

12-1-2009

Reconfigurable middleware architectures for large scale sensor networks

Sean M. Brennan

Follow this and additional works at: https://digitalrepository.unm.edu/cs_etds

Recommended Citation

Brennan, Sean M.. "Reconfigurable middleware architectures for large scale sensor networks." (2009).
https://digitalrepository.unm.edu/cs_etds/46

This Dissertation is brought to you for free and open access by the Engineering ETDs at UNM Digital Repository. It has been accepted for inclusion in Computer Science ETDs by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.

Sean M. Brennan

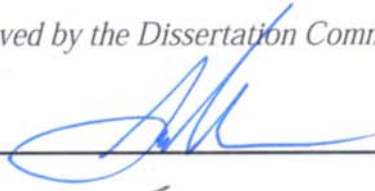
Candidate

Computer Science

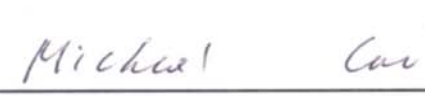
Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:



, Chairperson



Reconfigurable Middleware Architectures for Large Scale Sensor Networks

by

Sean M. Brennan

B.A., University of Iowa, 1994

M.S., University of New Mexico, 2003

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctorate of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2009

©2009, Sean M. Brennan

Dedication

For Annelly, Guendalyn and Sedrik.

Acknowledgments

I would first and foremost like to thank my committee chair, Barney Maccabe, for many, many years of mentorship, and for continuing to advise me despite his retirement from UNM. It was through him that I had the opportunity to work at Los Alamos National Laboratory (LANL) as a graduate research assistant, which led to a career as a research scientist. Though the road to this dissertation was long, winding, and challenging, I feel privileged to be Barney's last Ph.D. student.

I'd also wish to thank the other members of my committee, Wenbo He, Sudharman Jayaweera and Michael Cai for their vital feedback to my thinking and writing processes. Without their input, my explanations of SENSIX, and hence this dissertation, would have been far less clear. I particularly appreciate Michael's helpful guidance through the dissertation process.

The empirical experiments described herein were performed on LANL equipment funded by the NNSA Office of Nonproliferation Research and Development (NA-22) under the Distributed Sensor Networks with Collective Computation project (DSN-CC) led by Michael Cai, and previously by Angela Mielke.

I am also indebted to my kids, Guendalyn and Sedrik, for their assistance with bibliographical data entry and for their patience through this process. Finally, but most especially, I want to thank my wife, Annely, for making all this possible. Without her invaluable and ceaseless support - both moral and material - this research would never even have been started, much less completed. Thank you for absolutely everything.

This unclassified document is released as Los Alamos National Laboratory Unclassified Report, number LA-UR-09-07188.

Approved for public release; distribution is unlimited.

Los Alamos National Laboratory is operated by Los Alamos National Security, LLC for the U.S. Department of Energy under U.S. Government contract DE-AC52-06NA25396.

Reconfigurable Middleware Architectures for Large Scale Sensor Networks

by

Sean M. Brennan

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctorate of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2009

Reconfigurable Middleware Architectures for Large Scale Sensor Networks

by

Sean M. Brennan

B.A., University of Iowa, 1994

M.S., University of New Mexico, 2003

Ph.D., Computer Science, University of New Mexico, 2009

Abstract

Wireless sensor networks, in an effort to be energy efficient, typically lack the high-level abstractions of advanced programming languages. Though strong, the dichotomy between these two paradigms can be overcome.

The SENSIX software framework, described in this dissertation, uniquely integrates constraint-dominated wireless sensor networks with the flexibility of object-oriented programming models, without violating the principles of either. Though these two computing paradigms are contradictory in many ways, SENSIX bridges them to yield a dynamic middleware abstraction unifying low-level resource-aware task reconfiguration and high-level object recomposition.

Through the layered approach of SENSIX, the software developer creates a domain-specific sensing architecture by defining a customized task specification and utilizing object inheritance. In addition, SENSIX performs better at large scales (on the order of 1000 nodes or more) than other sensor network middleware which do not include such unified facilities for vertical integration.

Contents

List of Figures	xii
List of Tables	xiv
1 Integrating Wireless Sensor Networks into Large Systems	1
1.1 Wireless Sensor Networks	2
1.2 Distributed Computing through Object-Oriented Middleware	6
1.3 SENSIX by example	7
1.4 Hypothesis	11
1.5 Overview	12
2 Background	13
2.1 Wireless Sensor Networks	13
2.1.1 Example deployments	14
2.1.2 Unique issues	18

Contents

2.1.3	Run-time and operating systems	19
2.2	Object-oriented programming and design	21
2.3	Middleware	23
2.3.1	Distributed computing	23
2.3.2	Middleware in WSNs	28
3	The SENSIX Framework	33
3.1	Building a SENSIX architecture	35
3.2	The SENSIX run-time	38
4	The Reference Implementation	43
4.1	Object-Oriented Heterogeneous Sensing	44
4.2	Sensing and Tasking Model	47
4.3	Sensing Mini-language	50
5	Metrics, Experiments and Results	55
5.1	Middleware Emulation and Simulation	55
5.2	Tasking Case Studies	58
5.3	Task Complexity Metric	60
5.4	Messaging and Energy	62
5.5	Commodity tradeoff	67

Contents

6	Conclusions and Extensions to SENSIX	73
6.1	Unfinished aspects of SENSIX	74
6.2	Extensions to SENSIX	75
6.3	Autonomy for Ubiquitous Systems	76
	Appendices	80
A1	SENSIX IDL	80
A2	Sensing Mini-Language	85
A3	Generic Sensing IDL and Discovery IDL	87
	References	98

List of Figures

1.1	Wireless sensing in a healthcare context.	9
1.2	SENSIX as a bridge in the context of healthcare.	10
2.1	Basic CORBA architecture.	25
3.1	Wireless sensing in an ambulatory healthcare context.	34
3.2	Generic SENSIX build process.	36
3.3	SENSIX build process for an HL7-compliant system.	38
3.4	SENSIX layers.	39
3.5	SENSIX data return efficiency.	40
3.6	SENSIX Request object interactions.	41
4.1	An example supersystem deployment (and visualization output of the SENSIX deployment tool).	45
4.2	The SENSIX data fusion/network tasking cycle.	49
4.3	SENSIX Request/Response objects implementing the data fusion/net- work tasking cycle.	51

List of Figures

4.4	A UML Sequence diagram of the SENSIX run-time under the reference implementation.	52
5.1	Middleware task complexity in bytes: source code (top) and bytecode (bottom).	61
5.2	Compiled middleware engine size and mote capacity.	62
5.3	Task messaging average per node.	64
5.4	Task messaging average per node – closeup on SENSIX and Maté.	65
5.5	Middleware messaging over increasing scale (including CORBA).	68
5.6	CORBA influence on performance.	69

List of Tables

4.1	A comparison of data fusion models	48
5.1	Mote energy consumption in milliwatts	63
5.2	Percentage of allowable CORBA network infiltration given different middleware benchmarks.	70
A2.1	Task and aggregation domain-specific language	86

Chapter 1

Integrating Wireless Sensor Networks into Large Systems

This dissertation describes and analyzes a middleware approach that efficiently integrates wireless sensor networks with large distributed object systems, with a focus on scalability. Middleware is software that connects distributed components across different systems for the sake of interoperability, often masking the details of the requisite interactions. Whereas previous wireless sensor network software efforts created custom, ad hoc, and typically direct-to-the-user interfaces to the sensor network, this middleware approach, embodied in the SENSIX open-source implementation*, considers the sensor network as an extension of a larger system and supports a domain, language and operating system independent means of programming and interacting with the sensor network with an emphasis on seamless vertical integration. SENSIX unifies the low-level, event-driven, data-centric paradigm of wireless sensor networks and the high-level abstraction of object-oriented systems, without violating the principles of either. This dichotomy is the core issue that SENSIX addresses.

*<http://sensix.sourceforge.net>

Once SENSIX is programmed for a particular sensory domain, the run-time offers dynamic addition of nodes, functional reconfiguration and presents the sensors as software objects to other components beyond the tightly resource-constrained sensor network. Furthermore it achieves all this more efficiently in terms of both messaging and memory footprint than several leading wireless sensor network middleware implementations and, when balanced, is shown to be far more scalable in very large networks.

This introductory chapter reviews the unique challenges of sensor networks and how they compare with other computing paradigms. It then summarizes the SENSIX approach in the light of a concrete example before moving on to later chapters covering the state-of-the-art in wireless sensor networks and middleware, the SENSIX architecture, a reference implementation and an experimental evaluation of its performance.

1.1 Wireless Sensor Networks

Wireless Sensor Networks (WSNs) are very small form factor computing platforms, wirelessly inter-networked, sensing some surrounding environment in-situ. These cubic centimeter-scale (or smaller) nodes, called *motes*, differ from previous sensing architectures in three major ways: (a) they are self-powered, (b) they communicate wirelessly (usually through radio frequencies) and (c) they have some computation capability. The primary and most daunting constraint these networks face is severely limited power, and as such the minimization of energy consumption drives further constraints such as computation, communication and storage limitations. The wireless network is usually constrained to the very limited transmission range of ISM-band and 802.15.4 radios, although significantly wider ranges are available through WiFi or, more rarely, cellular networks. Messaging may be reduced by in-network

data aggregation. Exfiltration of important data out of the network is usually accomplished through multihop routing. It cannot be stressed enough that the most vexing hardware limitation of all in WSNs is that of the power supply, because this limitation touches every other part of the system: what and how well it can detect, how far it can communicate, how much it should compute, and how much data it may store. Environmental energy harvesting, wherein mote operational power is extracted from the surroundings (solar, vibration, etc.), can yield continuous power, but, as reflected in [54], at such a low rate that WSNs are still heavily power-constrained.

The term wireless sensor network has been used to include hardware nodes much larger than the mote as well. Herein, unless otherwise noted, WSN refers to mote-based sensor networks.

Power limitations are not the only concern in sensor networks; scalability is also prominent. Initially wireless sensor networks were seen as enabling a ‘real-world’ Internet: hundreds, thousands, even millions of these minuscule detecting/processing/communicating platforms cooperating and coordinating to get live data to end-users on the other side of the globe. WSNs have been cited as prime solutions for a number of potential and actual applications including ecological and environmental studies, human health observation, defense and nonproliferation surveillance, transportation traffic monitoring or manufacturing process tracking. Small-scale networks have been deployed for habitat monitoring [76, 107], volcanic seismology [112], agricultural micro-climate observations [66] and petroleum facility management [55]. A larger system (approximately 1200 nodes) has demonstrated perimeter surveillance [7, 10]. Despite these examples, long-term and/or large-scale systems in unattended deployments have not yet been demonstrated.

The wireless sensor network computing paradigm is data-centric and event-driven, meaning that the intermittent arrival of event data dictates the behavior of the sensing application. Due to the resource constraints, and contrary to traditional

client-server based distributed computing, WSN computation is highly asynchronous and communication is tightly controlled. Sensor network software must be robust and reduce costly and invasive user administration by being self-configuring. WSN messaging and communications must also be minimized in motes because wireless transmission is the single most costly activity in terms of energy. Sensory data character, plus the quality of a particular datum, will influence message priority and aggregation capability, both involved in reducing message volume. The natural spatial and temporal localization of events in the environment also gives network traffic a bursty nature, further emphasizing the need for bandwidth minimization. Violation of any of these data-centric principles degrades efficiency and, therefore, network lifespan.

Early WSNs were organized in a star topology. The current drive is towards greater multihop and ad hoc (self-organizing) networking. Typical fielded topologies are either single-tiered and fully ad hoc with under a hundred small sensor nodes multihopping data to a controlling laptop, or a two-tier system wherein clusters of small motes report via short multihop paths to more powerful gateway nodes which themselves may form their own multihop network of greater range to report data to a controlling laptop or exfiltration point. The 1200-node ExScal experiment [7] is an example of such a two-tiered topology.

With sensor networks acting as a cornerstone of the vision of ubiquitous computing (sensors and computational resources everywhere), WSNs have been optimistically projected to encompass many thousands of nodes *per user* [30]. This vision has yet to be realized. While part of the reason may have to do with the opportunity (or lack thereof) to assemble so many nodes into a network, the staggering effort required to complete ExScal implies that there is more to sensor network scaling than just funding issues. Wireless sensor network research, by and large, sees these networks as a stand-alone end unto themselves. At least among WSN hardware vendors, this

attitude has begun to change. WSNs are probably best utilized as the sensory leaf nodes of a much larger and potentially more complex system tree. Given the volume of data that a very large sensor network (or network of sensor networks) would generate, the most appropriate consumer to WSN data production is in fact another data system – not a human user. WSNs are inherently cyber-physical systems, and this meeting of physical and computing elements demands a different mode of thinking.

Wireless sensor networks with their constraints and capabilities are very different from other distributed computing paradigms. Mobile, ad hoc, socially-oriented computing devices, as embodied by the ubiquitous cell phone, though *somewhat* similar in their wireless networking and constraint bounds, are highly user interactive – eliminating the need for self-administration and mitigating energy constraints. Likewise, sensor networks, though highly parallel, are not similar enough to high performance computing systems – we cannot directly use a Message Passing Interface (MPI) programming library here. Although WSNs pass messages throughout the network, MPI (or for that matter the abstract models of PRAM, BSP, or LogP) is inappropriate because it assumes a homogeneous, low-latency, topologically static and, moreover, reliable network. Without an intervening layer to meet or mitigate those assumptions, MPI cannot be applied to sensor networks.

Nor do distributed object-oriented (OO) middleware approaches directly fit in such a constrained environment. Gone are the MPI requirements of homogeneity and static network topology, but for flexibility and dynamism, these OO systems ship inefficient objects across the network for remote operations – a very bad idea in an efficiency-sensitive wireless sensor network. OO systems also tend to have a larger memory footprint, which is why simpler procedural languages, like C, dominate embedded programming. Ultimately, like MPI, OO embodies high-level abstractions which are desirable for the power of their problem-space modeling, but at a cost to efficiency. Distributing objects remotely is, for efficiency reasons, arguably a

poor choice even in low-constraint environments, but given the prevalence of the object-oriented paradigm and the complexity of OO middleware implementations, this fundamental inefficiency is beyond the scope of this work.

1.2 Distributed Computing through Object-Oriented Middleware

The OO approach embeds behavior (as methods or function calls) in self-contained data structures called objects. These objects are meant to mock real-world ideas and forms, and interact by passing messages (through their methods) to each other, in contrast to direct manipulation embodied in the procedural approach. This encapsulation, governed by method interfaces, enables inheritance and polymorphism, which means that hierarchies of related objects (from general to specific) can be created. For instance, a ‘Cat’ class of object can derive much of its non-cat-specific functionality from a ‘Mammal’ class, from which ‘Dog’ also inherits. In a distributed environment, this message passing is taken to its logical conclusion, passing object messages through the network, instead of just within a single application process. This distributed middleware hides much, such as data marshalling and network arbitration among heterogeneous participants. Although this high-level abstraction yields greater power, particularly for modeling, it comes at a cost of greater internal complexity and more overhead – in both computation and communication.

The Common Object Request Broker Architecture (CORBA) is an object-oriented middleware that is valuable for its platform and language heterogeneity through the abstract Interface Definition Language (IDL), its runtime discovery (through dynamic invocation), its asynchronous messaging (distributed callbacks) and its existing high-level services (such as Security, Event and Trader services), but most of

all for its standardization. In fact, CORBA is *the* standard for distributed object computing.

CORBA is prevalent in, and has standardized services for, the domains of finance, healthcare, defense, manufacturing, telecommunications, and transportation. CORBA is in your last phone call, debit card transaction and in your Linux Gnome desktop. As such CORBA is commodity software. To qualify as commodity software, a product must be so widely used that it is itself a standard – in practice, if not in principle. Commodity is not an expressly positive denotation. On the advantageous side, commodity means software with high availability through several competing vendors, is extensively used and tested, has a familiar development environment with a shallow learning curve and is highly interoperable. Unfortunately, trying to be very general-purpose also means that it may be too broad a solution, be heavyweight (having excessive overhead) and be difficult to optimize. CORBA is all this.

1.3 SENSIX by example

An example to show the SENSIX approach in context is in the domain of healthcare. The CodeBlue project at Harvard has developed prototype notes that collect blood-oxygenation, electrocardiogram, electromyogram, or blood pressure data, as well as the software to serve these data on a PDA [74]. The role of CodeBlue is as a tool for emergency responders in disaster relief efforts. Such wireless monitoring might also be advantageous in a hospital setting, and almost certainly would be so in a nursing home or in telecare (where the patient, living at home, is monitored 24/7 through wearable devices).

The healthcare industry has been slow to go paper-less – and for good reason:

security and trust are vital concerns surrounding sensitive patient information. When and where it has gone electronic, healthcare had difficulty leveraging the promised potential due to software incompatibilities. This difficulty gave rise to Health Level Seven (HL7), a nonprofit, all-volunteer standards organization that now has global impact. The purpose of HL7 is to unify software interfaces for healthcare such that, for example, a patient could go to a new medical provider and have their entire record history available, including test results, greatly increasing the quality and efficiency of care.

HL7 standards are in turn based on middleware standards such as CORBA and the Service-Oriented Architecture (SOA). Chapter 3 addresses SOA; for now, this example focuses on CORBA.

Figure 1.1 represents an example CORBA/HL7 compliant system interfacing with wireless vital sign monitoring. The service components in rounded boxes here are either existing or upcoming CORBA standards. These standards provide object-oriented interfaces for clinical data collection, decision tools, integrated billing facilities, and even interfaces to national databases. The wireless body-area network which is monitoring the patient must interface with this coherent, standardized system, either locally, or remotely through telecare. The wireless network interacts with the Person Identification Service to securely associate sensor data with a particular patient medical record, and the Clinical Observations Access Service which collects the vital sign data. Figure 1.2 provides a detailed view of the boundary between the object-oriented CORBA/HL7 system and the body-area sensor network running the efficient TinyOS run-time with CodeBlue components. The SENSIX framework bridges this boundary, which is not only a software interface but also a divide between software paradigms.

Recent developments in the CORBA standard such as CORBA/e, a pared-down specification that targets embedded platforms [81], a mobile wireless InterORB Pro-

1.3. SENSIX BY EXAMPLE

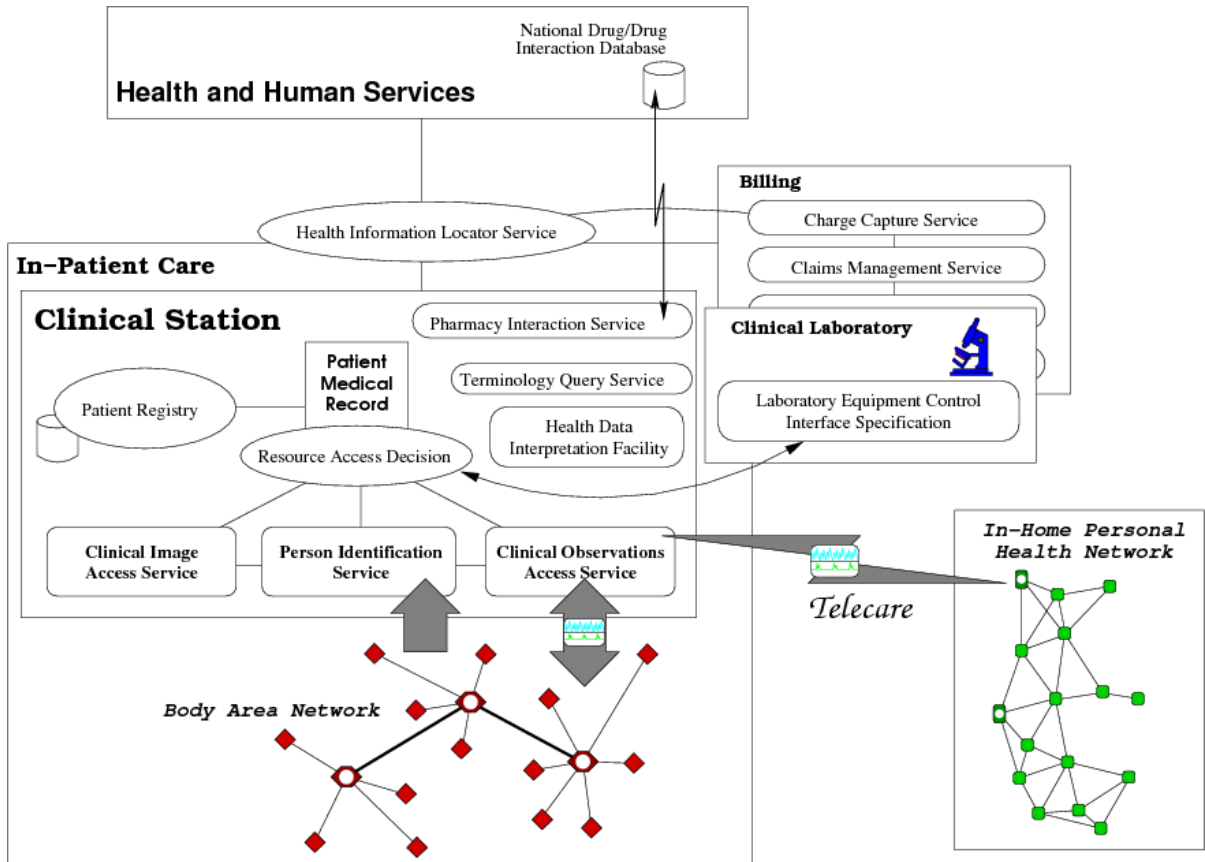


Figure 1.1: Wireless sensing in a healthcare context.

tolcol (IOP) [80] and a real-time specification [79], make CORBA more appropriate for certain wireless devices. However, these advances are still incompatible with WSNs: even the slimmest CORBA/e footprint is still around one megabyte (too big for motes), and the wireless IOP is not tuned for WSN-style efficiency. Moreover, the mere fact that CORBA maintains object state over the wire violates the efficiency principles of WSNs, so a direct graft of CORBA onto mote networks will not work well.

