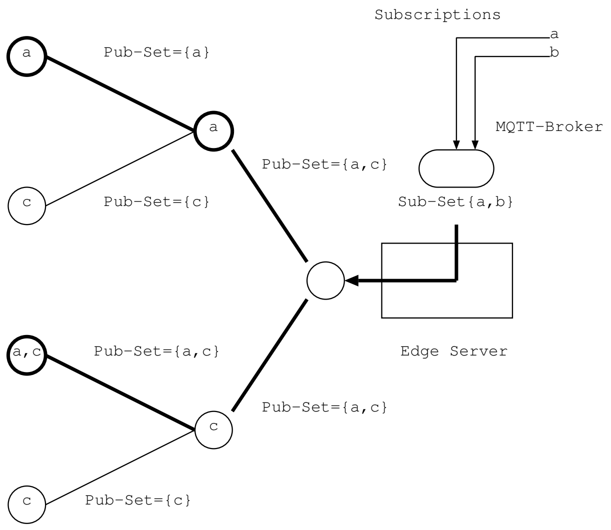


Fig. 8. Control Message Exchange

of three sensor types it is capable of sensing for: *a*, *b*, *c*. A given Messo node may support multiple different sensors. Each non-leaf nodes merges the sets from all its children with its own before fowarding on. There are two subscriptions on the channel for topics corresponding to sensor type *a* and *b*. The subscription set message is propagated down the tree to all those parts of the tree whose publication set has a non empty intersection with the subscription set; the path through the tree is shown in bold. A non-leaf node may receive the subscription set even though it itself has no sensors corresponding to the current subscriptions. In this case it forwards to the appropriate children but does not start publishing; all publishing nodes are shown in bold.

C. Messo Implementation on TinyOS

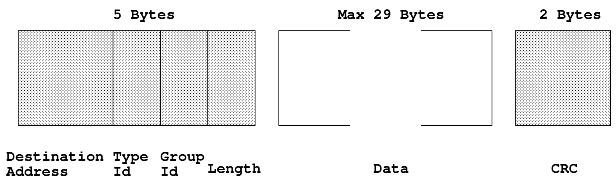


Fig. 9. TOS Active Message Format

We have implemented Messo on the open-source TinyOS operating system [6]. TinyOS is a simple event based operating system that runs on the Berkeley/Crossbow motes. Mica2 is a typical example of a current generation mote pocesssing a 8Mhz processor with a 128 Kbytes of ROM and 4 kbytes of RAM. TinyOS offers a set of libraries that allow application to read from a range of sensors type and send/receive values over a radio interface. As all of the system is open-source any part of it can be modified, but it is clearly beneficial to reuse existing libraries as much as possible. TinyOS offers an active message abstraction whose format is shown in Figure 9. Note that there is no source field in the TOS message header, i.e. a receiver does not know (unless the information is added to the payload) from whom the message was sent. The single byte group id allows multiple different application to share the same radio frequency and the single type id allows the purpose of the message to be identified. Messo is mapped directly onto the TOS message format. The 2 byte destination address is either the parent's address when a message is being sent up the tree or the reserved broadcast address when it is being sent down it. Type id's whose top most bit is set are considered control messages, currently there are only two: PUBLICATION-SET, SUBCRIPTION-SET, id's whose top most bit is zero are publication message on a topic defined by the remaining 7 bit integer, hence there are currently 127 possible topics within a given Messo network. The groupd id is not currently set.

TinyOS sensor reading are typically 16 bit integers, so a maximum of 14 distinct readings can be held within a given TOS message, less if additional information about the reading needs to be carried in the data; 29 topics types can be carried in a PUBLICATION-SET and SUBCRIPTION-SET

One mote is connected via a serial line to the edge server. This mote does not have any sensors but simply forwards TOS messages from the radio interface to the serial line and visa-versa. When a Messo/TOS message is received over this serial line a Messo/MQTT bridge running on the edge server converts the message into the appropriate MQTT format and publishes it into the MQTT broker. The mapping between the single byte Messo topic and the MQTT topic (a string) is held at the edge server. Conversely the current subscription set is extracted by the bridge and transmitted over the serial line, and then down through the tree of Messo nodes. A Messo client capable of storing ten sensors readings uses 12k of RAM and 1.5k of ROM on a Mica2 mote

IV. Edge Server Software Architecture

The edge server runs on a single processor, either a cabled PC or a lighter weight battery powered platform such as a PDA. Our software runs on a Java virtual machine, either standard or micro edition. The three major components within the architecture are: the application-specific code that processes the messages, message channels from which messages are received and to which messages are forwarded and the channel controller which tracks to where messages should be published and from where they can be subscribed.

