EmStar: An Environment for Developing Wireless Embedded Systems Software[*]

J. Elson[†], S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin

Center for Embedded Networked Sensing, University of California, Los Angeles

March 24, 2003

Abstract

Recently, increasing research attention has been directed toward wireless sensor networks: collections of small, low-power nodes, physically situated in the environment, that can intelligently deliver high-level sensing results to the user. As the community has moved into more complex design efforts—large-scale, long-lived systems that truly require self-organization and adaptivity to the environment—a number of important software design issues have arisen. The data reduction process is critical for meeting energy and channel capacity constraints by preventing raw sensor time-series from being delivered. However, the lack of raw data prevents the data reduction process itself from being evaluated. Simulation is difficult to apply; the network's physical situatedness makes it sensitive to subtleties of sensors and wireless communication channels that are difficult to model. A second problem that arises is that the traditional layered protocol stack, designed to emphasize conceptual abstraction and reusability, has too high of an efficiency cost in this domain where efficiency is paramount.

In this paper, we describe EmStar, a Linux-based software framework that addresses these issues. EmStar's novel execution environment encompasses pure simulation, true in-situ deployment, and a hybrid mode that combines simulation with real wireless communication and sensors situated in the environment. Each of these modes run the same code and use the same configuration files, allowing developers to seamlessly iterate between the convenience of simulation and the reality afforded by physically situated devices. We also describe a modular programming model that allows domain knowledge in one module to affect the modules around it, without sacrificing the advantages of reusability and abstraction that strict layering provides. Using several case studies, we show how EmStar has been applied to building real sensor network services.

[*]UCLA Center for Embedded Networked Sensing, Technical Report 9, March 2003. Under submission to SOSP 2003: the 19th ACM Symposium on Operating Systems Principles

[†]Correspondence author: jelson@cs.ucla.edu. EmStar development mailing list: emstar-users@cens.ucla.edu

1 Introduction

The recent proliferation of small, low-power hardware platforms that integrate sensing, computation, and wireless communication has led to widespread interest in the design of wireless sensor networks. Such networks are envisioned to be large-scale, dense deployments in environments where traditional centrally-wired sensors are impractical. For example, ubiquitous wiring is infeasible for microclimate studies [3, 13], groundwater contaminant monitoring, precision agriculture, and condition-based maintenance of machinery in complex environments.

A primary factor in the design of such sensor network systems is their finite energy source. As communication is the primary consumer of energy [14], and the low power budget of the nodes precludes communication beyond a short distance, research has been focused on maximizing local processing and minimizing the overhead of collaboration. Emerging designs allow users to task the network with a high-level query such as "notify me when a large region experiences a temperature over 100 degrees" or "report the location where the following bird call is heard" [8, 12].

The use of local processing, hierarchical collaboration, and domain knowledge to convert data into increasingly distilled and high-level representations—or, *data reduction*—is key to the energy efficiency of the system. In general, a perfect system will reduce as much data as possible as early as possible, rather than incur the energy expense of transmitting raw sensor values further along the path to the user. For a system designer, there is an unfortunate paradox intrinsic to this ideal: the data that must be discarded to meet the energy and channel capacity constraints are necessary for the evaluation and debugging of the data reduction process itself. How can a designer evaluate a system where, by definition, the information necessary for the evaluation is not available?

A second paradox arises in the structure of the software itself. As in any software system, it is important to promote code reuse. In sensor networks, the difficulty we described in evaluating a system's correctness makes it especially important to reuse algorithms that are known to be correct. However, the communication efficiency required has made it difficult to construct a traditional strictly-layered network stack. For example, routing is often influenced by higher layers such as data aggregation [8], user queries [12], and time synchronization [6]. Traditional abstractions such as TCP's *end-to-end* reliable and congestion-responsive streams, which have served Internet development so well, do not address the desire in sensor networks to encourage as much *hop-by-hop* processing of data as possible (e.g., aggregation and filtering). How can we create reusable components, beyond basic device drivers, when application and domain knowledge seems to seep into every layer?

As sensor network research has moved out of its infancy, its focus has started to shift away from short-lived, hand-configured demonstrations and toward the creation of real applications: long-lived, larger-scale sensor systems that must be robust and truly adaptive to their environment. This shift has shown the paradoxes in sensor network software design to be serious stumbling blocks.

This paper describes EmStar, our new Linux-based software framework that addresses the difficulties in creating robust software in the sensor network domain. Broadly speaking, its contributions fall into two areas. First, EmStar's execution environments address the problem of visibility into an in-situ system. EmStar provides a spectrum of run-time platforms—a pure simulation, a true distributed deployment, and two hybrid modes that combine simulation with real wireless communication and sensors in the environment. Each of these modes run the same code and use the same configuration files, allowing developers to seamlessly iterate between the convenience of simulation and the reality afforded by physically situated devices (Section 2).

Second, EmStar's programming model aims to promote software reusability while being more flexible than a strictly layered stack. As we will describe in Section 3, EmStar's modules may be flexibly interconnected using standardized interfaces; connections can be a flow of packets, stream data, state updates, or configuration commands. Our model also lets applications' domain knowledge affect modules that are common across applications, without making application-specific changes to those modules.

Section 4 describes several case studies of services developed using EmStar. Related work is reviewed in Section 5, and in Section 6 we describe our conclusions and future work.

2 Execution Environments

EmStar provides a diverse set of execution platforms, ranging from pure simulation to fully distributed in-situ operation. The same code and configuration files are used on each platform, making it possible for a developer to move seamlessly among the available modes. This is central to our approach of easing the path from concept to deployment and back again. We will describe each point along this spectrum in detail, but their character varies chiefly along two axes, as depicted in Figure 1:

- *Scale*—The number of nodes in the sensor network, and their geographic extent.

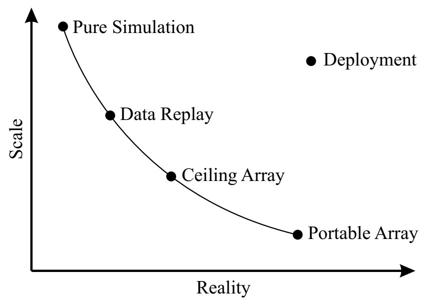


Figure 1: The spectrum of EmStar execution environments. Points along the arc allow high visibility into the system, enabling detailed analysis and improvement of its behavior. By understanding both the effects of scale (via simulation) and the effects of the real environment (via the ceiling and portable arrays), developers are more likely to create software that works properly when deployed on a large scale in the real world.

- *Reality*—The similarity of the platform, and the nature of its inputs, to a deployment in the application's intended target environment.

By definition, the most realistic possible platform is an autonomous wireless sensor network, including both hardware and software, deployed in its real environment. In contrast, a pure simulation is not realistic. For example, the behavior of the communication channel and sensor inputs are based on models that can never capture the full complexity of the real world. The range of hardware failure modes seen in harsh environments is also difficult to anticipate, and thus difficult to simulate.

Of course, for a sensor network to be deployed, it must eventually deal with reality. Unfortunately, reality imposes significant obstacles to understanding the behavior of the network in detail. Such an understanding is central to the development of algorithms and software. The most fundamental problem is the paradox we described in Section 1: the network's *raison d'être* is to filter, reduce, and summarize data in situations where transmitting complete sensor time-series to a central location for analysis is impossible. However, the discarded time-series are needed to evaluate whether the state of the environment was accurately reflected by the final, high-level sensing result. A simulation makes such an analysis possible because it offers complete visibility into a system—allowing the developer to save every sensor input and the state of intermediate computations at every node, if necessary.

This and other advantages of simulation make it a vital tool, but it has a critical drawback: its essential lack of reality can lead developers astray. Real communication channels in complex environments (e.g., indoors, or in dense foliage) are notoriously difficult to model accurately [7]. Connectivity is unpredictable and has been shown to vary significantly on both short and long timescales. The difference between the real and simulated channels can make it easy to write software that works *only* in the simulator. Software written in the sheltered environment of a deterministic channel, or in a simulator that has an overly simplistic noise model, often breaks when exposed to the real world for the first time.

For example, consider software that reliably delivers packets to the neighbors within a node's local radio range. In a real channel, a transient environmental effect might allow the delivery of a few packets from a far-away, normally unreachable neighbor. A developer that has never experienced these dynamics may write software that permanently adds a node to a neighbor-list whenever a packet is received. This algorithm may work in a simulator with a deterministic channel, or with a channel that produces packet loss on short timescales. In the real channel, a node will endlessly retransmit packets to a neighbor that will never acknowledge them.

