Emstar: a Software Environment for Developing and Deploying Heterogeneous Sensor-Actuator Networks

LEWIS GIROD[‡], NITHYA RAMANATHAN[†], JEREMY ELSON[♯],
THANOS STATHOPOULOS[†], MARTIN LUKAC[†], DEBORAH ESTRIN[†]

‡ MIT CSAIL
Cambridge, MA

† Center For Embedded Networked Sensing
University of California, Los Angeles
Los Angeles, CA

♯ Microsoft Research
Redmond, WA

Recent work in wireless embedded networked systems has followed *heterogeneous* designs, incorporating a mixture of elements from extremely constrained 8- or 16-bit "Motes" to less resource-constrained 32-bit embedded "Microservers".

Emstar is a software environment for developing and deploying complex applications on such heterogeneous networks. Emstar is designed to leverage the additional resources of Microservers by trading-off some performance for system robustness in sensor network applications. It enables fault isolation, fault tolerance, system visiblity, in-field debugging, and resource sharing across multiple applications.

In order to accomplish these objectives, Emstar is designed to run as a multi-process system and consists of **libraries** that implement message-passing IPC primitives, **services** that support networking, sensing, and time synchronization, and **tools** that support simulation, emulation, and visualization of live systems, both real and simulated. We evaluate this work by discussing the Acoustic ENSBox, a platform for distributed acoustic sensing that we built using Emstar. We show that by leveraging existing Emstar services, we are able to significantly reduce development time while achieving a high degree of robustness. We also show that a sample application was developed much more quickly on this platform than it would have been otherwise.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

Author's address: L. Girod, MIT CSAIL, 32-G918, 32 Vassar St. Cambridge, MA 02139.
This work was made possible with support from The Center for Embedded Networked Sensing (CENS) under the NSF Cooperative Agreement CCR-0120778, and the UC MICRO program (grant 01-031) with matching funds from Intel Corp.

General Terms: Design, Experimentation, Measurement

Additional Key Words and Phrases: Sensor Networks, Real Code Simulation, EmStar, TinyOS

1. INTRODUCTION

As new applications require more sophisticated tasks of wireless sensor-actuator networks, system complexity tends to increase. From the standpoint of minimizing overall complexity, the most expedient solution usually simplifies the end nodes by centralizing as much of the computation as possible. However, this strategy is impossible when the application involves sensors with high data rates such as audio [Wang et al. 2004] [Chen et al. 2003] or video, or requires low-latency actuation within an unreliable network [Merrill et al. 2004]—these cases force a distributed implementation.

These types of distributed system are notoriously difficult to deploy. Through our initial experiences in this area we identified several impediments to deploying distributed sensing systems, and outlined key system requirements necessary to overcome these hurdles.

First, hardware and software components are prone to fail as a result of harsh deployment environments, unexpected inputs, and unreliable hardware. This is particularly true during the development process, when the environment is not fully understood, and components of the hardware and software are still in the prototype stage. Because failures are common, systems must be able to continue functioning despite their presence. This leads to two separate but related system requirements: *fault isolation*—a failure in one software component (e.g. memory corruption) should not corrupt the entire system; and *fault tolerance*—a failure in one hardware or software component should not result in a system reboot.

Second, the ability to debug such a failure in the field is critical but extremely difficult. For example, in deploying 20 microphones, at least one is bound to fail. If the entire system fails to start because one software component cannot access this device, it will be much more difficult to find and fix the failure; thus ensuring *in-field debugging* and *system visibility* despite failures is critical.

Third, the high cost of development and deployment motivates platforms that support a broad class of applications [Girod et al. 2006], and deployments that can host multiple independent users. This is accomplished through *resource sharing* via reusable and composable services that arbitrate access to resources.

While many early projects produced inspiring demos (for example, many early demos in the NEST DARPA program), these efforts typically provided little leverage for follow-on work. Systems that make a good demo often disregard details that are critical for practical deployments, *e.g.*, setup and configuration is typically done by the developers themselves, and the system is often calibrated to fit the specific environmental conditions at the demo. Furthermore, the development process for demos typically cuts corners that rule out applying the system in a broader context.

One of the primary motivations for Emstar came from our experience developing an acoustic localization system [Girod and Estrin 2001; Girod et al. 2002; Girod et al. 2006]. This work went through several iterations of increasing sophistication, in which the robustness and applicability of the system improved.

In the earliest version [Girod and Estrin 2001], the software was written as a single monolithic process that processed audio I/O and transmitted packets via a radio module interfaced through the parallel port. Because this implementation was highly integrated, it was difficult to add more complex functionality such as the ability to time-synchronize over multiple RF hops. The implementation also made it difficult to implement other applications without integrating them with the localization system, because access to key hardware components (*e.g.*, the radio and the audio interface) were exclusively held by the localization software.

In the second version of the system [Girod et al. 2002], these problems were addressed by decomposing the system into services. For example, the radio module was managed exclusively by a service, but many applications could access it. By decoupling these services, their implementations could be made more sophisticated with lower overall complexity. However, new difficulties arose in *managing* a system composed of many interlocking services. The new system worked well as a demo, but was difficult to operate and was not robust to component failures, which would sometimes occur because of imperfections in the setup of the equipment, or when the system was tested for the first time in a new environment.

In the final version of the system [Girod et al. 2006], the ideas that compose Emstar were fully applied. The interlocking services from the previous version were updated to use the Emstar IPC and design principles, yielding a robust system that can be used by non-engineers, outside of the context of a demo. Ad-hoc, jury-rigged solutions to the problems of managing the system were replaced by reusable components supplied with Emstar, providing many common services from system health monitoring and fault reporting to web-based management tools. We discuss this final version in more detail in Section 4.

This incremental development process drove the development of Emstar. In this work we set out to provide a reusable set of services that speed the development of practical, deployable distributed sensing systems, within a common, extensible framework. The services and the framework speed development by providing a substrate for developing distributed sensor systems and by solving many common practical problems, lessening the temptation to just "hack it together" while saving considerable development time. Through this one-time investment in infrastructure, future systems can explore new application domains with increased sophistication.

Emstar's driving design goal is to enable and facilitate the deployment of resource-constrained wireless sensing systems. To this end, Emstar is an event-driven system comprised of a framework that enables fault tolerance and field preparation, a communication layer that facilitates fault diagnosis, and a set of commonly required services that enables quick construction of applications. What makes Emstar unique is its basic philosophy in which services are loosely couped through stateless status channels and an emphasis on making the interfaces between these services visible. What makes Emstar usable is the set of libraries, communication primitives (i.e. device patterns), easily composable services, visualization tools, and simulation framework that also compose this system.

In this article, we present the Emstar software system in detail. In Section 2 we explain the design decisions we made in the design of Emstar, and some of the trade-offs. In Section 3 we show that Emstar can be decomposed into several layers, and we explain each layer in more detail. In Section 4 we evaluate Emstar by describing how we have used it to build a complex distributed application. Section 5 contains Related Work and we conclude with lessons learned in Section 6.

2. DESIGN DECISIONS AND TRADE-OFFS

Based on the requirements of this application area as described in the previous section, we made a series of design decisions with corresponding trade-offs. These decisions resulted from early experiences with distributed wireless sensing systems, and from considering the ways in which existing programming interfaces meshed or clashed with the needs of these systems.

Overall the design is motivated by the desire to trade-off some performance for system robustness and visibility in developing applications for "microserver" sensor nodes. Emstar was *not* designed to write software for extremely constrained nodes such as the Berkeley Mote, and thus operates in a very different efficiency regime. While energy is ultimately still an important consideration, it has not been a central focus of Emstar development. In addition, since Emstar operates exclusively at the application layer, it cannot directly control hardware or process interrupts with low latency. Whenever events must be timestamped accurately, the timestamps must be captured from within the kernel and exposed to Emstar applications.

The two most influential decisions in the Emstar design were guided by our design philosophy: 1) The *multi-process service architecture* enables loosely coupled services; and 2) the *device file IPC* enables the interfaces between these services to be easily browseable from the command line. We discuss Emstar's key design decisions in the following sections.

2.1 Multi-process service architecture

Emstar is built as a collection of separate processes providing services. This decision is in fact two independent choices: first, the decision to provide a set of common services to support multiple applications, and second, the decision to implement these services as separate processes.

The first choice was made based on the observation that Microservers had sufficient resources to concurrently support multiple independent applications. The ability to support multiple applications was a requirement of some of the first instances of sensor nodes, notably the WINSNG 2.0 platform [Merrill et al. 2002] used in the DARPA SensIT program. In this program, several independent research groups ran applications on a single fielded network. Given that applications act independently, there was therefore a need to provide services that regulate access to shared resources.

The second choice, to implement the services as separate processes, was made to enable greater freedom in the implementation of the services as well as to provide fault isolation between services and to enable the system to tolerate faults in individual services more gracefully. By narrowing the scope of individual services, the initial implementation of each service was simplified, enabling implementations to grow in sophistication as requirements change. This approach also enabled us to readily compose a system from only the select services that were required for a given application. Building each service as a separate process also provided maximum flexibility of language choice and the ability to integrate arbitrary external components (such as proprietary code), rather than forcing the early adoption of any particular language or discipline.

By implementing Emstar as a group of processes instead of a single process we fulfilled several of our requirements for system robustness. Yet there are trade-offs to this approach, which are discussed in Section 2.1.3.

2.1.1 *Importance of fault isolation and fault tolerance.* Fault isolation and fault tolerance were critical elements of the reasoning for making this choice. Because in the early work in WINSNG we provided a platform to a set of customers, it was critical to be able to immediately distinguish between a failure of the platform services and a failure of the customers' applications. This principle holds true for today's implementation of Emstar, in which it is helpful to minimize the complexity of the Emstar client libraries in order to reduce the likelihood that an application crash is due to a library bug.

Fault isolation (i.e. the separation of services across process boundaries so a bug such as memory corruption in one process does not impact other processes) and fault tolerance (i.e. tolerance of service or application to failures by only rebooting the failed process instead of the entire application) are critical properties for fielded systems. Performing fielded experiments involves an immense investment of time and effort, and is in many cases time-critical (e.g. in the event that a scientific phenomenon only occurs at certain times of year). In addition, debugging systems while in the field is considerably more difficult. These two considerations suggest that there is a very high cost to bugs and failures that lie in the critical path of getting results. Thus, it is more desirable that the system limp along in spite of faults and still collect some data, than it is that the system suffer a catastrophic failure that requires in-field, high-pressure debugging. As long as the system is limping and the faults are reported and visible, the bugs can be diagnosed and fixed in parallel, or even left alone.

In our experience with these systems, we have had examples where systems exhibited failures only in the field, typically when a system is deployed in a new environment for the first time or when a system is re-deployed with modified system software. In these cases, we have made use of Emstar's fault-tolerance property. Because of the selection of service boundaries and interfaces, portions of the system can restart with momentary or even unmeasurable effects on application performance.