A. Application Boosters

In order to distinguish between the software running on the application server and the application-specific software running on the edge server we term the latter a *booster*. The name emphasizes that the edge servers are there to enhance performances. There may be many different boosters running on the same edge server. Boosters receive messages from one or more input message channels, process them and forward them to one or more other output message channels. The technologies that support the input and output message channels may be distinct.

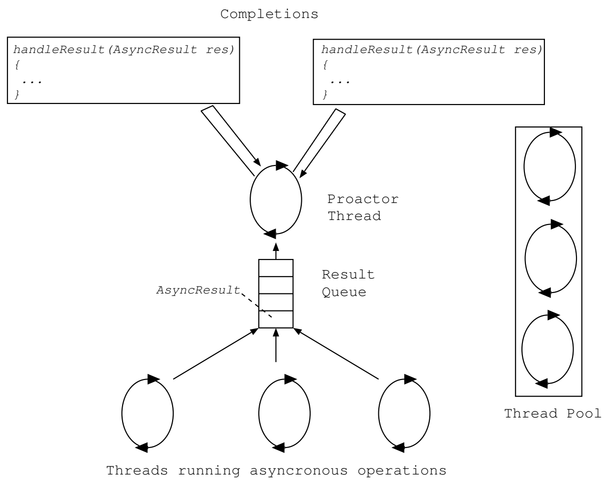

Completions
handleResult(AsyncResult res) { ... }
handleResult(AsyncResult res) { ... }
Proactor Thread
Result Queue
AsyncResult
Thread Pool
Threads running asyncronous operations

Fig. 10. Proactor model for supporting boosters

The edge server software architecture is built around a proactor [18]. There is one proactor instance per booster, cf. Figure 10. Boosters add asynchronous operations to the proactor which allocates them to threads taken from a thread pool. The asynchronous operations deliver the result from the operation to the proactor which then dispatches them to booster defined handlers. From the application programmers view the booster is defined by a list of asynchronous operations and the handlers that should be used to treat the results from those operations. The environment offers a set of standard asynchronous operations, these include those required to register and receive or transmit messages from different messaging systems such as the TAO/JacORB [2] implementation of the CORBA notification service, IBM Research's TSpaces [9] implementation of a tuplespace, IBM's commercial MQ messaging system and the SCADA sensor protocol MQTT [7] [2] as well as the Messo protocol explained in the previous section. In addition, there are non-messaging asynchronous operations of which the most useful are timers. Application programmers may add new asynchronous operations. Figure 10 shows the interaction between handlers, the proactors and the asynchronous operations.

[2]Note that MQTT was original known as MQIsdp.

Example: an application programmer wishes to process a set of messages received from a MQTT broker before transmitting them to an application server using MQ. The application running on the application server is implemented as a message bean and expects to receive messages using the Java Messaging System (JMS) API. A maximum of 100 MQTT messages are combined in 1 MQ message, but no message is held by the edge server longer than 30 seconds. The application programmers uses three asynchronous operations:

- *MQTTConsumerOp* passing the address of the broker service and the type of the message to consume;
- *JMSMQProducerOp* passing the address of the MQ broker and the type of the message to produce;
- *TimerOp* passing the expiry period.

The message types are simply represented as strings, which the *MQTTConsumerOp* maps to a filter expression at subscription to the MQTT broker and the *JMSMQProducerOp* maps to a JMS topic name. The application also writes the handler that does the application-specific processing, this handler takes *JMSMQProducerOp* as argument and is associated with *MQTTConsumerOp* and *TimerOp*. Every time an MQTT message is received it is dispatched to the handler, when either a 100 such messages have been received or the timer has elapsed the result is processed and the result placed in *JMSMQProducerOp*. This asynchronously formats the result as a JMS message and transmits it to the broker.

The approach is similar to the Rio architecture outlined in [21] in that Rio also uses asynchronous operations within a publish/subscribe framework to support distributed application specific processing in a fault tolerant way. It is distinct from Rio, in as much as Rio supposes one publish/subscribe mechanism — Sun's JavaSpaces — while we assume that multiple different messaging system are required. In addition, Rio uses Sun's Jini system for service discovery while, as explained in Section II-B we introduce a new component called a meta-channel.