EmStar is, in part, an attempt to balance the usefulness of a simulator with the need to write software that works in reality. To this end, we have implemented a spectrum of execution environments that fall on different points in the Scale/Reality space shown in Figure 1. EmStar allows developers get the basics of an algorithm working in a controlled environment (simulation); then, understand both the effects of scale (via a large simulation) and the effects of the real environment (via the ceiling and portable arrays). Code that has been debugged using all the modes has a good chance of working in a real-world deployment, where it must both be scalable *and* deal with the effects of the real environment. While deployed code may not work immediately, an immense amount of real progress can be made in a much more friendly environment.

In the following sections, we will describe each of EmStar's execution environments in more detail.

2.1 True Distributed Deployment

In a real deployment, autonomous and untethered nodes are deployed in a real environment, running a real application. Each node has a low-power radio and sensors, and runs an EmStar software stack. The scale of the deployment typically is limited by the hardware available. In most of our development, the goal is to reach this state.

For deployment "in the wild," our current prototype platform is the Compaq iPAQ 3760, which is a handheld, battery-powered device normally meant to be used as a personal organizer (PDA). The iPAQ provides a reasonable balance of cost, availability in quantity, and functionality. It has a 206MHz Intel StrongARM-1110 processor, 64MB of RAM, and 32MB of persistent FLASH. Our iPAQs use the "Familiar" distribution of Linux [1]. Each iPAQ is attached via a serial port to a Berkeley Mote [9], which is used as a low-power wireless transceiver and sensor interface.

[Although large and power-hungry relative to our ultimate desired target platform, the iPAQ serves as a reasonable stand-in until a smaller and lower-power equivalent "system-on-chip" is available. In addition, the Linux-based devices are only one layer of our tiered architecture—larger, more computationally endowed nodes are deployed in conjunction with smaller, less capable but more numerous nodes. The smaller nodes, which can't do much buffering or computation, have simple software for feeding data to and being tasked by the larger nodes. They, in turn, do the more complex aggregation, signal processing, routing, and so forth.]

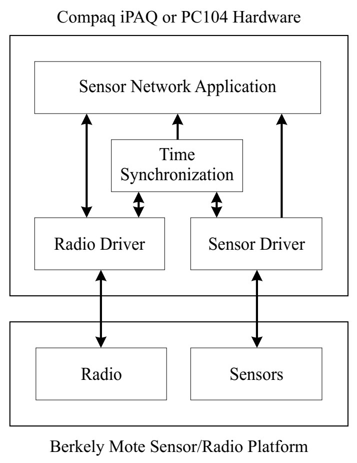

Compaq iPAQ or PC104 Hardware
Sensor Network Application
Time Synchronization
Radio Driver
Sensor Driver
Radio
Sensors
Berkely Mote Sensor/Radio Platform

Figure 2: A block diagram of a simple EmStar software stack. Each block represents a Linux process. Arrows indicate flows of packets, state, or other information. Details of the inter-process communication are described in Section 3.

As we will discuss in Section 3, each component of the EmStar stack is represented by a process with its own address space. The collection of processes is managed by *emrun*, which starts each process in the proper dependency order based on a configuration provided by the user. In a real deployment, the stack includes device drivers that provide interfaces to real physical channels, such as the network and sensors. Typically, there are several layers of common services on top of the physical interfaces, such as sensor calibration, neighbor discovery, routing and data dissemination protocols, time synchronization, acoustic ranging, and 3D multilateration. One or more sensor applications are at the top of the stack (Figure 2).

If a process terminates unexpectedly (e.g., due to a bug), it is automatically restarted; other modules in the stack can then reconnect to the failed module without losing their own state. This provides an important element of robustness in deployed systems where users are not available to manually recover from errors or restart failed processes. *emrun* is also responsible for configuring the verbosity of debug output of each process, and collecting the output into a temporary in-memory buffer. The buffer can be queried via the network if a high-level error is observed.

In this configuration, none of the elements of the system are tethered to an infrastructure, making true distributed deployment possible. However, as we discussed earlier, this same property makes the system difficult to control, observe, and debug. In addition, using real hardware has many logistical hurdles: programming, power, packaging, the coupling of sensors to the environment, and other hardware vagaries combine to add a lot of noise to the experimental process when dealing with a large number of nodes. In the early stages of an application's development, it is an obfuscating distraction that prevents developers from focusing

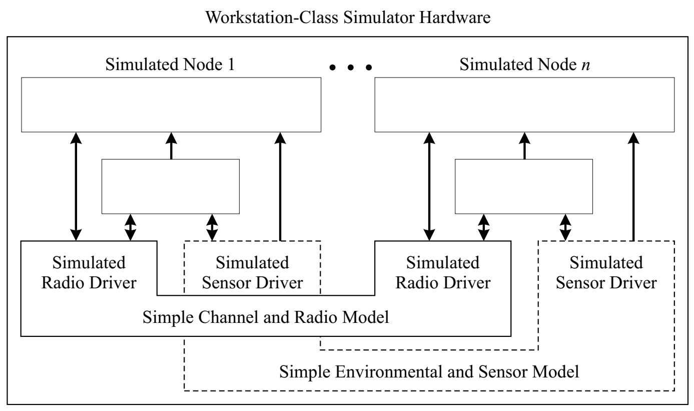


Figure 3: The structure of *emsim* in pure simulation mode. For each node, an instance of *emrun* is launched, creating a stack such as the one in Figure 2. However, instead of physical devices, simple radio channel and sensor input models moderate each node's interactions with the physical world. The channel simulator provides interfaces that emulate the behavior of the real device drivers. This allows the same services and applications run on top of the simulated device drivers without modification.

on the essence of the problem at hand. Parallel work by multiple developers is also difficult; most labs do not have enough deployable hardware for more than one developer to simultaneously test a large-scale deployment.

2.2 Pure Simulation

At the other end of the platform spectrum is *emsim*, a pure simulation environment. In this mode, multiple copies of *emrun* are started, each of which launches a copy of the same stack that is run in a real deployment. Each instance represents one simulated node, and is run in its own sandbox. As in reality, the nodes must interact via the "environment" and are not allowed to share state directly. Instead of using real radios and sensors, *emsim* provides a channel simulator that models the (simplified) behavior of the environment, based on a simulator configuration that defines aspects of the nodes such as their position and radio power. The channel simulator provides interfaces that match those of the real device drivers (Figure 3). The same services and applications can therefore run unchanged using the simulated device drivers.

Because the simulated and real platforms both run the same user code, read the same configuration files, and provide the same interfaces to the operating system and physical devices, developers are forced to think through and implement *every detail* of their algorithms early in the development process. Unlike more traditional simulators, developers are prevented from taking shortcuts or making unrealistic assumptions that later prevent the code from running on a real system. (The move to reality isn't always completely transparent, however. One group using EmStar recently developed a float-point signal correlator which worked fine on the x86-based simulator. When deployed on an iPAQ, they found it ran very slowly—the StrongARM does not have hardware floating point support.)

The main advantage of the simulator is that it offers complete visibility into the system being tested. Nodes running in simulation can easily log "distributed" events in their global temporal order. Practically infinite space is available for saving sensor "inputs," debugging messages, the intermediate results of computations, or any other information useful for understanding the system's behavior. As we will describe in Section 3, the EmStar programming model also allows interactive inspection of much of the system's internal state while the simulation is running. Since the simulator is a full-fledged desktop workstation, it is easy to use complex debuggers, visualization tools, memory checkers, and so forth.

In addition to visibility, simulators offer exceptional control. Unlike code running on distributed iPAQs, centrally simulated nodes can be instantly "placed" in any topology, or a random topology, via a configuration file. Systematic testing of a range of scenarios is easy, including configurations that might not be feasible to actually deploy due to cost or other constraints. Furthermore, while the real environment is constantly in flux, a simulation can be made completely deterministic; this is useful because many problems are easy to fix once they are consistently reproducible.

Simulations are also attractive because of their accessibility. The small, low-power nodes appropriate for real deployments have not yet been commoditized. It is still prohibitively expensive to give each developer enough real nodes to perform experiments on a significant scale. In contrast, a simulation machine is (currently) much more accessible than actual sensor network hardware; it is cheap, ubiquitous, and easy to use. This allows many developers to work in parallel rather than contending for limited real hardware. It also opens sensor network development to a much wider audience—enabling, for example, remote development, undergraduate and high school class projects, and tinkering by hobbyists. *emsim* is also useful because it can simulate larger numbers of nodes (hundreds) than may be available in reality—allowing developers to see the effects of scale long before it is possible to do so in the real world.

