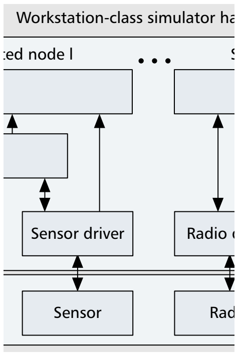
# EMSTAR: DEVELOPMENT WITH HIGH SYSTEM VISIBILITY

JEREMY ELSON, MICROSOFT RESEARCH
LEWIS GIROD AND DEBORAH ESTRIN, UCLA CENTER FOR EMBEDDED NETWORKED SENSING



EmStar's novel execution environments encompass pure simulation, true in-situ deployment, and hybrid modes that combine simulation with real wireless communication and sensors situated in the environment.

## 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. To make software robust, we must carefully observe its behavior and understand its failure modes. However, many of these failures are not manifested until deployment time. Channel and storage limitations make visibility into a deployed system difficult, hindering our understanding of failure modes. 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. In this article we describe EmStar, a PC-based software framework that aims to make development easier by improving system visibility. EmStar's novel execution environments encompass pure simulation, true in-situ deployment, and hybrid modes that combine simulation with real wireless communication and sensors situated in the environment.

## 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 [1, 2], groundwater contaminant monitoring, precision agriculture, and condition-based maintenance of machinery in complex environments.

As sensor network research has moved out of its infancy, its focus has started to shift away from short-lived hand-configured tests and demonstrations. We are seeing the emergence of real applications: longer-lived, larger-scale sensor systems that are situated in real environments and collect real data. The drive to deploy real systems has been slow and difficult; designing software for sensor networks is hard.

The difficulty's origin is a confluence of factors. First, sensor networks must have software that is autonomous and robust despite dynamics in the system and environment. For example, topologies must be discovered, not preconfigured; node failures must be automatically detected, as they cannot be manually repaired. Second, the dynamics are difficult to predict. Experience has shown that we learn about the many failure modes only from in situ deployments; simulation and analysis are not enough. Third, the constrained storage and channel capacity makes it difficult to collect data from a deployed system that sheds light on the details of its internal behavior. This lack of system visibility interferes with the feedback process that normally drives system development: design an algorithm, implement it, observe its behavior, and change the design based on the analysis.

This article describes EmStar, our PC-based framework that addresses the difficulties in creating robust sensor network software. EmStar's execution environments address the problem of visibility into an in situ system. It provides a spectrum of runtime platforms: pure simulation, 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.

## DESIGN OF AUTONOMOUS SOFTWARE

Sensor networks must be autonomous at a level required by few other distributed systems. They are in dramatic contrast to most existing computer systems, which can be designed with the assumption that users are on hand to solve problems. A user can exercise judgment about what to delete if her disk is full. If a network connection is broken, she can decide how often to try again and when to give up, considering the nature of the failure, cost of the connection, or importance of the work. She can recognize bugs in software and adapt the way she works to avoid encountering them again. She can even cope with partial hardware failures; for example, she can send a document to the office printer instead of using the one at home.

In a sensor network, humans are not in the loop. Its software must adapt to failures and unexpected conditions using only the mechanical intuition built into it by the system designer. Encoding intuition into software is not easy. While this may not be a crucial issue for short-lived demonstrations, it is a stumbling block for real deployments. As the lifetime of a system grows, so do the number and variety of unexpected situations encountered. Over time, nodes can run out of energy, produce bad data after overheating in the sun, be carried away to a new location by wind, flood the network due to software bugs, or be confounded by unusually noisy sensors. Even in fixed positions, the quality of radio frequency (RF) communication links (and thus nodes' topologies) can change dramatically due to the vagaries of RF propagation. These changes are a result of propagation's strong environmental dependence and difficult to predict in advance. In sensor networks, such situations are the rule rather than the exception.

Unfortunately, experience has taught us that it is hard to expect the unexpected. When designing a sensor network application, there are many lessons we learn only from in situ deployments; simulation and analysis are not enough. To write robust software, feedback from failures of early deployment prototypes is crucial. To understand these failures, we need *visibility* into a deployed system.