B. Management and Deployment

The OSGi is a consortium of software and hardware vendors concentrating on enabling dynamic services for networked appliances. In particular, the OSGi specifies a service platform for loading and executing the software on networked appliances [22]. Code is installed on this platform in the form of a *bundle*, a

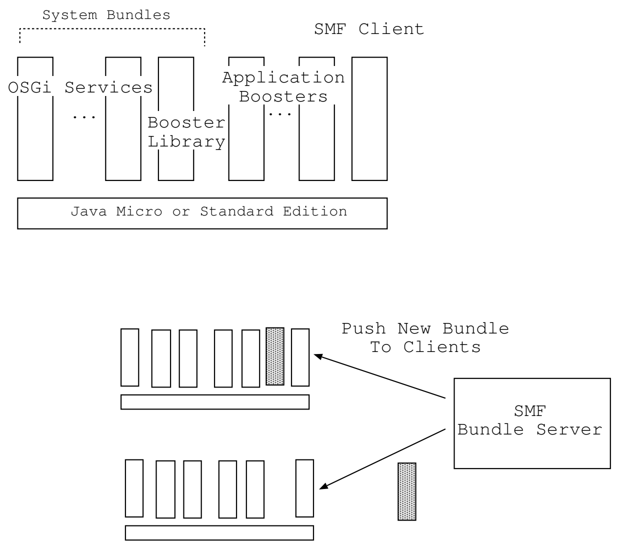


Fig. 11. Service Management Framework

bundle is just a jar file with additional meta information describing the resources that the bundle requires, the services that the bundle imports/exports and the means to activate/deactivate the bundle. Clients can request the installation of new bundles from a bundle server or the bundle server can push bundles to the clients. The bundle server keeps track of the dependencies between bundles and negotiates with the clients to ensure that an update will not render them inconsistent. The OSGi also specifies a set of standard bundles offering useful services for the platform. We use the IBM implementation of the OSGi platform – the Service Management Framework (SMF) – for the configuration management of the boosters. Figure 11 shows how the edge server is structured into several SMF bundles. Each booster is implemented as an active bundle, starting the bundle creates the appropriate asynchronous operations and the environment in which they run, stopping it remove them. The generic edge server code: proactor, thread pool, standard asynchronous operations, etc, is implemented as a library bundle.

C. *Edge Server Software Performance*

We measured the performance of our edge server using the a commercial traffic generator. We use simple UDP point-to-point communication rather than via a channel as this allows us to measure the performance of the edge server independently of the implementation of a channel. In addition, all TCP based communication is implicitly rate controlled by the receiver window size meaning that the sender can not offer arbitrary loads, furthermore channels that use protocols such as IIOP require that senders wait for application level acknowledgments from the channel before sending again. We use a *UDPConsumerOp* for the performance measurement that simply waits on a socket and dispatches the result to a handler. The sender writes a sequence number at the head of the packet payload and the handler uses it to determine the sending rate and the number of packets dropped.

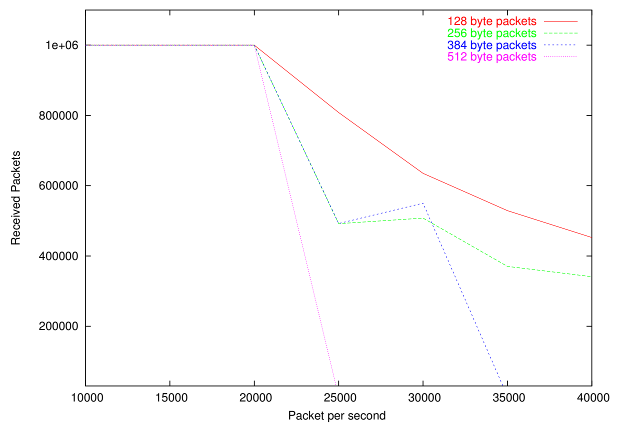


Fig. 12. Performance

Figure 12 shows the number of packets that the edge server receives at various sending rates for different packet lengths. In each test 1,000,000 packets are sent and the number that the actually application processes is recorded. For all packet lengths 20,000 packets per-second are comfortably processed without loss, above this level losses increase at a rate proportional to the packet length. Note that 20,000 512-byte packets per-second represents a rate of 82 Mbit/s.