Of course, the disadvantage of simulation is that it does not capture every aspect of the real world that can affect the outcome. This is an important problem in sensor networks; their function is often intimately tied to the world in which they are physically situated. However, early in the development of a new algorithm, subtle effects of the radio or sensor channels are often invisible compared to the basic problems encountered when writing any new software. When code is first written, even a trivial channel model will reveal fundamental design flaws and protocol bugs, sanity-check the offered load against the channel capacity, and let developers find common software problems such as memory overruns, broken interfaces, and plain coding errors. Inexperienced developers tend to spend particularly long dealing with these sorts of issues. In our experience, using the simulator makes the process much faster.

Because of the simplicity of our channel models (Figure 4), algorithms that are sensitive to the subtleties of the channel are

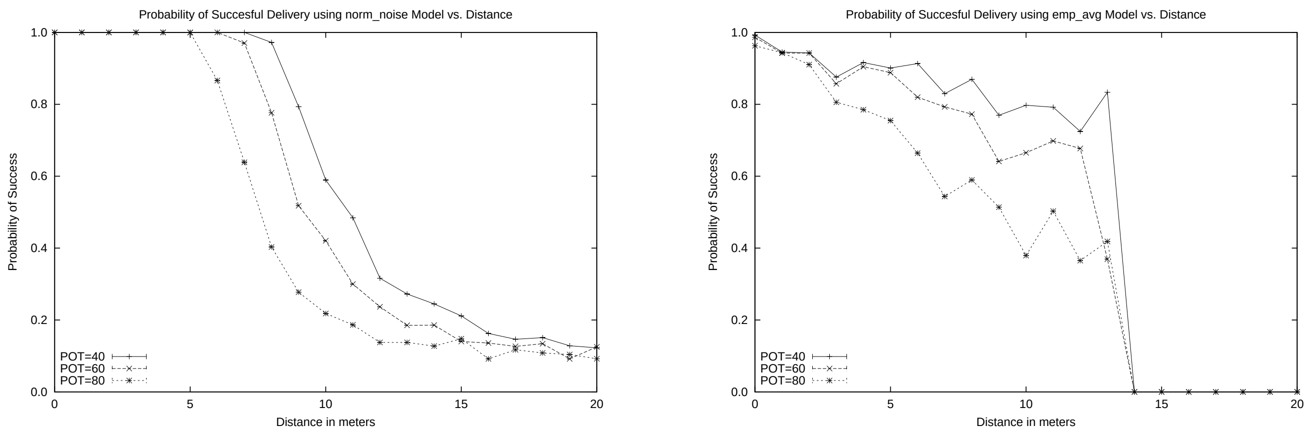


Figure 4: Two of the *emsim* channel models. Not shown is the "deterministic circle" model, in which nodes less than 8 meters apart can exchange packets with 100% reliability, while nodes separated by longer ranges can never exchange packets. While unrealistic, the determinism is helpful for debugging fledgling applications. *left*) The "normal noise" model is somewhat more realistic: as nodes are separated by greater distances, the loss rate gradually increases. It also has a basic model of the mote's potentiometer (POT) on transmit range. *right*) "Empirical average" is a statistical model based on experiments with real motes. We used connectivity data that were collected at various ranges and potentiometer settings as part of the ASCENT project [4].

not as well served by *emsim*. For example, our simulator would be a poor tool for testing a module that tries to deduce the range between two nodes based on radio signal strength. However, much of the supporting code surrounding channel-dependent algorithms *can* be effectively developed and tested in simulation—such as the network protocols and statistical algorithms required for a group of nodes to automatically schedule ranging experiments, share their deduced ranges, discard outliers, and synthesize what remains into a consistent, shared coordinate system.

Once EmStar code works in *emsim*, development can continue by using the modes that incorporate real channels, as we will describe in the coming sections. Debugging code while dealing with the vagaries of real RF propagation is slower and more difficult. However, since the code has already been vetted in the simulator, far less total time is required.

2.3 The Ceiling Array

Roboticist Rodney Brooks has famously observed, "The world is its own best model." This guidance is also apt for sensor networks which, like robots, are physically situated. The research community's past efforts have shown it is very difficult to model RF propagation for short-range, low-power radios in complex environments [7]. Indoor models are notorious because reflection, diffraction, and scattering are caused by both the structure itself and the objects inside it. Yet, our channel models are simplistic—instead of trying to predict these effects with great fidelity, the goal of our simulations is only to be good enough to support basic software development. In EmStar, realistic channels come from the ceiling array—a platform that uses the world as its channel model.

We permanently mounted a uniform array of 54 motes to the ceiling of our lab. The motes are all wired for power and have a

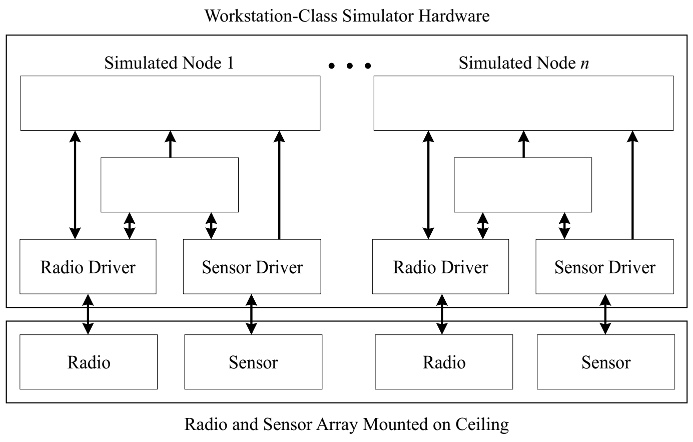


Figure 5: The structure of *emcee*—a hybrid mode that combines simulation with real channels. As with *emsim*, multiple instances of *emrun* are run, including real device drivers. The drivers are attached to 54 Berkeley motes permanently mounted on the lab ceiling. *emcee* lets developers experience real channel characteristics in a mode that is as easy to use and debug as a pure simulation.

serial-port connection back to a central simulation machine. As in a real deployment, each mote is programmed to be a wireless transceiver and sensor interface board. *emcee*, the ceiling array control program, is similar in most ways to *emsim*—all the instances of the node stack are run centrally. However, the channel simulator module is not used; instead, each simulated node is mapped to one of the motes on the ceiling. When a node sends a packet, it is transmitted and received by real motes, through the real channel (Figure 5).

The usefulness of the ceiling array relative to the simulator stems from the complexity of the real channel. The environment causes distortion and multipath fading; the effects include spatially correlated packet loss, asymmetric links, and non-monotonically degrading connectivity as range increases. Changes in the environment (e.g., motion of people and objects, electrical devices turning on and off, cell phone calls) also cause a variety of time-varying effects.

Figure 6 shows two of the experiments we performed on the ceiling array channel. For connectivity between a pair of nodes at fixed locations, the channel exhibits both short- and long-term time dependencies (left). Independent of the other effects, there are also spatial dependencies (right) with adjacent nodes demonstrating correlated losses, and different spatial regions showing significantly different behavior. While any one of these metrics may be easily simulated, it is difficult to capture all the various dimensions of RF propagation dynamics working together—especially when many of the dimensions are unknown, and many are uncharacterized.

The ease-of-use of the ceiling array has been a crucial feature. Applications that work in *emsim* can be tested on the ceiling array just by typing *emcee* instead. Because the motes are permanently programmed, powered, wired, and mounted, the ceiling array shields developers from most of the difficulty in dealing with large numbers of small devices, while still bringing important aspects of reality to bear. The hybrid simulations have many of the same advantages of *emsim*: simulated nodes run centrally, so debugging is facilitated by complete visibility into the system and a rich set of debugging and visualization tools. When the overhead of testing code on a real channel is so low relative to simulation, developers tend to test their code against the real channel early and often.

Developers can control the mapping of simulated nodes to physical motes, so varying topologies can be achieved by using different subsets of the ceiling motes. Of course, the diversity of topologies is constrained by the fixed locations of the motes. This is a limitation relative to the pure simulator, where arbitrary topologies are possible. In addition, while many simulator machines are available, there is only one ceiling array; contention for its use can be a problem. These kinds of constraints naturally arise when moving from a purely virtual to partially physical system.

Another important limitation of the ceiling array is that it represents one *particular* channel, and is not representative of *all* channels. RF propagation in our lab has interesting and important dynamics not seen in the simulator, but not all offices are the same, and none of them are likely to reflect the behavior of nodes in a forest or desert. This limitation is the motivation for our portable array.