For example, in one instance a rare combination of range errors caused a numerical algorithm in the multilateration component to cause an FPU exception. This behavior had never happened in lab environments, and only cropped up with certain types of error in the field. After the problematic range estimates were refined, the errors were eliminated and the numerical algorithm converged properly. However, had the whole system rebooted each time this bug occurred, the refinements would never happen and the system would be very unlikely to ever converge. While nobody would argue that buggy systems should be preserved, the ability to work around unexpected failures is a critical part of building a robust system.

2.1.2 *Selection of service boundaries.* A key element of the design of the Emstar services is the careful selection of service boundaries and interfaces. In general, Emstar services are delineated such that the interface provided by a service is transactional, delivers a series of messages, and exhibits a low traffic volume. That is, connections between client and service are generally loose, stateless couplings—most often simply a series of periodic state updates sent from a server to multiple clients. This approach eliminates many of the semantic difficulties that typically plague RPC systems [Tanenbaum and van Renesse 1988]; in particular, simplifying crash recovery and eliminating problems associated with loss of state and repeated execution semantics.

For example, consider the case of a neighbor discovery service. Rather than a model where a client queries the neighbor service to iterate through the neighbor list, the client of the Emstar neighbor service may retrieve a complete and consistent list at any time, and can register to receive a new neighbor report whenever the neighbor list changes. This approach serves to simplify the interface between client and server and to decouple the client from the server, reducing the importance of IPC performance by greatly limiting the frequency of IPC transactions. By relying wholly on passing complete messages, it also eliminates synchronization problems arising from concurrency and inconsistencies arising from service or client restart.

2.1.3 *Alternative solutions and trade-offs.* There are several trade-offs to a multiprocess architecture. The most obvious one is that defining IPC channels between processes is generally less efficient, less flexible and more complex than using simple function calls. In cases where memory protection is not required and language choice is not an issue (for example, if all applications are written in a common language and it is immune from memory errors), loading all applications into a single process space is more efficient and often simpler.

The other problem is that while partial restart may be helpful in deployment it is much more difficult to implement correctly than a full-system restart model. Prior work has addressed robustness by partial rebooting in several contexts, including Microreboot [Candea and Fox 2003] [Candea et al. 2004] for distributed systems and Nooks [Swift et al. 2004] for driver restart within a Linux kernel. The conclusion of these efforts has agreed with our own conclusions: enabling restart is often the best approach to designing robust complex applications, but it can be challenging to implement. In a careful implementation, partial restart works correctly and provides excellent robustness. However, if done incorrectly the system can experience various forms of livelock, or never properly recover.

2.2 Device File IPC

Another choice in the design of Emstar was to settle on a microkernel-like model for services. Rather than implementing interfaces to services using sockets, we implemented a kernel module called FUSD [Girod et al. 2004a] that enables a user-space process to expose device file interfaces by proxy through the kernel. These device files appear in the /dev filesystem and enable an interactive user to browse the state of an Emstar system in a similar way to the /proc and /sys filesystems which provide a view and control channel into a Linux kernel. This approach has certain advantages and disadvantages.

2.2.1 *Simple, library-optional interface.* The use of FUSD and device files for IPC is described in detail in Section 3. The key advantage of this approach is that no special library is required to access a FUSD-based IPC channel. Any software with access to the POSIX API can connect to the service by opening the device and making system calls. This means that these interfaces are accessible from the shell and from most simple UNIX utilities and scripting languages without additional support. Ousterhout's arguments on the relationships between scripting and system languages [Ousterhout 1998] apply here. Accessing Emstar services from scripts trades-off performance for significant ease of use and code re-use. By providing typeless ASCII interfaces in addition to typed binary interfaces, Emstar supports rapid development by "gluing" applications together.

Another advantage is that the blocking nature of a system call often simplifies the implementation of software that interfaces to these IPC channels. While notification that the channel is ready is signaled via select(), system calls on a FUSD device file are synchronous and blocking, which means that they can be used in simple, "straight-line" code without worrying about concurrency and synchronization. As long as the service hosting the interface is responsive, the latency of a call is generally low [Girod et al. 2004a]. The disadvantage of blocking semantics is the possibility of deadlock in the event of a cycle in the graph of FUSD clients and servers.

2.2.2 *No local-remote transparency.* One key disadvantage of FUSD IPC is that unlike sockets, FUSD devices do not natively provide local/remote transparency. However, this loss is less important when we observe that Emstar systems are typically embedded wireless systems for which remote access rarely behaves similarly to local access. This argument follows many similar arguments against the use of transparent RPC, e.g., in [Tanenbaum and van Renesse 1988]. For the common case of local access, FUSD achieves adequate performance and provides a convenient interface for clients that can be used directly via system calls. The less common case of remote access is addressed using the FUSDNet service, described in Section 3.3.5.1.

2.2.3 *Alternative solutions and trade-offs.* Tanenbaum and van Renesse [Tanenbaum and van Renesse 1988] cited a number of disadvantages to an RPC style interface, some of which we have already discussed: Emstar IPC is strictly local and tends to be used in a stateless, transactional manner. Many of the other concerns are addressed by the protocols that Emstar builds over the syscall RPC layer. Clients and servers are typically implemented as event-driven programs and thus can readily handle concurrent connections. Client-server role reversal is accomplished by dispatching events to the client via the select() system call, so that the client subsequently calls back to retrieve a message. "Multicast" communication from server to clients is achieved by notifying multiple clients at once and individually handling their retrieval requests; thus while the system calls are not "multicast", the semantics layered above can be.

When considering the implementation of IPC, there are many alternatives, most of which are based on sockets. Sockets have the advantage of higher performance and local-remote transparency, but have the disadvantages of not being browseable and of requiring a client library or a helper program to enable access. Using sockets would also eliminate the blocking RPC semantics that can provide a simpler "straight-line" programming model when making immediate calls to a service. On the other hand, a sockets-based solution would not be subject to deadlock.

Some libraries such as libasync [Dabek et al. 2002] enable asynchronous socket operations within an event-based system to appear as though they were "straight-line" blocking code. However, the very fact that these systems allow asynchronous operations means that their concurrency is more difficult to reason about. The fact that the code *appears* "straight-line" does not prevent other blocks of code from running concurrently. In contrast, using the Emstar event system and FUSD IPC calls, no two event handlers ever run concurrently.

By using the syscall API as its fundamental IPC mechanism, Emstar represents a simple implementation of a microkernel. As with many microkernels, the cost of crossing the kernel boundary on inter-service calls in Emstar is expensive. However, newer microkernels such as L4 achieve reasonably comparable performance [Härtig et al. 1997] to monolithic kernels using memory mapping mechanisms to share memory for large data transfers. Such an approach could also be used in Emstar for large memory transfers where buffer copies across kernel boundaries hurt performance. However, so far, the added complexity of shared memory has not been required as most of our applications have not required high data rates; thus the system robustness achieved with multiple processes outweighs the IPC performance hit.

2.3 Event driven vs. polling vs. threads

While Emstar does not mandate a particular execution model, the requirements of embedded sensing dictate a system architecture optimized for efficiency in idle periods. Whereas servers are typically designed to use polling to optimize for peak performance under high load, sensor systems are more often designed to stay vigilant but under-utilized for long periods of time. While servers stand to lose nothing by doing more work when under-utilized, sensor systems can extend system lifetime by triggering processing only when events occur.

In general, interleaved, interacting events are the common case for distributed sensing systems. Similar to the principles of "reactive robotics", distributed sensing systems tend to operate in a "reactive" mode in which their immediate behavior is heavily influenced by asynchronous and concurrent sensor inputs [Brooks 1986]. This reactive style fits well within an event-driven programming model, because events and inputs of different types arriving asynchronously must be integrated to influence the immediate behavior of the system. While threads can also be used to achieve this type of concurrency, synchronizing different types of events often makes threaded implementations more complex.

Most services in Emstar are implemented using a common set of helper libraries that impose a event-based execution model. In order to maintain responsiveness, no event handler can block. Long running computations are run in separate "compute threads", with message queues linking the thread with the main event loop.

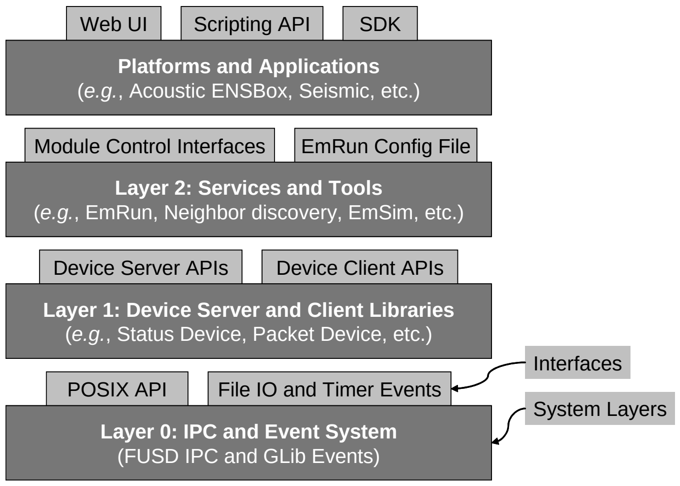

Web UI
Scripting API
SDK
Platforms and Applications
(e.g., Acoustic ENSBox, Seismic, etc.)
Module Control Interfaces
EmRun Config File
Layer 2: Services and Tools
(e.g., EmRun, Neighbor discovery, EmSim, etc.)
Device Server APIs
Device Client APIs
Layer 1: Device Server and Client Libraries
(e.g., Status Device, Packet Device, etc.)
Interfaces
POSIX API
File IO and Timer Events
System Layers
Layer 0: IPC and Event System
(FUSD IPC and GLib Events)

Fig. 1. Decomposition of the Emstar framework into layers and interfaces.

3. HOW EMSTAR WORKS

Software frameworks are by nature difficult to conceptualize because they exist only as the foundation and structure of a set of applications. However, a framework can be captured by describing the services and interfaces it provides, and the structure it imposes on an application. Emstar can be decomposed into three distinct layers connected by interfaces, as shown in Figure 1.

Layer 0 consists of the low-level code necessary for IPC and for implementing events. This layer motivates the construction of all Emstar applications as server and client processes, and is thus important to understand. Layer 1 is the heart of Emstar—libraries implementing common patterns of communication between client and server processes. Layer 2 consists of the set of re-usable tools and services, *e.g.*, time synchronization, routing, simulation, visualization, and system management, included with Emstar and used by applications written within Emstar. The layer at the top of the figure represents the platforms and applications built using Emstar, for example the Acoustic ENSBox system described in Section 4. Each of the underlying layers is described in more detail in this section.

3.1 Layer 0: FUSD Syscall IPC and the Event System

The lowest layer of Emstar is composed of FUSD [Girod et al. 2004a; Elson 2002] and the GLib event library. Together these provide two interfaces: the POSIX API used to connect clients and servers through character device files, and an I/O and Timer event API. It is important to note that while the existing version of Emstar implements these interfaces using FUSD and GLib, these components can be readily exchanged for other implementations that provide equivalent functionality. For example, while on Linux we use FUSD for IPC, much of Emstar could be ported to another operating system simply by replacing FUSD and GLib. In this section we describe the current Linux-based implementation.