V. Example Application

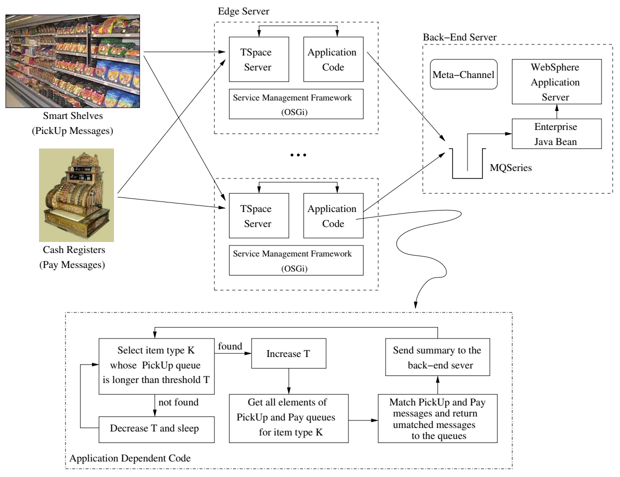

Edge Server
TSpace Server
Application Code
Service Management Framework (OSGi)
Smart Shelves (PickUp Messages)
Cash Registers (Pay Messages)
...
Back–End Server
Meta–Channel
WebSphere Application Server
Enterprise Java Bean
MQSeries
Select item type K whose PickUp queue is longer than threshold T
found
Increase T
Send summary to the back–end sever
not found
Decrease T and sleep
Get all elements of PickUp and Pay queues for item type K
Match PickUp and Pay messages and return umatched messages to the queues
Application Dependent Code

Fig. 13. Architecture of the example application

Our example application tracks the “in-flight time” of items within a supermarket. Every time an item is picked up within the system, a *pick-up* message is generated; every time an item is payed for, a *pay* message is generated. An application running on a WebSphere [8] application server uses this information in order to determine when an item has been in-flight for a suspiciously long time. This helps the supermarket to identify in real time when items are being stolen.

As Figure 13 shows, we use a tuplespace server [9] implementation and IBM’s commercial WebSphere MQ messaging system to support two types of message bins. The tuplespace server accepts raw *pick-up* or *pay* messages, whereas MQ accepts summary messages. The tuple servers run on the edge servers; MQ is run on the application server, and a single meta-channel (based on the MQTT microbroker) is also run on the application server. Application-specific code running on the edge servers matches *pick-up* and *pay* messages in order to calculate the average in-flight time of different item types, e.g. razor blades, and to identify individual items that have been in-flight significantly longer than the average. They place the summary messages on WebSphere MQ, which are subsequently picked up by the back-end application running on the WebSphere Application Server. The application-specific piece of code uses the meta-channel to identify where the raw messages are being published. Edge servers can be added and removed dynamically from the system, without requiring manual reconfiguration of the running components.

It is advantageous to wait until numerous raw item messages are available before processing, as this increases the ratio of raw to summary messages; however a raw message should not be left unprocessed for too long as this delays the time taken to indicate that the item has gone missing. The algorithm that the application-specific code executes is summarized in Figure 13. Note that, in scheduling theory terms, it is not work conserving, i.e. the application will sleep if there is not “enough” work to do. The reason for this is that other tasks running on the same edge server should be given a chance to execute. The application tracks the rate at which items arrive using the threshold value. Each time the number of items found is greater than the current threshold, the threshold value is increased by an amount proportional to the difference between the number of items found and the current threshold; for a fixed item arrival rate R and fixed service time S, the threshold thus approaches $S \times R$ asymptotically. Note that if S increases, for example when MQ becomes loaded, the threshold also increases, reducing the number of summary messages that are being sent to the application server. The threshold is divided by two when not enough work is found. Decreasing exponentially allows us to react quickly to a reduction in the arrival rate.

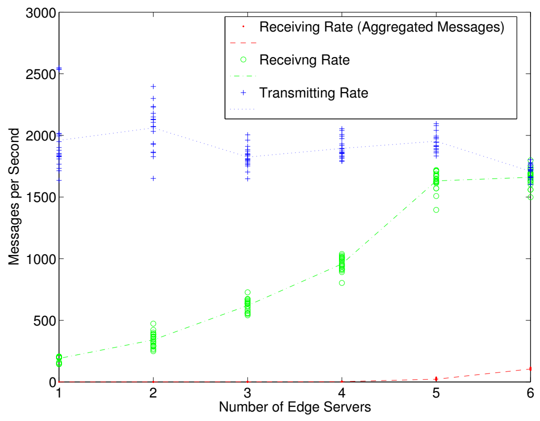


Fig. 14. Performance of Edge Servers

Figure 14 shows how the number of messages per second varies as we add additional edge servers to the system. The system reaches stability at six edge servers. The bottom line shows the receive rate of aggregated messages at the application server. The ratio of this to the receive rate of raw messages is the edge server gain. At equilibrium, the gain is about 10, but arbitrarily high gains can be achieved at the cost of increased delay. The more than linear increase in the performance is explained by the fact that the time to write a tuple is independent of the number of tuples in the space, but the time to read a specified tuple is linear with the number of tuples in the space. Hence, there is a double gain in adding a new edge server: there are more readers and the average queue length is shorter.