2.4 The Portable Array

Software-wise, the portable array is identical to the ceiling array: it uses *emcee* to run simulated instances of the stack centrally, and connects each instance to a mote that is wired to the simulator. However, instead of using a server attached to motes permanently mounted on the ceiling, the portable array uses a laptop and "loose" battery-powered motes that can be placed anywhere.

The portable array is useful for exposing applications to the characteristics of the intended deployment environment, while using a platform that still has most of the conveniences of pure simulation. Such experience can be invaluable—the communication channel and sensor responses can differ significantly in an area of sparse trees vs. an area with dense low brush. The portable array allows developers to confront these issues before the system is deployed in an inaccessible area with limited diagnostic output.

The disadvantages of the portable array are mostly practical. Unlike the ceiling array, which is always ready at the touch of a button, use of the portable array involves a trek to a foreign environment with a box full of motes and hundreds of feet of cable. These logistical concerns are not trivial: research in wireless sensor networks exists because of the desire to observe environments where a large-scale deployment of wired sensors is infeasible.

The inconvenience of deploying the portable array is the price paid for an almost completely realistic in-situ deployment that still has the complete visibility of a simulation. It also prevents the portable array from growing to a large number of radio and sensor interfaces. For this reason, the portable array differs from true deployment in one key area, as we saw in Figure 1: scale.

2.5 Data Replay

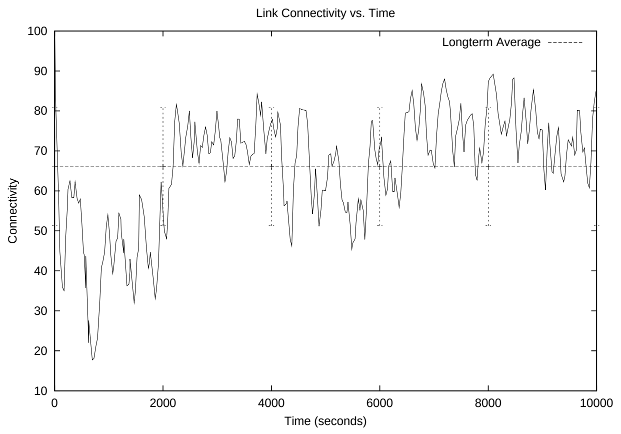
The final EmStar platform, Data Replay mode, has not yet been used externally and is part of our future work. However, we will describe it briefly to show how it will fit into the spectrum of other platforms.

In the existing platforms, nodes run with either a channel that is completely simulated (as with *emsim*), or a real channel (as with *emcee*, or a real deployment using only *emrun*). Data replay mode will add a new dimension: the sensor inputs will be recorded or taken from other sources such as existing seismic arrays or vehicle transportation data. Later, these stored sensor time-series will be played back in real-time to an otherwise simulated set of nodes. Data replay mode is essentially a trace-driven simulation, where the trace is a time-series of sensor values.

Data Replay mode will be valuable to help develop algorithms that have dependencies on the behavior of sensors, in cases where the sensors have already been well-characterized. For example, the seismology community keeps databases of time-series data from seismometers, annotated with global timestamps and positions. This kind of data will be used to feed a simulated seismometer as part of an EmStar simulation, facilitating development of algorithms for automatic event detection and localized collaboration.

3 Programming Model

In the design of EmStar and applications, we have developed an approach which has been a good fit to the wireless embedded systems space. We have previously alluded to the requirements



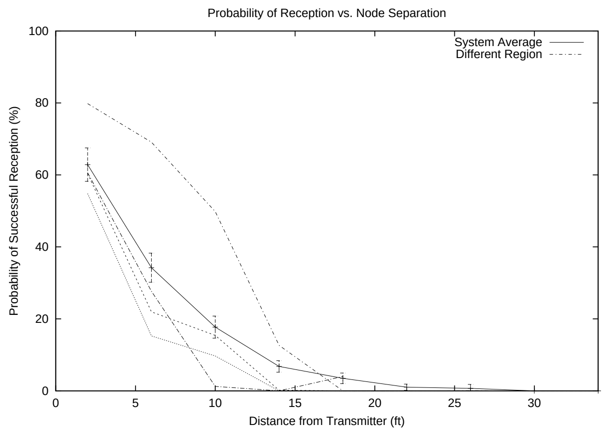


Figure 6: a) (*left*) A pronounced change in the probability of reception between a fi xed pair of nodes within the duration of a single experiment. b) (*right*) The effects of spatially correlated noise. The lower three curves represent three nodes in a particular region of the network that has lower-than-average probability of reception. In contrast, the top curve shows a curve from a region that displays exceptional short range propagation characteristics, then quickly falls to below the average at longer distances.

of these applications, which can be summarized as three general categories: reactivity, robustness, and modularity.

Reactivity is important to these applications; they must be able to respond to dynamics in both the system and the environment. System dynamics include the number of nodes operating at any given time, how much power or buffer space they have remaining, and other more specific information about the state of the system. Environmental dynamics include variations in the radio environment, as well as variations in the environment that might affect a sensing application.

Robustness is especially important to these applications because wireless embedded systems are much more difficult to manage centrally. When developing distributed systems where network bandwidth is not a scarce resource, a centralized management approach is often the most expedient solution. While centralized management can be done through a debugging harness (such as a second, wired network), the characteristics of wireless networking mean that a deployed system must survive without it. Therefore, in order to iterate between development and deployment, local mechanisms will be required that keep individual systems running.

Modularity is critical to the development of any complex system, and EmStar is no exception. What is somewhat different about EmStar is the difficulty of separating the system into clearly defined layers. For example, many applications do not use an end-to-end routing layer. Instead, packets often need to be reprocessed at every hop along the way, for example so that similar data can be aggregated or summarized. This does not mean that there is no utility in having a routing module; rather that this module will need a richer interface that enables the application to provide input to the routing decisions. The impact of this kind of design is that the number and complexity of the inter-module dependencies grows.

Given these requirements, we will now describe the facilities we have developed in support of these requirements.

3.1 FUSD–A Framework for User-Space Devices

To support IPC mechanisms for modular and robust design, we developed FUSD, a Framework for User-Space Devices.[1] FUSD (pronounced *fused*) is essentially a microknerel core implemented on top of the monolithic Linux kernel; it allows device-file callbacks to be proxied into user-space and implemented by user-space programs instead of kernel code. Such device drivers can create device files that look and act just like any normal file under `/dev`; other user-space processes can open a FUSD driver's device and execute system calls on the file descriptor, just as with a normal kernel-implemented device.

Of course, as with almost everything, there are trade-offs. User-space drivers can be significantly slower and higher-latency than kernel drivers because they require three times as many trips through the kernel, and additional memory copies, per user system call (see Figure 7). User-space drivers can not receive interrupts, and do not have the full power to modify arbitrary kernel data structures as kernel drivers do. Despite these limitations, we have found user-space device drivers to be a powerful programming paradigm with a wide variety of uses, as we will see in the next section.

FUSD drivers are conceptually similar to kernel drivers: a set of callback functions called in response to system calls made on file descriptors by user programs. FUSD's C library provides a device creation function, fusd_register(), which is similar to the kernel's devfs_register_chrdev() function. fusd_register() accepts the device name and a structure full of pointers. Those pointers are callback functions which are called in response to certain user system calls—for example, when another process tries to open, close, read from, or write to the driver's device. The callback functions are generally written conform to the standard definitions of POSIX system call behavior. In many ways, the

[1]FUSD was developed by the authors, in part with support from Sensoria Corporation.

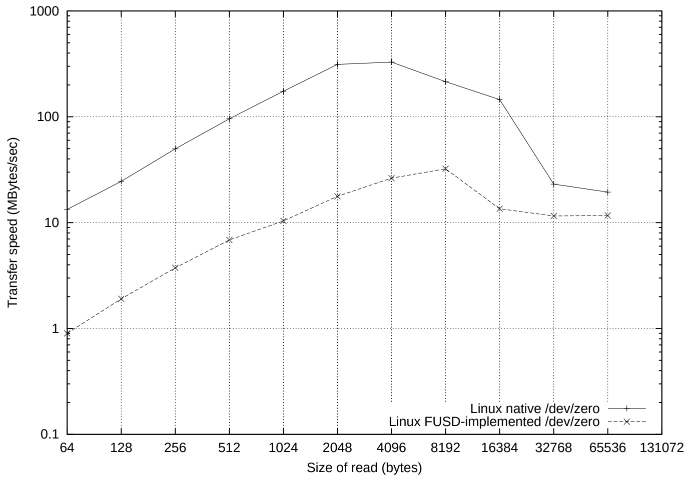


Figure 7: Throughput of the Linux kernel's native /dev/zero vs. an equivalent FUSD driver. The test program timed a read of 1GB of data from test device on a 550 MHz AMD K6 with 64K cache. We tested `read()` sizes ranging from 64 bytes to 64 Kbytes. Larger read sizes are higher throughput because the cost of a system call is amortized over more data.

user-space FUSD callback functions are identical to their kernel counterparts.