## The Importance of System Visibility

Imagine we want to build a simple network of temperature sensors that can report the average room temperature inside a building. We want to maximize the lifetime and scalability of the network by minimizing the number of energy-consuming radio transmissions, so we design the network to organize itself into a hierarchy. Each node is programmed to report its temperature observation to its parent in the aggregation tree. Internal nodes average the readings of their children and forward just a weighted average — but not all the raw data — up the tree. The node at the root of the tree is connected to a base station, informing the user of the system-wide average. This scheme scales nicely because only a single datum is transmitted over each link in the tree, independent of network size.

Suppose we write a software prototype, load it on a dozen sensors spread around a room, and turn it on for the first time. After a few minutes, an answer pops out: *77 degrees*. Is that correct? How can we tell if the software is working properly? Was there a bug in our code that performs time-weighted averaging? Is the spatial aggregation working? Should we have used a different filtering algorithm to eliminate noise from the sensor? Does the average really represent all the sensors, or did half of them malfunction? Was the routing hierarchy stable, or constantly in flux due to variations in link quality? Were packets lost in routing loops? Unfortunately, all we have is a single number — 77 degrees. We have no data to answer these questions.

In many networked systems, the cost of gathering the data needed for analysis and debugging of a deployed system has a relatively low marginal cost. An Internet router, for example, transports billions of data bits every second. Additional transmission of a few thousand bytes of system analysis data has a negligible impact. In sensor network the situation is reversed. Sensor networks are designed to carry very low-rate data; typical bandwidths are a few kilobits per second, duty cycles are low, and storage capacity is limited. Additional data for debugging and analysis, such as routing tables and link status matrices, are hundreds of times larger than, for example, a simple temperature reading. Consequently, it can be impossible to collect such meta-data from a deployed system without dramatically changing the network under observation. The cost of system visibility is not marginal as in a router; in sensor networks it dominates.

Low bandwidth and short range are a necessity in sensor networks because the network has finite energy, and communication is the primary consumer of this precious resource [3]. Sensor nodes and their batteries are both small; in typical deployments, recharging is infeasible. The use of local processing, hierarchical collaboration, and domain knowledge to convert data into increasingly distilled and high-level representations — *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?

## EmStar

EmStar is our Linux- or Windows-based framework for developing sensor network software. It addresses the issue of efficiently gaining experience with real-world deployments without sacrificing visibility into the system that is crucial for understanding it.

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. EmStar can provide both the convenience of simulation and the reality afforded by physically situated devices.

We will describe each point along this spectrum in detail, but their character varies chiefly along two axes, as depicted in Fig. 1:

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.

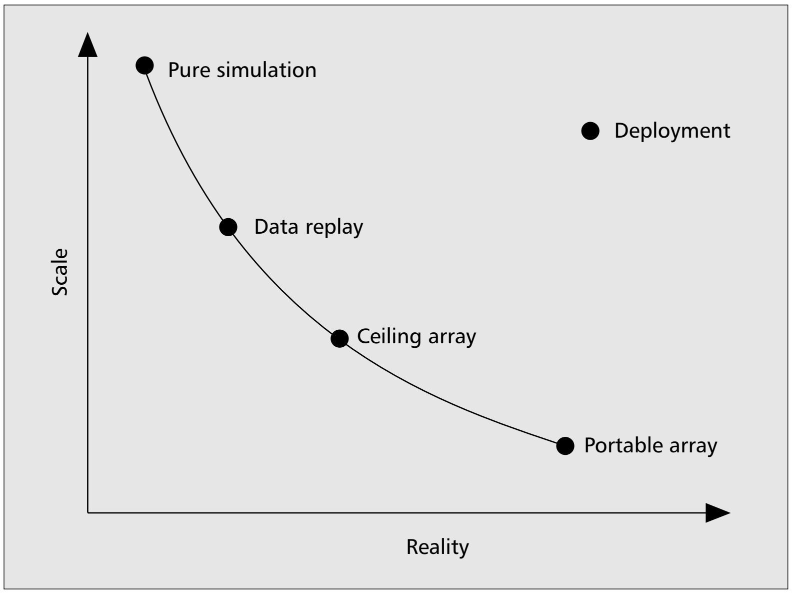


**Figure 1.** *The spectrum of EmStar execution environments. Points along the arc allow high visibility into the system, enabling detailed analysis and improvement of its behavior. By understanding the effects of both scale (via simulation) and the real environment (via the ceiling and portable arrays),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