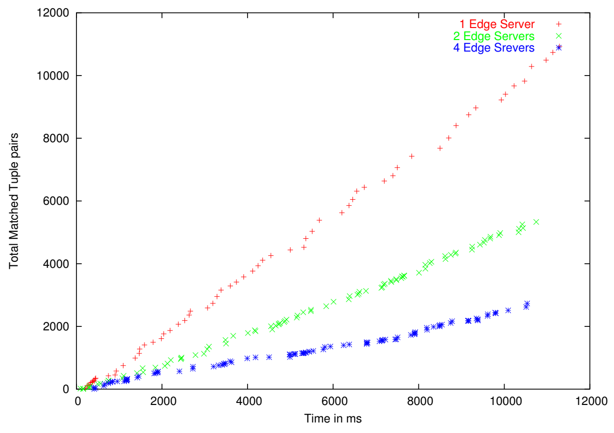


Fig. 15. Load-Balancing Edge Servers

Figure 15 shows how the cumulative number of tuples matched by a given edge server varies as the number of edge servers increases. As the number of servers increases by two, the cumulative number of tuples a given edge server matches is divided by almost exactly two, which demonstrates that the load is well balanced among the edge servers.

VI. Conclusion

Work within the area of sensor networks has concentrated on the set of problems related to the power efficient forwarding of data across those networks. In a commercial setting data extracted from sensors will be typical destined to an application server. In this paper we have looked at a the set of techniques which enable the integration of sensor networks into the enterprise network. The use of edge servers to preprocess data at the boundary between the sensor and enterprise network helps in scaling but makes the system more prone to failure. We have proposed an means of creating networks of edge servers within failure is handled gracefully. We have shown through the use of appropriate protocols how even very simple sensors can be connected into the edge server network. We have described our implementation of a proof of concept system and given performance results for a novel sensor based application.

References

[1] G. Banavar, M. Kaplan, K. Shaw, R. E. Strom, D. C. Sturman, and W. Tao. Information flow based event distribution middleware. In *Proceedings of Middleware Workshop at the International Conference on Distributed Computing Systems*, 1999. http://www.research.ibm.com/gryphon.

[2] G. Brose. Jacorb: Implementation and design of a java orb. In *Procs. of DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems*, Cottbus, Germany, Sept. 1997.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[4] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114 – 131, June 2003.

[5] J. M. Hellerstein, W. Hong, and S. R. Madden. The sensor spectrum: technology, trends, and requirements. *SIGMOD Records*, 32(4):22–27, 2003.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[7] IBM. Telemetry integration. http://www-306.ibm.com/software/integration/mqfamily/integrator/telemet%ry/.

[8] IBM. Websphere application server. http://www-306.ibm.com/software/webservers/appserv/was.

[9] IBM. TSpaces Programmer's Guide. 2002. http://www.almaden.ibm.com/cs/TSpaces.

[10] C. Intanagowiwat, 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.

[11] S. Madden. *The Design and Evaluation of a Query Processing Architecture for Sensor Networks.* PhD thesis, University of California, Berkeley, 2003.

[12] D. Mladenié, W. F. Eddy, and S. Ziolko. Exploratory analysis of retail sales of billions of items. In *Computing Science and Statistics, Volume 33 , Frontiers of Data Mining and Bioinformatics*, Costa Mesa, CA, June 2001.

[13] OMG. *CORBA Notification Service Specification.* Object Management Group Publication, August 2002.

[14] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture, 2002.

[15] S. Rooney, D. Bauer, and P. Scotton. Distributed Messaging Using Meta Channels and Message Bins. In *IM 2005, Integrated Network Management, Nice France*, pages 703–716, May 2005.

[16] S. Rooney, D. Bauer, and P. Scotton. Edge Server Software Architecture for Sensor Applications. In *Saint 2005, Symposium on Applications and the Internet, Trento Italy*, pages 64–73, Feb 2005.

[17] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[18] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume 2, chapter 3, pages 215–260. Wiley, 2001. ISBN 0-471-60695-2.

[19] D. C. Schmidt, M. Stal, H. Rohert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* Wiley and Sons, 2000.

[20] A. Soppera, T. Burbridge, and B. Briscoe. "Generic Announcement Protocol for Event Messaging". In *London Communications Symposium (LCS),University College London*, September 2003.

[21] Sun Microsystems. Rio architecture overview, v1.0, 2000. Technical report, Sun Microsystems, Inc.

[22] The OSGi Alliance. OSGi Service Platform, Release 3, 2003. http://www.osgi.org.