The GLib event library is a standard component of most Linux distributions that provides an interface for registering File I/O and timer event handlers. Emstar registers events through the GLib API, and the GLib library runs a main loop that calls select().

FUSD is a custom micro-kernel interface implemented in Linux that allows user-space server processes to register character device files and handle system calls on those devices. FUSD provides a convenient way to enable cross-process message-passing, while at the same time exposing shell-accessible interfaces to internal state and control functions. In some respects, FUSD is similar to the AT&T Plan 9 system [Pike et al. 1990], but has the advantage of running over any platform running Linux rather than requiring the port of a complete OS to the latest embedded hardware. FUSD also has much in common with the procfs and sysfs features of Linux, which expose control and status interfaces to in-kernel features; the difference being that FUSD exposes interfaces to user-space processes. For example, Emstar services typically forgo the standard UNIX methods of logging and status—plain files in /var/log and /var/run—for active FUSD device interfaces that retain "backwards compatibility" with plain files.

FUSD implements the inter-process communication for the multiple processes that make up an Emstar system. Emstar clients interface to services through POSIX system calls on a FUSD device file that is registered by a server. This simplifies the implementation of clients by reducing all interaction with a server to a combination of straight-line blocking RPC calls and event notification via select(). Although the underlying implementation is wholly grounded in POSIX calls and the FUSD device API, most developers will operate at the higher layer of abstraction provided by libraries that support common device patterns and device clients. In the next few sections we describe FUSD and show how FUSD client-server connections provide fault isolation.

3.1.1 *System Calls as Blocking RPC.* A system call on a FUSD device represents a blocking RPC call to the server, brokered by the kernel. For example, consider the following snippet of client code:

In line 4, the read() system call results in the following sequence of events, corresponding to the diagram in Figure 2:

(1) Client process traps into the kernel and blocks.
(2) Kernel marshals the arguments to the read() call into a FUSD message.
(3) Kernel queues the FUSD message for the server process bound to "/dev/test", and wakes the server.
(4) Server reads and processes any previously queued FUSD messages.
(5) Server reads out the FUSD message and processes it.
(6) After processing the message, Server marshals a response and writes the response message to the kernel.
(7) Kernel passes the response back to the Client and the Client's system call returns with a result code.

From this sequence of events, it is important to note that the client blocks for the entire duration of the system call. Even if the system call is a "non-blocking"

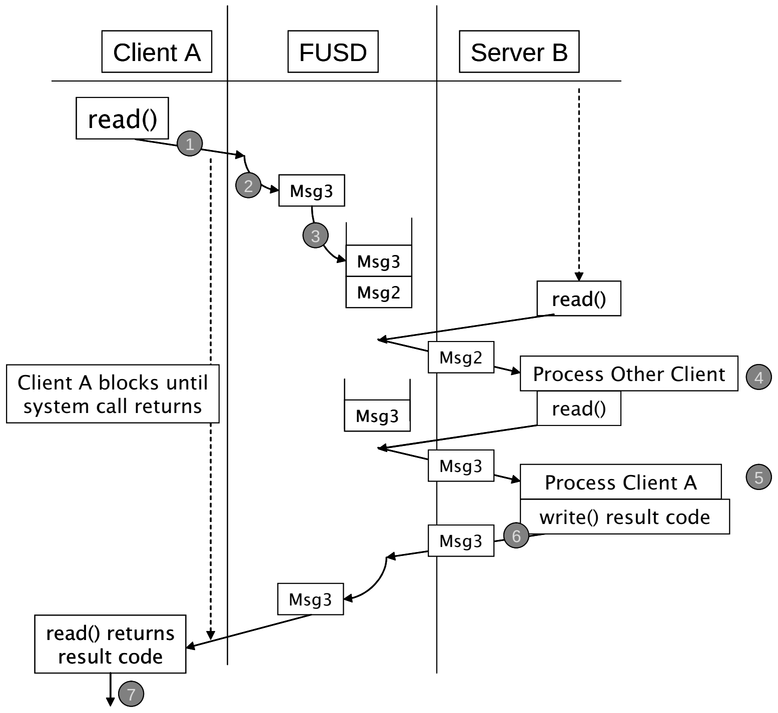


Fig. 2. Message timing diagram of a FUSD call. The middle column of the diagram represents the FUSD kernel module.

call such as a read() on a file descriptor that is configured non-blocking, the client will still be blocked until the server processes the message and returns a response, e.g. EAGAIN to indicate that the server is not ready. This means that if the server is unresponsive the client can block in a system call for arbitrary amounts of time. Slow response times could be caused either by errors in the implementation of the server, or if the server is busy handling calls from other clients. Under high system load, scheduling latency can cause a significant increase in response time. However, under normal loading conditions experiments show that inter-process latency is about 66 microseconds on a typical PC platform.

Despite this drawback of a blocking RPC model, there is also considerable benefit to this approach. First, synchronous RPC calls can be made in a straightforward coding style, as opposed to using completion callbacks. Each syscall synchronously returns a result code with a minimum of latency. Lengthy operations are typically structured as a request which is accepted or rejected quickly in one RPC call, followed by notification when the requested operation completes.

Second, syscalls are a very basic interface accessible from any POSIX application, with no library required beyond the standard UNIX interface libraries. Emstar device file interfaces are browseable within the device filesystem, and in many cases can be accessed directly by existing UNIX programs such as cat. The syscall interface is also narrow and therefore readily ported to operating systems other than Linux.

Because FUSD is central to the Emstar system, understanding its performance is important; it is investigated further in Section 3.2.4. However, it is also important to consider that Emstar systems are typically designed in a way that limits the importance of FUSD performance. Unlike module interfaces based on function call APIs (*e.g.*, TinyOS or RPC), Emstar uses FUSD to support control traffic between loosely coupled modules, with application-specific caching on the client side. For example, the Emstar neighbor service *publishes* its list to its clients once, rather than servicing numerous requests to access table elements. Since most Emstar services interface to some physical entity that changes state relatively slowly, caching at the client works quite well.

3.1.2 *Client-Server Connections in FUSD.* Using FUSD, client and server are distinct roles. A process becomes a server by registering a new device file and handling operations on that device file. A process becomes a client by opening a FUSD device file. By successfully opening a device file, a connection is established between client and server, which is named on the client side by the file descriptor returned by open(). The client may invoke RPC calls to the server at any time by making a system call on that file descriptor. The client may also listen for asynchronous notification on that file descriptor using select() or poll(). Thus, to communicate from server to client, the server first notifies the client and the client then calls back to the server. For example, the server might indicate "readable" and the client responds by calling read().

This asymmetric relationship has advantages. The primary advantage is fault isolation: clients can be written to lower standards than servers without compromising the integrity of the system. That is, while an incorrectly implemented server can permanently block a client, an implementation error in a client can't cause a server to fail. This is a consequence of the blocking semantics: while a server can potentially cause a client to block, a client can at worst only pass malformed arguments in a system call which are either caught by the kernel or should be rejected by the server process.

The FUSD client-server connection provides certain guarantees ensured by the kernel that enable fault isolation. The kernel ensures memory fault isolation between the client and server processes. Any pointers provided as arguments to a system call are checked for validity in the kernel and transferred to the memory space of the destination process, thus protecting the client and server from each other. In addition, in the event that a client or server terminates unexpectedly, any open connections are automatically cleaned up. When a client terminates with open connections, close() messages are generated and sent to the servers handling those connections. When a server terminates with active clients, those clients' file descriptors are immediately notified with exception signals and any future system calls on those descriptors will return EBADF error codes.

3.1.3 *FUSD Dependency Graphs.* An Emstar system typically involves dozens of components, each of which hosts multiple servers and is client to several other components. For example, the acoustic localization system described in this document, along with all of its Emstar subcomponents, is composed of 23 components and 182 device file interfaces. While processes are often both servers and clients of

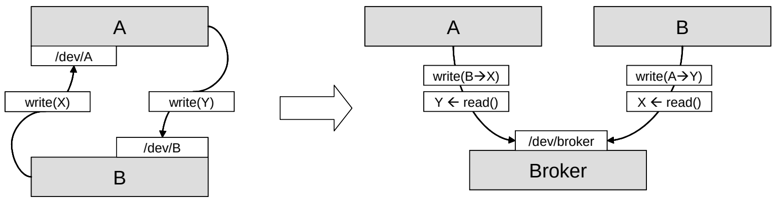


Fig. 3. A dependency loop, and using a broker service to break the loop.

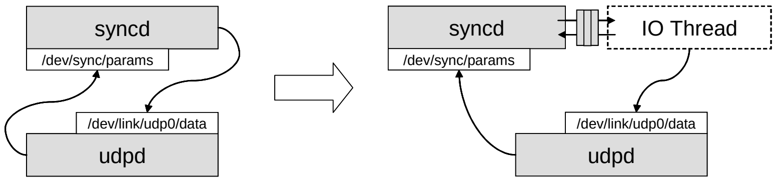


Fig. 4. Diagram showing how to use a thread and a queue to break a FUSD dependency loop.

other processes, there is a requirement that the dependency graph of clients and servers be loop-free. This stems from the blocking nature of the system calls. A loop in the dependency graph introduces the potential for deadlock, as shown in the left side of Figure 3. In the diagram, process A is blocked in write() system call as a client of process B, but before handling that call process B attempts to make a write() call back to process A. When a process is blocked in a system call as a client, it cannot respond as a server.

Figure 3 also shows a way to resolve this type of circular dependency. The most common solution is to restructure the system to add a new service that can act as an application-level "broker". Building systems as a collection of services tends to lend itself naturally to this type of structure, because most services naturally act as a broker in the course of their operations.

Note that unlike a generic message-passing mechanism, an application-level broker still serves as a service endpoint from the perspective of a client. Errors in the implementation of a service are handled by avoiding client/server protocols based on hard state.

In some cases, there is no natural decomposition into strictly layered services. When a strict layering is inconvenient, a thread and a queue can be used to break a loop, as shown in Figure 4. Essentially, this decouples the client and server portions of a process, with a queue between them.

3.2 Layer 1: Emstar Device Patterns and Libraries

Layer 1 of the Emstar design is a layer of libraries that comprise the heart of Emstar, supporting the implementations of all of the tools, services, and applications built within the framework. Using FUSD, it is possible to implement character devices with almost arbitrary semantics. FUSD itself does not enforce any restrictions on the semantics of system calls, other than those needed to maintain fault isolation between the client, server, and kernel. While this absence of restriction makes FUSD a very powerful tool, we have found that in practice the interface needs

Pattern Name	Description
Status Device	Presents current status on demand, and notification of status change.
Packet Device	Send and receive small packets on a best-effort basis, with per-client queuing.
Command Device	Presents usage information when read, accepts command strings via write().
Query Device	Synchronous RPC with a single round-robin queue for transactions.
Sensor Device	Streaming or buffered interface to a buffered sequence of samples measured from a sensor.
Log Device	Buffer of recent log messages
Option Device	/proc-style runtime-configurable option
Directory Device	Internally stores a mapping from strings to small integers, and allows clients to access and add to the mapping.