The proxying of kernel system calls is implemented using a combination of a kernel module and cooperating user-space library. The kernel module implements a device, `/dev/fusd`, which serves as a control channel between the two. When a program calls fusd_register(), it uses this channel tell the kernel module the name of the device being registered. The kernel module, in turn, registers that device with the kernel proper using devfs. devfs and the kernel don't know anything unusual is happening; it appears from their point of view that the registered devices are simply being implemented by the FUSD module.

Later, when the kernel makes a callback due to a system call (e.g. when the character device file is opened or read), the FUSD kernel module's callback blocks the calling process, marshals the arguments of the callback into a message and sends it to user-space. Once there, the library half of FUSD unmarshals it and calls whatever user-space callback the FUSD driver passed to fusd_register(). When that user-space callback returns a value, the process happens in reverse: the return value and its side-effects are marshaled by the library and sent to the kernel. The FUSD kernel module unmarshals this message, matches it with a corresponding outstanding request, and completes the system call. The calling process is completely unaware of this trickery; it simply enters the kernel once, blocks, unblocks, and returns from the system call—just as it would for any other blocking call.

One of the primary design goals of FUSD is *stability*. A FUSD driver can not corrupt or crash any other part of the system, either due to error or malice. Of course, a buggy driver may corrupt itself (e.g., due to a buffer overrun). However, strict error checking is implemented at the user-kernel boundary which prevents drivers from corrupting the kernel or any other user-space process—including other FUSD drivers, and even the processes using the devices provided by the errant driver.

3.2 IPC in EmStar

Using FUSD, we have built a broad class of inter-process communication mechanisms by varying the semantics of the user-space drivers' responses to POSIX system calls. For example, our audio server can return stream data (audio time-series after time synchronization and convolution) by implementing a `read()` system call that delivers the next chunk of data, just as in the kernel's audio device driver. Packet-based IPC is also easy: the drivers for our radio devices define the semantics of `read()` to always return exactly one packet, starting from the packet boundary, similar to the `recvfrom()` system call. For remote procedure calls, a client can open a server's device, write a command and marshalled argument list, and later read back any results of the call. Event notification for any of these types of IPC (e.g., next packet is available, or RPC call has completed) is implemented using `select()`—FUSD-based servers can trigger file descriptors to become readable or writable. This capability was easily integrated into an event system, GLib, that enables dynamic registration of events in response to the status of file descriptors.

FUSD itself allows the construction of devices with arbitrary semantics. However, in our work on EmStar we have found that a relatively select set of device semantics accounted for most of our needs. We have called these device types status devices, packet devices, and command devices.

Status devices provide a view into the current state of some aspect of a module, with notification on state change. For example, our radio device drivers maintain state variables about the radio's configuration, and statistics on the number of packets received and transmitted. This information about the current radio state is exposed through a status device. To access this status information, a client simply opens the device file and reads from it. Status devices can return either binary data (e.g., C structures), which is useful for IPC, or human-readable (ASCII) data, which is useful for interactive debugging.

In addition to reporting the status, the modules can notify clients when the status changes. This is an important feature for building reactive systems, because it allows changes seen by one module to quickly propagate to others. In the case of radio status, clients often want to receive notification of changes to radio configuration parameters, such as the radio being off or its MTU changing, so that they can adapt their behavior accordingly.

Whenever a module decides to notify its status-device clients of a change, the clients' file descriptors become readable according to `select()`. The client receives the new information on its next read. The semantics of what constitutes a change requiring notification, and which clients are notified, is defined by the module. For instance, changes to the packet transmission statistics do not trigger notification, while changes to the MTU do.

An important feature of status devices is that they do not guarantee to deliver all "intermediate" states, only the most recent state at the time of the read. That is, if the state changes several times but the client fails to read in those intermediate times, its eventual read will only see the most recent state.

Packet devices provide an interface to a stream of messages in and out of a module. For example, packet devices are used by our radio device drivers to deliver packets to modules that use the radio, and accept packets for transmission from them. Packet devices implement per-client input and output queues, along with a standard API to configure and query configuration such as the maximum queue length. Packet devices also support per-client packet filtering.

Command devices provide a way to configure a module. Clients open command devices and write simple configuration strings that are parsed in a module-specific way. For example, the radio potentiometer on a Berkeley mote can be changed by writing `POT=55` to its device driver. Of course, commands can be written programmatically by another module, or interactively (e.g., using `echo`) from the command-line. Command devices are also an easy way for the client to trigger events in the server.

To simplify implementation of servers, most of the server side details of these three IPC mechanisms are implemented by libraries. Adding a new status device to provide a new debugging output only requires a call to a constructor and the definition of a callback function to generate the status output.

On the client side, the interface is fairly simple – at the lowest level it is all POSIX calls such as open(), read(), write() and select(). However, it has been useful to have helper libraries that make it easy to integrate device clients into the GLib event system. These libraries also enable new features supporting robustness. For instance, the client library opens the device, registers the event, and waits for data to be ready. In the event that the server crashes, the client will see the file descriptor close unexpectedly. In this case, *emrun* restarts the failed process, as we described in Section 2.1. The client library will automatically reopen the device and reread it, transparently to the client application. This "crashproofing" feature makes the system much more resilient to programming errors that might otherwise cause much of the system to restart, or worse, become non-functional.

3.3 Modularity in EmStar

Reactive systems are often complex, and understanding complex systems requires modularity. In EmStar we follow the UNIX doctrine of building a system from the combination of many small processes. These small processes communicate through the IPC mechanisms we described above. As we described in Section 2.1, *emrun* is responsible for process management, which includes starting each processes in its correct dependency order, restarting failed process, collecting log messages, and so forth.

Having many small processes has numerous advantages in terms of modularity and robustness, but one of the biggest potential problems is transparency into the operation of the system: what are all those processes doing? In EmStar, the same mechanisms that provide IPC also allow interactive inspection of the system's state.

As we mentioned earlier, status devices can provide state either as binary IPC *or* as human-readable output. The two modes are synchronized, and any number of clients can read simultaneously and get copies of the same information. This makes it very easy for a developer to see exactly what information is flowing along any IPC pathway in the system, using a tool no more complex than `cat`. We have found this to be a major improvement in system visibility over other IPC mechanisms. For example, with socket-based IPC, it is difficult for a user or debug process to get a *copy* of the information flowing from one process to another, and the information may be in binary format. Beyond plain `cat`, more complex debug programs (e.g., EmView, the visualizer) gather state from many status interfaces and synthesize it into a more comprehensive display. In addition, using `echo`, users can interactively change the configuration of modules via command devices and observe the resulting behavior.

Another issue that crops up with many interdependent processes is the potential for circular dependencies. *emrun* checks to ensure the module dependency graph is acyclic. However, in some cases, this can prevent the correct IPC channels from forming if the desired flow of information between modules is inherently cyclical. While this at first appears to be a serious limitation, there is usually a straightforward solution. Typically this problem is avoided by more finely decomposing the functionality of the modules, allowing two clients to both configure a common server and receive notification when the server has been reconfigured.

To give a concrete example of this idea, recall our example of a radio with a status device that reports whether the radio is asleep or awake. Processes that use the radio (e.g., neighbor discovery, or routing algorithms) need to know whether the radio is awake before sending a message. Suppose now that a particular application has a module that tries to conserve power by duty-cycling the radio, using specialized knowledge of the application domain. When it commands the radio to turn off, it also indirectly communicates that information to all the processes using radio—because the radio status changes. The users of the radio don't need an *explicit* meta-data channel to the application-specific module. In fact, the radio clients don't even need to know that the application module exists.

This example also demonstrates a key facet of our approach for preserving modularity and code re-use, while still allowing specialized domain knowledge to affect otherwise generic modules. In this example, the domain-specific knowledge was encapsulated in the power-conserving module (the reasons *why* the radio is turned off) while the more generic information (that the radio *is* off) was transmitted to clients that could use that information.

4 Case Studies

In this section, we describe several examples of sensor network services that were developed using EmStar. Each was initially developed using *emsim*, allowing new developers to become familiar with the framework and run sample applications without the need for real sensor hardware. Later, as the basics were fleshed out, developers began to iterate between the simulator and initial experimentation with the ceiling array. Finally, as implementation seemed to be complete, one of the groups set up a portable array in a nearby wooded area.