- *Scale*: The number of nodes in the sensor network and their geographic extent
- *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 earlier: 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 [4]. Connectivity is unpredictable and has been shown to vary significantly on both short and long timescales. The difference between 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 faraway, normally unreachable neighbor. A developer who 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 Fig. 1. EmStar allows developers to get the basics of an algorithm working in a controlled environment (simulation); then understand the effects of both scale (via a large simulation) and 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 describe each of EmStar's execution environments in more detail.

## 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.

Each component of the EmStar stack is implemented in 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 (Fig. 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 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 on the essence of the problem at hand. Parallel work by multiple developers is also difficult; most laboratories do not have enough deployable hardware for more than one developer to simultaneously test a large-scale deployment.

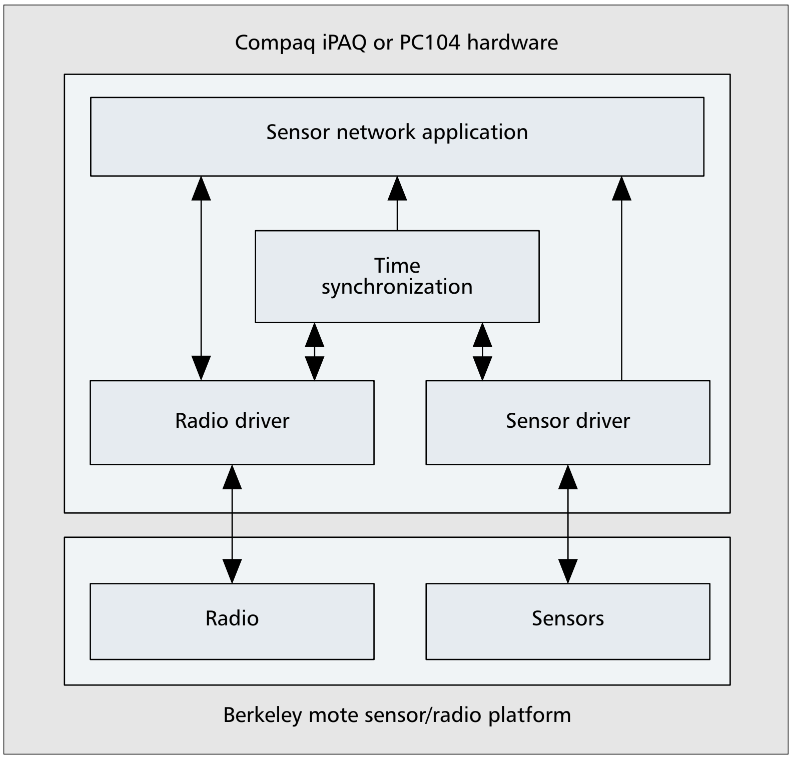

Compaq iPAQ or PC104 hardware
Sensor network application
Time synchronization
Radio driver
Sensor driver
Radio
Sensors
Berkeley mote sensor/radio platform

**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 interprocess communication are described in [5].*

## 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 (Fig. 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 is not always completely transparent, however. One group using EmStar recently developed a floating-point signal correlator which worked fine on the x86-based simulator. When run on a StrongARM-based embedded node, it ran very slowly: the StrongARM has no 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 described in more detail in [5], EmStar's 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 nodes, 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. 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)

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.