There are a number of conflicting goals when attempting to accomplish a total clinical observation system with wearable devices as illustrated in Figure 1.1. First and foremost is standards compliance, which ultimately means interoperability with

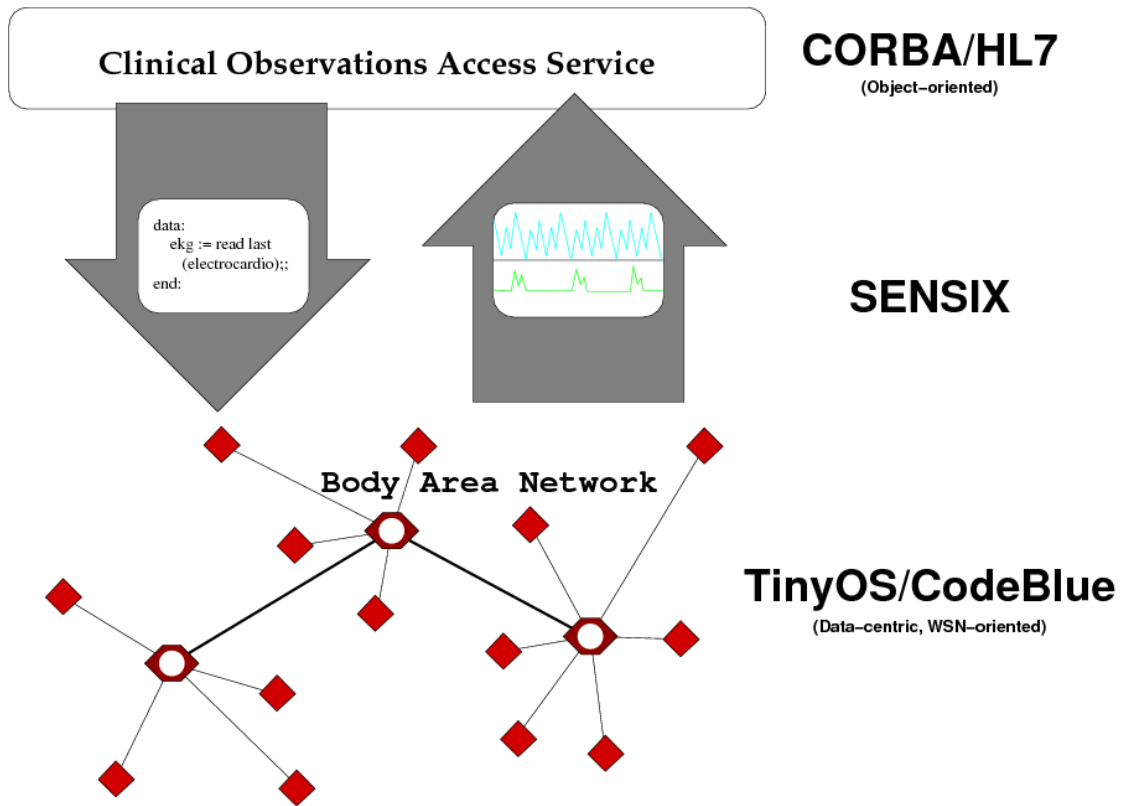


Figure 1.2: SENSIX as a bridge in the context of healthcare.

CORBA. Second is typical software engineering practices, with a toolset that is dominated by object-oriented methods and approaches. Third is software flexibility: the ability to reconfigure, recombine, extend and expand modular objects to achieve new functionality. Lastly comes the counter-point of the event-driven data-centricity of WSNs and their inherent incompatibility with the distributed object paradigm. Any solution to this incompatibility must honor the important aspects of each side.

The SENSIX framework provides just such a solution. Leveraging multiple layers to achieve controlled ad hoc networking (Access layer), reconfigurable and parametric sensing (Functor layer), domain-specific functionality (Morphic layer) and an object-oriented presentation (Transformative layer), SENSIX bridges the data-centric and object-oriented approaches independent of the operating system, the implementation

language or the sensing domain. We can apply SENSIX equally to complex defense systems or wide-area, extended-term ecological monitoring.

1.4 Hypothesis

The hypothesis for this dissertation is:

The SENSIX approach scalably bridges commodity object-oriented middleware and event-driven, data-centric sensorware without the loss of the dynamic flexibility of the former or the constraint-sensitive efficiency of the latter.

To show this, I qualitatively examine the flexibility of the object-oriented side of SENSIX and quantitatively evaluate the efficiency of the mote side.

The primary contributions of SENSIX and this dissertation are:

- a bridging facility between mote software and object-oriented software systems,
- a means of defining WSN applications independent of programming language, network protocol, or mote infrastructure/operating system,
- a reference implementation of a tasking mini-language to guide other domain-specific efforts and
- an efficient means of achieving very large scale sensing networks.

While SENSIX achieves its bridging capability through its multiple layers, its language, protocol and infrastructure independence is largely a by-product of the mote IDL compiler and Transformative layer. Scalability, when the network is properly balanced, comes about mostly due to efficient implementation of the Functor layer.

1.5 Overview

- **Chapter 2** reviews recent WSN research, CORBA middleware and tradeoffs, and work in WSN reconfigurable middleware.
- **Chapter 3** delves into the details of the SENSIX software architecture as a bridging facility and returns to the healthcare example to show how it is applied.
- **Chapter 4** discusses a reference implementation that embodies generic, mote-compatible sensing and explores the tradeoffs involved in using SENSIX throughout a large heterogeneous sensory system that uses far more than just mote-scale platforms.
- **Chapter 5** proves the hypothesis of this dissertation by using this implementation in analytical and empirical efficiency measurements, comparing against representative WSN middleware implementations while addressing SENSIX dynamism and flexibility.
- **Chapter 6** explores incomplete aspects of SENSIX, improvements and future directions.
- **The Appendices** contain SENSIX IDL code, an encoding table for the reference implementation mini-language and the IDL for the reference implementation.

Full source code for SENSIX is available on SourceForge at <http://sensix.sourceforge.net>.

Chapter 2

Background

To make the dichotomy between the wireless sensor network and object-oriented computing paradigms apparent, this chapter reviews the issues of WSNs and OO programming, as well as middleware approaches for both, with particular attention on object-oriented CORBA.

2.1 Wireless Sensor Networks

Wireless sensor networks are a relatively new field. Spawned by the Defense Advanced Research Projects Agency (DARPA) in 1991, the Network Embedded Software Technology (NEST) program (beginning with the Smart Dust project) aimed at producing millimeter-scale general-purpose sensor platforms capable of full situational monitoring. Throughout the lifetime of this initiative, the University of California (UC) has been at the forefront, with UC Berkeley leading in systems engineering - so much so that the MICA platform and TinyOS software [49] that they developed came to dominate the field. This field of research has now moved into an era of industrial development, sprouting numerous start-up companies. Though the

initial projects are now completed and there is now a wide variety of university and industry innovation, the earliest UC work continues to frame the discourse in this field. One of the first papers from Estrin et al., detailed their data-centric, application specific and localized approach to these systems, demonstrated through directed diffusion routing [30]. This data-centrism has become an underlying assumption in WSN development. Several years later, it is again UC that articulates the developing vision of wireless sensor networks. These networks are to be untethered, unattended (perhaps even unreachable), very large scale in coverage, either mobile or exhibiting “stationary mobility” due to dynamic environmental variability (i.e. mobile obstacles to sensing), autonomous, localized and dynamic (i.e. rapidly switching from low activity to high activity upon event detection) [29].

2.1.1 Example deployments

One of the distinguishing characteristics of WSN systems research is the fact that, from early on, experimental sensor network deployments were multidisciplinary – combining WSN system development with productive sensing in some other science domain. The data concerns of these other domains have driven WSN development. These selected experiments exemplify aspects of sensor network energy-efficiency and data-centricity, issues in WSN software robustness and scalability, and the opaque nature of these non-interactive systems.

Storm Petrels

In this first example project, a UC Berkeley team deployed about 150 motes monitoring temperature, humidity, burrow occupancy (ambient light) and barometric pressure to reveal micro-climate details of Storm Petrel nests in a nature preserve on Great Duck Island, Maine [104, 105]. Operating over 123 days, these nodes, run-

2.1. WIRELESS SENSOR NETWORKS

ning TinyOS (see Section 2.1.3), were divided into one single-hop network and one multihop network.

This project used the Crossbow MICA2DOT, a round, slightly smaller repackaging of the popular MICA2 mote. Both are single board computers with an 8-bit 4 MHz microcontroller, a 433 MHz radio offering up to 40 Kbps, a 512 KB flash memory and an on-board Analog-to-Digital Converter (ADC). On separate boards, usually linked to the main board by a 51-pin connector, are a variety of low-power sensors. The MICA2 is the same length and width as the pair of AA batteries it uses, while the MICA2DOT is 25 mm in diameter and uses a 3V coin cell battery.

The main sensor network systems insight revealed by this project was that while duty-cycle has a strong effect on network lifetime, multihopping drastically reduces lifetime (by about half). Also a variety of fault modes were seen here: in sensor readings, networking, and total node failure.

California Redwoods

This next project looked at micro-climatic effects in the boughs of a giant redwood tree in Northern California. Again MICA2DOTs were deployed, this time in special packaging, to measure temperature, humidity, and incident and reflected light. The motes were running TinyOS, and, instead of a custom binary application, the TinyDB reconfigurable middleware (see Section 2.3.2). This motes used multihop routing, facilitated by the MintRoute component of TinyOS, to a base-station at the bottom of the tree.

The UC Berkeley team found that “battery failure was correlated with most of the outliers in the data” [107]. Mote sensors were calibrated before deployment, so when unexpected data noise turned up, the team was able to isolate the issue as due to physical node orientation.

Volcán Reventador

This three week deployment of 16 motes in Ecuador, studied earthquake wave propagation and volcanic source mechanisms. This project's goals were vastly different from those of the preceding projects. Specifically, this volcanology study required high data-fidelity and throughput over a widely separated network in a much more event-driven application [113]. These motes also ran TinyOS, but were TMote Sky units (predecessor of the TelosB), similar to the MICA2, but with less available memory, a 16-bit processor and an additional ADC board.

Geophones and microphones measured seismic and acoustic phenomena. The software used circular buffer logging to overcome the low radio data rate, since events were sparse but intense. To avoid false detections, the motes voted on event occurrence and, upon a successful election, the local logging laptop would instigate a data pull from the motes. Sensors were deactivated during this data pull, and so data on sequential events was lost. The logging laptop represented a single point of failure, and, as such, was responsible for several days of lost data when its batteries drained.

As opposed to the more gradual and easier to capture events of the previous deployments, this project pushed the boundaries of unattended, low-power sensing, demonstrating the need for greater emphasis on total systems research, particularly in vertical integration.

LOFAR-agro

This deployment of 150 motes over three months sought to monitor micro-climate conditions in a potato crop to detect and isolate the occurrence of a fungal disease [66]. This project used TNode motes, which are very similar to MICA2DOTs, running TinyOS with MintRoute and the T-MAC Medium Access Control layer pro-

to col. It also used Deluge, a wireless reprogramming facility which transmits the application binary image, flashes it into memory and reboots into the new image.

This project failed to gain sufficient data to achieve its crop monitoring goal. As a result and unlike other deployment studies, this paper emphasizes the software engineering issues involved in deploying WSNs, isolating key weaknesses:

- need for development formality
- insufficiency of the existing Application Programming Interfaces (APIs)
- lack of software robustness
- pre-deployment testing
- lack of fine-grained, component-level diagnostic tools
- super-linear increase in complexity with a linear increase in scale
- battery exhaustion

Except for battery exhaustion, these issues are well known and studied distributed computing and software engineering problems. The unique constraints brought on by requirements for low-power only make these already hard problems that much harder.

In the rush to solve the resource-constraint issues of WSNs, it is vital *not* to ignore the complexity of distributed computing or the pitfalls of software engineering, nor to discard the partial solutions we already have for them.

ExScal

Our final example, a series of scalability experiments led by Ohio State University (OSU), still under the auspices of NEST, uncovers additional issues. The first exper-

iment was a deployment of just more than 90 MICA2s for intrusion detection, and one of the key conclusions of this study was “that communication in dense sensor networks is significantly unreliable” [6].

This project then grew significantly in scale: 983 MICA2-based eXtreme Scale Motes (XSMs) equipped with infrared, magnetic and acoustic sensors, plus 203 Crossbow Stargates, equipped with GPS. The Stargate is a traditional single board computer running Linux with a 400 MHz Intel Xscale processor, 64 MB of memory, and Ethernet, Serial, USB, and PCMCIA interfaces. These were arranged in a five-row deep planned perimeter topology, tasked with detecting and classifying intrusions (human or vehicle).

This experiment found increased messaging reliability due to multi-tier network heterogeneity, versus their previous homogeneous experiment. The authors also claim overall reliability of 73%, including routing yield and hardware, software and localization faults. However, this deployment used in-network Sensor Network Management System (SNMS) querying to provide ground truth – which admittedly only covers “about half of the deployed network” [10].

These OSU experiments emphasize the opacity of WSNs and the importance of network autonomy. Here, much more attention was given to vertical integration issues, but there was no apparent attempt to use standard tools beyond those already prevalent in WSNs.

2.1.2 Unique issues

As these example deployments demonstrate, in-situ wireless sensor network computing is drastically different from programming for business applications. The uniqueness of WSNs revolve around their special emphasis on “energy efficiency, robustness, and scalability” [90]. The focus on energy efficiency is seen most strongly in

the WSN principle of data-centrism, particularly in networking. The data-centric principle means what it says: data is central, everything else is overhead. Deployment, network layers and applications are all driven by the data. Since the data is associated with events: quick, intense and bursty as in volcanology, or slow, gradual and leisurely as in micro-climatology, WSNs are also event-driven. Energy-efficient scalability is also prominent in the wireless sensor networking literature, yet software scalability is hardly addressed. Equally, a strong emphasis on overall robustness is evident in the form of networks that are self-organizing and self-healing, but little or no application of this in the sensor software development process.

Thus, while sensor network software is very adept at energy efficiency, it tends to disappoint in terms of scalability and particularly robustness, again leading back to distributed computing and software engineering concerns. The focus of the work described in this dissertation is to unify these issues to achieve robust software scalability efficiently, using common tools as appropriate and wherever possible.

For WSNs, these tools are in the form of embedded operating systems and specialized programming environments.

2.1.3 Run-time and operating systems

Addressing these issues starts at the level of the mote operating system (OS). At this level, the first choices are made in the tradeoff between low-level efficiency and high-level power and convenience. The following exemplar operating systems exhibit incremental advancement in features, carefully choosing and justifying the tradeoffs.

TinyOS

TinyOS, open-source software from UC Berkeley, is not an operating system per se, but rather a collection of components “wired” together, that upon compilation provide a feature-full run-time environment. TinyOS is event-driven; specifically it uses a split-phase programming model, which is closer to the action of hardware interrupts: the scheduler can interrupt a task, switch to handlers for sensor data or incoming packets, and return to the long-running task. It is optimized for code plus data size, is statically linked, and all memory is preallocated – there is no heap. TinyOS’s companion language, nesC, serves as the means to tie those components together into an application, and to create new components. NesC also performs concurrency analysis at compile time to avoid race conditions [49].

TinyOS cannot, however, dynamically load a new executable without a complete image replacement and reboot, and thus lacks fine-grained flexibility. NesC’s programming model, though based on C, is sufficiently different that it has a notoriously steep learning curve (see [66, 77]). Nonetheless, it is the de facto standard tool for WSN programming, with its componentization being a key feature.

SOS

SOS improves on TinyOS by providing dynamic memory allocation, loadable modules and a kernel. Application programming is in C, a much more common and familiar language. The kernel takes care of network messaging, dynamic memory allocation, and module loading and unloading [43]. For over-the-network reprogramming, TinyOS requires that the entire mote image be shipped and the mote reboot, whereas SOS enables the use of smaller modules that can be dynamically loaded. SOS is still event-oriented like TinyOS, but in terms of network reprogramming efficiency, SOS wins because of the finer granularity of its modules. However, SOS does

not emphasize software component recomposition like TinyOS does.

MANTIS

MANTIS introduces preemptive thread scheduling. A preemptive scheduler is particularly important to enforce fairness among threads or tasks [14]. MANTIS also provides multi-granular code injection: at the level of the whole system, a single thread, or even thread variables. This provides dynamic and efficient run-time reprogrammability, even after a network has been deployed. MANTIS offers POSIX-style system calls in C and a remote interactive shell. Of these three, MANTIS is the most familiar to the UNIX programmer, offering the most high-level OS services.

There are many more operating systems that are WSN-specific and offer additional features, like Nano-RK [31], Contiki [28] and eCOS [70], and still others that are merely real-time embedded-platform-specific, such as QNX [47] and VxWorks.

2.2 Object-oriented programming and design

In stark contrast to the low-level attention that WSNs require is the high-level abstractions of object-orientation. The OO approach has so thoroughly saturated software engineering that it has become a commodity technology – a de facto standard.

Object-oriented programming is an imperative (versus declarative) style in which the developer defines abstract data types that encapsulate state and behavior. Though there are numerous OO languages (C++, Java, Python, Ruby, C#, Eiffel, Smalltalk and more), they share some features beyond this information-hiding and behavioral interfaces – such as object inheritance, functional polymorphism, late-bound dynamic dispatch and open recursion (a ‘self’ or ‘this’ pointer). Run-time objects are

self-contained, interacting only through their methods, or function calls. This approach encourages fine-grained modularity, ‘real-world’ modeling and strong typing. Inheritance, whereby an object gains the functionality of an ancestor, and polymorphism, where a named method may have object-specific functionality, enable both static (compile-time) and dynamic (run-time) software reuse, which itself leads to greater extensibility, reliability and maintainability.

Although any Turing-complete language can accomplish these same features (since all Turing-complete languages are mathematically equivalent), the focus of object-oriented programming is to make these desirable outcomes easier to realize. There is a price to be paid, however. Dynamic dispatch, which enables polymorphism and open recursion, means that a method call must, at run-time, route through a lookup table to use the correct executable code. As a result of this complexity, OO run-time libraries (or interpreters) tend to be quite large – often far too much to be used in embedded systems.

From the point of view of application design, OO is a bottom-up strategy (reducing large problems into smaller, more manageable components) as opposed to top-down (detailed stepwise instruction). The use of object-oriented programming should encourage high module cohesion and low module coupling, increasing reuse. Object-oriented software engineering tools and processes abound for most OO languages and these concepts dominate the field of software engineering:

- use cases
- Unified Modeling Language (UML) diagrams
- Class-Responsibility Collaboration (CRC) cards
- design patterns
- design by contract

- unit testing frameworks
- self-documentation
- coupling, cohesion and complexity metrics
- Rational Unified Process (RUP)/eXtreme Programming (XP)/Scrum

2.3 Middleware

Middleware is supportive software that hides direct operating system and network software interfaces from an application. The purpose of this masking may be for mere simplification, portability, or, most often, for a unified distributed computing environment across diverse hardware, network, and/or software platforms. This section looks at both traditional distributed computing middleware and that which is specifically for WSNs.

2.3.1 Distributed computing

Distributed computing has its own constraints. Waldo et al. note that distributed systems must (and often don't) take into account: latency, memory access, concurrency and partial failure [111].

One of the earliest and certainly most successful distributed computing middleware implementations is the Transmission Control Protocol and Internet Protocol (TCP/IP) networking stack, particularly as embodied through Berkeley sockets. This bedrock of the Internet was so successful that it is now embedded in the kernel of most operating systems, and is the assumed base of the remaining middleware discussed in this subsection.

The Open Software Foundation's Distributed Computing Environment (DCE) is a far more complete programming environment, providing IDL defined Remote Procedure Call (RPC) service, POSIX threads, a Naming Service, Kerberos security, the Distributed File System and the Distributed Time Service [26]. Originally supporting just C, it expanded to support object-orientation and C++. Microsoft's Distributed Component Object Model (DCOM) is an extension to the Component Object Model (COM). COM builds on the (non-OO) DCE RPC with a layer that enables object-orientation. CORBA is an object-oriented standard for distributed computing from the Object Management Group (OMG), a large consortium of software vendors. CORBA uses an ORB (Object Request Broker), as opposed to RPC, to provide location transparency for distributed objects. Object-oriented DCE, DCOM and CORBA were all concurrent competing technologies in the 1990s.