Table I. Device Patterns currently defined by the Emstar system.

of most applications fall into well-defined classes, which we term *Device Patterns*. Device Patterns factor out the device semantics common to a class of interfaces, while leaving the rest to be customized in the implementation of the service. Table I shows a list of Emstar Device Patterns.

The Emstar device patterns are implemented by libraries that hook into the GLib event framework. The libraries encapsulate the detailed interface to FUSD, leaving the service to provide the configuration parameters and callback functions that tailor the semantics of the device to fit the application. For example, Figure 5 shows a diagram of a Status Device server. In this diagram, the dotted box represents the portion implemented by the device pattern library. Because the library encapsulates all of the FUSD-related details, a minimal server implementation is less than 10 lines of C.

Emstar client implementations are trivial, because services are interfaced directly through the POSIX API. Many Emstar services provide human-readable device interfaces that can be accessed using simple shell commands such as cat and echo. For access from C programs, Emstar provides a thin client library that integrates with the GLib event system and supports added features such as *crashproofing*, that limits the impact of service failures by automatically re-connecting to a failed service.

The following sections will describe a few of the Device Patterns and client libraries defined within Emstar. Most of these patterns were discovered during the development of services that needed them and later factored out into libraries. In some cases, several similar instances were discovered, and the various features amalgamated into a single pattern.

3.2.1 *Status Device.* The Status Device pattern provides a device that reports the current state of a service or application. The exact semantics of "state" and its representation in both human-readable and binary forms are determined by the service. Status Devices are used for many purposes, from the output of a neighbor discovery service to the current configuration and packet transfer statistics for a radio link. Because they are so easy to add, Status Devices are often the most

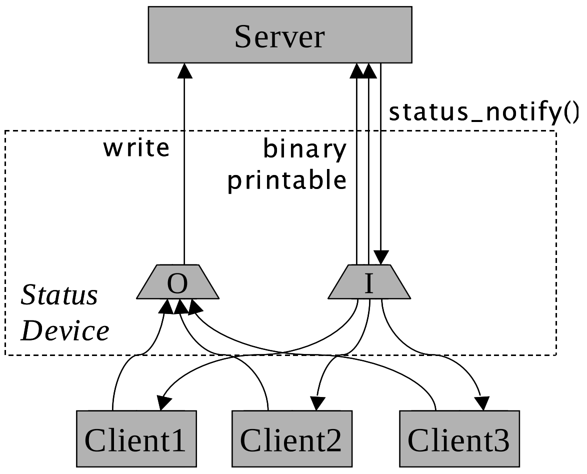


Fig. 5. Block diagram of the Status Device pattern. The functions binary(), printable(), and write() are callbacks defined by the server, while status_notify() is called by the server to notify the client of a state change.

convenient way to instrument a program for debugging purposes, such as the output of the Neighbors service and the packet reception statistics for links.

Status Devices support both human-readable and binary representations through two independent callbacks implemented by the service. Since the devices default to ASCII mode on open(), programs such as cat will read a human-readable representation. Alternatively, a client can put the device into binary mode using a special ioctl() call, after which the device will produce output formatted in service-specific structs. For programmatic use, binary mode is preferable for convenience and compactness. Binary mode responses aim to provide standard, well-defined interfaces for services that provide similar functions.

Status Devices support traditional read-until-EOF semantics. That is, a status report can be any size, and its end is indicated by a zero-length read. But, in a slight deviation from traditional POSIX semantics, a client can keep a Status Device open after EOF and use poll() to receive notification when the status changes. When the service triggers notification, each client will see its device become readable and may then read a new status report.

This process highlights a key property of the status device: while every new report is guaranteed to be the current state, a client is not guaranteed to see every intermediate state transition. The corollary to this is that if no clients care about a particular state, no work is done to compute it. Applications that desire queue semantics should use the Packet Device pattern (described in Section 3.2.2).

The possibility of dropping intermediate states requires that services implement a "soft-state" style of interface, in which the complete current state is always reported in each message. This style has the advantage of simplifying the implementation and eliminating a wide array of potential bugs in which implementation errors cause the client to become out of sync with the server. While this approach becomes inefficient as the update rate or the message size increases, in practice our services do not exhibit high loads in either event rate or the size of the data required to represent each event.

Like many Emstar device patterns, the Status Device supports multiple concurrent clients. Intended to support one-to-many status reporting, this feature has the interesting side effect of increasing system transparency. A new client that opens the device for debugging or monitoring purposes will observe the same sequence of state changes as any other client, effectively snooping on the "traffic" from that service to its clients. The ability to do this interactively is a powerful development and troubleshooting tool.

A Status Device can implement an optional write() handler, which can be used to configure client-specific state such as options or filters. For example, a routing protocol that maintained multiple routing trees might expose its routing tables as a client-configurable status device. In such a design, the client might request updates for a particular sink tree by opening the device and configuring it by writing the node ID of the sink into the open device file.

3.2.2 *Packet Device.* The Packet Device pattern provides a read/write device that provides a queued multi-client packet interface. This pattern is generally intended for packet data, such as the interface to a radio, a fragmentation service, or a routing service, but it is also convenient for many other interfaces where queue semantics are desired.

Reads and writes to a Packet Device must transfer a complete packet in each system call. If read() is not supplied with a large enough buffer to contain the packet, the packet will be truncated. A Packet Device may be used in either a blocking or poll()-driven mode. In poll(), readable means there is at least one packet in its input queue, and writable means that a previously filled queue has dropped below half full.

Packet Device supports per-client I/O queues with client-configurable lengths. When at least one client's output queue contains data, the Packet Device processes the client queues serially in round-robin order, and presents the server with one packet at a time. This supports the common case of servers that are controlling access to a rate-limited serial channel.

To deliver a packet to clients, the server must call into the Packet Device library. Packets can be delivered to individual clients, but the common case is to deliver the packet to all clients, subject to a client-specified filter. This method enhances the transparency of the system by enabling a "promiscuous" client to see all traffic passing through the device.

3.2.3 *Client Libraries and Crashproofing.* Crashproofing is intended to prevent the failure of a lower-level service from causing exceptions in clients that would lead them to abort. It achieves this by encapsulating the mechanism required to open and configure the device, and automatically triggering that mechanism to re-open the device whenever it closes unexpectedly.

The algorithm used in crashproofing is described in Figure 6. The arguments to this algorithm are the name of the device, and two callback functions, config and handler. The config function configures a freshly opened device file according to the needs of the client, e.g. setting queue lengths and filter parameters. The handler

```
WATCH-CRASHPROOF($devname$, CONFIG, HANDLER)
 1  $fd \leftarrow$ OPEN($devname$)
 2  if CONFIGURE($fd$) $< 0$ goto 11
 3  $crashed \leftarrow$ FALSE
 4  $resultset \leftarrow$ POLL($fd, \{input, except\}$)
 5  if $crashed$
 6      then $status \leftarrow$ READ($fd, buffer$)
 7            if $status < 0$ abort
 8            if $devname \in buffer$ goto 1
 9      else
10            if $except \in resultset$
11              then CLOSE($fd$)
12                    $fd \leftarrow$ OPEN("/dev/fusd/status")
13                    if $fd < 0$ abort
14                    $crashed \leftarrow$ TRUE
15            elseif $input \in resultset$
16              then $status \leftarrow$ READ($fd, buffer$)
17                    if fatal error goto 11
18                    if $status \geq 0$ HANDLER($buffer, status$)
19  goto 4
```

Fig. 6. "Crashproof" auto-reopen algorithm.

function is called when new data arrives. Note that in the implementation, the call to poll() occurs in the GLib event system, but the fundamental algorithm is the same.

A client's use of crashproof devices is completely transparent. The client constructs a structure specifying the device name, a handler callback, and the client configuration, including desired queue lengths, filters, etc. Then, the client calls a constructor function that opens and configures the device and starts watching it according to the algorithm in Figure 6. In the event of a crash and reopen, the information originally provided by the client will be used to reconfigure the new descriptor. Crashproof client libraries are supplied for both Packet and Status devices.

3.2.4 *Impact of FUSD and Emstar Devices on Latency.* While Emstar process isolation has many advantages, passing messages though successive user-space services increases latency. To quantify the latency costs of FUSD and of some representative Emstar services, we sent a series of loopback UDP "ping" messages and measured the latency costs incurred by layering additional modules over an Emstar link device.

Figure 7 shows the results of our experiment, running on a 700 MHz Pentium III. The udp-raw curves represent a non-Emstar benchmark, in which we used a UDP socket directly. The udp-dev curves represent a minimal Emstar configuration, in which we used the Emstar UDP Link device. For a two-layer stack, we added the Emstar Link Quality Estimation service, represented by the +linkstats curves. For a three-layer stack, we added the Fragmentation service over Link Quality, shown by the +frag curves.

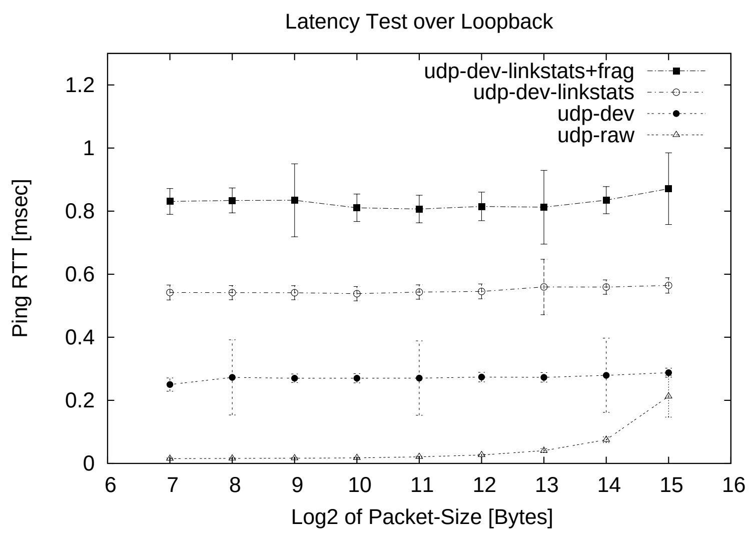


Fig. 7. Measurements of packet latency through a stack of Emstar link devices on a 700 MHz Pentium III running the 2.4.20 kernel. Each curve shows the average round-trip delay of a ping message over the loopback interface, as a function of packet length, through different Emstar stacks. Both graphs show that performance is dominated by per-packet overhead rather than data transfer.

In this experiment we measured the round-trip times for 1000 packets and averaged them to estimate the latency for our four configurations. Since the latency over loopback is negligible (shown in the "udp-raw" curve), all of the measured latency represents Emstar overhead. In each case, a ping round trip traverses the stack four times, thus is approximately 4x the latency of a single traversal. Thus, the data show that crossing an Emstar interface costs about 66 microseconds on this architecture, without a strong dependence on the length of the message being passed. This means that timing jitter at the Emstar application level is large relative to interrupt latency. Emstar assumes that all timing-critical operations and all time-stamping is performed within the kernel (as is the case for any user-level Linux software).