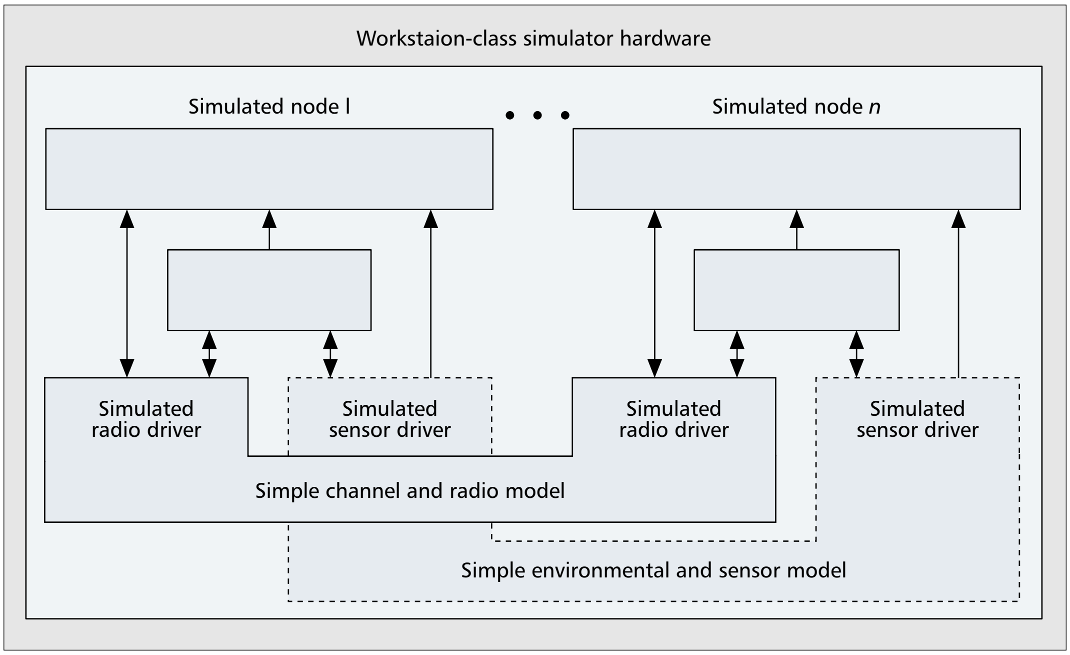


**Figure 3.** *The structure of* emsim *in pure simulation mode. For each node, an instance of* emrun *is launched, creating a stack such as the one in Fig. 2. However, instead of physical devices, simple radio channel and sensor input models moderate each node's interactions with the physical world.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

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 times 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 (Fig. 4), algorithms that are sensitive to the subtleties of the channel are 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 described 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.

## THE CEILING ARRAY

Roboticist Rodney Brooks has famously observed, "The world is its own best model." This guidance is also apt for sensor networks that, 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 [4]. 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.