Since that time, Microsoft has switched to .NET which supports both binary and XML data, and more web-oriented approaches, such as XML-RPC, SOAP and others, offer eXtensible Markup Language (XML) based data exchange, collectively known as SOA. Java supports both CORBA and the less complex Remote Method Invocation (RMI).

The Internet Communications Engine (ICE) from ZeroC is a new, non-standard incarnation of CORBA. It offers mappings for C++, Java, C#/.NET, Python, PHP, Objective-C and Ruby (all of which exist for CORBA as well). Its overall architecture is almost identical to CORBA (although the names of components have been changed).

CORBA in depth

Since SENSIX is implemented with CORBA, this subsection examines this architecture in greater detail. Figure 2.1 demonstrates both the client/server basis of

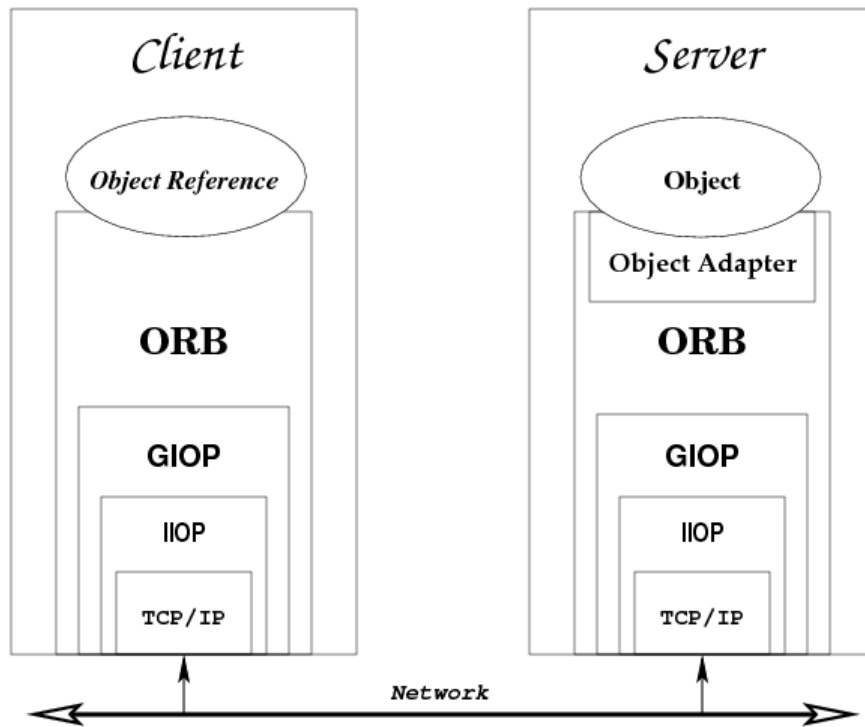


Figure 2.1: Basic CORBA architecture.

CORBA, and the layering of CORBA which allows for individual components to be swapped out. For instance, The Internet Inter-ORB Protocol (IIOP) which uses TCP/IP can be replaced by DIOP (Datagram IOP) to use UDP, or SSLIOP for access to the Secure Sockets Layer (SSL), or LW-IOP (for lightweight) which is optimized for wireless communications. The abstract GIOP (General IOP) gives the framework for these specific implementations, consisting of the Common Data Representation (CDR), which maps data types to on-the-wire representation, Interoperable Object References (IORs), essentially non-local object pointers and message formats (Request, Reply, etc.) for managing object requests and so forth.

The Object Adapter (again an abstract facility, usually implemented by the Persistent Object Adapter) serves as a generic interface between the ORB and served objects. It registers object implementations, generate IORs, maps the reference to

the implementation, invokes methods and activates or deactivates the object [96].

In addition to basic remote object access, OMG has also defined other standards, categorized as CORBA services, and both horizontal and vertical CORBA facilities. CORBA services serve bootstrapping needs such as locating objects (Naming Service, Trader Service) and security. Horizontal CORBA facilities are widely usable modules such as user interfaces, system management and task management, while vertical CORBA facilities are industry specific (telecommunications, manufacturing, finance, etc.) [96]. OMG has also produced standard subsets of the full CORBA specification in the form of CORBA/e for embedded platforms, with both compact and micro profiles [81], which perform much better than full CORBA [36] at the expense of less functionality.

The reasoning for basing the SENSIX framework on CORBA is two-fold. First, CORBA is a commodity technology. Commodity means open standards, multiple vendors and wide acceptance. TCP/IP is the ultimate software commodity. Commodity software yields competitive cost, low differentiation in implementations and, thus, high interoperability, high reliability (to stay competitive), familiar development skills and knowledge and an existing, well-supported infrastructure. Compared to ‘boutique’ software which results in high-priced vendor lock-in with a high potential cost for failure and requiring specialized developer knowledge, commodity computing is by far preferable. CORBA, as the only mature and complete standard for distributed object-oriented computing, is distinctly qualified as commodity. Second, CORBA out-performs its direct (i.e. platform independent) competitors. SOAP and XML-RPC have significantly higher network bandwidth consumption than CORBA [38, 83]. While Java RMI and CORBA have similar network performance [38], CORBA scales better in the face of increased load than RMI [56].

CORBA is dead

CORBA is periodically declared obsolete, most recently by Michi Henning, co-author of “Advanced CORBA Programming with C++” and currently chief scientist of ZeroC. In a series of articles across several publications and online forums, Henning criticizes CORBA’s technical flaws and particularly the design-by-committee process [46].

CORBA usage and development is very active, particularly in telecommunications, manufacturing, government and healthcare. Because it is a mature standard, much of this work is hidden. It’s not new, there’s no hype, CORBA just works.

CORBA is put to work for diverse customers such as travelocity, weather.com and the Keck Observatory. Example deployments of CORBA also include wood product manufacturing, food packaging, fueling systems, traffic management, steel manufacturing, shipping, visual effects creation and bottling beer [100].

Which is not to say that CORBA does not have true problems, the most predominant of which are a direct consequence of a standards process run by competing vendors. The CORBA standard tends to be under-specified, and, at times, incoherent due to committee in-fighting, leading to subsequent problems with vendor implementation interoperability. Another often-raised issue is that CORBA servers are unreachable behind firewalls. This is a red-herring: there are IIOP-proxies that can forward information through port 80, as well as CORBA-aware firewalls. These problems and more are also being mitigated by ongoing research and development ([3, 8, 19, 37, 58, 61, 84, 93, 94, 108, 110] represent a recent sampling) and an active standards process.

There is, however, the core issue of potentially invalid across-the-wire object pointers (IORs) and high-overhead state transmission. This is not necessarily a fundamental and inherent issue for CORBA, as evidenced by recent specification trends.

CORBA is, ever-so-slowly, moving toward greater applicability in the mobile and embedded market. CORBA/e, LW-IOP, and Lightweight Services (Event, Naming and Time, optimized for resource-constrained systems [78]) demonstrate a trend toward greater efficiency. For mote-based WSNs, however, this trend is not nearly enough.

2.3.2 Middleware in WSNs

Sensor network middleware, unlike the traditional version, usually just masks the complexity of distributed sensing on one particular mote infrastructure to allow programming the sensor network as a whole. There is an enormous amount of research activity in middleware for sensor networks with no real consensus on what is the best approach. Römer et al. proposed three design principles for WSN middleware: localized and adaptive fidelity algorithms, data-centric communication and application knowledge in the nodes themselves [90]. These principles can be seen to a greater or lesser degree in the WSN middleware surveyed here, falling into the paradigm categories of databases, mobile agents, virtual machines, neural networks, web services and novel techniques.

Databases

TinyDB [75], Cougar [16] and SINA [101, 52] all treat the sensor network as a virtual database, queried through an SQL-like language. In TinyDB, for example, each sensor node has its own query processor. The network forms a spanning tree rooted at the user. Once a query reaches the leaves, they respond periodically with readings. Parent nodes may aggregate or merely collate their own data with that of their children, iterating to the root. ESS [42] and DsWare [69] also fall into this category. Note that SQL is *not* a Turing-complete language and hence not fully expressive – a distinct drawback.

Mobile Agents

While the mobile agent concept for sensor networks has been explored fairly extensively (for example [91, 15, 86, 21]), actual mote-compatible implementations are far fewer. Agilla accepts tasks written in an assembly-like language for injection into the network [33]. Agent code and state propagation is governed by `move` and `clone` instructions specified over a grid topology. For efficiency reasons, Agilla mixes its metaphors by also providing a neighbor-accessible tuple-space within each agent for inter-agent coordination. SensorWare similarly transmits mobile scripts for stateful execution, but does not support notes [17]. In contrast, Kensho provides a group programming facility using publish/subscribe interactions [50]. Here application code moves fluidly through the network in response to local network state and activity, as opposed to explicit instructions in individual nodes. Finally, Impala from the ZebraNet project also embodies the mobile agent paradigm [72]. However, its goals in an extremely mobile, often disconnected, herd tracking mission are highly specialized.

Virtual Machines

Similar to the mobile agent concept, in that arbitrary code from the network can be run, virtual machine middleware is more general because it does not associate state with code updates. In fact, Agilla was built on top of the Maté Virtual Machine (VM). “Maté is a bytecode interpreter that runs on TinyOS” [67]. Maté uses viral code infection to reprogram a network. Because VM execution involves code interpretation, there is a significant run-time overhead cost compared to native binary code. Maté is itself a framework under which specific customized VMs can be created. Its default incarnation is called Bombilla. Support for constructing other VMs is pending, so further references to the Maté implementation actually refer to

Bombilla. Maté proves to be fairly similar to SENSIX at the mote level in two key ways. Both allow for domain-specificity: Maté by customized VMs, SENSIX through specialized IDL; and provide a default implementation, which for Maté is Bombilla. They both are also tuned for efficiency, discussed in detail in Chapter 5.

MagnetOS provides an object-oriented virtual machine as a single system image [11]. The programmer sees a single Java VM mapped over the sensor network. Despite its sensor-centricity and power-awareness, it is based on Java/Linux and therefore inapplicable on motes.

Neural Networks

Intuitively, the neural network approach seems like a good fit for constrained sensors. Both [87] and [63] post-process previously collected data for aggregation. Oldewurtel and Mähönen, however, implemented an artificial neural net on Telos nodes running TinyOS for light, temperature and humidity readings [82]. Although ‘fuzzy’ event detection requires a preprocessing step, bit-error correction is quite strong. Note that this paradigm only addresses data aggregation within the network.

Web Services

Using SOAP within WSNs has been proposed multiple times [25, 2, 4, 23], citing programming simplicity and flexibility. Janeček manages to squeeze a SOAP server (minus XML parsing) into a footprint suitable for a mote [53], but none of these proposals address the very significant added network overhead of textual (XML) transmission.

Other

Other notable middleware that do not fit into the categories above include EnviroTrack, MIREs, MiLAN, Kairos, DFuse, AutoSeC and ATaG. These middleware range from general-purpose to fairly application-specific. EnviroTrack is a TinyOS-compatible middleware that facilitates network-wide object tracking by using nearest-neighbor groupings of sensory context to aggregate data [1]. EnviroTrack is rather limited in that it focuses only on tracking types of tasks. MIREs concentrates on network interactions, featuring a publish/subscribe service for inter-application network messaging, also implemented for motes [99]. In contrast, ATaG (Abstract Task Graph) emphasizes dynamic tasking across the network governed by graphs of information flow, and adjusts to network conditions at run-time [9]. MiLAN also uses task graphs, but concentrates on providing abstractions that bridge network device heterogeneity [45]. Both DFuse [88] and AutoSeC [44] are service-oriented. DFuse separates structure, correlation and compute services specifically to facilitate data fusion. AutoSec provides generic and dynamic service composition, similar to CORBA at a high level. Kairos is a macroprogramming middleware that emphasizes global behavior, and uses only three primitives (variable read/write, iterating through one-hop neighbors, and addressing arbitrary nodes) in its programming model [39]. Not all of these middleware are meant for use in mote-level WSNs, and many are not governed by the data-centric and event-driven principles of constraint-bound sensor networks.

Finally, there are several papers that touch on issues particularly relevant to SENSIX. Lake proposes IDL-driven sensor fusion in [65]. Here both sensor objects and fusion collectors are defined in language independent IDL code. Though developed independently, this concept is central to the SENSIX reference implementation as described in Chapter 4. Both Villanueva et al. and Buckl et al. propose CORBA-on-the-mote implementations. Buckl in [20] emphasizes domain-specific languages,

while Villanueva in [109] focuses on compatibility with ICE. The SENSIX approach contends that pushing CORBA onto mote sensor networks is fundamentally wrong. Like the SOAP implementations above, these direct CORBA implementations ignore the extreme constraints of mote networks, particularly in network messaging costs.

It is clear that object-oriented software engineering principles and tools would bring a welcomed maturity to the challenges of WSN software development. It is also clear that there are fundamental incompatibilities between the high-level OO approach and low-level WSN constraints. In bridging these two, a secondary, but by no means distant, concern is to use pre-existing, preferably commodity, software tools and infrastructure wherever possible to avoid duplication of effort. For SENSIX, this means unifying TinyOS and all that it offers for WSNs, with CORBA and its vast array of distributed object services. This facilitates an object-oriented style of software engineering for sensor networks. SENSIX is the first step in creating energy efficient, robust and scalable sensory solutions with familiar and appropriate tools.

Chapter 3

The SENSIX Framework

Tanenbaum et al. have justly critiqued, “although researchers have developed effective solutions to *network* problems easily simulated in the lab, they have not addressed the *systems* challenges associated with deployment in real-world conditions,” [106]. This fact was particularly revealed in the example deployments discussed in Chapter 2. SENSIX begins to address some of these distributed systems and software engineering issues guided by the supreme WSN principles of energy efficiency, robustness and scalability.

Chapter 1 introduced the concept of integrating wearable vital sign monitoring devices into an HL7-compliant system. This chapter will continue to use this example to illustrate SENSIX concepts and usage, but keep in mind that the SENSIX approach is not domain-specific and can be applied wherever a wireless sensor network must interface with object-oriented software.

As mentioned in Chapter 2, XML-based SOA has been proposed for WSNs elsewhere [4, 24]. However, everything that is inappropriate about the object-oriented approach inside a WSN is doubly so with a human-readable data format. Therefore, while SOA is a valid part of HL7 standards, it is too extreme in its inefficiency to

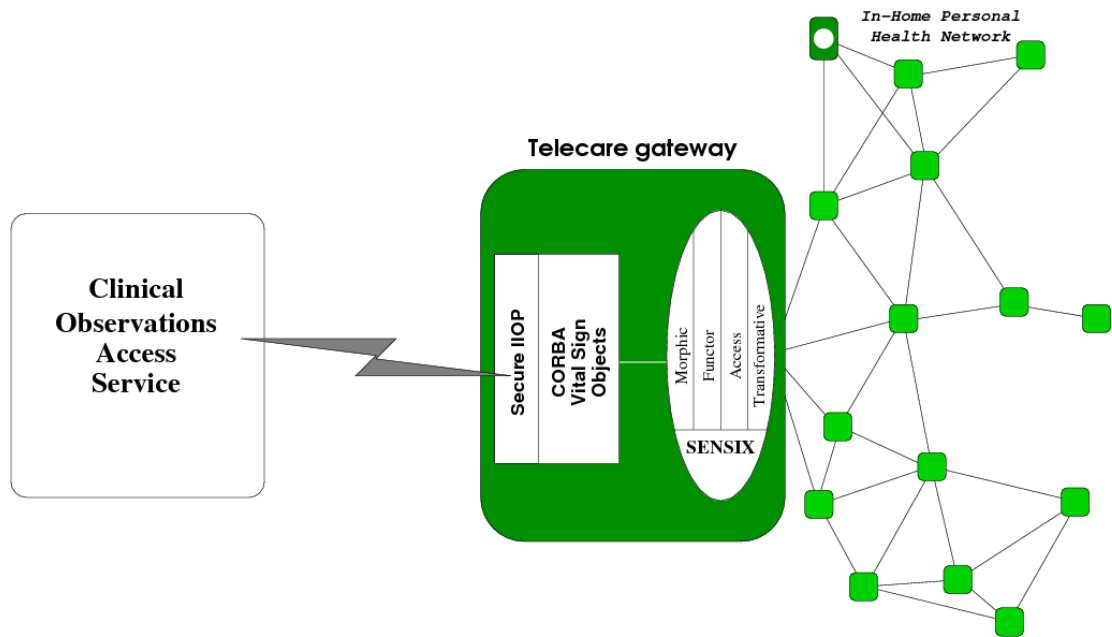


Figure 3.1: Wireless sensing in an ambulatory healthcare context.

serve as a direct WSN interface. Moreover, CORBA has standardized services that provide conversions to and from SOA protocols, which can be placed where it is appropriate – at the boundary to the user.

Figure 1.1 on page 9 depicts the overall HL7-compliant system and Figure 3.1 provides details of the ambulatory (in-home telecare) portion. In the in-home health network, the telecare gateway bridges the OO and WSN segments of the total system before transmitting data to the remote collection service. Given this comprehensive and coherent environment of the CORBA/HL7 system, tacking on multiple wearable vital sign WSNs through arbitrary, ad hoc software is a poor fit. Instead of considering the sensor network in isolation as most middleware implementations do, SENSIX achieves seamless vertical integration, resolving paradigmatic differences through its layered approach.

There are a number of conflicting requirements here. The first and most general is

that of the object-oriented versus event-driven and data-centric software paradigms. The object-oriented approach is important for its dynamic flexibility and its domain modeling; whereas within the WSN, the event-driven, data-centric paradigm is vital to preserve constraint efficiency. OO is highly abstract software; event-driven means the software is much closer to the hardware (event handlers are triggered by hardware interrupts); likewise, the data-centric approach in its asynchronous data transfer is very concrete. For this domain, the ability to dynamically add new software components or service abstractions, and the means to efficiently accommodate the added load of a security layer are crucial.

Next, and more specifically, is the conflict between the ease of adding, replacing or removing wearable devices and the need for network separation and security. We do not want two patients to pass in the hall and have their monitors reassign based on proximity, yet swapping out sensors should be as simple as their physical removal.

Finally, the wearable system needs to allow for dynamically individualized parameterization, per patient and per sensor. A caregiver should be able to alter alarm parameters, for instance, on the fly in response to a changing treatment protocol.

These requirements touch SENSIX's software reconfigurability, network reconfigurability and bridging capabilities in a domain-specific manner. First we'll examine the SENSIX build process with respect to this example, then how the SENSIX runtime fulfills these requirements and resolves the conflicts.

3.1 Building a SENSIX architecture

Through its multi-step build process, the SENSIX framework produces domain-specific architectures, which are themselves further specified into deployment-specific implementations. The sensor network architect coordinates with the user/program-

3.1. BUILDING A SENSIX ARCHITECTURE

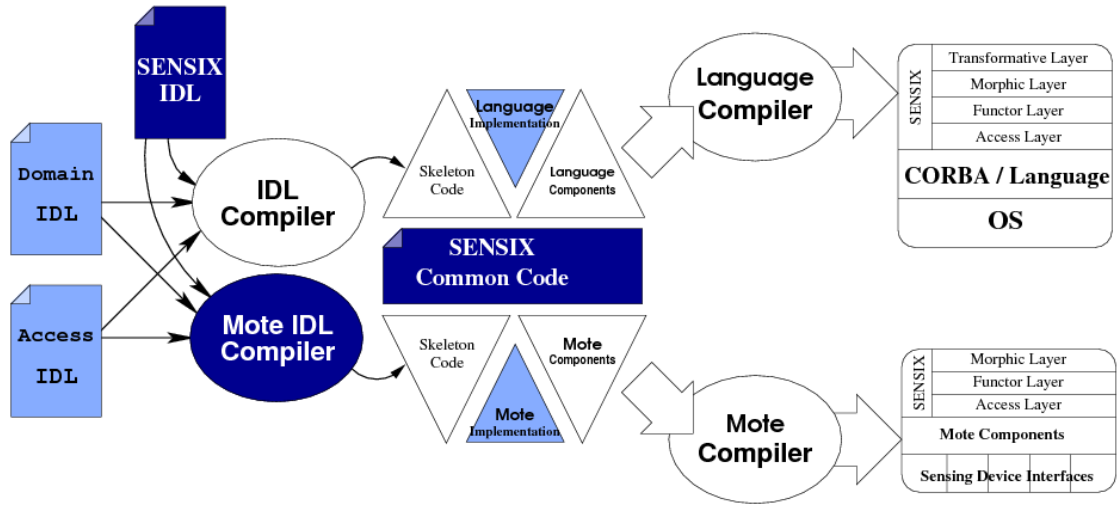


Figure 3.2: Generic SENSIX build process.

mer to define the system. The architect is a WSN expert involved only in this one-time process, whereas the user/programmer is a domain expert who administers and reconfigures the system frequently. Figure 3.2 shows the generic build process for creating a SENSIX deployment: code provided by the architect is lightly shaded, code and tools provided by SENSIX are shaded darkly, and the remainder are typical programming tools and components. This overall process is an extension of, and very similar to, building a CORBA-only application. The top half of the figure corresponds to the CORBA side of the bridging facility and the bottom half is devoted to the mote side of the bridge. The initial input consists of IDL code, part of the CORBA system. To implement SENSIX, this consists of three parts: SENSIX-specific IDL, domain-specific IDL and IDL that defines network access.