3.3 Layer 2: Emstar Components, Services, and Tools

Layer 2 in the Emstar design is a collection of reusable components, services, and tools that address common needs in embedded networked systems. This spans a wide range of functionality including device drivers, routing algorithms, time synchronization services, and distributed collaboration services. Table II lists the most commonly used Emstar services.

In this section we first detail three of the most fundamental services used in Emstar systems. First we describe Emstar's collection of network components, which enable Emstar systems to bridge between 802.11 and Mote networks, and enable the development of new network protocols that emphasize local communication over end-to-end connections. Second, we discuss the EmTOS library, a TinyOS [Hill et al. 2000] platform that enables TinyOS code to be shared between Motes and

Service Name	Description
UDPd	Provides a packet interface to IP networks.
TOSNic	Provides a packet interface to Mica2, MicaZ and Telos mote networks.
Fragmentation	Fragmentation layer for sending large packets via small-MTU links.
Neighbord	Sends periodic beacons to discover neighbors.
Linkstats	Link quality estimator based on measuring packet loss.
RNPLite	Low-overhead link quality estimator based on measuring packet loss.
Blacklist	Drops packets based on a specified policy about link quality.
Flooding	Flood routing.
DSR	Distributed Source Routing.
Sink	Generic routing module, routes packets based on tables provided by an external service, such as StateSync.
StateSync	Reliable multicast state dissemination protocol, with two flavors: flooding and multicast.
Centroute	Microserver-based system for managing Mote networks. Incorporates lightweight Mote registration protocol and unicast routing to and from Motes.
TimeSync	Time conversion service that enables conversions between CPU clock time, other system clocks (e.g. GPS or audio codec), and the CPU clocks of neighboring nodes.
GSync	Global time broadcast service, makes a single time reference available on all nodes in the network.
Audio Server	Continuous multichannel recording service with 8 seconds of internal buffering and integration to the TimeSync system.
EmRun	Manages startup, shutdown, and logging of processes in an Emstar system.
EmProxy	Daemon for monitoring and controlling groups of nodes via UDP broadcast.
Directory Service	Creates a persistent local mapping of string values to integers.
IP Connector	Tunnels IP packets via a link interface.

Table II. Partial list of services currently implemented within the Emstar system.

Emstar systems. Third, we discuss the EmRun service, a daemon that starts up, shuts down, and monitors a running Emstar system, providing centralized logging and fault reporting.

Finally, we describe a collection of tools that help developers design, implement, and deploy new systems, including tools for simulation, deployment, remote access, and visualization.

3.3.1 *Network Stack Components.* Emstar includes a suite of components that are used and combined to provide network functionality tuned to the needs of wireless embedded systems. These components include "link drivers", that implement the lowest layer interfaces to network resources, pass-through modules that implement various types of filter and passive processing, and routing modules that provide facilities that provide network-layer interfaces that route messages among one or more link-layer interfaces. Each network component provides a standard *Link Device* interface, composed of a packet device for data traffic, and command

and status devices for configuration and status. The Link Device interface provides a uniform interface to a basic set of common features such as neighbor list, data transfer, and channel selection, but is extensible to support hardware-specific features in the interim before the uniform abstraction can be extended. This represents a different approach from the recently proposed SP abstraction [Polastre et al. 2005], in which all hardware-specific details are hidden behind a rigid abstract interface.

Emstar implements several "link drivers", providing interfaces to radio link hardware including 802.11, and several flavors of the Mica Mote. The 802.11 driver overlays the socket interface, sending and receiving packets through the Linux network stack, and optionally integrating feedback from the MAC layer about RSSI, precise timing, and transmission failures. Two versions of the Mote driver exist, one that supports both Berkeley MAC [Hill et al. 2000] and SMAC [Ye et al. 2002] on Mica2, and a new version that supports only BMAC but adds support for newer platforms such as Telos. While the Link Device provides an abstraction for many common radio features, it does not require that this abstraction be complete or opaque. Low level information about the link's capabilities and current status, *e.g.*, transmit power, MTU, and channel, is exposed so that applications can make intelligent decisions about how and when to use them.

Link interfaces are also used to construct modules that sit in the middle of the stack, passing packets through to lower layers, possibly analyzing or modifying them along the way. A pass-through module is both a client of a lower Link device and a provider of an upper Link device. To simplify the implementation, some of the work of proxying status and command interfaces is done by a library. In some cases, the implementation of a pass-through involves implementing a single function that transforms a packet from above and sends it below, and vice versa.

Linkstats, Blacklisting, and Fragmentation are examples of pass-through modules. Linkstats adds a small header to each packet and counts gaps in sequence numbers to estimate link quality. Blacklisting uses the output of a neighbor discovery module to block traffic on links that are not bidirectional. Fragmentation breaks large packets up into smaller packets.

Link interfaces are also used to provide interfaces to routing modules. These are *network* layer interfaces rather than *link* layer interfaces, so the source and destination addresses are usually interpreted as network layer IDs i.e. Node IDs) rather than link layer IDs (i.e. Interface IDs). The simplest routing module is the flooding module, which accepts messages, adds a sequence number, and re-sends each message exactly once. There is also an implementation of DSR and a generic routing module called sink that uses routing tables provided by another module to route messages to a specified destination.

3.3.2 *Supporting Heterogeneity with EmTOS.* EmTOS [Girod et al. 2004b] is a TinyOS [Hill et al. 2000] platform target that enables TinyOS applications to run as a single module in an Emstar system. EmTOS implements a shim layer that translates between the TinyOS system API and the Emstar API, enabling a TinyOS application to run on a Microserver. EmTOS also enables a TinyOS application to provide services to other Emstar applications by hosting Emstar device interfaces. Because EmTOS emulates the TinyOS API, applications that require direct access to Mote hardware or that rely on precise timing will not work properly in EmTOS.

The key advantage of EmTOS is that it allows TinyOS/NesC code to be shared between TinyOS Motes and Linux Microservers. Consider the case of a Mote network in which a complex protocol is implemented. Without EmTOS, there are two ways to interface that network to a Microserver: either by re-implementing the protocol on the Microserver, or by loading a specialized image on the Mote directly connected to the Microserver. In the latter case, the Mote must implement the protocol and translate it into TOS_Msg structures that can be interpreted on the Microserver side (e.g. in Java via MIG). However, EmTOS provides a third alternative: to run the NesC protocol code directly on the Microserver, in a special TinyOS application that uses any convenience afforded to Linux applications.

To better support Emstar integration, EmTOS provides NesC interfaces that support the native Emstar IPC mechanisms, including Status Device, Packet Device, and Link Device. EmTOS also enables the Emstar simulation environment to simulate heterogeneous networks composed of both Motes and Microservers [Girod et al. 2004b]. Using EmTOS as a building block, we have developed more sophisticated Mote-Microserver collaboration tools, including the concept of Mote Herding [Stathopoulos et al. 2005].

3.3.3 *EmRun Services.* EmRun is a program that parses a configuration file that describes an Emstar system, launches the system, and provides several centralized services. Among the services it provides are automatic re-spawning with visibility into the status and history of processes, centralized in-memory logging, process responsiveness tracking, and fault reporting.

3.3.3.1 *Respawn.* Process respawn is neither new, nor difficult to achieve, but it is very important to an Emstar system. It is difficult to track down every bug, especially ones that occur very infrequently, such as a floating-point error processing an unusual set of data. Nonetheless, in a deployment, even infrequent crashes are still a problem. Often, process respawn is sufficient to work around the problem; eventually, the system will recover. Emstar's process respawn is unique because it happens in the context of "crashproofed" interfaces (Section 3.2.3). When an Emstar process crashes and restarts, Crashproofing prevents a ripple effect, and the system operates correctly when the process is respawned.

When processes die unexpectedly, EmRun tracks the termination signal and last log message reported by the process. This information can be accessed from the last_msg device, that reports the count and circumstances of all process terminations along with their final message.

3.3.3.2 *In-Memory Logs.* EmRun saves each process' output to in-memory log rings that are available interactively from the /dev/emlog/* hierarchy. This design decision stemmed from experience with hardware platforms in which writing logs to flash posed serious practical problems. While in principle saving persistent logs to a large flash device should not be a problem, in practice many platforms do not include a suitable device. In our early experiences with embedded Linux platforms we found that frequent writes to flash often caused system performance problems. When multiple applications needed access to flash and erase operations were occurring, flash accesses would sometimes block for long periods of time.

Defaulting to in-memory logs addressed both of these problems at once. In the event of a critical failure or user command, an external component saves a snapshot of the current logs to persistent storage on flash. As logging capabilities improve, this design choice may need to be revisited.

3.3.3.3 *Process Responsiveness Tracking.* An unresponsive server can cause performance bottlenecks for an entire Emstar system. To address this, EmRun tracks the responsiveness of all processes in the system. The EmRun client library includes a timer event that sends a periodic heartbeat message to EmRun. EmRun tracks the arrival of these messages and compares the arrival time to the scheduled send time according to the timer. The discrepancy in the times reveals an estimate of the responsiveness of the process, since whenever the timer fires, that process is also free to respond to I/O events.

3.3.3.4 *Fault Reporting.* Fielded systems often have the possibility of unexpected faults. For example, our acoustic systems sometimes experience wiring problems that cause one or more channels of the microphones or the speaker to fail. It is also possible for our acoustic systems to be incorrectly set up, for example if the battery pack for the microphone array is disconnected or if the array's wires are crossed. A driver that detects a fault can publish this fault through a centralized reporting service provided by EmRun. This fault report is only available locally through the faults device file, but other modules can publish that fault information over the network to other nodes and to the user.

The fault reporting API is very simple. A process reports a fault by opening a device file, writing in a string describing the fault, and keeping that file open until the fault is corrected. When the file is closed (whether by the application or by the process terminating), the fault will be removed from the list. Other processes may monitor the fault list using the standard Status Client library.

3.3.4 *EmSim: the Emstar Simulator.* Transparent simulation at varying levels of accuracy is crucial for building and deploying large systems [Elson et al. 2003] [Levis et al. 2003] [Girod et al. 2004b]. EmSim enables "real-code" simulation at many different accuracy regimes. EmSim runs many virtual nodes in parallel, each with its own device hierarchy. Because Emstar applications always interact with the world through standard interfaces such as Link devices and Sensor Devices, EmSim can transparently run nodes in simulation by presenting identical interfaces to simulated or remote resources.

For operations in pure simulation, a radio channel simulator and a sensor simulator can provide interfaces to a modeled world, or re-play conditions that have been observed empirically. EmSim also supports "emulation mode", in which real hardware such as radios and sensors can be accessed remotely. This yields a very convenient interface to a testbed, because the entire application can run centrally on a single simulation server, while the radio traffic or sensor data comes from a deployed testbed. We have found that using real radios is far superior to attempting to model radios, especially when there may be bugs or glitches in the operation or performance of the radios.