While the efficacy and "user-friendliness" of a software development environment are difficult to quantify, we feel that these examples suggest, anecdotally, that EmStar makes the process of developing software for sensor networks more efficient. In addition, the visibility into the system that EmStar provides let developers iterate on their designs based on meaningful feedback—including quantitative metrics that would not be available in a pure wireless system. Such an approach made developers more confident that the software's inner workings would be correct once the code *was* executed in a pure wireless environment. This was appreciated as an improvement over black-box development that relies on a high-level "it seems to work" judgment.

4.1 Reliable NeighborCast

Network-wide flooding is commonly used in sensor networks by query distribution protocols [8, 12]. While flooding is often thought of as sufficiently reliable due to its inherently high redundancy, our initial experience with motes in an earlier outdoor deployment showed that even flooded queries tend to get lost due to spatio-temporal correlation of link interference and fading conditions. While many link-layer retransmission protocols exist for wireless unicast traffic, reliable broadcasts to a set of local neighbors are usually not considered. We therefore used EmStar to develop Reliable NeighborCast, or RNC. This case study presents an initial implementation of RNC, and our experiences with EmStar.

The algorithm for improving reliability is simple. RNC adds small sequence numbers to packets. Since it is impractical to expect a separate acknowledgment from each of the receivers individually, we use an asynchronous acknowledgment mechanism. Each node keeps the sequence numbers of the packets it most recently received from its neighbors. Periodically, each node broadcasts this state. When the aggregate acknowledgments are received by a sender, it can re-send any packets that the receiver is missing.

We wanted to debug the details of the algorithm in a simulation. *emsim* prevented us from taking shortcuts, such as implicitly sharing global information between physically distributed nodes or using global time. While the initial implementation required more time than a less realistic simulation, it allowed us to eliminate algorithmic bugs and run a sanity check (see Figure 8).

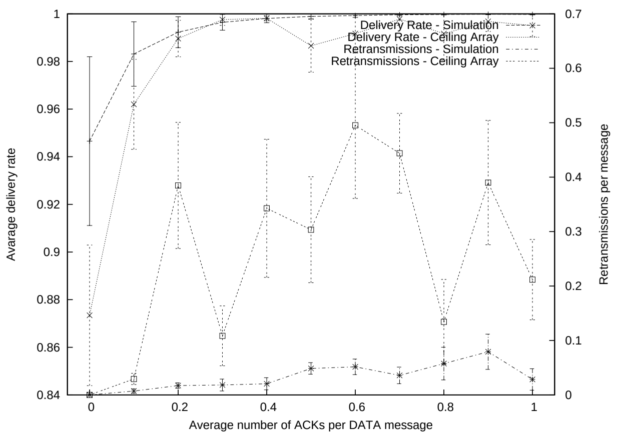


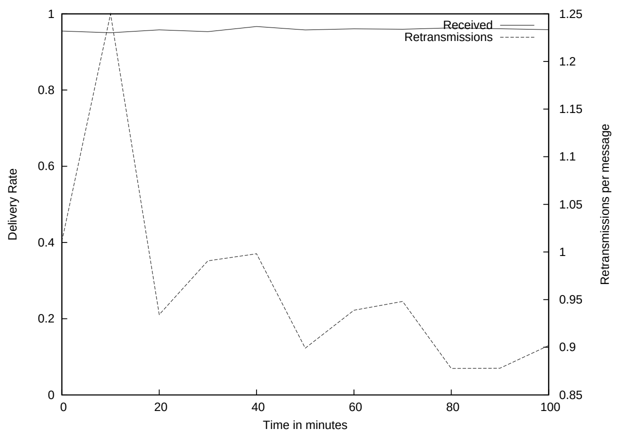
Figure 8: The top curves display the average delivery rate over a single hop as a function of a number of acknowledgements per data message. The bottom curves are plotted against the right Y-axis and show the average number of retransmissions per data message on the same X-axis. Results include simulation and ceiling array.

Single-hop RNC

Our simulation consisted of nodes in a 3×3 grid, all within radio range of each other. All nodes broadcast periodic messages and recorded statistics about the messages received from other nodes. As we increased the acknowledgment rate with respect to the data rate, the reliability increased. Simulation results verified the correctness of the basic functionality. However, it was still unclear if persistent retransmissions would increase contention and cause loss due to collisions on a real channel, thus potentially altering the results radically.

We then ran the same set of experiments on the ceiling array. This is when the time invested into a complete implementation of all the interfaces paid off: we did not have to make any code changes to repeat the above experiment on real hardware and a real channel. The results are shown in Figure 8. Even though the absolute values of the reception ratio were different between the simulation and the ceiling array, the trends were clearly the same for both.

As we mentioned earlier, in practice fading and/or interference are often correlated in time and space. This encouraged us to look at the performance of our mechanism over time. We conducted two longer-term experiments in different locations and compared the results. One experiment was performed on the lab ceiling array; the other was performed on a lawn outside the building using a portable array. The results of these experiments are shown in Figure 9. The graph on the left shows the performance on the ceiling array. There was clearly much less variance in the results. The graph on the right shows that interference was much less uniform in the outdoor experiment; in particular, two dips in the delivery rate correspond the cell phone calls received by the person performing the experiment. The low delivery rate in the beginning of the experiment is a consequence of small adjustments to node locations.



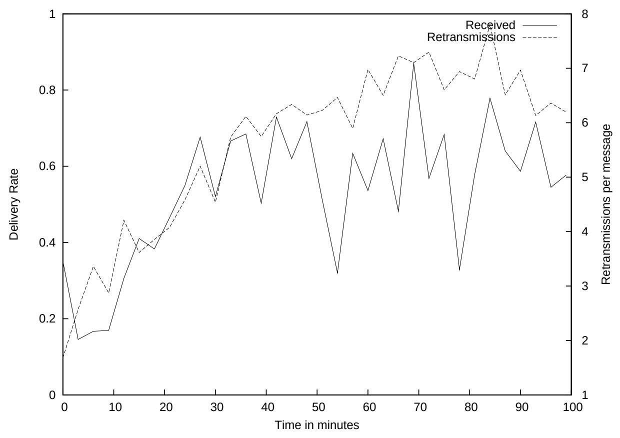


Figure 9: Single hop delivery rate and retransmissions over time. *left*) Ceiling array results. *right*) Portable array results. The outdoor portable array clearly has more variance in reception and an order of magnitude increase in the number of retransmissions.

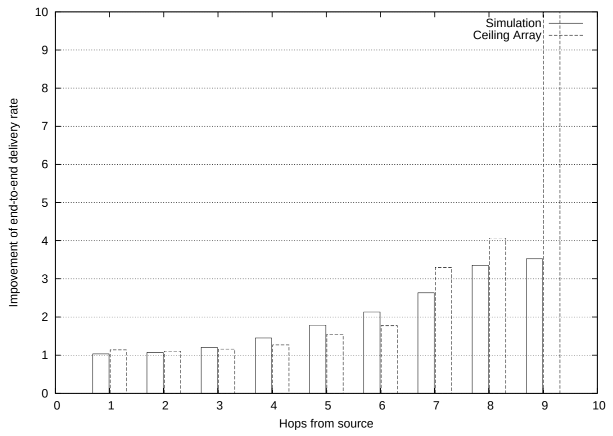


Figure 10: Improvement of the delivery rate of flooding over RNC over normal flooding. Values greater than 1 signify an improvement of the end-to-end delivery rate. Results include simulation and ceiling array.

The use of the portable array exposed the undesirable behavior of our simple retransmission scheme in the presence of an unstable channel. The lack of an adaptive back-off mechanism resulted in excessive retransmissions in the experiment outdoors.

Flooding with RNC

The above experiments allowed us to verify RNC's performance within a single neighborhood. The next step was to verify that the improvement in the reliability within a neighborhood would improve the performance of flooding in a multi-hop network.

Figure 10 shows that RNC improved the performance of pure flooding, especially for longer paths. The Y-axis shows the ratio of the delivery rate achieved by RNC-assisted flooding to that of plain flooding; values greater than 1 indicate improvement. The simulation results agree with the results from our ceiling array.

EmStar made all of these configurations easy. Flooding, RNC, and the underlying datalink device (whether simulated or real) all communicate using interchangeable interfaces. This made it trivial to test different configurations, just by changing how the modules are interconnected. Our test application could run using the plain radio; using the flood daemon, which sent packets to the radio; using RNC, which sent packets to the radio; or, using the flood daemon, which sent packets to RNC, which sent packets to the radio. Each stack configuration could be used on the ceiling array or in the simulator.