The SENSIX IDL (see the code listing in Appendix A1) defines the generic Functor layer of the SENSIX run-time. Functors are usually described as function objects, or independent software components that have, or embody a single method. On the object-oriented side of the bridge, this definition holds true; but on the mote side, a functor is just a data query or other simple sensory-oriented function. In imple-

3.1. BUILDING A SENSIX ARCHITECTURE

mentation, a functor resembles code from a functional programming language like LISP, and indeed, following functional programming principles, at the mote level it is stateless and its data is immutable.

The domain-specific IDL uses inheritance to declare what particular functors are available to the overall application. As in the UNIX philosophy per Raymond [89], this allows the creation of an application-specific mini-language. Elsewhere, such tightly focused languages are also termed domain-specific languages and this is a long-studied, yet active, field of research [13, 34, 85]. Some examples from UNIX that were directly involved in typesetting this document are latex, make, m4 and awk. Each has its own syntax and semantics; some are Turing-complete, some are not; none are meant to serve as a general-purpose programming language, but rather to do the task at hand in the most flexible way.

The access IDL defines the means by which new nodes that are added to the network are integrated into the software system. In other words, do new nodes broadcast their presence and capabilities, or register with a central server, or something else entirely? This is also where security services are invoked.

All this code is input to both the CORBA IDL compiler (available from various commercial and open-source vendors) and the SENSIX mote IDL compiler. The output is intermediate source code, called skeleton code, that is specific to CORBA on the one hand, and the chosen mote infrastructure on the other. At this point the user provides implementation details (the domain-specific ‘how’ of the functors) to fill in the gaps of the generated skeleton code. If the implementation language is the same for both, this code will be very similar for both CORBA and the motes.

All this, including the SENSIX-specific common code, is sent to the respective language compilers to produce the SENSIX run-time for the specified platforms. Figure 3.3 duplicates Figure 3.2 except where it is specific to our healthcare example.

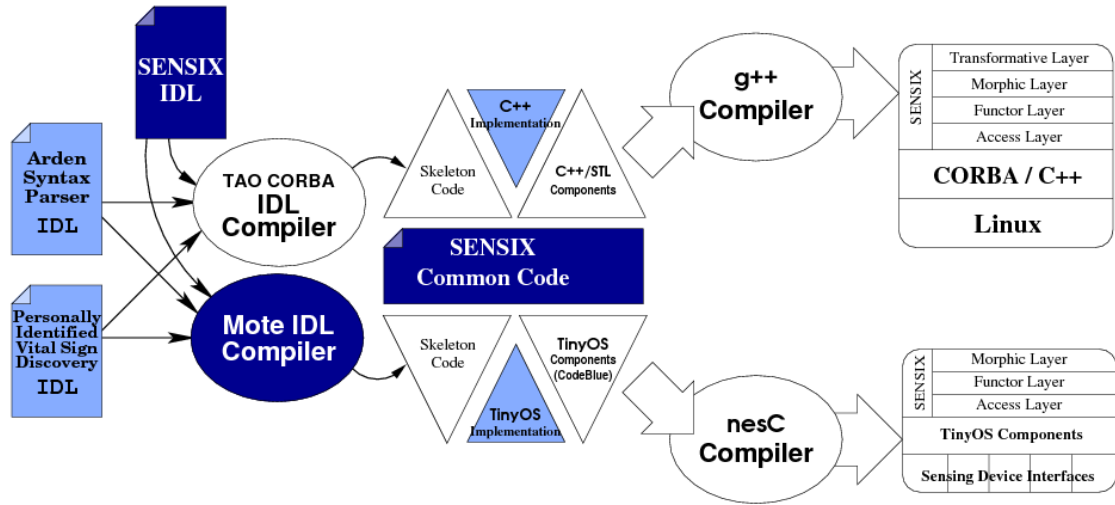


Figure 3.3: SENSIX build process for an HL7-compliant system.

Here the domain IDL will embody HL7 Reference Information Model concepts, particularly the Arden Syntax for vital sign queries, which interfaces with the Clinical Observations Access Service. The access IDL will interface with CORBA’s Person Identification Service to establish the patient-local network and initialize security. Here the CORBA implementation will likely be C++, and the mote implementation would probably use TinyOS plus CodeBlue components.

3.2 The SENSIX run-time

This recomposable build process already begins to shape the run-time interactions. Once a node has joined a network through the access processing, it will await a functor command to parameterize its sensing tasks. Figure 3.4 shows the conceptual layers of SENSIX. The Access layer corresponds to the functionality defined in the access IDL described above. The Functor layer is also already familiar from the SENSIX IDL definition above.

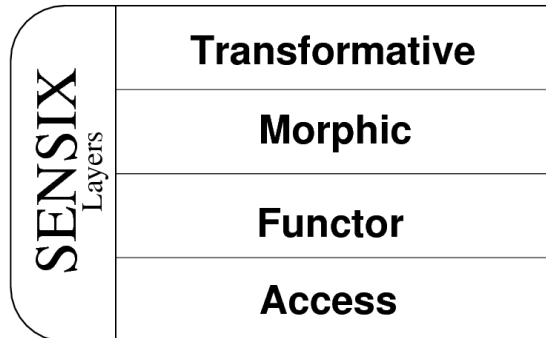


Figure 3.4: SENSIX layers.

The Morphic layer is called so because it is defined by the domain IDL which gives SENSIX the domain-specific ‘shape’ of its functionality. The root “morph” comes from the Greek, meaning ‘form’ or ‘shape’.

Finally, each mote infrastructure has its own packet structure or data encoding; the Transformative layer converts this packetized functor or tagged data to a CORBA object and vice versa. This Transformative layer, specific to each mote infrastructure, is included in the SENSIX common code.

An early SENSIX implementation returned data within the functor that generated it in an object-like style, but performance testing rapidly showed this to be an unnecessary and costly overhead. Instead, currently, tagged functors have a sequence number built in, and returned data is simply marked with that sequence number. Figure 3.5 shows the difference between the two approaches in terms of the average bytes transmitted per mote. This example is returning a total of 128 bytes of actual sensory data for both approaches (over a period of time); the discovery and command phases are identical, but the data portion of the right bar also includes the extra overhead of packing this data into a functor representation for network transmission. The left bar, on the other hand, only includes a node ID and a functor-tracking ID with the sensor data, reducing data transmission by half of

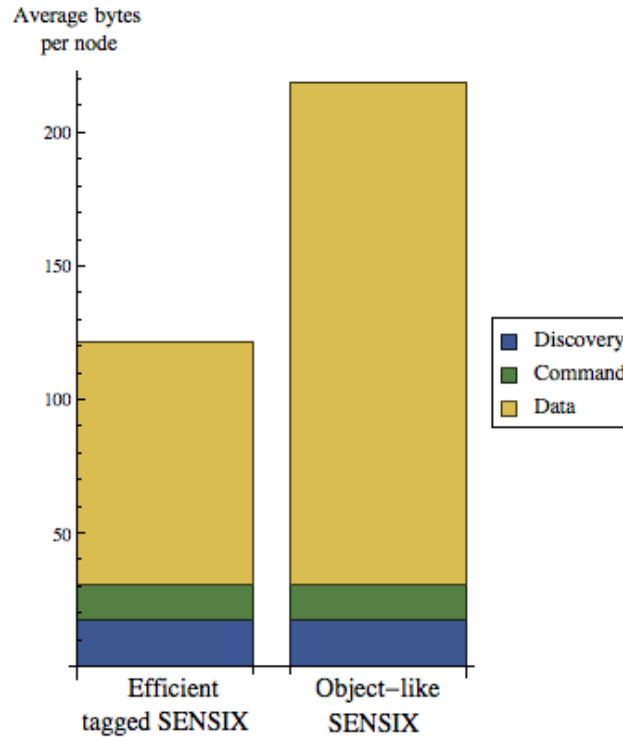


Figure 3.5: SENSIX data return efficiency.

the object-like style. This demonstrates the power of the data-centric principle to eliminate excessive network overhead, which can become quite significant over large network scales and/or lifetimes.

Thanks to the inheritance of the domain IDL interface from the SENSIX functor IDL, the functor provides the programming interface to the sensor network. The flow of control starts on the CORBA side: a functor object is invoked by a Request object's `apply` method. The Transformative layer converts the object and transmits the functor. On reaching the appropriate node(s), the functor guides sensor node activity – potentially over a long term. Data is either collated, aggregated or immediately transmitted, marked with a unique functor identifier. The Transformative layer loads the data into the original functor object and performs a `dataready` callback on the Request object. Multiple functors may be running on a single node,

sharing a single sensor or multiplexing across several. At this level, there is no indication of network protocols or routing within the WSN, those details are provided in the mote implementation code and, except where it impacts the Access layer, SENSIX is independent of those choices.

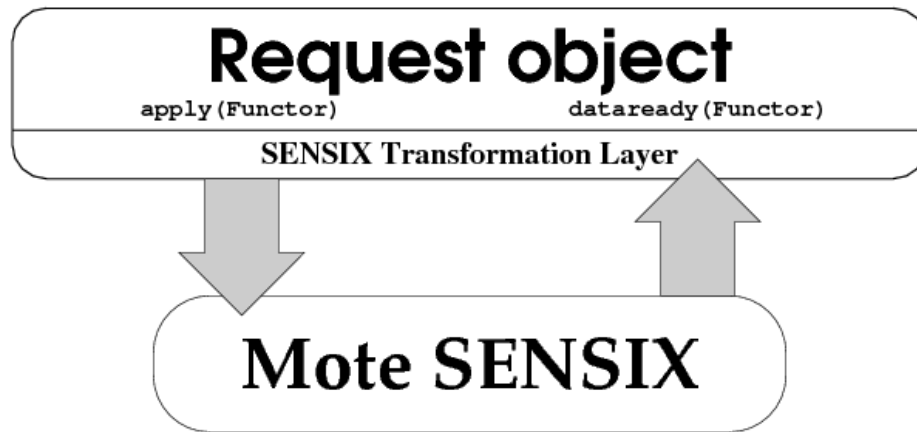


Figure 3.6: SENSIX Request object interactions.

Appendix A1 contains the IDL code that defines these objects. This IDL defines a single type for both integer and floating-point data, the above Request object, the Functor base class, and a Task object that is responsible for Functor tracking. All these objects are implemented in the SENSIX common code.

Notice that nowhere in this description of architecting the SENSIX framework has a specific application been overlaid onto the system. SENSIX requires only that the capabilities of the sensor network be defined. If the domain IDL closely models the sensors that are resident on the motes, then the flexibility of the overall system with respect to the sensing portion of the network is maximized and any constraints are entirely hardware driven, since SENSIX exposes this model in an object-oriented way. This, then, is the power of such a hybrid object-oriented/event-and-data-driven system. As a collection of independent software components, distributed object systems maximize flexibility and dynamic reconfigurability. Individual objects do

not have any fixed relationship to each other except in whatever limitations their interfaces impose. These objects can be rearranged and re-composed to provide new functionality during run-time. SENSIX elevates WSNs to this same level of flexibility while remaining sensitive to the constraints that make WSNs unique and challenging.

This chapter has shown how SENSIX takes advantage of its own separation of layers to split the CORBA and mote implementations. SENSIX also provides flexibility through both the generic and recomposable nature of the build process, and the dynamic and re-taskable run-time environment.

While this chapter examined SENSIX from the point of view of an example in healthcare, the next chapter describes a generic reference implementation. The domain and access specific portions of SENSIX, which have been hypothetical so far, will be fleshed out in detail. The next chapter will also look at SENSIX in a nested, as opposed to merely bridging, configuration.

Chapter 4

The Reference Implementation

Wireless sensor network middleware should provide “mechanisms for formulating complex high-level sensing tasks, communicating this task to the WSN, coordination of sensor nodes to split the task and distribute it to the individual sensor nodes, data fusion for merging sensor readings of the individual sensor nodes into a high-level result, and reporting the result back to the task issuer . . . [and] for dealing with the heterogeneity of sensor nodes” [90]. SENSIX accomplishes all of this.

While Chapter 3 described the SENSIX architecture and how it provides sensitivity to the target sensory domain, that is only part of the story. SENSIX also enables strong scalability and the reference implementation of SENSIX is particularly tuned to this, while providing generic sensing. Our previous example of healthcare highlighted the bridging aspect of SENSIX and its application in a pre-existing software infrastructure. This example is also an *urban* sensing domain, wherein the CORBA side of SENSIX has very low constraints on power consumption and benefits from wired network bandwidth. However, in settings without ready access to electrical infrastructure, even the higher capacity nodes running the CORBA side will be constraint-bound (though less so than motes). To make these scaling concepts

more concrete, this discussion will invoke a wide-area environmental science example which is itself a system of systems, or supersystem.

4.1 Object-Oriented Heterogeneous Sensing

Contrary to small scale WSNs composed of tens of motes, or large scale systems like ExScal's 1200 nodes, this supersystem example involves tens to hundreds of thousands of nodes, here termed a very-large-scale system. In this supersystem, nodes are classified by level: of size, computational capability, sensory range, communications range and on-board power – all of which are considered to grow proportionally.

At level one, the lowest level, are the leaf nodes of the total system tree: motes that sense micro-climate information within relatively small spatial bounds. These motes can form a peer-to-peer network, but their primary communication is with at least one level-two node. Level two nodes are significantly larger than motes, with greater resources available. These nodes may also contribute to sensory data, but at a different level of perceptivity. Whereas the motes gather fine-grained data, level two nodes have a broader but coarser field of view. Level two nodes have a much wider transmission range, are more sparse and directly administer level one motes.

The relationship of level three nodes to level two is similar, and so on up the physical hierarchy. The network between levels one and two is physically separate from that between two and three because motes tend to use different network technologies, and more importantly, this prevents throughput reductions brought on by overlapping networks which will hinder scaling. This separation is continued up the system hierarchy, as is the trend to more resources and rougher granularity. Figure 4.1 presents an example topology of such a supersystem.

This topology does not explicitly address network density. Although Gupta and

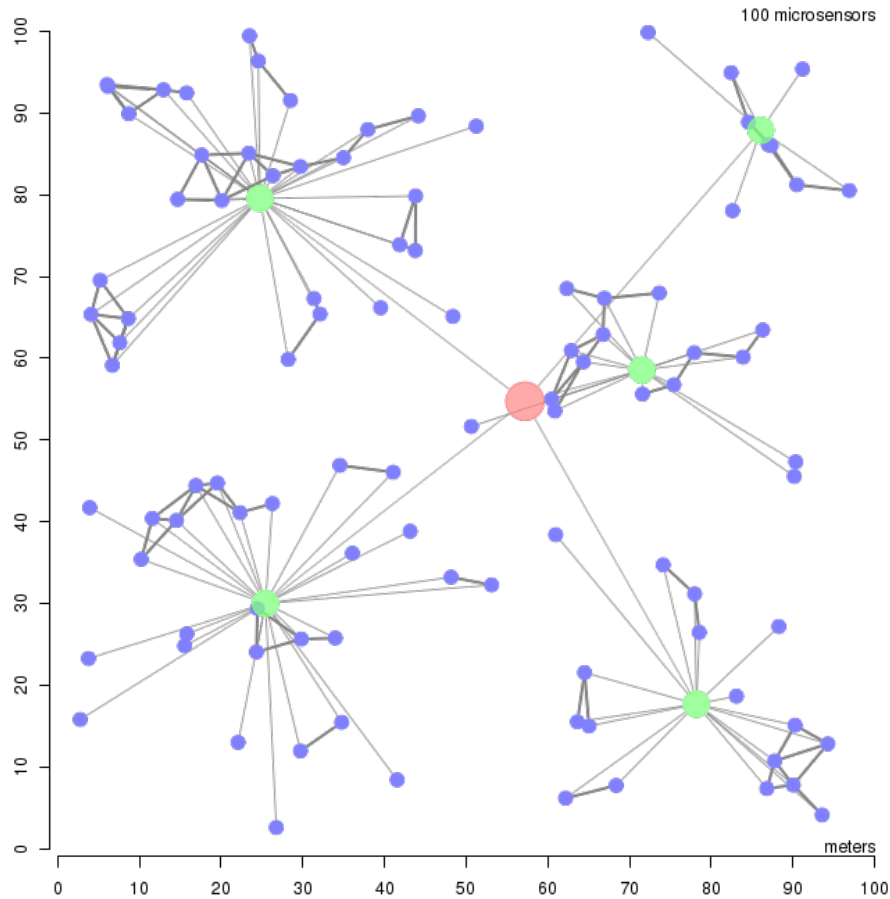


Figure 4.1: An example supersystem deployment (and visualization output of the SENSIX deployment tool).

Kumar show that per node throughput scales as $O(\frac{1}{\sqrt{N}})$, where N is the number of nodes per unit area [41, 40], Shepard shows an adaptive means around that limitation [95]. Hence this discussion ignores such potential limitations.

Why this topological complexity? First is the advantage in sensing. Like the fable of the blind men describing an elephant, motes may be seriously limited in their sensory extent, but a multi-level system offers increasingly broader overviews, potentially capturing macro-phenomena that motes alone cannot. Such a supersystem bridges the complementary perspectives of local phenomena observation and remote

sensing. Remote sensing refers to a stand-off acquisition of information on objects or phenomena from distant platforms such as aircraft or spacecraft. Remote sensors cannot yield the detail or diversity of in-situ sensors, but they can provide a broad picture and set the environmental context for data from the ground. This supersystem example uniquely offers an entire spectrum of viewpoints from micro to macro. The second reason for this topology is increased scalability. For a datum to traverse an ad hoc network, in the worst case it must be forwarded $O(N)$ hops, where N is the size of the network. The physically-based topology described here accomplishes the same traversal in $O(\log N)$ hops. While such network scalability may be a mere optimization, as described in [115, 71], sensory scalability may be vital to the correct completion of the supersystem's mission.

In our multi-tiered, example ecological study, stationary motes are very adept at uncovering the micro-climate where an elk forages, but are significantly challenged in tracking and discriminating between individual animals. However, these same motes can cue more powerful and more sparsely distributed nodes for which localized tracking is more feasible. These, in turn, can coordinate with higher-level identification and classification nodes.

SENSIX was itself inspired by such a multi-tiered mission. At Los Alamos National Laboratory, the Distributed Sensor Networks with Collection Computation project (DSN-CC) was tasked with creating a prototype traffic monitoring WSN. The system ultimately combined disparate sensory data from MICA2 motes and Crossbow Stargates, feeding into a decision engine. The motes ran TinyOS and were the most reliable segment of the system. The remainder of the software was, due to extreme time constraints, a rapidly-prototyped hack of sockets linking bare algorithms, and was prone to sudden and mysterious failure, exasperated by hardware malfunctions in a harsh environment. Ours was not the only project to experience such difficulties. Others, such as LOFAR-agro [66], had difficulty primarily at

the mote level, while ExScal [10] was quite challenged at multiple levels of network administration despite new in-network tools created specifically to reduce human intervention. SENSIX was born from exactly these issues of vertical integration with standard software development and engineering tools.

The DSN-CC traffic monitoring system used the TCP/IP network protocol for its reliability, and this approach has been proposed in [27] as well. However, the assumptions inherent in TCP/IP make it, like CORBA, inappropriate for highly constrained portions of such a supersystem. Delay Tolerant Networking can address some of these assumptions such as continuous connectivity [32], but not (like the CORBA/e specification) WSN-specific efficiency issues. These tradeoffs in commodity network protocols mirror those of CORBA commodity middleware.

The SENSIX reference implementation embodies a whole-system remedy. To achieve network and sensory coverage, SENSIX offers a deployment tool. It assumes random placement of level-one motes when there is no a priori information on the sensing environment. Based on given radio ranges, it minimizes the number of nodes of each subsequent level while maintaining full network coverage, as seen in the example in Figure 4.1. If sensory coverage information is available, the tool can utilize such feature maps to constrain level-one placement.

These network-forming aspects all fit within the Access layer of SENSIX. The Morphic layer represents the sensory tasks that the network will perform. The next section discusses the models that back the reference implementation's Morphic layer.

4.2 Sensing and Tasking Model

Supersystem middleware must allow for heterogeneity: of sensors, of processors, of energy consumption, of data storage capacity and of communication modalities. To

4.2. SENSING AND TASKING MODEL

achieve dynamic reconfigurability the middleware layer must also handle high-level task decomposition down into the network as well as the resultant data fusion upward, all in a resource sensitive manner. Assuming such a deployment depicted (without features) in Figure 4.1 and a network topology as described above, the SENSIX reference implementation handles ad hoc network and sensory capability discovery, task decomposition down into the hierarchy tree and data aggregation back to the root.