These different simulation regimes speed development and debugging; pure simulation helps to get the code logically correct, while emulation in the field helps to understand environmental dynamics before a real deployment. However, because the Emstar simulator does not provide tight timing accuracy and glosses over hardware details, timing- and hardware-related bugs that appear in deployment may be difficult to reproduce in simulation. Simulation and emulation do not eliminate the need to debug a deployed system, but they do tend to reduce it.

In all of these regimes, the Emstar source code and configuration files are identical to those in a deployed system, making it painless to transition among them during development and debugging. This serves to eliminate accidental code differences that can arise when running in simulation requires modifications. EmSim can also simulate heterogeneous networks that contain both Motes and Emstar systems, by running the Mote code inside an *EmTOS* wrapper [Girod et al. 2004b]. Other "real-code" simulation environments include TOSSim [Levis et al. 2003] and SimOS [Rosenblum et al. 1997], but Emstar is the only environment that readily supports heterogeneous networks and "emulation" using real hardware.

3.3.5 *Remote Access Methods.*

As an IPC mechanism, FUSD has the benefit of being fast, deterministic, and synchronous, making straight-line programming of sequential calls possible. These properties make it easy to communicate between processes on a single node, but unlike sockets, they do not provide any native remote access mechanism. In addition, some languages such as Java are designed to handle everything in terms of sockets, and don't have complete support for POSIX system calls.

To address these concerns, we have implemented several remote access mechanisms to Emstar. These mechanisms enable access to Emstar services over the network, and can also simplify the integration of Emstar with other systems. The three remote access mechanisms supported by Emstar are FUSDnet, the Emstar HTTP server, and EmProxy.

3.3.5.1 *FUSDnet.*

FUSDnet is a remote access protocol based on FUSD. Using FUSDnet, any FUSD device can be accessed remotely via a sockets protocol.

A server that wants to enable incoming remote connections must set a special flag when it registers the device. This flag will register the device with the FUSDnet daemon, which listens for incoming requests and demultiplexes them to the appropriate server. A client that wants to connect to a remove FUSD service must run a client program that opens a socket to the remote node, requests a connection to the specified device, and creates a local stub device. Once the connection is established, the local stub device will be an exact mirror of the remote device. A system call made on the stub will be marshaled and transferred via the socket to the remote server, where it will be handled and a response returned. Thus, FUSDnet provides transparent access to remote FUSD services.

FUSDnet is transparent, but it is recommended only for use in conditions with reliable and deterministic network links between client and server. FUSDnet might be a convenient way to link two Emstar systems together if they have a wired Ethernet link between them, but it would not be so appropriate if those two systems are physically separate and are connected wirelessly. For such situations, protocols designed for slow or unreliable links would be preferred.

3.3.5.2 *HTTP.* With the advent of the Web, HTTP has become one of the most universally implemented protocols. Recognizing this, Emstar supports an HTTP gateway that enables remote access to FUSD devices. This access is implemented by CGI scripts that enable access to Status Devices, Command Devices, and Log Devices via simple URL formats. This approach can easily be extended by adding additional CGI scripts to handle other device types.

The Emstar HTTP server integrates with EmRun to provide a default web page that shows the node's current status and can integrate sub-pages for each running process. This "Node Page" can make it easy for novice users to browse the status of a node using a web client. The HTTP service also enables integration with Java and other software that can readily access services via HTTP, and allows those programs to run remotely.

3.3.5.3 *EmProxy.* EmProxy is a remote access protocol that exposes real-time state changes via a best-effort UDP protocol. Unlike FUSDnet and HTTP, which use TCP to reliably connect to a specific node, EmProxy provides a broadcast interface to control and monitor groups of nodes over a broadcast network. Because EmProxy can operate over broadcast to groups of nodes, it is very useful in deployed environments where many nodes are involved but only a subset of those nodes are reachable at any given time. The use of UDP enables EmProxy to report status at high rates, dropping messages rather than buffering them when the rate exceeds capacity.

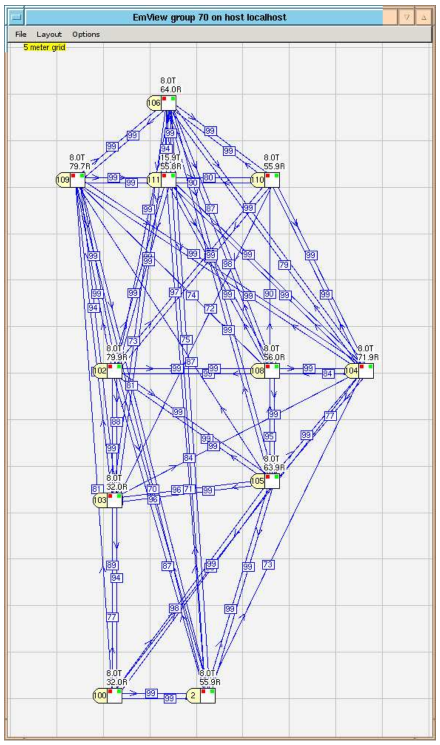
An EmProxy client connects to EmProxy by periodically sending a request message that lists a set of Status Devices to monitor. The EmProxy service opens those Status Devices and monitors them for notification. Whenever notification is triggered, EmProxy reads the new state and reports that back to the requester via UDP. The request string can include options and arguments to limit the rate at which replies are reported, to automatically re-read the device periodically, and to set the mode in which the device is read (e.g. binary, ASCII, XML, etc.)

EmProxy also supports the ability to run shell commands and report back the results. This broadcast remote shell is very useful for managing and controlling groups of nodes in a deployed setting.

3.3.6 *Deployment Tools.* Emstar includes a number of tools and facilities designed to aid deployments. When working on a deployment the two primary issues to address are finding out the state of the nodes and controlling the nodes. Emstar provides several mechanisms that address these issues: rbsh (Remote Broadcast SHell), IP routing, and efficient state flooding.

3.3.6.1 *rbsh.* The rbsh program is an invaluable tool for dealing with collections of nodes. It provides a simple shell prompt interface, but when commands are written at the prompt, they are broadcast out over a selected network, and the command script is run on each node. The results of the command are then reported back and collated at the prompt.

In a deployed setting, this provides a fast and convenient way to send commands to all reachable nodes without needing to know which nodes exist or which are reachable. In addition, relative to tools based on ssh and other connection-oriented protocols, there is no need to maintain connections to remote nodes, nor to time



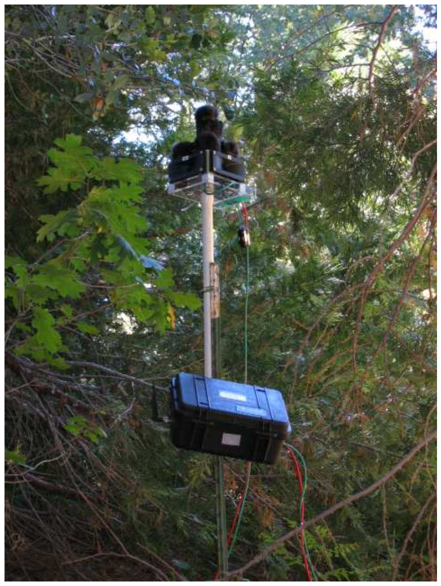


Fig. 8. (a) Screen shot of EmView, the Emstar visualizer, taken from our James Reserve deployment. (b) Photo of one of the Acoustic ENSBox nodes, deployed at the James Reserve.

out broken connections. The result is a simple, fast, and generally intuitive shell interface.

We have also had success using rbsh in scripts to control groups of nodes while running experiments and to implement simple forms of coordination without writing specialized application code.

3.3.6.2 *IP Routing.* In a deployed setting, being able to telnet across the network is very useful. For this reason, even if the application itself does not require end-to-end IP routing, IP routing can be very useful for debugging a deployment. Often full pairwise routing is not needed, and routing along a tree is sufficient. Emstar provides IP routing by combining native Emstar routing facilities with the *IP Connector.* The IP connector provides a regular IP network interface that applications such as telnet and ping can use, but tunnels the traffic on that interface through a Emstar Link device.

3.3.6.3 *State Flooding.* One of the most challenging parts of a deployment is determining what is happening in the network. For example, it is important to know the link quality observed between different nodes in the system so that gaps in connectivity can be corrected. It is also important for the user to be aware of faults that may have occurred in the deployment.

To address these needs, Emstar includes StateSync, an efficient reliable state dissemination mechanism [Girod et al. 2005]. The StateSync protocol defines a local retransmission protocol, and is provided in two versions: the flooding version and the tree-based multicast version. The flooding version is ideal for debugging and fault reporting, because it does not rely on any form of routing; it exchanges messages peer to peer at a low rate to flood the current state of a set of variables to its neighbors with a maximum hopcount to limit propagation. Reliability is achieved by triggering local retransmission when a gap is detected in the sequence of messages. This mechanism is described in more detail in [Girod et al. 2005].

In our acoustic deployment, we use this mechanism to flood reported faults and neighbor link quality. This enables us to use a laptop to observe the network from anywhere in the field, quickly getting a picture of the link quality in the network, and immediately seeing any reported faults.

3.3.7 *Visualizing Emstar Systems.* EmView is a graphical visualizer for Emstar systems. Through an extensible design, developers can easily add "plugins" for new applications and services. Figure 8(a) shows a screen-shot of EmView displaying real-time state of a running deployment at the James Reserve. In this instance, the data was being displayed live from our state flooding protocol.

EmView uses the EmProxy protocol to acquire status information from a collection of nodes. Although the protocol is only best-effort, the responses are delivered with low latency, such that EmViewcaptures real-time system dynamics. In order to support heterogeneous networks, EmView first requests a configuration file from each node that details how to visualize the services on that node. Based on that file, EmView then follows up with a request for node status as needed. This design enables EmView to visualize any Emstar system without needing to be informed up front about the details of what software or services are present in each system.

4. EVALUATION

To evaluate the utility of Emstar we present a case study in which Emstar was used to build a sophisticated Microserver application. The Acoustic ENSBox [Girod et al. 2006; Girod 2005], shown deployed in the James Reserve in Figure 8(b), is a platform for distributed acoustic sensing. It represents a complete vertical implementation of a sensor-actuator network, from hardware through the sophisticated application software shown in the architectural diagram in Figure 9. We have used this system to detect and localize animal calls in several field experiments [Ali et al. 2007; Trifa et al. 2007].

In a typical deployment, a cluster of ENSBoxes is positioned on tripods in an area of interest. Each node runs an Emstar stack and hosts a 4-channel microphone array and 4 piezo tweeters, capable of sampling and processing all four channels at 48 KHz. Nodes in the system communicate via an 802.11 interface with an external antenna; an on-board mote radio provides an interface to networks of motes.

In this section, we evaluate the effectiveness of Emstar as an application framework by describing how it enabled the implementation of two applications: a sophisticated acoustic self-calibration service, and a simple marmot detection application.