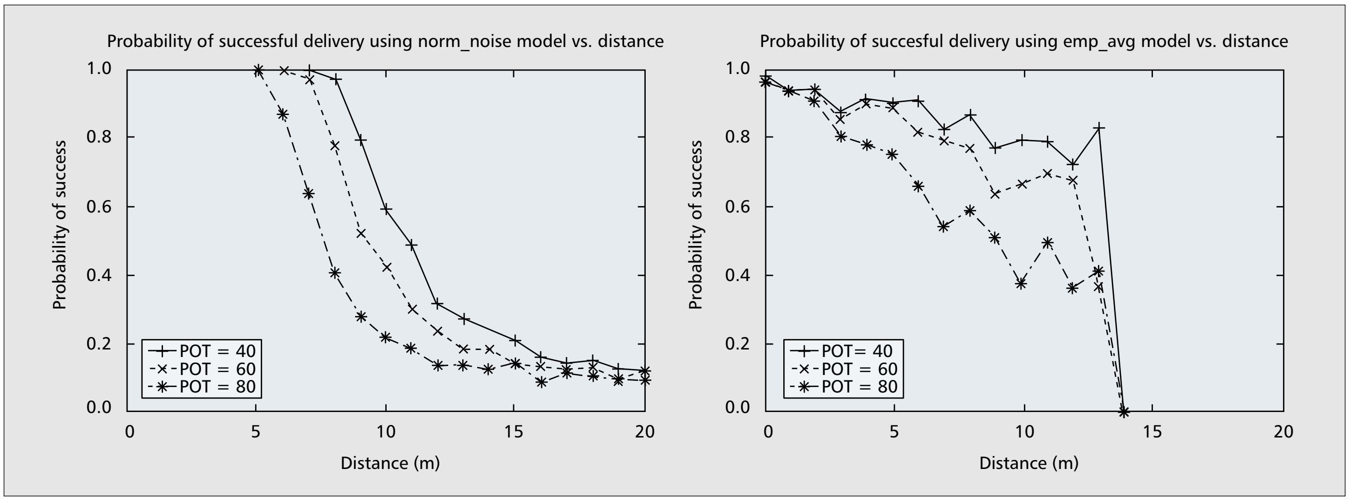


**Figure 4**. *Two of the* emsim *channel models. Not shown is the deterministic circle model, in which nodes less than 8 m apart can exchange packets with 100 percent reliability, while nodes separated by longer ranges can never exchange packets. While unrealistic, the determinism is helpful for debugging fledgling applications. a) 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. b) 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 [6].*

We permanently mounted a uniform array of 55 motes to the ceiling of the CENS systems laboratory, pictured in Fig. 5. The motes are all wired for power and have a 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 (Fig. 6).

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 7 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

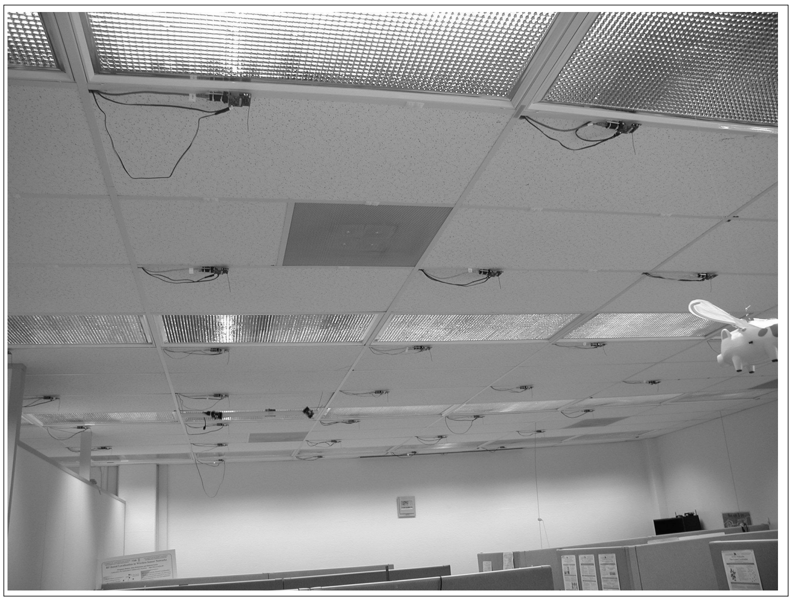


**Figure 5**. *Fifty-five wireless network interfaces, mounted on the ceiling of our laboratory and wired back to a simulation server, help to bridge the gap between simulation and reality. Developers can learn about the effects of real wireless channels in an environment as convenient as simulation.*