JDL	Boyd	SENSIX
	act	evaluation
refinement } impact }	decide	inferencing
situation } object }	orient	{ follow multi-target target tracking
data	observe	

Table 4.1: A comparison of data fusion models

The task decomposition and data aggregation in this SENSIX implementation are governed by a data fusion model that derives from several standard models, such as the Joint Directors of Laboratories’ (JDL, a DoD R&D committee) data fusion model [114]. The JDL model separates data fusion processing into levels that assess object, situations, threats (impact) and the fusion process itself. Subsequent work has added a sub-object level and defined how these levels may interact [102, 73]. The Boyd control cycle [18] is another prominent data fusion processing model, also known as an OODA loop for its linked phases: observe→orient→decide→act. Bedworth and O’Brien integrate these models, and others, in the Omnibus model, itself a looped construct [12]. Table 4.1 shows how the SENSIX data fusion model fits alongside both the OODA loop and the five-level JDL model. Whereas these

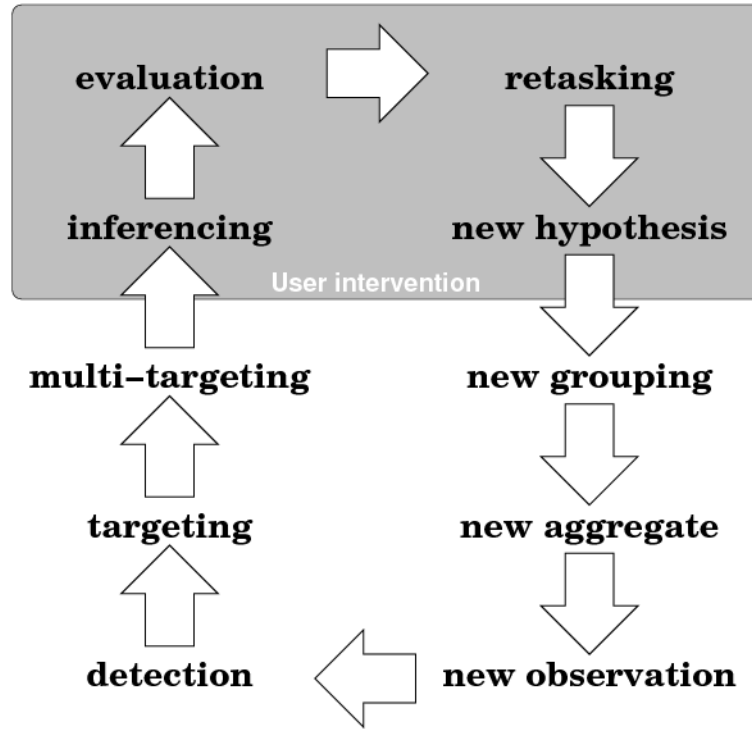


Figure 4.2: The SENSIX data fusion/network tasking cycle.

other models seek to be general and open to interpretation, the SENSIX model is fairly specific to sensory data fusion.

Figure 4.2 shows this SENSIX fusion model as a cyclic process. Note that, as currently implemented, the inferencing, evaluation, retasking and new hypothesis steps require human input. The remaining steps describe the task decomposition within the hierarchical supersystem as sensory viewpoints shrink and become more specific, and then the corresponding data aggregation in the reverse direction. Not reflected in these diagrams is the fault modeling that is necessary, but often ignored in data fusion models. In the SENSIX model, faults are handled as close to their origin as possible, and are propagated through the network only as far as they are relevant. This data fusion model therefore implies that decisions, however simple, are being made at every level of the supersystem tree. How all this is accomplished

and realized is defined by the SENSIX reference implementation's domain-specific mini-language.

4.3 Sensing Mini-language

The table in Appendix A2 details the functor extensions of the reference implementation language. Appendix A3 lists the corresponding IDL definitions. There are three categories of functors here: data, series and aggregate, each encoded as a single byte with limited parameters available. This encoding scheme ensures efficient network representation, while functor nesting allows for task complexity. Listings 5.1 through 5.4 on pages 58 and 59 show several simple examples of SENSIX task source code.

Task decomposition can either (a) be explicitly specified by including a hierarchical level parameter (see Appendix A2) to each functor which specifies at what tier the functor is stripped, or (b) follow the default decomposition which strips aggregate and series functors right before the mote leaf level. Figure 4.3 illustrates the functor object method calls that implement the SENSIX data fusion model cycle. Expanding on the bridging facility of Figure 3.6 on page 41, this figure demonstrates how SENSIX extends the CORBA client/server infrastructure into a peer-to-peer configuration. This is primarily accomplished by combining server and client processes on each node. The `apply` call sends a Functor across the network, and `devolve` triggers task decomposition. The `dataready` call receives collected data, while the `aggregate` function performs Functor-governed data fusion.

Normal CORBA usage requires a central Naming Service to find remote objects; the reference implementation of SENSIX bypasses this limitation by using a discovery phase. In discovery, nodes broadcast their presence to their neighbors. Aligned with the network topology described above, such messages are separated by a node's

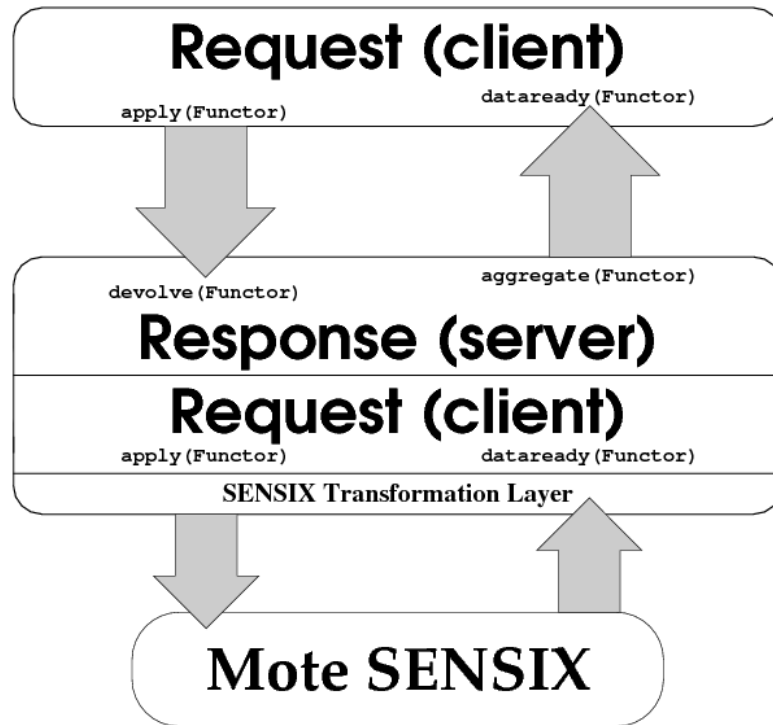


Figure 4.3: SENSIX Request/Response objects implementing the data fusion/network tasking cycle.

position in the hierarchy. Ancestor messages announce higher-level nodes, descendant messages the presence and capabilities (and in CORBA, the IOR of those capability objects) for use by parent nodes, and sibling messages share this information among peers. These discovery message types are distinct because, on a particular node, they may be sent out on separate network interfaces. Figure 4.4 details this process in the style of a Unified Modeling Language (UML) Sequence diagram, along with the tasking process. Although this sequence shows ancestor messages triggering descendant messages, the addition of a node anywhere in the network will cause it to send all three message types, resulting in a cascade through the network.

For tasking the network, the user provides a LISP-like nesting of functors. These include the data functors, `sense` and `peaksense`; the latter ignores values that are

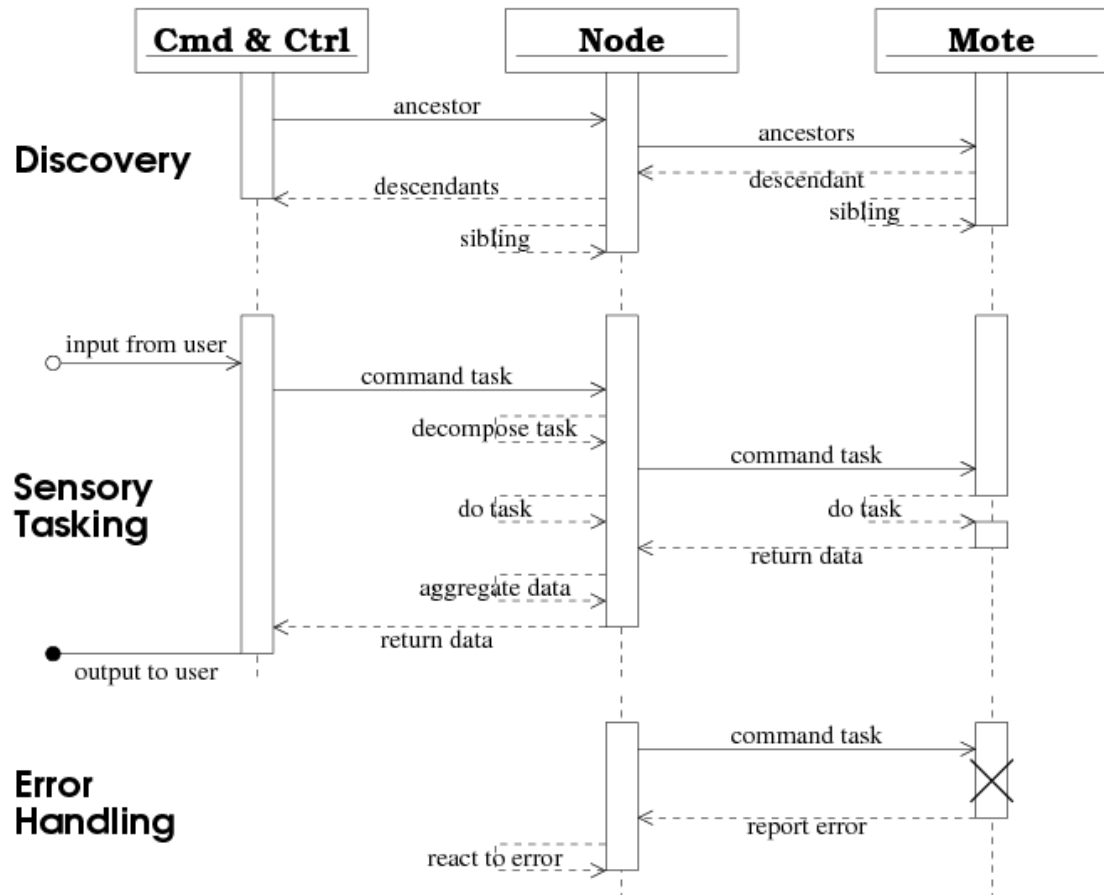


Figure 4.4: A UML Sequence diagram of the SENSIX run-time under the reference implementation.

not beyond a certain threshold. Both require a sensor parameter and may take an optional sampling rate parameter. The `timeseries` functor collects readings over a time span and must envelop at least one data functor (functors may be combined with logical AND and OR). Likewise, the `spatialseries` functor combines readings over a certain location. Although GPS is available for motes, it is power-hungry. Instead, motes are considered to have the coordinates of their parent in the hierarchy. This is reasonable considering the several meter resolution of basic GPS, but may be supplemented by other WSN-specific localization schemes.

Finally, the aggregate functors all require one or more series functors. `Sum`, `delta`, `mean` and `sigma` each have their mathematical meaning. `Recite` simply collates readings. `Lambda`, however, can run arbitrary bytecode over its constituent data. Here, this is implemented as Java bytecode injection and obviously is not applicable to the motes. Under certain mote infrastructures that support it, such as Mantis (but not TinyOS), `lambda` may be implemented for motes with on-demand compilation of code modules to binary.

Although a three-level nesting of these functors is readily apparent, a task can nest functors almost arbitrarily, combining newly sensed data with aggregated results at any level. As the sensory tasking phase of Figure 4.4 demonstrates, the symmetric task decomposition and data aggregation process, spawned by user input and resulting in output to the user/programmer, allows any mix of data acquisition and fusion.

The error handling phase of Figure 4.4 shows the fault model mentioned above, wherein errors (represented by the X) are handled as close to the source as is reasonable. Rather than just report the error to the user, the supersystem attempts to resolve it reactively. This may consist of re-commanding a task, rerouting communications, or running a different task.

This mini-language couples the Access and Morphic layers of SENSIX under the physically hierarchical assumption that data density, energy budget and transmission range all increase as the network tree is traversed from leaves to root. Such coupling is not necessary; in fact decoupling these layers would lead to greater software reuse by reducing the interdependency that now exists.

4.3. *SENSING MINI-LANGUAGE*

This chapter detailed the task decomposition and data aggregation cycle and the physical network tree of the SENSIX reference implementation. There now only remains the question of SENSIX bridging performance and the extent of the impact of CORBA and object-orientation on such a vertically integrated supersystem. The next chapter utilizes the network topology and data fusion modeling as implemented here to evaluate the performance of SENSIX by way of specific tasking case studies.

Chapter 5

Metrics, Experiments and Results

While previous chapters covered high-level SENSIX concepts and its low-level implementation details, this chapter will show how this approach performs with respect to other sensor network middleware in terms of task complexity, data aggregation, and messaging overhead. This chapter also presents the tradeoff of commodity object-orientation, and discusses the circumstances that may influence the adoption and extent of a commodity approach.

5.1 Middleware Emulation and Simulation

Out of the myriad of sensor network middleware mentioned in Chapter 2, I have selected three that represent disparate middleware paradigms: namely, TinyDB which treats the sensor network as a database to be queried, Maté where the network is a virtual machine, and Agilla based on mobile agents. The main criterion for selecting these implementations is that they actually exist in implementation and are readily available in the TinyOS repository. Other reconfigurable middleware were either not implemented (MiLAN), not publicly available (Cougar, SINA, Impala, DSWare), or

did not fit into motes or the TinyOS simulation framework (like SensorWare).

To run large scale experiments, simulation is a must. A true test-bed deployment would be preferred, but at scales above 1000 nodes this is cost-prohibitive. In order to simulate potentially large scale sensor networks, I modified the built-in TinyOS simulator, TOSSIM [68], to work within a cluster computing environment. To accommodate this, these modifications (a) allow node numbering that does not directly index state structures in the simulation process, (b) routes out-of-process messages through a centralized networking simulation server, and (c) requests ADC input from a centralized sensory simulation server. The networking server acts to join the TOSSIM instances into a single network. It accepts messages, calculates noise and collisions, and re-injects those messages as appropriate. TOSSIM, which handles local collisions and noise, delivers an injected packet as-is, and so requires extra (non-local) noise and collision handling by the networking server. To produce sensor data, the sensory simulator loads a file produced by a GUI program that combines the predefined sensor network node layout with point-and-click user input defining the behavior of sensory phenomena.

In contrast to simulation frameworks, such as ns-2, that focus exclusively on network interactions, this TOSSIM-based framework is far less concerned with the fidelity of the network simulation and more intent on component software interactions. Since TOSSIM is merely a recompilation of the same mote TinyOS code to the x86 architecture, with some additions for simulation effects, this framework is more emulative than ns-2 – which may in fact be a better choice for multihop networks [51]. Whereas simulation explicitly and abstractly models the system or interactions being studied, emulation focuses on reproducing the external behavior of the system (*not* the internal state). This approach also avoids the problems of model validation that plague wireless network simulations [5, 64, 62]. Emulation is also appropriate in this case because internal state is far less important here than the interactions of

system components. The CORBA/Java portion of SENSIX ties into this TOSSIM framework via the Transformative layer, but otherwise does not simulate a *wireless* network. Again the aim of this emulative approach is not to analyze network protocol performance, but to compare the efficacy and relative efficiencies of the software systems in question.

The core comparison here is among Maté, Agilla, TinyDB and the mote-side of SENSIX, all instrumented in the same way with identical network layouts, sensory inputs and timing. Sensory input is produced as described above, and since these middleware implementations are used in a reactive fashion, the timing is closely tied to this input.

Networking assumptions were trickier to unify. As covered in Chapter 4, large-scale SENSIX is organized such that the network topology dictates a routing tree. In contrast, Maté and TinyDB use the standard MintRoute component from TinyOS. Agilla assumes a grid network (addresses are in the form of an x, y grid position). In the end, the simplest solution was to share the same network topology and TOSSIM connectivity graph among the four middleware. The SENSIX discovery phase, MintRoute, and a diagonal grid for Agilla all come to the same conclusion for the routing tree. The arrangement of the other middleware into this topology nullifies any advantage the SENSIX implementation may derive due to a networking hierarchy. The connectivity graph for TOSSIM was created using the deployment tool described in Section 4.1.

Liu et al. establish a lower bound limit for such multi-tiered topologies: for n nodes in tier levels i and j , $n_j \geq \sqrt{n_i}$, where $i < j$ [71]. Tier level numbering increases from leaves to root. For these experiments, individual tier node counts were set at $n_j \approx \sqrt{n_i}$, with non-integer counts always increasing to the next integer. This also very roughly corresponds to the relationship between the transmission ranges for motes (on the ground), 802.11-based radios, and long-range ISM-band (i.e. 900

5.2. TASKING CASE STUDIES

Listing 5.1: Basic reading task

```
recite(spatialseries(sense(sensor=light , rate=5)))
```

Listing 5.2: Event detection task

```
recite(spatialseries(peaksense(sensor=light , rate=5, low threshold=2) &&  
                    peaksense(sensor=accel , rate=5, high threshold=3)))
```

MHz) radios.

These experiments were run on the custom N -sim cluster at Los Alamos National Laboratory. The cluster nodes booted a Debian Live netboot image, and were administered using the kanif tool based on TakTuk [22].

5.2 Tasking Case Studies

Exhaustive empirical testing and comparison is just not feasible, therefore I have selected four tasks implemented in each middleware version. These tasks represent the common missions of intrusion detection, tracking, and environmental monitoring, plus a minimal single-reading task. These tasks are represented in the SENSIX reference mini-language (as source code, not bytecode) in Listings 5.1 and 5.2, and 5.3 and 5.4 on the following page. For the remainder of this discussion, these tasks will be labeled as follows: the basic reading task is a minimal single-reading task (from an ambient light sensor); the event detection task performs intrusion detection by a reduction of ambient light and an increase in accelerometer vibration at the same time; the event tracking task traces the intrusion by monitoring the same readings as the second task, but over a time series; the environmental monitoring task watches for a significant increase in ambient light and temperature across several nodes to detect a fire (environmental monitoring). These tasks are linked together

Listing 5.3: Event tracking task

```
recite(timeseries(peaksense(sensor=light , rate=5, low threshold=2) &&
                  peaksense(sensor=accel , rate=5, high threshold=3), duration=10))
```

Listing 5.4: Environmental monitoring task

```
recite(spatialseries(peaksense(sensor=light , rate=5, low threshold=2) &&
                     peaksense(sensor=temp , rate=5, high threshold=3) &&
                     peaksense(sensor=siblings , low threshold=2)))
```

through predetermined and scripted triggers in order to demonstrate the dynamic reconfigurability of the network.

These task cases are easily applicable to the multi-tier environmental science example, yet equally feasible for a mote-only deployment under the same network topology.

Not all middleware are created equal. The last three tasks were the most difficult to realize in TinyDB, as a result of its query abstraction, more than anything else. In contrast, Agilla and Maté, which are much more flexible and expressive, also required *much* more extensive source code. SENSIX, embodied in the reference implementation, manages to be both concise and expressive for this set of tasks. There certainly are other tasks for which this reference implementation is not suited, but unlike Agilla and Maté (in the form of Bombilla), with SENSIX that is easily remedied. The earlier healthcare application provides just such a counter-example: the reference implementation does not directly support data streaming, which is likely a primary mode for a vital sign body area network. Such domain-specificity, as provided by SENSIX, ensures a close and efficient fit of tasks to desired outcomes.

In comparing metrics among these four middleware, it is important to disregard the impact of the object-oriented side of SENSIX for now, since the other middleware are mote-only. The next two sections look at strictly mote-only tasking and messag-

ing metrics, but Section 5.5 does look at the effect of CORBA on overall supersystem performance.

5.3 Task Complexity Metric

A typical metric of source code complexity counts lines of code (LoC). Since task source code is widely varying here: for TinyDB it is SQL-like, for Agilla assembly-like, for Maté C-like, and for SENSIX Lisp-like, using a strict LoC metric would be invalid. However, since whitespace is syntactically meaningless across all four middleware, except to separate tokens, counting non-whitespace characters as in the top half of Figure 5.1, seems to have more validity. A better metric looks at the bytecode representation of each. This is a byte count of the task code as it is transmitted wirelessly. This bytecode metric answers concerns, raised by Kaner and Bond in [57], that the reputed measurement actually means what we think it does. The bytecode metric directly impacts the network performance shown in Figure 5.3.

The top half of Figure 5.1 looks similar to the bottom bytecode bar graph, except that the scale is about an order of magnitude larger. The Agilla and TinyDB bytecode have a direct correspondence to the source code complexity, but SENSIX has a slight improvement and Maté a rather significant improvement. The tradeoff between task code complexity and the expressiveness of the middleware abstraction is very apparent here. Agilla sacrifices efficiency in propagating tasks to achieve general purpose utility, whereas TinyDB appears to do just the opposite. SENSIX, too, chooses efficiency over expressiveness at the level of a particular implementation, but does not suffer as a result thanks to its domain-specific build process.