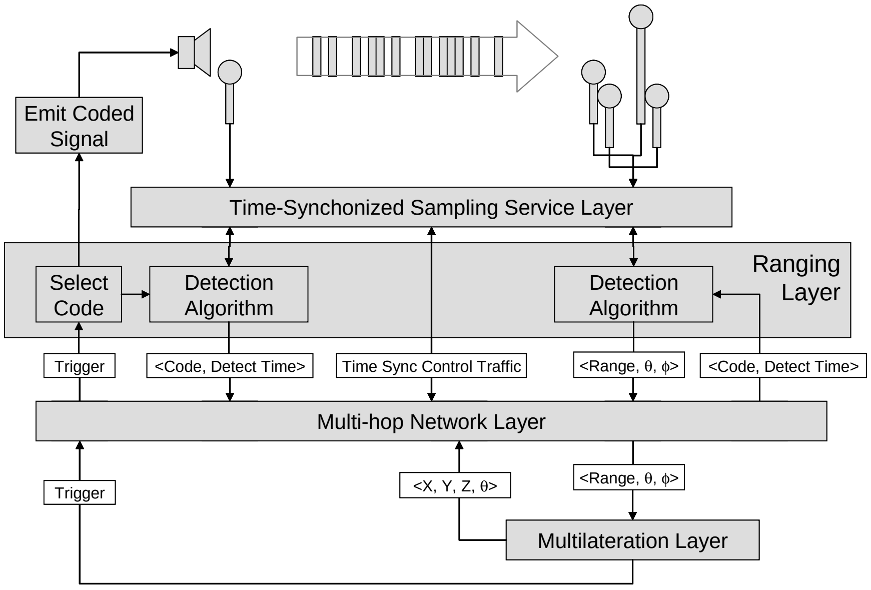


Fig. 9. Simplified architectural diagram of the Acoustic ENSBox automated position estimation system. The "Time Synchronized Sampling Layer" is composed of the TimeSync service and the Audio service. The "Multi-hop Network Layer" is composed of the UDPd packet service, RNPLite link estimator, the Flood routing service, and the StateSync reliable state dissemination service. The "Multilateration Layer" and the "Ranging Layer" are two separate processes that together make up the position estimation service.

4.1 Acoustic Self-Calibration

The ENSBox platform includes an integrated self-calibration mechanism that computes highly accurate position and orientation estimates for each node. When the calibration system is activated, it emits acoustic signals into the environment and detects them at other points to autonomously estimate a high-accuracy 3D position map for all nodes in the system. This map may then be used by higher-level applications such as bearing-based localization applications and algorithms [Wang et al. 2005; Wang et al. 2004; Chen et al. 2003; Ali et al. 2007].

4.1.1 *Leveraging Emstar's Wealth of Services.* The implementation of the Acoustic ENSBox software stack, shown in Figure 9, uses most of the components of Emstar described in this article. EmRun starts the system up, keeps processes running in the event of a failure, and supports fault reporting, local logging, process responsiveness monitoring, and other diagnostics. Faults and network connectivity information is propagated throughout the network via the state flooding facility. From a laptop at the edge of the network, EmView can visualize the deployment, faults can be observed, and rbsh can broadcast shell commands to the whole network for control or diagnosis. Figure 8(a) shows a screen shot of EmView captured during the James Reserve deployment.

The Acoustic ENSBox uses Emstar services to provide a synchronized sampling layer [Elson et al. 2002b]. The Audio service continuously samples the audio hardware and provides an interface to the last 8 seconds of sampled audio data using the sensor device pattern. The Audio service also provides time conversion data to the Time Synchronization service, enabling applications to directly convert CPU timestamps to and from points in the audio time series. To enable conversions between the clocks of neighboring nodes, the Time Synchronization service implements Reference Broadcast Synchronization (RBS) [Elson et al. 2002a]. The time synchronization API provides access to this entire graph of conversions, while the sensor client API provides access to specified ranges of audio samples from the recent past. Should a node discover an interesting feature at a particular point in its audio stream, these facilities make it easy to communicate that information to nodes one or more hops away, who in turn can retrieve and analyze the data recorded at the same time at their location.

The Acoustic ENSBox also uses StateSync [Girod et al. 2005] to share acoustic ranging and position estimation results with other nodes. The StateSync facility, also built using Emstar, provides a publish-subscribe interface over an efficient reliable multihop broadcast. Designed to make soft-state system designs more efficient, applications can publish and re-publish their current "state" via StateSync. The StateSync system will deliver the current state reliably and deliver efficient diffs to that state whenever a change is made. StateSync can be used to implement a variety of coordination mechanisms; Acoustic ENSBox uses it to coordinate ranging experiments and to publish the results of ranging and position estimation.

The re-usable Emstar services described above represent over 50K lines of C. Using these facilities substantially reduced the complexity and time required to construct the acoustic self-calibration system. While the self-calibration implementation is still substantial (15K lines), it is focused exclusively on the signal processing and estimation details required to estimate the ranges between nodes and to construct a coordinate system.

In addition, by implementing common services as separate modules we have been able to generalize them across applications and to support multiple concurrent applications. In contrast to the first iterations of this system we described in Section 1, the Emstar based implementation has enabled animal call detection and localization software to run in parallel with the self-calibration system.

4.1.2 *Emstar's Robustness.*

Using Emstar uniquely enabled us to build and deploy a robust system. Before we left the lab, we used the real-code simulation and emulation capabilities of EmSim to debug many of the underlying components. For the networking portions, we used emulation mode on a 13-node 802.11 testbed that tested our networking components on a real (albeit different) radio channel. Using emulation mode, we were able to test our networking algorithms using a real 802.11 channel, subjecting them to real channel-loss and timing characteristics. This enabled us to debug problems relating to timing, congestion, and link asymmetry.

After getting the system working in the lab, we performed several test deployments at the James Reserve, each running the system continuously for 24 hours. Once in the field, Emstar's robustness paid off in three specific ways.

First, there were a few unexpected failures of underlying software components, particularly the Time Synchronization and the Link Quality Estimation services. These failures were reported, the services automatically restarted, and clients of those services automatically reconnected. These failures did not cause more than a few minutes' disruption in the application, and so we were able to ignore those problems during our experiment. If these faults had caused a total system restart it would have been much more disruptive because many services would need to go through a time-consuming process to rediscover their state. For example, the Acoustic Ranging service would need to re-range to all of its neighbors, which takes minutes to complete. Moreover, in some cases a complete system restart would simply have re-created the failure mode, whereas restarting an individual service would enable the system to continue to move forward and collect data.

Second, because our acoustic arrays are prototypes and are not completely hardened, we experienced problems with microphone wiring and low voltages on the microphone batteries. These faults were detected by the Audio service and reported to EmRun with low latency. StateSync was then used to reliably disseminate observed faults, so that they were visible from a laptop anywhere in the field. Thus, whenever a fault arose, we would walk out to that node and debug it.

Third, some hardware failures caused software lock-ups. In one case, problems with the flash filesystem caused the application to block while attempting to write. This problem was detected and reported by EmRun as an unresponsive process. While in this instance there was no way to fix the problem in the field, Emstar enabled the problem to be diagnosed on-line and the node was removed from the experiment. Without a framework that was monitoring distributed software and hardware components, it would have been more difficult to track down the source of the problem.

Arguably, if these debugging facilities and robustness features did not already exist, system developers would need to construct them in order to test and deploy their system in difficult environments.

4.2 A Simple Detection Application

Our second use case is the implementation of a simple animal-call detection algorithm described in [Ali et al. 2007]. This detection algorithm was written in about a week, and comprised about 800 lines of C. The application monitors a single audio channel, filtering the signal and detecting an onset of energy in a particular band. When an event is thus detected, the time of the event is flooded to all nodes within N hops. A separate event recording application receives these notifications, aggregates them whenever notifications from different nodes overlap, and records the event to flash. Thus, whenever any node in the network detects an event, all nearby nodes record precisely synchronized segments of audio for later processing.

Just as in the case of the localization system, Emstar's existing services and components vastly reduce the work required to develop this application. Approximately 50% of the detector implementation was devoted to the signal processing algorithms themselves; the remainder provided control and status interfaces for run-time re-configuration of parameters. However, Emstar services addressed most of the remaining details, from time-synchronized sampling at multiple nodes and multi-hop networking to logging and centralized fault reporting.

5. RELATED WORK

Throughout the paper we have mentioned work related to individual components or aspects of Emstar. See Section 2.1.3 for a discussion of Microreboot [Candea and Fox 2003] [Candea et al. 2004] and Nooks [Swift et al. 2004]; see Section 2.2.3 for a discussion of alternative IPC mechanisms. In this section we discuss two categories of work related to Emstar: software systems and simulators.

5.1 Software Systems

The system most similar to Emstar is TinyOS [Hill et al. 2000]. TinyOS addresses the same problem space, only geared to the much smaller Mote platform. As such, much TinyOS development effort must focus on reducing memory and CPU usage. By operating with fewer constraints, Emstar can focus on more complex applications and on improving robustness in the face of growing complexity. A key attribute of TinyOS that Emstar lacks is the capacity to perform system-wide compile time optimizations. Because Emstar supports forms of dynamic binding that do not exist in TinyOS, many compile-time optimizations are ruled out.

Player/Stage [Vaughan et al. 2003] is a software system designed for robotics that supports "real-code" simulation. In many respects, Player and Emstar have a similar philosophy. Both focus considerable effort on ease of system integration, because just as in typical Emstar systems, the robots Player is designed to support must interface to many peripherals and devices that often do not support any pre-existing standardized interface. Both are designed to support a soft-state system design typical of reactive robotics systems, in which different components are loosely connected by data paths that pass along the latest state and notify on change. The main difference lies in the design of the system and the IPC mechanisms. Player is based on simple sockets protocols, which have the advantage of remote transparency, but are not as easily browsed and provide less diagnostic capability.

CARMEN[1] is another software framework designed to support development and real-code simulation in the field of robotics. Like Player, CARMEN uses sockets-based IPC with a loose publish-subscribe message-passing semantics. Unlike Player, in which clients and servers connect directly to each other, CARMEN routes all messages through a central message router that plays a role similar to FUSD in Emstar.

5.2 Simulation Environments

Several other simulation environments have been applied to sensor network systems, including TOSSIM [Levis et al. 2003], Avrora [Titzer et al. 2005] and *ns*-2[2]. TOSSIM and Avrora both support real-code simulation of Mote-based systems. The *ns*-2 simulation environment is commonly used to model the behavior of Internet protocols.

Like EmTOS, TOSSIM [Levis et al. 2003] is implemented as a TinyOS "platform", and simulates code running above a TinyOS API. TOSSIM can been used to simulate above either a very low-level API or over a higher message-level API. Simulations above the low-level API can be used to simulate behavior within the

[1] CARMEN (Carnegie Mellon Robot Navigation Toolkit), `http://carmen.sourceforge.net/doc/`.
[2] *ns*-2, the Network Simulator, `http://nsnam.isi.edu/nsnam/index.php/Main_Page`

radio stack, whereas simulations above the message-level API are more scalable. TOSSIM does not run real-time, but guarantees proper ordering of messages and proper timing of timer interrupts. However, it does not take execution time into account.