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 simu-

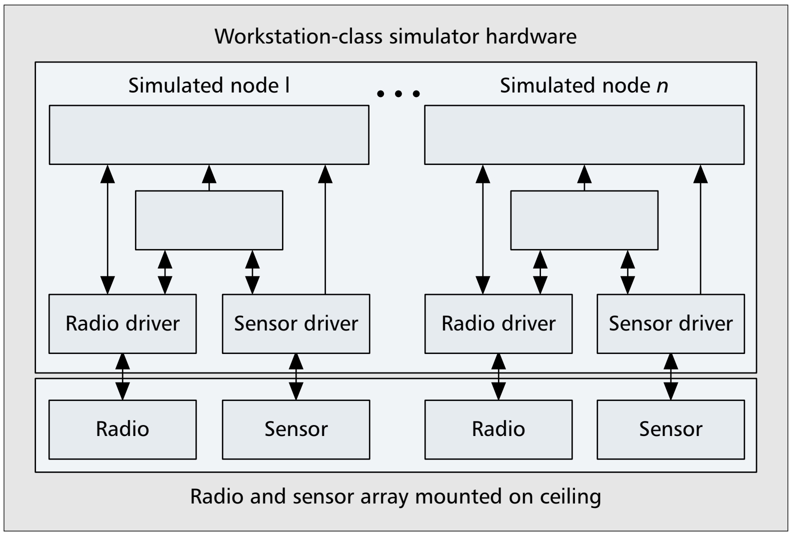


**Figure 6**. *The structure of* emcee*, a hybrid mode that combines simulation with real channels. As with* emsim*, multiple instances of* emrun *are run, including real device drivers. The drivers are attached to 54 Berkeley motes permanently mounted on the laboratory ceiling.*

lation, 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 a 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 laboratory 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.

## 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 seen in Fig. 1: scale.

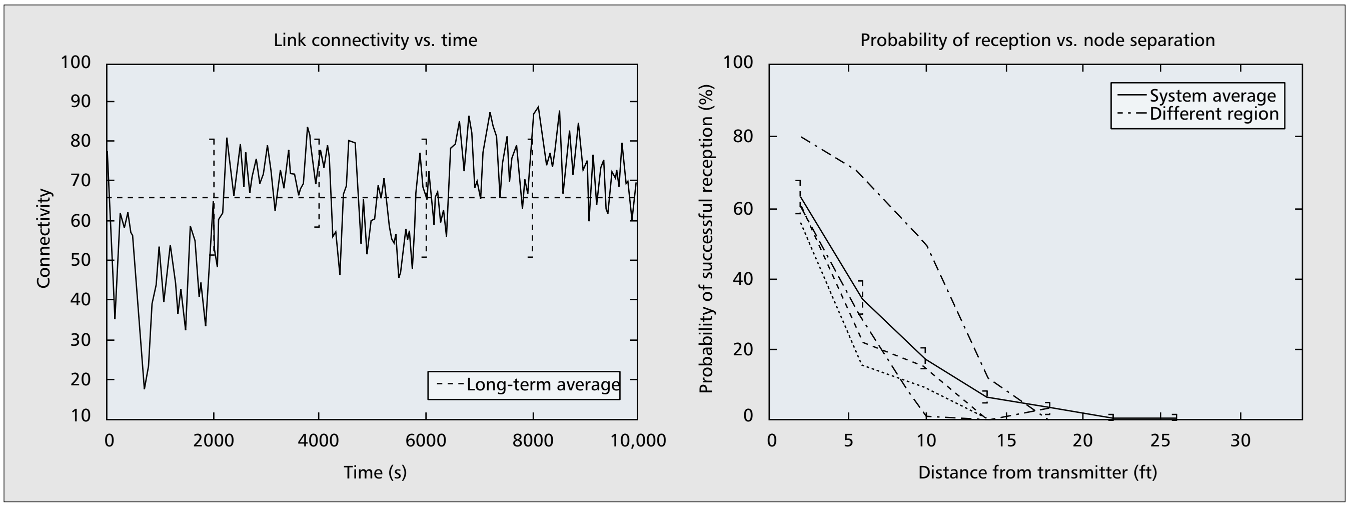


**Figure 7**. *a) A pronounced change in the probability of reception between a fixed pair of nodes within the duration of a single experiment; b) the effects of spatially correlated noise. The lower three curves represent three nodes in a particular region of the network that has lower than average 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.*

## DATA REPLAY

In the previously described 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 adds a new dimension: sensor inputs can be recorded or taken from other sources such as existing seismic arrays or vehicle transportation data. Later, these stored sensor time-series can 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 is 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.