5.3. TASK COMPLEXITY METRIC

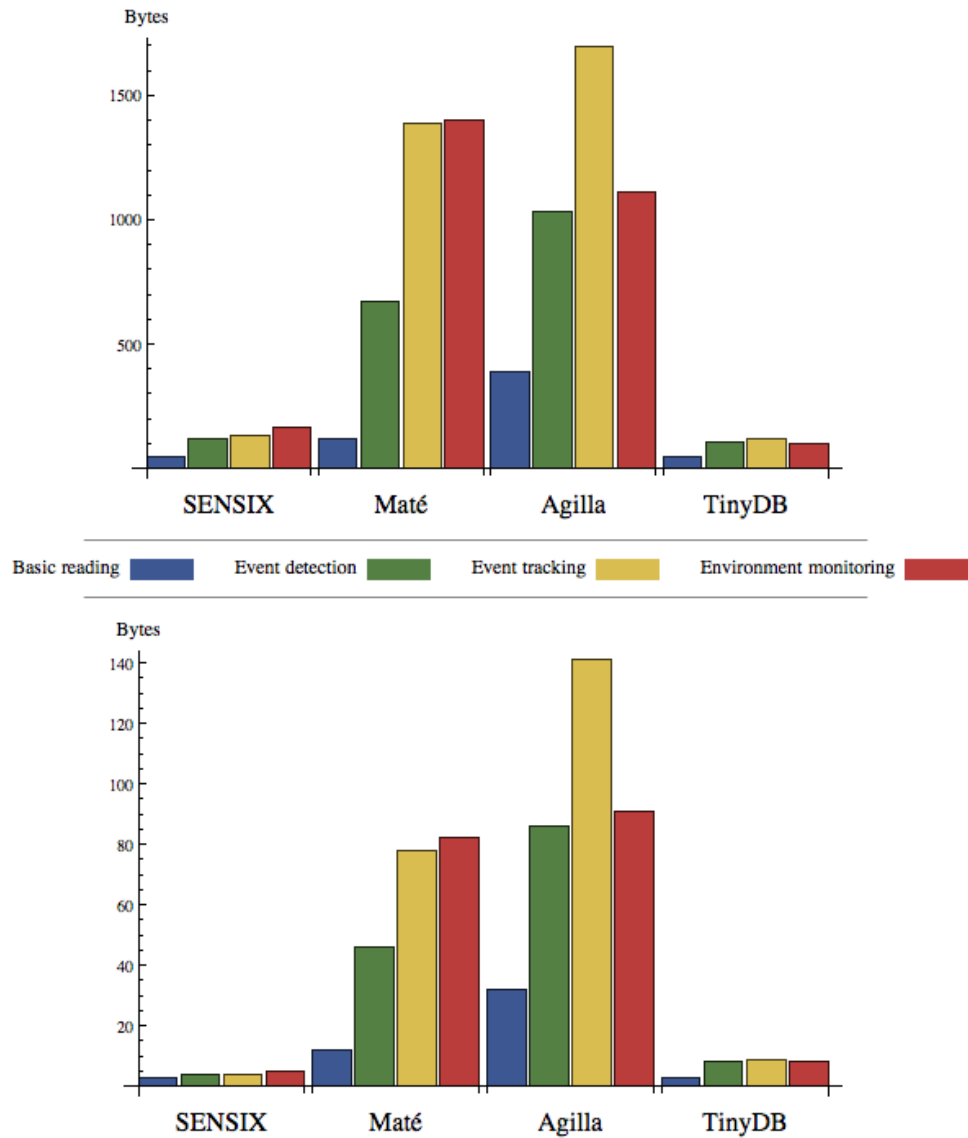


Figure 5.1: Middleware task complexity in bytes: source code (top) and bytecode (bottom).

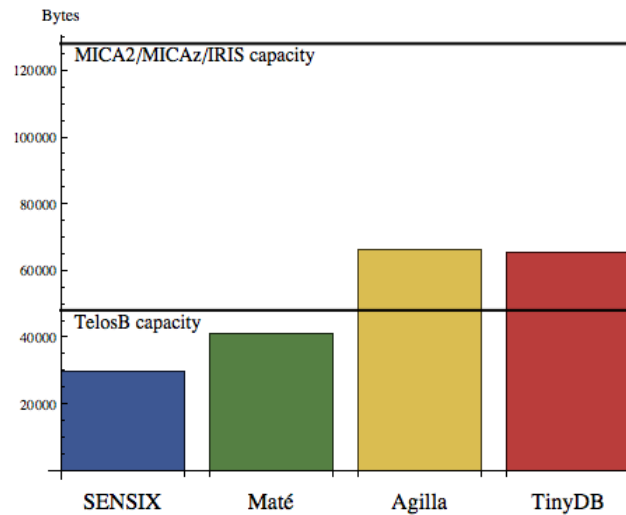


Figure 5.2: Compiled middleware engine size and mote capacity.

Figure 5.2 shows another source code based metric that does not have a direct impact on task reconfiguration, but is important nonetheless. The metric is that of the compiled footprint size, which has an impact on additional mote components that may be included. SENSIX leaves plenty of room for expansion, even for the flash-memory-poor TelosB.

5.4 Messaging and Energy

For WSNs the most important metric is energy consumption. Mote power usage is dominated by message transmissions, as is readily seen in Table 5.1, which summarizes the power constraints listed on the datasheets of the most common motes. As such, minimizing the bytes transmitted wirelessly reduces the energy expenditure of the unit, and of the network as a whole. There are other potential issues, such as network bottlenecks, that may be detrimental to network lifespan, but they are not considered here because they are network topology and protocol dependent, and

	$\mu\text{C}/\text{CPU}$	TX/RX
MICA2	24	81/30
MICAz	24	57/54
TelosB	6	57/54
BTnode	24	81/30
TMote Sky	18	53/59
IRIS	24	51/48
TinyNode 584	6	180/42
eyesIFX	7	36/28
FireFly	18	51/57

Table 5.1: Mote energy consumption in milliwatts

thus out of scope for this evaluation.

One hundred simulations were run, as described above, for network scales of 100, 1000 and 10000 nodes. Although 100000-node simulations were planned, they were deemed to be unnecessary, and sensory data production at this scale also proved extremely time consuming. TOSSIM has a ‘packet’ debug switch that outputs so much information about network transmissions that it can interfere with simulation performance. Instead, each middleware was instrumented such that it tracked the number of bytes sent, binned into one of two types (three for SENSIX), namely task propagation, or data return, or, for SENSIX alone, capability discovery. The task complexity metric above directly impacts the empirical task propagation measure.

Besides scaling and task complexity, there are other variables that can shift the outcome of the relative middleware transmission efficiencies, namely the number of bytes returned per node, the extent of data aggregation in the network, and, specific to the SENSIX discovery phase, the number of sensory capabilities.

Figure 5.3 shows the average bytes transmitted per node over the lifetime of each task. Variation over individual simulations is very small: the average is approximately 0.1 percent of the total throughput. This is intentional. Network topology,

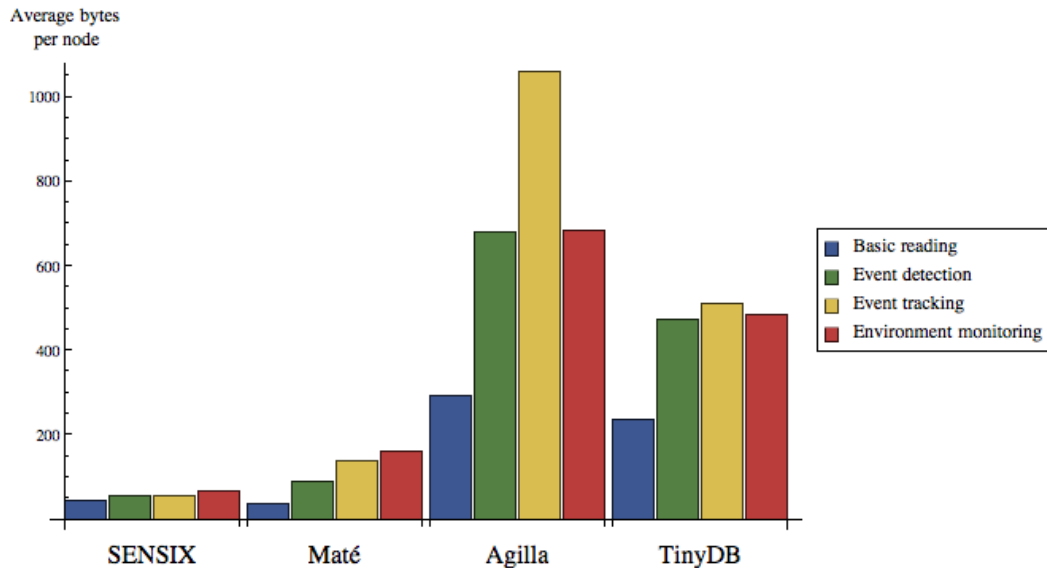


Figure 5.3: Task messaging average per node.

network density and event timing between simulation runs was tightly controlled. This was to eliminate wireless network effects as much as possible, focusing instead on the software overheads. Because SENSIX and Maté are difficult to see here, Figure 5.4 presents the same data, focusing on just those two middleware.

After determining that TinyDB traded expressiveness for efficiency, why does it perform so (relatively) poorly? The answer is two-fold: first Maté goes to great lengths to ensure that every network transmission is minimal, with excellent results; second the tradeoff between expression and efficiency is neither proportional nor straightforward. Increased expressivity may positively impact efficiency, especially for complex tasks, and when retasking is infrequent compared to the volume of data returned. It seems that TinyDB, particularly when not utilizing its aggregation attributes, is too simplistic to scale well under a variety of task complexities.

Agilla’s performance is strongly degraded by its wrapping the returned data in a mobile agent, coupled with its large bytecode representation. When the volume

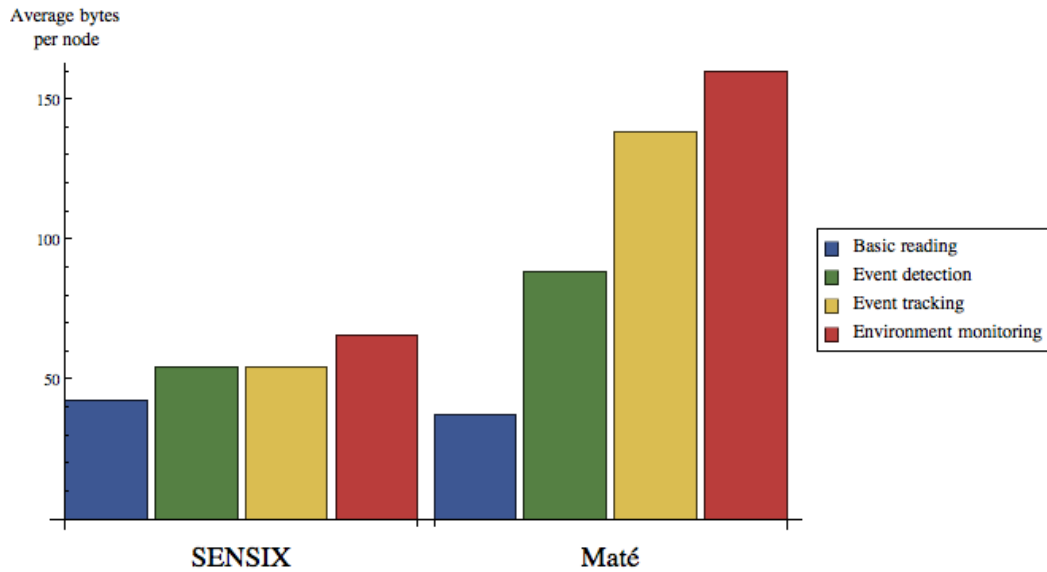


Figure 5.4: Task messaging average per node – closeup on SENSIX and Maté.

of data returned per node is very large, Agilla does much better; however, this is in opposition to normal WSN principles of in-network data aggregation and network responsiveness.

Maté performs remarkably well. Investigating the reasons for this performance led to the SENSIX optimization mentioned in Section 3.2 – wherein the functor which wrapped returning data was discarded in favor of tagging and tracking. At the Transformative layer where functor objects are turned into mote functors, the Task object assigns a unique tag to the mote functor. When this functor’s data returns from the mote, the tag allows the data to be matched to the original functor object. Maté, too, assigns version numbers to its code capsules, but does not need to provide tracking beyond this since it does not provide vertical integration. Maté seeks to be general-purpose, which drives the size of its in-network task representation, and this in turn causes it to not perform as well as SENSIX. The SENSIX reference implementation does not try to be general-purpose at this level of deployment, but pushes those decisions domain-specificity into the architecture build phase.

Maté also incurs a performance hit during execution because it is interpreting instructions. In contrast, SENSIX functors are binary executables waiting to be activated. Also, for run-time systems that support it, bytecode functors (like `lambda`) can run arbitrary functionality, but the energy-cost of this is made more explicit by the extra step it requires in tasking the network.

The average number of capabilities per node only affects the SENSIX discovery phase, causing its throughput to grow minimally. An increase in the number of bytes returned per node (normally directly related to task complexity) causes the data messaging phase to grow, but disproportionately among the different middleware due to wrapping and potential aggregation. Data aggregation is not expressed in these relatively simple tasks, and this lack has a detrimental effect on TinyDB. Post-experiment analysis indicates that the extent of aggregation has a large influence on performance, and warrants its own metric. However, that has no bearing on this discussion, because while variation along such an aggregation metric will show vast improvement with increased aggregation for TinyDB, Agilla will still perform poorly due to stateful transmissions, and Maté and SENSIX will continue to perform well thanks to their data-centric networking.

Task complexity has an effect well predicted by the bottom bar graph of Figure 5.1. The effect of network scaling is rather flat in terms of average bytes per node, as reflected in Figure 5.5. This means that overall network scaling of throughput is linear for mote-only SENSIX, Maté, Agilla and TinyDB. Whether this simulation result is reflective of reality is discussed further in the next section.

5.5 Commodity tradeoff

Up to this point, the comparisons have all involved mote-only middleware. The CORBA side of SENSIX has been assumed to be running on low-constraint units and thus external to the measured WSN. CORBA should not run on motes, but what if, as in the supersystems of Chapter 4, we wish to run the CORBA side on more constraint-driven devices. How does CORBA affect the network?

Consider an example network deployment composed of iMote devices which can run either TinyOS or Linux with CORBA/Java. Chapter 4 described the physically tiered network topology that the SENSIX reference implementation assumes. Under this topology, the CORBA side of SENSIX always occupies the tiers nearest to, and including, the root. An example full-SENSIX deployment, then, would run CORBA SENSIX in the top three tiers of iMotes, and mote SENSIX in the bottom two (most populous) tiers.

In addition to the mote simulation described above, a set of full-SENSIX simulations were run, varying not just network scale but also the percentage of the network devoted to the CORBA side of SENSIX. CORBA instrumentation to produce tasking versus data versus discovery byte counts was included as well. The number of tiers of the topology that CORBA occupied was varied, and the variations performed for the mote-only runs were also included here. The graph in Figure 5.5 shows the average bytes transmitted per node as both network size and task complexity vary independently. Task complexity is unified for all four middleware here as a single number. Under this unification, the basic reading task has a metric of 2, the metric for event detection is 4, and both the event tracking and environmental monitoring tasks have a metric of 5. Here CORBA occupies all but the bottom one or the bottom two tiers, which corresponds very roughly to two percent and one percent of the network, respectively. A 3D surface is fitted to the empirical measures here to

5.5. COMMODITY TRADEOFF

clarify overall trends.

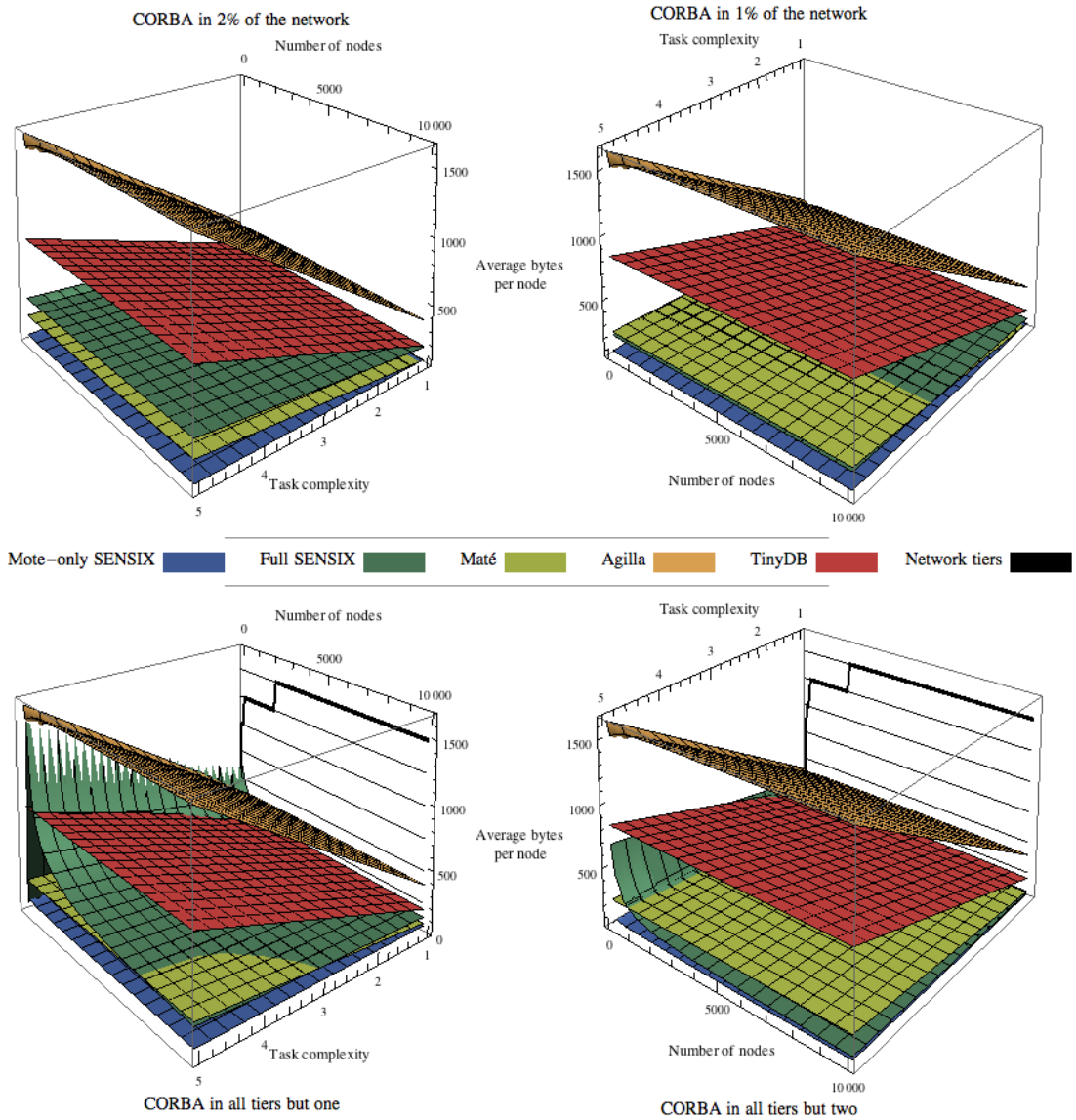


Figure 5.5: Middleware messaging over increasing scale (including CORBA).

In the lower graphs of Figure 5.5, it is apparent that the CORBA side of SENSIX can have enormous impact when it infiltrates too far into the network. When this occurs, it also seriously magnifies the impact of increasing task complexity. However, in the lower right graph, SENSIX performs better than Maté at scales of greater than

1000 nodes, when the CORBA side occupies three out of five and then four out of six tiers. The lower left graph shows how task complexity contributes, when CORBA occupied 5 of the 6 tiers. The top graphs in this figure show that these interactions can be controlled better by keeping the percentage of CORBA infiltration steady as the network scales.

Figure 5.6 shows the effect of varying CORBA penetration into the network. Here this is varied by tier depth and by corresponding percentage. At scales below 1000 nodes, it is apparent that the tiered approach has difficulty balancing the increased throughput of CORBA at all. Again with CORBA as a strict percentage of the network this is a linear relationship.