Avrora [Titzer et al. 2005] is an AVR/Mote binary emulator that provides a cycle-accurate emulation environment with energy accounting and an interface to a radio channel model. Avrora executes a complete binary Mote image just as a real Mote would, and all hardware interrupts work as in a real Mote. Unlike the Emstar simulator or TOSSIM, Avrora properly accounts for processing time and will deliver interrupts with completely accurate timing. However, Avrora is very resource intensive relative to other simulators.

The *ns*-2 simulation environment has been adapted for use in wireless settings, including the use of different MAC layers and integrated support for common IP protocols. While wireless support was not central to the design of *ns*-2, it includes good support for MAC layer details, particularly of 802.11. The most significant drawback to *ns*-2 is that it is not a *real-code* simulation environment, instead requiring considerable amounts of glue to link real code to the simulation engine.

Relative to Avrora and TOSSIM, Emstar provides a reduced-accuracy simulation environment, but provides two important additional capabilities: the ability to run using real hardware to provide a radio or sensor channel, and the ability to simulate systems composed of different types of hardware, such as different types and configurations of Motes as well as Microservers. Emstar also is relatively easy to integrate with other simulation tools; for example with Avrora to include a few cycle-accurate Motes within a larger simulation.

6. CONCLUSION AND LESSONS LEARNED

We have found Emstar to be a very useful development environment for Linux-based and heterogeneous embedded sensor-actuator networks. We use Emstar at CENS in several current development efforts, including the Acoustic ENSBox project [Girod et al. 2006] [Girod 2005], the Cyclops heterogeneous camera network, a 50-node seismic deployment [Lukac et al. 2006], and the James Reserve ESS microclimate sensing system. In support of these projects, we continue to enhance Emstar with new components and tools. We also support several other groups using Emstar, including groups at Ohio State [Arora et al. 2004] [Arora et al. 2005], MIT, and USC/ISI.

Although Emstar has been used quite successfully internally and at a few external sites, it has not been easy to export to a broader community, and since the beginning of the project the basis for some of our original assumptions have changed.

First, while the multi-process service model has positive benefits for robustness when the services are implemented appropriately, new users have had difficulty correctly developing their own services. While Emstar applications are generally single processes and are easier to implement, implementing services correctly is more challenging. We are investigating the use of a monolithic system approach using a type-safe, garbage collected language and a uniform component model. In this context, we would try to retain the system visibility and robustness features, while greatly simplifying the system from the developer's perspective. However, we have not yet come to a conclusive approach that achieves these ends, and are considering a range of different points on the spectrum from multi-process systems to monolithic.

Second, configuration information in the system tends to be set either by command line arguments or dynamically through a Command Device. While Command Device enables dynamic run-time configuration, it also has the effect of distributing configuration into many different places and it means that the restart semantics must include dynamically re-issuing configuration commands. The standard UNIX approach of reading a configuration file on startup and on SIGHUP would greatly simplify the restart semantics and would reduce the incidence of incorrect implementations. This might be implemented as a centralized configuration service, similar to the Windows registry.

Third, the Status Device has been incredibly useful as a construct, but the reliance on FUSD has introduced several difficulties, from not directly supporting remote access to requiring compilation of a kernel module as a prerequisite to using the system. We are considering implementing a new version of Emstar that bases its IPC on sockets and eliminates the kernel module dependency. We are also considering implementing a specialized "inter-positioning" interface that allows cheap and reliable logging of outputs or traffic on a particular IPC interface.

Fourth, some more recent applications have needed to handle data rates that impose noticeable overhead via message passing. This has led to plans to implement a shared memory interface for the sensor device interface. In addition, it is more efficient and more natural for high rate network services such as DTN "bundle transfer" to use direct TCP connections to neighboring nodes, even if they implement a control and discovery protocol over the link device [Lukac et al. 2006].

As we continue work on this system, we are investigating ways to address these issues, keeping the good features of Emstar while fixing it to better address the needs of the user community.

REFERENCES

Ali, A., Collier, T., Girod, L., Yao, K., Taylor, C., and Blumstein, D. T. 2007. An empirical study of acoustic source localization. In *IPSN '07: Proceedings of the sixth international conference on Information processing in sensor networks.* ACM Press, New York, NY, USA.

Arora, A. et al. 2004. A line in the sand: A wireless sensor network for target detection, classification and tracking. *Computer Networks 46,* 5 (Dec.), 605–634.

Arora, A. et al. 2005. Exscal: Elements of an extreme scale wireless sensor network. In *IEEE RTCSA*. IEEE.

Brooks, R. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation 2,* 1.

Candea, G., Cutler, J., and Fox, A. 2004. Improving availability with recursive micro-reboots: A soft-state system case study. *Performance Evaluation Journal 56,* 1-3 (March).

Candea, G. and Fox, A. 2003. Crash-only software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX).* Lihue, HI, USA.

Chen, J., Yip, L., Elson, J., Wang, H., Maniezzo, D., Hudson, R., Yao, K., and Estrin, D. 2003. Coherent Acoustic Array Processing and Localization on Wireless Sensor Networks. *Proceedings of the IEEE 91,* 8 (August).

Dabek, F., Zeldovich, N., Kaashoek, F., Mazières, D., , and Morris, R. 2002. Event-driven programming for robust software. In *Proceedings of the 2002 SIGOPS European Workshop.* Saint-Emilion, France.

Elson, J. 2002. *FUSD: Framework for User Space Devices.*

Elson, J., Bien, S., Busek, N., Bychkovskiy, V., Cerpa, A., Ganesan, D., Girod, L., Greenstein, B., Schoellhammer, T., Stathopoulos, T., and Estrin, D. 2003. EmStar: An Environment for Developing Wireless Embedded Systems Software. Tech. Rep. CENS Technical Report 0009, Center for Embedded Networked Sensing, University of California, Los Angeles. March.

Elson, J., Girod, L., and Estrin, D. 2002a. Fine-grained network time synchronization using reference broadcasts. In *OSDI*. Boston, MA, 147–163.

Elson, J., Girod, L., and Estrin, D. 2002b. A wireless time-synchronized COTS sensor platform, part i: System architecture. In *IEEE CAS Workshop on Wireless Communications and Networking*.

Girod, L. 2005. A self-calibrating system of distributed acoustic arrays. Ph.D. thesis, Univerity of Caliornia at Los Angeles.

Girod, L. et al. 2004a. Emstar: a software environment for developing and deploying wireless sensor networks. In *Proceedings of the 2004 USENIX Technical Conference*. USENIX Association, Boston, MA.

Girod, L. et al. 2004b. A system for simulation, emulation, and deployment of heterogenous sensor networks. In *Proceedings of the second international conference on Embedded networked sensor systems*. ACM Press.

Girod, L., Bychkovskiy, V., Elson, J., and Estrin, D. 2002. Locating tiny sensors in time and space: A case study. In *in Proceedings of ICCD 2002 (invited paper)*. Freiburg, Germany.

Girod, L. and Estrin, D. 2001. Robust range estimation using acoustic and multimodal sensing. In *International Conference on Intelligent Robots and Systems.*

Girod, L., Lukac, M., Parker, A., Stathopoulos, T., Tseng, J., Wang, H., Estrin, D., Guy, R., and Kohler, E. 2005. A reliable multicast mechanism for sensor network applications. Tech. rep., CENS. April 25, 2005.

Girod, L., Lukac, M., Trifa, V., and Estrin, D. 2006. The design and implementation of a self-calibrating distributed acoustic sensing platform. In *ACM SenSys*. Boulder, CO.

Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wolte, J. 1997. The performance of -kernel-based systems". In *Proceedings of the 16th 16th ACM SOSP*.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Arhitectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. ACM., Cambridge, MA, USA, 93–104.

Levis, P., Lee, N., Welsh, M., and Culler, D. 2003. TOSSIM: Accurate and Scalable Simulations of Entire TinyOS Applications. In *Sensys*. Los Angeles.

Lukac, M., Girod, L., and Estrin, D. 2006. Disruption tolerant shell. In *Proceedings of the 2006 ACM SIGCOMM Workshop on Challenged Networks*. Pisa, IT.

Merrill, W., Girod, L., Schiffer, B., McIntire, D., Rava, G., Sohrabi, K., Newberg, F., Elson, J., and Kaiser, W. 2004. Dynamic networking and smart sensing enable next-generation landmines. *IEEE Pervasive Computing Magazine*.

Merrill, W. M., Sohrabi, K., Girod, L., Elson, J., Newberg, F., and Kaiser, W. 2002. Open standard development platforms for distributed sensor networks. In *SPIE Unattended Ground Sensor Technologies and Applications IV*. 327–337.

Ousterhout, J. K. 1998. Higher level programming for the 21st century.

Pike, R., Presotto, D., Thompson, K., and Trickey, H. 1990. Plan 9 from bell labs. In *Proceedings of the Summer 1990 UKUUG Conference*. 1–9.

Polastre, J., Hui, J., Levis, P., Zhao, J., Culler, D., Shenker, S., and Stoica, I. 2005. A unifying link abstraction for wireless sensor networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys05)*. ACM.

Rosenblum, M., Bugnion, E., Devine, S., and Herrod, S. 1997. Using the simos machine simulator to study complex computer systems. *ACM TOMACS Speical Issue on Computer Simulation*.

Stathopoulos, T., Girod, L., Heidemann, J., and Estrin, D. 2005. Mote herding for tiered wireless sensor networks. Tech. Rep. CENS-TR-59, University of California, Los Angeles, Center for Embedded Networked Computing. Dec.

Swift, M. M., Annamalai, M., Bershad, B. N., and Levy, H. M. 2004. Recovering device drivers. In *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation*. San Francisco.

Tanenbaum, A. S. and van Renesse, R. 1988. A critique of the remote procedure call paradigm. In *EUTECO '88 Proceedings*. 775–783.

Titzer, B. L., Lee, D. K., and Palsberg, J. 2005. Avrora: Scalable sensor network simulation with precise timing. In *IPSN '05: Proceedings of the Fourth ACM/IEEE International Conference on Information Processing in Sensor Networks*.

Trifa, V., Girod, L., Collier, T., Blumstein, D. T., and Taylor, C. E. 2007. Automated wildlife monitoring using self-configuring sensor networks deployed in natural habitats. In *The 12th International Symposium on Artificial Life and Robotics (AROB)*.

Vaughan, R. T., Gerkey, B., and Howard, A. 2003. On Device Abstractions For Portable, Resuable Robot Code. In *In IEEE/RSJ IROS 2003*. Las Vegas, USA.

Wang, H., Chen, C.-E., Ali, A., Asgari, S., Hudson, R. E., Yao, K., Estrin, D., and Taylor, C. 2005. Acoustic sensor networks for woodpecker localization. In *SPIE Conference on Advanced Signal Processing Algorithms, Architectures and Implementations*.

Wang, H., Yao, K., Pottie, G., and Estrin, D. 2004. Entropy-based sensor selection heuristic for localization. In *Symposium on Information Processing in Sensor Networks (IPSN04)*.

Ye, W., Heidemann, J., and Estrin, D. 2002. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of IEEE INFOCOM*.

Received December 2005