4.2 Adaptive Topology Control

Much sensor net research focuses on conserving energy, a vital resource in this domain. A powered-on, idle radio is a significant waste of energy; as hardware is optimized, it is expected to become the dominant source of waste [14]. However, in a redundantly dense deployment of nodes, a large subset may be able to power their radios off, while those that remain on still maintain a usable (e.g., connected) topology. A number of adaptive topology schemes have been developed to achieve this goal [4, 5, 15, 18].

In this case study, we investigate the effectiveness of two such schemes—ASCENT [4] and CTC [18]. With CTC, each node measures connectivity within a two-hop radius by sending heartbeat messages, and chooses a few nodes with large energy reserves to maintain the topology when the radios of other nodes are powered off. ASCENT measures connectivity with all one-hop neighbors by sending heartbeat messages and by gathering connectivity statistics of all data traffic (even traffic not addressed to a node). It locally decides to power on or off additional radios based on the channel dynamics and the degree of connectivity required by the application.

We implemented ASCENT and CTC using a number of fine-grained modules, so that other developers could re-use as much of our work as possible. The first is the *LinkStats* module, which adds a monotonically increasing sequence number to each packet sent by any process on the node. It monitors such packets arriving from other nodes, and maintains detailed packet statistics

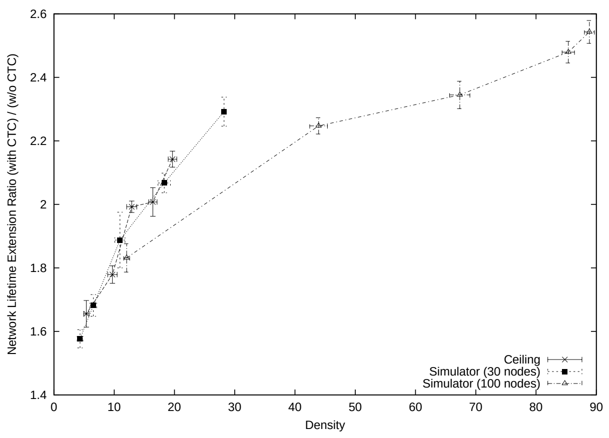


Figure 11: CTC on the ceiling testbed and in small and large-scale simulations. Though the results differ from the simulator quantitatively, the trends are the same. We achieved the expected result on a real channel though most of the development was done in the ease of simulation.

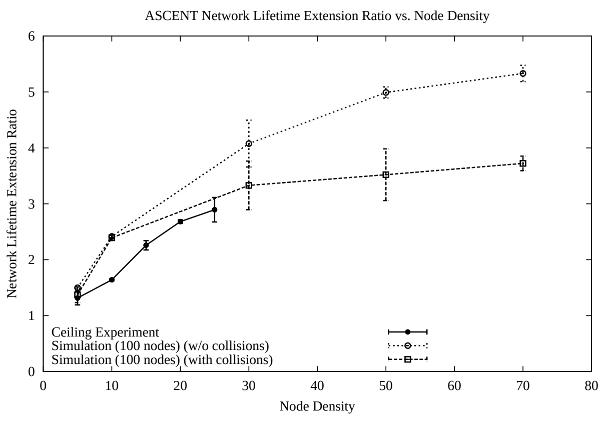


Figure 12: ASCENT on the ceiling testbed and in a large-scale simulation. The differences between the simulator and the real experiments are not signifi cant, in particular when using the collision model in the simulator. The trends as we increase density are the same in both simulation and real experiments.

for high precision connectivity measurements without increasing channel use (but, slightly reducing the maximum data payload). The second module is *Neighbor Discovery*, which sends and receives heartbeat messages, and maintains a list of active neighbors. Third, to evaluate both schemes, we created a module that acts as a simulated battery for each node. It counts packets sent and received, idle time, and radios powering on and off; energy is deducted from an initial supply accordingly. It runs in two modes: either it merely tracks energy usage or it actually shuts off a node when that node runs out of energy. Finally, we created the ASCENT and CTC protocol implementations themselves, which use information provided by the other modules. However, by modularizing our implementation, other applications can (for example) measure their own energy use, or take advantage of our detailed connectivity statistics, even if they are not using the ASCENT or CTC algorithms per se.

Because ASCENT and CTC exploit redundant nodes, varying node density (i.e. the number of nodes within one radio range) should affect the outcome. For this reason, in our experiments we vary density and measure by how much both ASCENT and CTC extend network lifetime. To create different densities, a fixed number of nodes were randomly distributed in areas of various sizes. Data traffic was generated at random from any node in the network, and flooding was used as the routing algorithm.

Our first simulator experiments used 30 nodes and the simple “deterministic circle” propagation model; this was primarily for development, and we do not show the results here. After the code was debugged, we ran CTC simulations using one of the non-deterministic channel models (emp_state, similar to the emp_avg model shown in Figure 4). We did not use the simulator’s channel contention model. Figure 11 plots the number of nodes using CTC that still have energy remaining over time, and clearly shows that network lifetime increases as a function of density.

Our simulation results seemed reasonable, but they rely on the accuracy of the channel model. Because ASCENT and CTC measure connectivity, it is important to experiment on a channel that more closely resembles those in the real world. Further, we had not been using the contention model; packet collisions, a harsh reality of the real world, thus did not occur in our simulations. For these reasons, we used the ceiling array for our next experiment. EmStar made it trivial to set up this experiment—the code did not even need to be re-compiled and utility scripts aided in adjusting the transmit power of the radios in order to vary node density. As Figures 11 and 12 show, our results hold when using a real channel with collisions.

Based on these initial experiments, we are confident that the simulator’s channel model was not wholly different from reality—at least, for the purposes of our specific case study, although further improvement could be achieved. Satisfied with the simulator’s predictive power, our next step was to see if our results held at a larger scale. We thus went back to the simulator and ran experiments on a network of 100 nodes, still with varied density. Figures 11 and 12 show that even at a larger scale, network lifetime is improved at higher densities for both ASCENT and CTC, respectively.

After performing the ceiling experiments and large scale simulations, we wanted to verify that the quantitative differences were in fact due to collisions. We went back to the simulator and ran experiments with varied density and using the collision model. Figure 12 shows that the results of the simulation with collisions are much closer to the results of the ceiling experiments. This result can be explained as follows. During ASCENT initialization, all the nodes have their radios on and they gather connectivity information. At high densities, and with flooding as our routing algorithm, the amount of channel contention is large, and the packet losses perceived by the nodes are high. Under these conditions, ASCENT is forced to increase the number of nodes with the radios on to maintain a usable topology, thus increasing energy consumption.

Using EmStar, our ASCENT and CTC experiments were simplified because the same code (in fact, the same binary file) could be tested with the un-modelled quality of a real channel, as well as with the scale necessary for any useful deployment but only available in a simulator. It also allowed us to easily go back and forth between experiments and simulations to analyze the different factors that affect our algorithms.

Because both protocols were evaluated using the same tools (and, in the case of the ceiling array, motes in the same physical locations), it is possible to draw a meaningful comparison between them based on our experiments. We can see from the figures that CTC and ASCENT perform similarly at low densities using the ceiling array. At simulated large densities without simulated collisions, CTC lifetime improvement grows more slowly than ASCENT. However, ASCENT is more susceptible to the effects of collisions, as we explained earlier.

Having EmStar will allow us to further test the algorithms using different scenarios and metrics. We plan to further evaluate both algorithms at large densities with collisions, as well as measuring performance of the algorithms using other metrics such as path quality.

5 Related Work

Many recent simulator and emulator projects have illustrated the philosophy of running simulations with the same code as is used on real systems. In many ways, EmStar is similar to the Netbed project (based on Emulab [17]), which aims to provide a single system and configuration interface for code running in pure simulation, in network emulation mode, and distributed across the Internet. Similarly, ModelNet [16] runs real user code on "edge" nodes, and simulates Internet core routers. These systems are meant for Internet development, where the only interaction with the "environment" is a wired communication channel, and thus easier to emulate without resorting to physical interfaces. In contrast, EmStar's execution platforms include physically situated modes because the sensor domain requires it.

TOSSIM [11] is notable because it is used in the sensor network community for testing the same TinyOS code as runs on motes before the motes are deployed. However, the combination of TinyOS and TOSSIM has only two modes: pure simulation and real deployment. EmStar adds intermediate modes such as the ceiling and portable arrays, which provide a combination of simulator-like visibility with a real channel.