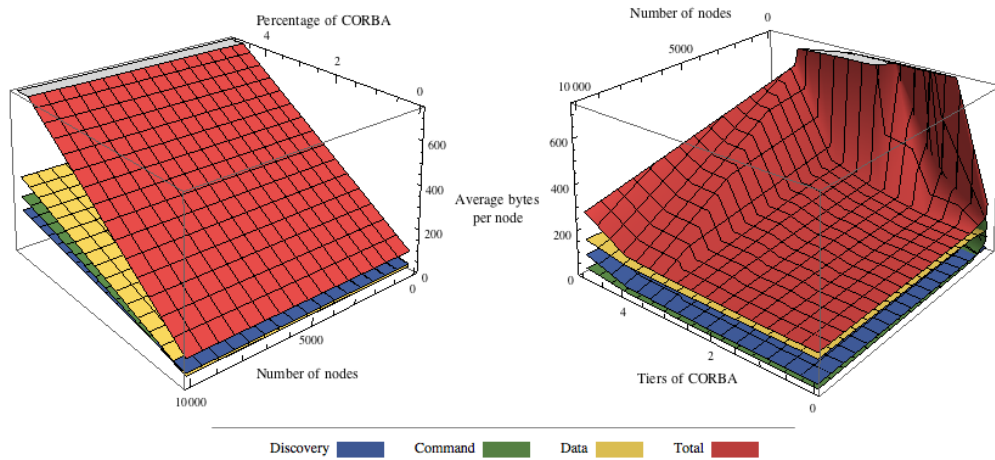


Figure 5.6: CORBA influence on performance.

As Figure 5.5 shows, it is clear that Maté would serve well as a benchmark for CORBA network infiltration. However, the above mentioned variables of task complexity, data return volume, aggregation, and capabilities reported by SENSIX all have disparate effects of these middleware.

Table 5.2 represents the range of the percentage of allowable CORBA network infiltration when benchmarked against each of the three alternative middleware. Here

the minimum network scale is 1000 nodes. CORBA infiltration is most restricted by extensive discovery (many capabilities), particularly large data sets and poor aggregation. The minimum values here represent special circumstances which are favorable to the middleware in question, so the median is a better benchmark for a CORBA upper bound. The TinyDB maximum is also a highly unfavorable circumstance and not reflective of a valid benchmark.

	Min	Median	Max
Maté	0.4	1.1	4.2
TinyDB	0.7	4.5	20.3
Agilla	1.0	8.9	14.7

Table 5.2: Percentage of allowable CORBA network infiltration given different middleware benchmarks.

In general, a CORBA infiltration of about one to two percent is the best guideline. It is worth repeating that this comparison is for *resource-bound* CORBA nodes. Also note that this benchmark considers identical resource availability (iMote versus iMote). Where node resources become less restrictive, this upper bound too relaxes. When node constraints disappear, this upper bound also disappears. Furthermore, this evaluation examines CORBA using IIOP; use of the more efficient LW-IOP is certain to produce better performance.

From these experiments, there appears to be no upper bound on the scalability of SENSIX, however, this may be partially an artifact of simulation. In [35], Ganesan et al. found that a relatively small scale network of approximately 150 nodes exhibited irregular and sometimes counter-intuitive wireless behavior. This complexity and irregularity is not well-reflected in simulation. The topology of the SENSIX reference implementation, deployed in reality, would minimize those effects, but not eliminate them. Properly optimized network protocols go a long way to mitigating these problems of long links, deep fades and asymmetry.

SENSIX software scalability may be affected by high node density. With the density bounded by the $n_j \approx \sqrt{n_i}$ rule, this topology experiences no degradation with scale. However, other, more dense topologies might. Thanks to the low data rate of mote communications, such degradation will be bound by wireless throughput limitations, and not computational limitations at the SENSIX Transformative layer.

This chapter found that, in comparison to three leading WSN middleware implementations,

- the SENSIX task representation is smaller and thus more efficient,
- the SENSIX middleware engine is more compact and more extensible,
- mote-only SENSIX uses less bandwidth over a variety of tasks,
- SENSIX continues to perform well despite variation in task complexity, data volume and data aggregation and
- if kept within a small percentage of the network, the CORBA side of SENSIX is competitive in terms of network efficiency.

Although a hierarchical topology was used in the SENSIX reference implementation, it is not a requirement. Other topologies should not see any degradation of performance beyond that induced by the wireless network.

This chapter evaluated SENSIX performance analytically for its task representation complexity and empirically through simulation for its messaging overhead. In this case-study evaluation, SENSIX out-performs several WSN middleware. Additionally, the overhead of the CORBA side of SENSIX proves to be manageable.

5.5. *COMMODITY TRADEOFF*

The final chapter discusses incomplete portions of the SENSIX framework and further directions for improvement.

Chapter 6

Conclusions and Extensions to SENSIX

The preceding chapters have described the SENSIX architecture, its reference implementation, and its analytical and empirical performance. These performance evaluations show that the SENSIX middleware provides object-oriented flexibility and data-centric efficiency that scales better than leading WSN middleware paradigms, supporting the hypothesis of this dissertation. There is one caveat: SENSIX requires balance to perform well, specifically the CORBA side of SENSIX cannot be permitted to dominate more than a small portion of the network, unless that portion is constraint-free (or nearly so). As an interesting side note, the Maté middleware offers performance close to SENSIX, and the customizable virtual machine paradigm is similar to the SENSIX approach. Unfortunately Maté does not extend its VM paradigm beyond the mote network, whereas SENSIX explicitly does make this extension. This high-level object-oriented abstraction coupled with low-level data-centric efficiency is unique to SENSIX.

Thus, the primary contributions of SENSIX and this dissertation are:

- a bridging facility between mote software and object-oriented software systems,
- a means of defining WSN applications independent of programming language, network protocol, or mote infrastructure/operating system,
- a reference implementation of a tasking mini-language embodying task decomposition and data aggregation and
- an efficient means of achieving very large scale sensing networks.

There are some secondary, incidental contributions also:

- an extended TinyOS-based WSN simulator, capable of simulating several hundred thousand, perhaps even millions of motes and
- a hierarchical supersystem deployment-planning tool.

6.1 Unfinished aspects of SENSIX

SENSIX is a work in progress. Unfortunately, the mote IDL compiler, a key piece of SENSIX, is not complete. Attempting to use the official CORBA 3.0 grammar has broken this tool, which is currently based on the Spirit library from the Boost C++ template library collection. The reference implementation was completed by hand.

Although Spirit is easy to write for (the grammar looks very similar to Extended Backus-Naur Form), being template-based it is nearly impossible to debug since symbol resolution is highly obfuscated. Continuing work on SENSIX will switch to Lex/Yacc or ANTLR to complete a CORBA 3.0 compatible mote compiler.

Other issues are both less pressing and less difficult. The SENSIX common code is fully fleshed out for Java, but not complete for the C++ or C language versions, nor are there any other languages included yet. There are both TinyOS and Mantis versions of the SENSIX common code, but there is not yet any Mantis Transformative layer, and it might be desirable to include other mote infrastructures as well. TinyOS 1.x is fully supported, but TinyOS 2.x support is incomplete.

6.2 Extensions to SENSIX

The object-oriented side of SENSIX is not limited to just CORBA. Rather CORBA demonstrates that this concept of mixing paradigms can be applied with any distributed middleware. Although it is very Microsoft-specific, SENSIX can work with the .NET framework and its Windows Communication Framework (WCF), Microsoft's latest competitor to CORBA and descendant of DCOM. Likewise, the Java language-specific Java RMI and Enterprise Java Beans (EJB) could quite easily replace the CORBA portion of SENSIX. Web services (SOAP, REST, WSRF and the like) are also a possibility. Each of these alternatives has its advantages and its drawbacks – lack of efficiency and/or platform universality being common to most.

One of the major advantages of CORBA, of course, is that it has no platform limitations and a very wide variety of languages from which to choose. ICE from ZeroC can also be a CORBA substitute in SENSIX with very little alteration. The only material differences are (a) ICE is not bound by CORBA standards, (b) ICE focuses hard on performance, and (c) ICE follows an open source development model with ZeroC as the final arbiter of conflicts. Item (a), while it provides opportunities for improvement, may actually be detrimental because it creates vendor lock-in. Item (b) is nothing new, most CORBA vendors emphasize performance in their implementations and it is a common selling-point. As for the effectiveness of item

(c), only time will tell if design-by-committee is actually worse than a single-vendor solution (which ICE will continue to be so long as ZeroC is the final arbiter of change). Nonetheless, SENSIX could very easily integrate with ICE.

The future of WSNs is certain to see increased miniaturization; Moore's Law is alive and well at the scale of the mote. The Sun SPOT, approximately the same size as a MICAz, boasts a 32 bit ARM9 core that runs the Squawk Java VM directly on the hardware (without an intervening operating system) [97]. Squawk is Java 2 Micro Edition (J2ME) Connected Limited Device Configuration specification (CLDC) compliant, which means that CORBA may be out of reach, but Java RMI is not [103]. Squawk introduces an interpreted, object-oriented run-time and an IPv6 stack at the mote level. As the cubic centimeter scale lowers its constraints, cubic millimeter scale devices fill the high constraint niche. The Spec node prototype has an 8-bit RISC core, 3K of memory, ADC, radio, UART and SPI all in a 5 mm² chip [48]. In short, as long as there is utility in highly constrained sensor nodes, there will be a use for SENSIX.

6.3 Autonomy for Ubiquitous Systems

Blending WSN data into other software systems with SENSIX is just the first step. Ultimately, as WSNs are integrated into large ubiquitous systems, the complexity of software will become overwhelming. These systems therefore need to become *autonomic* computing resources, being self-configuring, self-optimizing, self-healing, self-adapting and self-protecting [60, 59]. Such auto-management concepts already permeate WSN research in the goals of being self-organizing, self-healing, self-optimizing, self-diagnostic and self-sustaining [92]. Software that realizes these sensor network concepts must also be extended to the larger supersystem. SENSIX has preliminary support for such system self-regulation. On the mote side, both the

energy used by a functor and the time it takes to respond can be tracked, and this information propagated and accumulated in the functor object. In addition, SENSIX can close the loop on the task decomposition/data aggregation flow (see Figure 4.2 on page 49) by using aggregated data plus metadata, such as energy usage, as feedback to a decision-making engine, which in turn guides a task planning engine to retask the sensory network. Currently this consists of PROLOG for decision-making and Simple Hierarchical Ordered Planner for Java (JSHOP) for task planning, but these are not yet hooked into the SENSIX data flow. SENSIX also allows for task prioritization, which could be used by motes that are low on energy to decide to refuse low priority tasks, thereby extending lifespan. Such a facility effectively yields a self-scheduling of resources.

This proposed autonomic system would also require additional services, such as are the norm for distributed operating systems. These common OS services are interprocess communications (usually in the form of RPC), synchronization, resource scheduling, a file system, naming and security [98]. Of these, SENSIX already covers IPC and scheduling. There are WSN components that offer time synchronization and security, but these need to be tied-in with the larger system. While some services, such as a distributed file system, do not and should not have a place in the mote portion of this system, others such as node naming are vital. In the case of naming, any implementation is likely domain-specific and easily resides in SENSIX's Access layer, as does mote security. However, whereas distributed operating systems are supposed to emphasize transparency, anywhere WSNs are involved it is important to not mask too much – particularly location and faults. The occurrence of failures and errors can be even more vital than sensory data. SENSIX has support for that too, as mentioned in Chapter 4.

6.3. AUTONOMY FOR UBIQUITOUS SYSTEMS

SENSIX exposes networked sensors as distributed objects, offering customized and seamless vertical integration with object-oriented systems. This vertical integration benefits wireless sensor networks by allowing high-level abstractions to administer and carefully infiltrate the sensor network, and by providing a new toolset with which to develop WSN software. This vertical integration also has the potential to provide benefits in the opposite direction as well, pulling WSN-style autonomy into wider, ubiquitous systems.

Appendices

Appendix 1

SENSIX IDL

```
#define IDL
#include "sensix.h"

module sensix {
    union Data switch (char) {
        case 'd':
        case 'i':
        case 'u':
        case 'x':
            long long ireresult;
        case 'f':
        case 'e':
        case 'g':
        case 'a':
            double fresult;
    };
};
```

```
typedef sequence <Data> Series;

enum FunctorError {
    Refused, Incapable
};

typedef sequence <FunctorError> ErrorList;

interface Functor; // forward declaration

typedef sequence <Functor> FunctorList;

// basic distributed functional object
interface Functor {
    const octet ID = INVALID;
    readonly attribute octet identifier;
    readonly attribute unsigned long sequencer;

    attribute octet priority;
    attribute FunctorList subfunctors;

    attribute Series results;
    attribute ErrorList errors;
    string asString();
};
```



```
////////////////////////////////////
```

```
// does actual data collection
```

```
interface Capability {  
    readonly attribute octet identifier;  
  
    // load local data into a Functor  
    void acquire(in Functor f);  
};
```

```
// 'client' side
```

```
interface Response {  
    // callback from remote server  
    oneway void aggregate(in Functor f);  
  
    // tasking out to local client; returns error code  
    long devolve(in Functor f);  
  
    // cancel out to local client; returns error code  
    long detask(in Functor f);  
};
```

```
// 'server' side
interface Request {
    // tasking out to remote server
    oneway void apply(in Functor f, in Response callback);

    // cancel out to remote server
    oneway void cancel(in Functor f);

    // callback from local client
    void dataready(in Functor f);

    // add/register local capability
    void addCapability(in octet type, in Capability c);
};
```

```
////////////////////////////////////
```

```
typedef sequence <Request> ReqList;
```

```
// provides Functor tracking
interface Task {
    readonly attribute Functor f;
    readonly attribute Functor superf;
    readonly attribute Response callback;
    readonly attribute ReqList subtasks;
```

```
attribute long numPeers;  
attribute boolean cancelled;  
  
void addSubtask(in Request r);  
};  
};
```

Appendix 2

Sensing Mini-Language

The following table shows the SENSIX reference implementation mini-language encoding. As described in Chapter 4, the functors here are divided into the categories of data, series, aggregate, network and location. The components column lists optional (in parenthesis) and required parameters for each, which are themselves categorized as general metadata, or sensor types. The data, series and aggregate functors are the Morphic (domain-specific) portion of this particular SENSIX architecture, while the network and location functors fall within the Access layer. Not shown here is an internal power functor that tracks battery status, which can be seen in the IDL code of Appendix A3.

Appendix 2. Sensing Mini-Language

Level	Symbol	Encoding	Meaning	Components
data	α	0xE1	basic signal, a single sample	$\varsigma, (\nu, \iota)$
	β	0xE2	local maxima (peak sensing)	ε^+ and/or ε^- , $\varsigma, (\nu, \iota)$
series	Θ	0xC8	time series	α or β, N , and $\tau, (\iota)$
	Ψ	0xD8	spatial series	α or β, N , and $\rho, \theta, T, (\iota)$
aggregate	I	0xDF	recite (list)	Θ or $\Psi, (\iota)$
	Σ	0xD3	sum	Θ or $\Psi, (\iota)$
	Δ	0xC4	max variation (difference)	Θ or $\Psi, (\iota)$
	\bar{x}	0xAA	mean	Θ or $\Psi, (\iota)$
	σ	0xF3	statistical deviation	Θ or $\Psi, (\iota)$
	Λ	0xCB	aggregate function	bytecode and Θ or $\Psi, (\iota)$
network	Γ	0xC3	ancestors	N
	A	0xC1	siblings	N
	K	0xCA	descendants	N
location	T	0xD4	GPS position	$\phi, \lambda, (\zeta)$
	ϕ	0xF6	latitude	—
	λ	0xEB	longitude	—
	ζ	0xE6	elevation/altitude	—
	ρ	0xF1	polar distance (in meters)	—
	θ	0xE8	polar angle (in radians)	—
metadata	ς	0xF2	sensor type (see sensors)	light, temp, etc.
	ι	0xE9	hierarchical level	—
	ν	0xED	sample frequency	—
	ε^+	0xE5	upper threshold	—
	ε^-	0xA7	lower threshold	—
	Φ	0xD6	energy cost	—
	τ	0xF4	time (duration/cost)	—
	δ	0xE4	task priority	—
	N	0xCD	series/array size	—
sensors	light	0x01	ambient light	—
	temp	0x02	thermistor	—
	accel	0x03	accelerometer	—
	mag	0x04	magnetometer	—
	mic	0x05	microphone	—
	humid	0x06	humidity	—
	press	0x07	barometric pressure	—
		0x08 - 0x3F	... other sensors	—

Table A2.1: Task and aggregation domain-specific language

Appendix 3

Generic Sensing IDL and Discovery IDL

Sensing IDL

```
#include "sensix.idl"

module sensix {
    module sensing {

        interface Location : Capability {
            const octet ID = GPS;

            readonly attribute double latitude;
            readonly attribute double longitude;
            readonly attribute double altitude;
            readonly attribute unsigned long long time;
        };
    };
};
```

```
interface Power : Capability {
    const octet ID = BATTERY;

    readonly attribute double ratio;
};

////////////////////////////////////

interface Sensory : Functor {
    const octet ID = INVALID;

    readonly attribute octet level;
    readonly attribute octet sensor;
    attribute unsigned long long timeused;
    attribute double energyused;
};

typedef sequence <Sensory> SensoryList;

////////////////////////////////////
// Flexible component interfaces:
```

```
interface Sense : Sensory {
    const octet ID = ALPHA;

    attribute double rate;
};

interface PeakSense : Sense {
    const octet ID = BETA;

    attribute double highthreshold;
    attribute double lowthreshold;
};

interface Collection : Sensory {
    attribute Sensory sense;
};

typedef sequence <Collection> CollectionList;

interface TimeSeries : Collection {
    const octet ID = THETA;

    attribute unsigned long long duration;
};
```



```
interface SpatialSeries : Collection {
    const octet ID = PSI;

    attribute double distance;
    attribute double angle;
};
```

```
interface Aggregate : Sensory {
    readonly attribute SensoryList senses;
    attribute CollectionList collectors;
};
```

```
interface Recite : Aggregate {
    const octet ID = IOTA;
};
```

```
interface Sum : Aggregate {
    const octet ID = SUMMA;
};
```

```
interface Delta : Aggregate {
    const octet ID = DELTA;
};
```

```
interface Mean : Aggregate {
```

```

    const octet ID = BARX;
};

interface Sigma : Aggregate {
    const octet ID = SIGMA;
};

typedef sequence <octet> ByteCode;

interface Lambda : Aggregate {
    const octet ID = LAMBDA;

    attribute ByteCode code;
};
};
};

```

Discovery IDL

```

#include "sensix.idl"

module sensix {
    module discovery {
        #define IDL
        #include "sensix.h"
        #include "discovery_service.h"
    }
}

```

```

typedef sequence <octet> CapabilityList;
typedef sequence <NodeId> NodeList;

exception DiscoveryException {
    DiscoveryError error;
    string description;
};

typedef sequence <Request> RequestList;

////////////////////////////////////

struct AnnounceHeader { // size 8
    octet magic; // 'D'
    octet version; // 0xMm (Major, minor)
    octet flags; // 0 - big endian, 1 - little endian
    octet announceType; // 1, 2, 3, or 4 (see below)
    unsigned long announceSize; // see below per type
};

struct Announce { // type 1, size 5
    NodeId nodeId;
    octet hLevel;
};

```

```

struct Report { // type 2, size 6
    NodeId nodeId;
    octet hLevel;
    octet cType;
};

struct Require { // type 3, size 10
    NodeId nodeId;
    octet hLevel;
    octet cType;
    NodeId targetId;
};

struct Share { // type 4, size 6 + strlen(ior)
    NodeId nodeId;
    octet hLevel;
    octet cType;
    string ior;
};

////////////////////////////////////

// hierarchical context
interface Family {
    const octet ID = INVALID;

```

```
    readonly attribute NodeList nodes;

    // relation query
    boolean findNode(in NodeId id);
};

// units higher in the hierarchy
interface Ancestors: Family {
    const octet ID = ANCESTORS;
};

// units at your level or below in the hierarchy
interface Peers: Family {
    const octet ID = INVALID;
    readonly attribute CapabilityList capabilities;

    // query peers for a capability
    RequestList queryNetwork(in octet capability)
        raises (DiscoveryException);

    // query a particular peer
    Request queryNode(in NodeId id, in octet capability)
        raises (DiscoveryException);
};
```

```

// units at your level in the heirarchy
interface Siblings: Peers {
    const octet ID = SIBLINGS;
};

// units lower in the heirarchy
interface Descendants: Peers {
    const octet ID = DESCENDANTS;
};

////////////////////////////////////

// collection of all related nodes
interface Others {
    readonly attribute Ancestors ancestors;
    readonly attribute Siblings siblings;
    readonly attribute Descendants descendants;

    // query node existence
    boolean findNode(in NodeId id);

    // query all siblings and descendants
    RequestList queryNetwork(in octet capability)
        raises (DiscoveryException);

```

```
// query one particular sibling or descendant
Request queryNode(in NodeId id, in octet capability)
    raises (DiscoveryException);
};

// this unit
interface Self {
    readonly attribute octet level;
    readonly attribute NodeId identifier;
    readonly attribute CapabilityList capabilities;

// register a capability with the discovery service
void registerObject(in octet capability, in Request obj)
    raises (DiscoveryException);

// proxies:
boolean findNodeInFamily(in NodeId id)
    raises (DiscoveryException);
RequestList queryFamilyNetwork(in octet capability)
    raises (DiscoveryException);
Request queryFamilyNode(in NodeId id, in octet capability)
    raises (DiscoveryException);

boolean findNodeInDescendants(in NodeId id)
    raises (DiscoveryException);
```

```

RequestList queryDescendantNetwork(in octet capability)
    raises (DiscoveryException);
Request queryDescendantNode(in NodeId id, in octet capability)
    raises (DiscoveryException);
CapabilityList descendantCapabilities()
    raises (DiscoveryException);
NodeList descendantNodes()
    raises (DiscoveryException);

boolean findNodeInSiblings(in NodeId id)
    raises (DiscoveryException);
RequestList querySiblingNetwork(in octet capability)
    raises (DiscoveryException);
Request querySiblingNode(in NodeId id, in octet capability)
    raises (DiscoveryException);
CapabilityList siblingCapabilities()
    raises (DiscoveryException);
NodeList siblingNodes()
    raises (DiscoveryException);

boolean findNodeInAncestors(in NodeId id)
    raises (DiscoveryException);
NodeList ancestorNodes()
    raises (DiscoveryException);
};
};
};

```


References

- [1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. In *Proceedings of the IEEE International Conference on Distributed Sensor Networks (ICDCS'04)*, 2004.
- [2] Sangim Ahn and Kiwon Chong. Building a Bridge for Heterogeneous Sensor Networks. In *Proceedings of the Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous System and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*, 2006.
- [3] Abdulmalik Al-Gahmi, Ciju John, Jonathan E. Cook, and Bo Du. Supporting Quick and Dirty CORBA Introspection and Manipulation. In *Proceedings of the 10th Working Conference on Reverse Engineering WCRE'03*, 2003.
- [4] Adil Al-Yasiri and Alan Sunley. Data aggregation in wireless sensor networks using the SOAP protocol. In *Proceedings of the Conference on Sensors and their Applications (SENSORS'07)*, volume 76 of *Journal of Physics Conference Series*, 2007.
- [5] Todd R. Andel and Alec Yasinac. On the Credibility of Manet Simulations. *Computer*, 39(7):48–54, 2006.
- [6] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A Line in the Sand: a wireless sensor network for target detection, classification, and tracking. *Computer Networks*, 46(5):605–634, 2004.