## CONCLUSIONS AND FUTURE WORK

Robust software is needed to make the sensor network vision a reality. To make software robust, designers must experience the failure modes of sensors in the real world — early and often. EmStar helps to ease the transition from prototype to working system by increasing system visibility and thereby shedding light on failure modes. Developers get the basics of an algorithm working in a controlled environment (simulation); then understand the effects of both scale (via a large simulation) and 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.

While EmStar has proved useful, there is still much work to be done. For example, our work with *emcee* so far has focused on communications research. By adding sensors to the array, we can gain experience with real sensor inputs earlier than in a fielded deployment. In addition, we plan to create a public testbed based on EmStar, allowing researchers without access to large-scale hardware resources to test their algorithms in the real world, rather than remaining limited to simulation-based research.

We anticipate that these and other improvements will be useful, but ultimately the test will be the success of fully fielded systems that have 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 multihop 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.

We plan to create a public testbed based on EmStar, allowing researchers without access to large-scale hardware resources to test their algorithms in the real world, rather than remaining limited to simulation-based research.

## ACKNOWLEDGMENTS

EmStar was made possible through the support of 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.

EmStar is a group effort; contributors include Naim Busek, Vlad Bychkovskiy, Alberto Cerpa, Deepak Ganesan, Ben Greenstein, Eric Osterweil, Nithya Ramanathan, Roy Shea, Fabio Silva, Tom Schoellhammer, Thanos Stathopoulos, Hanbiao Wang, and Yan Yu. More information can be found on our project Web page, http://cvs.cens.ucla.edu/emstar.

## REFERENCES

[1] A. Cerpa *et al.*, "Habitat Monitoring: Application Driver for Wireless Communications Technology," *Proc. SIGCOMM Wksp. Commun. Latin America and the Carribean*, Costa Rica, Apr. 2001.
[2] A. Mainwaring *et al.*, "Wireless Sensor Networks for Habitat Monitoring," *Proc. 1st ACM Wksp. Wireless Sensor Networks and Apps.*, Atlanta, GA, Sept. 28, 2002.
[3] G. Pottie and W. Kaiser, "Wireless Sensor Networks," *Commun. ACM*, vol. 43, no. 5, May 2000, pp. 51–58.
[4] H. Hashemi, "The Indoor Radio Propagation Channel," *Proc. IEEE*, vol. 81, no. 7, July 1993, pp. 943–68.
[5] L. Girod *et al.*, "Emstar: A Software Environment for Developing and Deploying Wireless Sensor Networks," *Proc. 2004 USENIX Tech. Conf.*, Boston, MA, June 2004.
[6] A. Cerpa and D. Estrin, "ASCENT: Adaptive Self-Configuring Sensor Networks Topologies," *Proc. INFOCOM 2002*, New York, NY, June 23–27, 2002.

## BIOGRAPHIES

JEREMY ELSON (jelson@microsoft.com) is a researcher in the Networked Embedded Computing laboratory of Microsoft Research. He earned his Ph.D. in computer science in 2003 at the University of California at Los Angeles' (UCLA's) Center for Embedded Networked Sensing. His dissertation work on time synchronization in low-power wireless sensor networks, advised by Prof. Deborah Estrin, earned the Edward K. Rice Outstanding Graduate Student award. His other research interests include operating system issues and programming models in distributed self-organizing networks.

LEWIS GIROD is a Ph.D. candidate in computer science at UCLA. He received his B.S. and M.E. in computer science from Massachusetts Institute of Technology in 1995. After working at LCS for three years in the area of Internet naming infrastructure, he joined Deborah Estrin's group in 1998. His research focus is the development of robust networked sensor systems, specifically physical localization systems that use multiple sensor modalities to operate independent of environment and deployment.

DEBORAH ESTRIN (Ph.D., MIT 1985) is a professor of computer science at UCLA and director of the NSF Science and Technology Center for Embedded Networked Sensing (CENS) http://cens.ucla.edu. She helped to define the research agenda for wireless sensor networks, chairing a 1998 DARPA study and a 2001 National Research Council study. Her research addresses protocols for autonomous, distributed, and physically coupled wireless systems, with a particular focus on environmental monitoring applications. Her earlier research focused on Internet scaling and routing.