Pure simulators play a very valuable but different role than EmStar. For example, *ns-2* [2] has been optimized to handle very large simulations, offers scripting of scenarios far complex than currently possible in *emsim*, and has a number of detailed wireless channel models that are good enough for evaluating many protocols. However, *ns-2* does not run real code; this is an important drawback if the goal is to deploy code, or use simulation to track down a bug observed in code that has already been deployed. In addition, most pure simulators do not provide support for physical interfaces such as sensors.

Though meant for different domains (sensor networks vs. Internet routers) EmStar is similar to Click [10] in its design philosophy and programming model. Click and EmStar both let developers create protocol stacks by creating a graph composed of small modules that have standard interfaces. If Click code is run in the kernel, it allows interactive inspection and modification of state using devices in the `/proc` filesystem. Similarly, EmStar allows debugging using Status Devices, as we described in Section 3.2. In contrast, we use our FUSD microkernel extension to facilitate this feature for user-space code. In addition, FUSD lets our interfaces be more expressive: in addition to packets, interfaces can be programmed with stream, remote procedure call, or event-notification semantics.

6 Conclusions and Future Work

The sensor network community has moved out of its infancy and is now embarking on more serious design efforts—large-scale, long-lived systems that truly require self-organization and adaptivity to the environment. A number of important software design issues have arisen that are unique to this domain. First, the use of local processing, hierarchical collaboration, and domain knowledge to convert data into distilled, high-level representations—or, *data reduction*—is key to meeting the energy and channel capacity constraints of the system. However, the data that are discarded are necessary for the evaluation and debugging of the data reduction process itself. A second problem is that a traditional layered protocol stack, which is designed to emphasize conceptual abstraction and reusability, has too high of an efficiency cost in this domain where efficiency is paramount.

In this paper, we described EmStar, an environment for development of Linux-based wireless sensor network software. EmStar provides a modular programming model that allows domain knowledge in one module to affect the modules around it, without sacrificing the advantages of reusability and abstraction that strict layering provides. Services may be flexibly interconnected using standard interfaces; the connections can transmit a flow of packets, stream data, state updates, or configuration commands.

EmStar code may be run on a diverse set of execution platforms; each run the same code and use the same configuration files, making it easy for developers to seamlessly iterate among all the modes. In addition to a true distributed deployment platform, EmStar provides a number of development modes that facilitate debugging and evaluation:

- *Pure simulation*—Simple models of the communication and sensor channels define (for example) the effective range of each packet and the input of sensors. Simulation offers complete control over and visibility into the system being tested. For example, systematic testing of scenarios is possible, and behavior can be made deterministic so as to make problems more consistently reproducible. Many developers can work in parallel; simulators are ordinary PCs, which (currently) are more ubiquitous than the specialized hardware used in sensor networks.

- *Ceiling array*—We permanently mounted an array of low-power wireless transceivers on the ceiling of our lab, all wired back to a simulator-class machine. As in pure simulation, ceiling array node instances are run centrally. However, instead of using a channel model, the simulated nodes communicate by sending and receiving real packets through the real channel. This lets developers experience real channel dynamics without sacrificing the ease of development afforded by the simulator.

- *Portable array*—A portable version of the ceiling array allows radio and sensor interfaces to be taken to the real environment that will be used for deployment. Logistical limitations prevent large-scale testing, but this mode is valuable because it provides high visibility into a system running in its actual target environment.

EmStar allows developers get the basics of an algorithm working in a controlled environment (simulation); then, understand both the effects of scale (via a large simulation) and the effects of the real environment (via the ceiling and portable arrays). Code that has been debugged using all the modes has a good chance of working in a real-world deployment, where it must both be scalable *and* deal with the effects of the real environment. While deployed code may not work immediately, an immense amount of real progress can be made in a much more friendly environment.

In the future, there are still a number of important features we would like to incorporate into EmStar:

Non-realtime simulations—Currently, pure simulations run in real-time. Faster-than-realtime simulations would be useful to speed up the process of performing many simulations with varying configurations. Slower-than-realtime simulations are also useful in support of very large simulations, where the resources of the simulation machine (CPU speed, memory capacity and bandwidth, etc.) are less than to the sum of the resources of the nodes being simulated.

Parallel simulations—Although *emsim* can make full use of multiple processors on an SMP machine, we would like to enable larger simulations by allowing them to be split across multiple computers. We hope to apply a partitioning scheme similar to that described in ModelNet [16].

Data replay mode—As we mentioned in Section 2.5, Data Replay mode has not yet been widely used, and will complete our spectrum of execution environments. This mode will be especially useful for domains in which large databases of time-series data are already available (e.g., seismology).

Hybrid simulation and reality—Currently, EmStar platforms require that nodes either be all simulated or all real. We would like to enable hybrid modes that combine both types of nodes, allowing the study of scale and real channel characteristics in a single experiment.

Physically situated sensors in emcee—So far, development on the ceiling and portable arrays has focused on reacting to characteristics of the real communications channel. We would like to use these tools to develop algorithms that are reactive to real dynamics seen in physically situated sensors, as well.

We anticipate that these improvements will be useful, but ultimately our ambition is to implement a fully-fielded system that has grown up in the EmStar development environment. Two such systems are in development: a 100-node tiered-architecture microclimate array, and a 50-node collaborative multi-hop seismic array. We are working with our partners in the natural sciences to create a system that is both scientifically useful and advances the state of the art in sensor system design.

Project URL

Code and documentation for EmStar can be found at the URL: http://cvs.cens.ucla.edu/emstar

Acknowledgments

EmStar was made possible through the support of the National Science Foundation Cooperative Agreement #CCR-0120778, and matching grants from Intel Corporation and Sun Microsystems. Additional support was provided by the DARPA NEST program (the "GALORE" project, grant F33615-01-C-1906) and the University of California MICRO program (grant number 01-031). Sensoria Corporation provided valuable feedback and support.

Many individuals made valuable contributions to EmStar, including Simon Han, John Heidemann, Eric Osterweil, Nithya Ramanathan, Roy Shea, Fabio Silva, Mani Srivastava, Hanbiao Wang, and the intrepid students of CS213, who braved using EmStar while it was still very raw.

References

[1] Familiar Linux. http://www.handhelds.org.

[2] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[3] Alberto Cerpa, Jeremy Elson, Deborah Estrin, Lewis Girod, Michael Hamilton, and Jerry Zhao. Habitat monitoring: Application driver for wireless communications technology. In *Proceedings of the SIGCOMM Workshop on Communications in Latin America and the Carribean*, Costa Rica, April 2001.

[4] Alberto Cerpa and Deborah Estrin. ASCENT: Adaptive self-configuring sensor networks topologies. In *Proceedings of the Twenty First Annual Joint Conference of the IEEE Computer and Communications Societies*

(INFOCOM 2002), New York, NY, USA, June 23–27 2002. IEEE.

[5] Benjie Chen, Kyle Jamieson, Hari Balakrishnan, and Robert Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. *ACM Wireless Networks*, 8(5), September 2002.

[6] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine–grained network time synchronization using reference broadcasts. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163, Boston, MA, December 2002.

[7] Homayoun Hashemi. The indoor radio propagation channel. *Proceedings of the IEEE*, 81(7):943–68, July 1993.

[8] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the Symposium on Operating Systems Principles*, pages 146–159, Chateau Lake Louise, Banff, Alberta, Canada, October 2001. ACM.

[9] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Arhitectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Cambridge, MA, USA, November 2000. ACM.

[10] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[11] Phil Levis and Nelson Lee. Simulating tinyos networks. http://www.cs.berkeley.edu/ pal/research/tossim.html.

[12] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Boston, MA, USA, December 2002.

[13] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the First ACM Workshop on Wireless Sensor Networks and Applications*, Atlanta, GA, USA, September 28 2002.

[14] G. Pottie and W. Kaiser. Wireless sensor networks. *Communications of the ACM*, 43(5):51–58, May 2000.

[15] C Schurgers, V Tsiatsis, and M Srivastava. STEM: Topology management for energy efficient sensor networks. In *IEEE Aerospace Conference*, pages 78–89, March, 2002.

[16] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 271–284, Boston, MA, December 2002.

[17] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[18] Ya Xu, Solomon Bien, Yutaka Mori, John Heidemann, and Deborah Estrin. Topology control protocols to conserve energy in wireless ad hoc networks. Technical Report 6, University of California, Los Angeles, Center for Embedded Networked Computing, January 2003. submitted for publication.