- [7] Anish Arora, Rajiv Ramnath, Emre Ertin, Prasun Sinha, Sandip Bapat, Vinayak Naik, Vinod Kulathumani, Hongwei Zhang, Hui Cao, Mukundan Sridharan, Santosh Kumar, Nick Seddon, Chris Anderson, Ted Herman, Nishank Trivedi, Chen Zhang, Mikhail Nesterenko, Romil Shah, Sandeep Kulkarni, Mahesh Aramugam, Limin Wang, Mohamed Gouda, Young ri Choi, David Culler, Prabal Dutta, Cory Sharp, Gilman Tolle, Mike Grimmer, Bill Ferreira, and Ken Parker. ExScal: Elements of an Extreme Scale Wireless Sensor Network. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, pages 102–108, Washington, DC, USA, 2005.
- [8] Alexander B. Arulanthu, Carlos O’Ryan, Douglas C. Schmidt, Michael Kircher, and Jeff Parsons. The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, volume 1795 of *Lecture Notes in Computer Science*, April 2000.
- [9] Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The Abstract Task Graph: A Methodology for Architecture-Independent Programming of Networked Sensor Systems. In *Proceeding of the Workshop on End-to-End, Sense-and-Respond Systems, Applications, and Services (EESR'05)*, June 2005.
- [10] Sandip Bapat, Vinodkrishnan Kulathumani, and Anish Arora. Analyzing the Yield of ExScal, a Large-Scale Wireless Sensor Network Experiment. In *Proceedings of the 13TH IEEE International Conference on Network Protocols (ICNP'05)*, pages 53–62, Washington, DC, USA, 2005.
- [11] Rimon Barr, John C. Bicket, Daniel S. Dantas, Bowei Du, T.W. Danny Kim Bing Zhou, and Emin Gun Sirer. On the Need for System-Level Support for Ad hoc and Sensor Networks. *ACM SIGOPS Operating Systems Review*, 36(2), 2002.
- [12] Mark Bedworth and Jane O’Brien. The Omnibus Model: A New Model of Data Fusion? *IEEE Aerospace and Electronic Systems Magazine*, 15(4), April 2000.
- [13] J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8), August 1986.
- [14] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. MAN-TIS OS: An Embedded Multithreaded Operating System for Wireless Micro

Sensor Platforms. *Mobile Networks and Applications: Special Issue on Wireless Sensor Networks*, 10(4), January 2005.

- [15] Pratik K. Biswas. Architecting multi-agent systems with distributed sensor networks. In *International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, April 2005.
- [16] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. In *Proceedings of the Second International Conference on Mobile Data Management (MDM'01)*, January 2001.
- [17] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys'03)*, 2003.
- [18] John Boyd. A Discourse on Winning and Losing, 1987. Maxwell AFB lecture.
- [19] Adam Buble, Lubomír Bulej, and Petr Tůma. CORBA Benchmarking: A Course with Hidden Obstacles. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.
- [20] Christian Buckl, Stephan Sommer, Andreas Scholz, Alois Knoll, and Alfons Kemper. Generating a Tailored Middleware for Wireless Sensor Network Applications. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, 2008.
- [21] Min Chen, Taeyoung Kwon, Yong Yuan, and Victor C.M Leung. Mobile agent based wireless sensor networks. *Journal of Computers*, 1(1), April 2006.
- [22] B. Claudel, G. Huard, and O. Richard. TakTuk, Adaptive Deployment of Remote Executions. In *Proceedings of the International Symposium on High Performance Distributed Computing (HPDC)*, 2009.
- [23] Flávia C. Delicato, Paulo F. Pires, Luiz Rust, Luci Pirmez, and José Ferreira de Rendez. Reflective Middleware for Wireless Sensor Networks. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC'05)*, March 2005.
- [24] Flávia Coimbra Delicato, Paulo F. Pires, Luci Pirmez, and Luiz Fernando Rust Da Costa Carmo. A Service Approach for Architecting Application Independent Wireless Sensor Networks. *Cluster Computing*, 8, 2005.

- [25] Flavia Coimbra Delicato, Paulo F. Pires, Luci Pirmez, and Luiz Fernando Rust da Costa Carmo. A Flexible Web Service based Architecture for Wireless Sensor Networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)*, 2003.
- [26] John Dilley. OODCE: A C++ Framework for the OSF Distributed Computing Environment. In *Proceedings of the USENIX 1995 Technical Conference*, 2005.
- [27] Adam Dunkels, Juan Alonso, Thiemo Voigt, Hartmut Ritter, and Jochen Schiller. Connecting Wireless Sensornets with TCP/IP Networks. In *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*, February 2004.
- [28] Adam Dunkels, Björn Groönvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (EmNetS-1)*, Tampa, Florida, November 2004.
- [29] Deborah Estrin, David Culler, Kris Pister, and Gaurav Sukhatme. Connecting the Physical World with Pervasive Networks. *IEEE Pervasive Computing*, 1(1):59–69, 2002.
- [30] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: scalable coordination in sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 263–270, 1999.
- [31] Anad Eswaran, Anthony Rowe, and Raj Rajkumar. Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*, December 2005.
- [32] Kevin Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '03)*, 2003.
- [33] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Mobile Agent Middleware for Sensor Networks: An Application Case Study. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN'05)*, April 2005.
- [34] Martin Fowler. A Pedagogical Framework for Domain-Specific Languages. *IEEE Software*, 26(4), 2009.

- [35] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. Complex Behavior at Scale: An Experimental Study of Low-Power Wireless Sensor Networks. Technical Report CSD-TR 02-0013, UCLA, 2002.
- [36] Victor Giddings. Not Your Father’s CORBA - An Architecture for Embedded and Real-Time Systems. *RTC*, October 2008.
- [37] Christopher D. Gill, Jeanna M. Gossett, David Corman, Joseph P. Loyall, Richard E. Schantz, Michael Atighetchi, and Douglas C. Schmidt. Integrated Adaptive QoS Management in Middleware: A Case Study. *Real-Time Systems*, 29(2-3), 2005.
- [38] N. A. B. Gray. Comparison of Web Services, Java-RMI, and CORBA Service Implementation. In *Proceedings of the Fifth Australasian Workshop on Software and System Architectures*, 2004.
- [39] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macroprogramming Wireless Sensor Networks using *Kairos*. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS’05)*, June 2005.
- [40] Piyush Gupta, Robert Gray, and P. R. Kumar. An Experimental Scaling Law for Ad Hoc Networks, May 2001. Univ. of Illinois at Urbana-Champaign.
- [41] Piyush Gupta and P. R. Kumar. The Capacity of Wireless Networks. *IEEE Transactions on Information Theory*, 46(2), March 2000.
- [42] Richard Guy, Ben Greenstein, John Hicks, Rahul Kapur, Nithya Ramanathan, Tom Schoellhammer, Thanos Stathopoulos, Karen Weeks, Kevin Chang, Lew Girod, and Deborah Estrin. Experiences with the Extensible Sensing System ESS. Technical report, Center for Embedded Network Sensing, University of California, Los Angeles, 2006.
- [43] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler, and Mani Srivastava. SOS: A dynamic operating system for sensor networks. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys’05)*, pages 163 – 176, 2005.
- [44] Qi Han and Nalini Venkatasubramanian. AutoSeC: An Integrated Middleware Framework for Dynamic Service Brokering. *IEEE Distributed Systems Online*, 2(7), 2001.
- [45] Wendi B. Heinzelman, Amy L. Murphy, Hervaido S. Carvalho, and Mark A. Perillo. Middleware to Support Sensor Network Applications. *IEEE Network*, January – February 2004.

- [46] Michi Henning. The rise and fall of corba. *ACM Queue*, 4(5), 2006.
- [47] Dan Hilbebrand. An Architectural Overview of QNX. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, 1992.
- [48] Jason Hill, Mike Horton, Ralph Kling, and Kakshman Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47(6), 2004.
- [49] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, 2000.
- [50] James Horey and Arthur B. Maccabe. Designing a Dynamic Middleware System for Sensor Networks. Technical Report TR-CS-2005-12, University of New Mexico Computer Science Department, 2005.
- [51] Svilen Ivanov, Andr Herms, and Georg Lukas. Experimental Validation of the ns-2 Wireless Model using Simulation, Emulation, and Real Network. In *Proceedings of the 4th Workshop on Mobile Ad-Hoc Networks (WMAN'07)*, 2007.
- [52] Chaiporn Jaikaeo, Chavalit Srisathapornphat, and Chien-Chung Shen. Querying and Tasking in Sensor Networks. In *Proceedings of SPIE, the International Society for Optical Engineering – Digitization of the Battlespace V and Battlefield Biomedical Technologies II*, volume 4037, 8 2000.
- [53] Jan Janeček. Efficient SOAP processing in embedded systems. In *Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)*, May 2004.
- [54] Xiaofan Jiang, Joseph Polastre, and David Culler. Perpetual Environmentally Powered Sensor Networks. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN'05)*, April 2005.
- [55] Ian Johnstone, James Nicholson, Babar Shehzad, and Jeff Slipp. Experiences from a wireless sensor network deployment in a petroleum environment. In *Proceedings of the 2007 International Conference on Wireless Communications and Mobile Computing (IWCMC'07)*, pages 382–387, 2007.
- [56] Matjaz Juric, Ales Zivkovic, and Ivan Rozman. Comparison of CORBA and Java RMI Based on Performance Analysis. In *Proceedings of (MHSS'98)*, 1998.

- [57] Cem Kaner and Walter P. Bond. Software Engineering Metrics: What Do They Measure and How Do We Know? In *Proceedings of the 10th International Software Metrics Symposium*, 2004.
- [58] Jaakko Kangasharju. Implementing the Wireless CORBA Specification. Master's thesis, University of Helsinki, May 2002. Laudatur Thesis.
- [59] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1), January 2003.
- [60] Jana Koehler, Chris Giblin, Dieter Gantenbein, and Rainer Hauser. On Autonomic Computing Architectures. Technical report, IBM ZRL Research Report 3487, 2003.
- [61] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6), 2002.
- [62] David Kotz, Calvin Newport, Robert S. Gray, Jason Liu, Yougu Yuan, and Chip Elliott. Experimental Evaluation of Wireless Simulation Assumptions. In *Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM'04)*, pages 78–82, 2004.
- [63] Andrea Kulakov, Danco Davcev, and Georgi Stojanov. Learning patterns in wireless sensor networks based on wavelet neural-networks. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, 2005.
- [64] S. Kurkowski, T. Camp, and M. Colagrosso. MANET Simulation Studies: the Incredibles. *SIGMOBILE Mobile Computing and Communications Review*, 9(4), October 2005.
- [65] Ron T. Lake. Distributed Event Driven Architectures for Evolutionary Sensor Fusion. In *Proceedings of SPIE, the International Society for Optical Engineering – Sensor Fusion: Architectures, Algorithms, and Applications II*, volume 3376, 1998.
- [66] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, April 2006.
- [67] Philip Levis and David Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the International Conference on Architectural Support*

for *Programming Languages and Operating Systems (ASPLOS'02)*, December 2002.

- [68] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys'03)*, pages 126 – 137, 2003.
- [69] Shuoqi Li, Sang H. Son, and John A. Stankovic. Event detection services using data service middleware in distributed sensor networks. In *Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks (IPSN'03)*, volume 2634 of *Lecture Notes in Computer Science*. Springer, 2003.
- [70] Suet-Fei Li, Roy Sutton, and Jan Rabaey. Low Power Operating System for Heterogeneous Wireless Communication Systems. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power 2001*, September 2001.
- [71] Benyuan Liu, Zhen Liu, and Don Towsley. On the Capacity of Hybrid Wireless Networks. In *Proceedings of the Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies (InfoCom'03)*, volume 2, April 2003.
- [72] Ting Liu and Margaret Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the Ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)*, 2003.
- [73] James Llinas, Christopher Bowman, Galina Rogova, Alan Steinburg, Ed Waltz, and Frank White. Revisiting the JDL Data Fusion Model II. In *Proceedings of the 7th International Conference on Information Fusion*, volume 2, 2004.
- [74] Konrad Lorincz, David Malan, Thaddeus Fulford-Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoff Mainland, Steve Moulton, and Matt Welsh. Sensor Networks for Emergency Response: Challenges and Opportunities. *IEEE Pervasive Computing*, October 2004.
- [75] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. *ACM SIGOPS Operating System Review*, 36(SI), 2002.
- [76] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings*

of the ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02), Atlanta, GA, September 2002.

- [77] M. S. Nassr, J. Jun, S. J. Eidenbenz, J. R. Frigo, A. A. Hansson, A. M. Mielke, and M. C. Smith. Development, implementation, and experimentation of parametric routing protocol for sensor networks. In *Proceedings of SPIE - the International Society for Optical Engineering*, volume 6394, 2006.
- [78] Object Management Group. Lightweight services. <http://www.omg.org/cgi-bin/doc?formal/04-10-01>, October 2004. Version 1.0.
- [79] Object Management Group. Real-time CORBA. <http://www.omg.org/cgi-bin/doc?formal/05-01-04>, January 2005. Version 1.2.
- [80] Object Management Group. Wireless Access and Terminal Mobility. <http://www.omg.org/cgi-bin/doc?formal/2005-05-02>, May 2005. Version 1.2.
- [81] Object Management Group. Common Object Request Broker Architecture - for embedded. <http://www.omg.org/cgi-bin/doc?ptc/2006-05-01>, May 2006. Draft Adopted Specification.
- [82] Frank Oldewutel and Petri Mähönen. Neural Wireless Sensor Networks. In *Proceedings of the International Conference on Systems and Networks Communications (ICSNC '06)*, October 2006.
- [83] Mike Olson and Uche Ogbuji. Messaging technologies compared. *IBM developerWorks*, July 2002.
- [84] Carlos O’Ryan, Douglas C. Schmidt, and J. Russel Noseworthy. Patterns and Performance of a CORBA Event Service for Large-Scale Distributed Interactive Simulations. *International Journal of Computer Systems Science and Engineering*, 17(2), 2002.
- [85] Michael Pfeiffer and Josef Pichler. A Comparison of Tool Support for Textual Domain-Specific Languages. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, October 2008.
- [86] Hairong Qi, Xiaoling Wang, and S. Sitharama Iyengar. Multisensor Data Fusion in Distributed Sensor Networks Using Mobile Agents. In *Proceedings of 5th International Conference on Information Fusion*, 2001.
- [87] Huang Qi, Xing Tao, and Liu Hai Tao. Vehicle Classification in Wireless Sensor Networks Based on Rough Neural Network. In *Proceedings of the Second IASTED International Conference on Advances in Computer Science and Technology (ACST'06)*, 2006.

- [88] Umakishore Ramachandran, Rajnish Kumar, Matthew Wolenetz, Brian Cooper, Bikash Agarwalla, Junsuk Shin, Phillip Hutto, and Arnab Paul. Dynamic Data Fusion for Future Sensor Networks. *ACM Transactions on Sensor Networks*, 2(3), August 2006.
- [89] Eric S. Raymond. *The Art of UNIX Programming*. Addison Wesley, 2004.
- [90] Kay Römer, Oliver Kasten, and Friedemann Mattern. Middleware Challenges for Wireless Sensor Networks. *Mobile Computing and Communications Review*, 6(2), 2002.
- [91] Mark T. Keane Rónán Mac Ruairí. An energy-efficient, multi-agent sensor network for detecting diffuse events. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'07)*, 2007.
- [92] Linnyer Beatrys Ruiz, Jose Marcos Nogueira, and Antonio A. F. Loureiro. Sensor Network Management. In Mohammad Ilyas and Imad Mahgoub, editors, *Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems*, chapter 3. CRC Press, 2005.
- [93] G. Sudha Sadasivam and Dr. A. Chitra. Certain Improvements in Marshalling. *Academic Open Internet Journal*, 11, September 2004.
- [94] Douglas C. Schmidt and Steve Vinoski. Object Interconnections: An Overview of the OMG CORBA Messaging Quality of Service (QoS) Framework. *C++ Report*, 12(3), March 2000.
- [95] Timothy J. Shepard. A Channel Access Scheme for Large Dense Packet Radio Networks. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '96)*, August 1996.
- [96] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley and Sons, 1996.
- [97] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java on the Bare Metal of Wireless Sensor Devices: The Squawk Java Virtual Machine. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE'06)*, June 2006.
- [98] Pradeep K. Sinha. *Distributed Operating Systems: Concepts and Design*. Wiley-IEEE Press, 1996.

- [99] Eduardo Souto, Germano Guimarães, Glauro Vasconcelos, Mardoqueu Vieira, Nelson Rosa, and Carlos Ferraz. A message-oriented middleware for sensor networks. In *MPAC '04: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, 2004.
- [100] Malcolm Spense. CORBA still delivering, May 2006. Keynote speech at OMG Spring Technical Meeting.
- [101] Chavalit Srisathapornphat, Chaiporn Jaikaeo, and Chien chung Shen. Sensor information networking architecture and applications. *IEEE Personal Communications*, 8, 2001.
- [102] Alan N. Steinburg, Christopher L. Bowman, and Franklin E. White. Revisions to the JDL Data Fusion Model. In *Proceedings of SPIE, the International Society for Optical Engineering – Sensor Fusion: Architectures, Algorithms, and Applications III*, volume 3719, 1999.
- [103] Sun Microsystems, Inc. Connected Limited Device Configuration Specification, March 2003. Version 1.1, Java 2 Platform, Micro Edition (J2ME).
- [104] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An Analysis of a Large Scale Habitat Monitoring Application. In *Proceedings of the Second International Conference on Embedded Networked Sensor Systems (SenSys'04)*, November 2004.
- [105] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. Lessons From A Sensor Network Expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN'04)*, 2004.
- [106] Andrew S. Tanenbaum, Chandana Gamage, and Bruno Crispo. Taking Sensor Networks from the Lab to the Jungle. *IEEE Computer*, 39(8), August 2006.
- [107] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. In *Proceedings of the Third International Conference on Embedded Networked Sensor Systems (SenSys'05)*, pages 51–63, 2005.
- [108] Petr Tůma and Adam Buble. Open CORBA Benchmarking. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'01)*, July 2001.

- [109] F. J. Villanueva, D. Villa, F. Moya, J. Barba, F. Rincón, and J. C. López. Lightweight Middleware for Seamless HW-SW Interoperability, with Application to Wireless Sensor Networks. In *Proceedings of the Conference on Design, Automation, and Test in Europe*, 2007.
- [110] Steve Vinoski. Scalability Issues in CORBA-based Systems. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'00)*, 2000.
- [111] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A Note on Distributed Computing. In *Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet (MOS'96)*, 1997.
- [112] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 381–396, 2006.
- [113] Geoffrey Werner-Allen, Konrad Lorincz, Matt Welsh, Omar Marcillo, Jeff Johnson, Mario Ruiz, and Jonathan Lees. Deploying a Wireless Sensor Network on an Active Volcano. *IEEE Internet Computing*, 10(2), March – April 2006.
- [114] Franklin E. White. A model for data fusion. In *Proceedings of the 1st National Symposium on Sensor Fusion*, volume 2, 1998.
- [115] Kaixin Xu, Xiaoyan Hong, and Mario Gerla. An Ad Hoc Network with Mobile Backbones. In *Proceedings of the IEEE International Conference on Communications (ICC'02)*, volume 5, 2002.