USENIX Association

Proceedings of the General Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

EmStar: a Software Environment for Developing and Deploying Wireless Sensor Networks

Lewis Girod Jeremy Elson Alberto Cerpa
Thanos Stathopoulos Nithya Ramanathan Deborah Estrin

Center for Embedded Networked Sensing
University of California, Los Angeles
Los Angeles, CA 90095 USA
{girod,jelson,cerpa,thanos,nithya,destrin}@cs.ucla.edu

Abstract

Many Wireless Sensor Network (WSN) applications are composed of a mixture of deployed devices with varying capabilities, from extremely constrained 8-bit "Motes" to less resource-constrained 32-bit "Microservers". EmStar is a software environment for developing and deploying complex WSN applications on networks of 32-bit embedded Microserver platforms, and integrating with networks of Motes. EmStar consists of **libraries** that implement message-passing IPC primitives, **tools** that support simulation, emulation, and visualization of live systems, both real and simulated, and **services** that support networking, sensing, and time synchronization. While EmStar's design has favored ease of use and modularity over efficiency, the resulting increase in overhead has not been an impediment to any of our current projects.

1 Introduction

The field of wireless sensor networks (WSNs) is growing in importance [1], with new applications appearing in the commercial, scientific, and military spheres, and an evolving family of platforms and hardware. One of the most promising signs in the field is a growing involvement by researchers *outside* the networking systems field who are bringing new application needs to the table. A recent NSF Workshop report [4] details a number of these needs, building on early experience with deployments (e.g. GDI [7], CENS [23], James Reserve [26]).

Many of these applications lead to "tiered architecture" designs, in which the system is composed of a mixture of platforms with different costs, capabilities and energy budgets [5] [21]. Low capability nodes, often Crossbow Mica Motes [24] running TinyOS [17], can perform simple tasks and provide long life at low cost. The high capability nodes, or *Microservers*, generally consume more energy, but in turn can run more complex software and support more sophisticated sensors. EmStar is a software environment targeted at Microserver platforms.

Microservers, typically iPAQ or Crossbow Stargate platforms, are central to several new applications at CENS. The Extensible Sensing System (ESS) employs Microservers as data sinks to collect and report microclimate data at the James Reserve. A proposed 50-node seismic network will use Stargates to measure and report seismic activity using a high-precision multichannel Analog to Digital Converter (ADC). Ongoing research in acoustic sensing uses iPAQ hardware to do beamforming and animal call detection. Although EmStar systems do not target Motes as a platform, EmStar systems can easily interoperate with Motes and Mote networks.

In this paper, we intend to show how EmStar addresses the needs of WSN applications. To motivate this discussion, Figure 1 details a hypothetical application for which EmStar is well-suited. In this example, several nodes collaborate to acoustically localize an animal based on its call—an improved version of our system described in [8]. The large dashed box shows how the system might be implemented by combining existing EmStar components (gray boxes) with hypothetical application-specific components (light gray dashed boxes). Because EmStar systems are composed from small reusable components, it is easy to plug new application-specific components into many different layers of the system.

Although most of the implemented components in the diagram are described in more detail later in the paper, we will briefly introduce them here. The emrun module serves as a management and watchdog process, starting up, monitoring, and shutting down the system. The emproxy module is a gateway to a debugging and visualization system. The udpd, linkstats, neighbors and MicroDiffusion modules implement a network stack designed to work in the context of wireless links characterized by highly variable link quality and network topology. The timehist, syncd, and audiod modules together implement an audio sampling service that supports accurate correlation of time series across a set of nodes. The hypothetical modules include FFT, which computes a streaming Fourier transform of the acoustic input, detect, which is designed to detect a particular acoustic signature, and collab_detect, which orchestrates collaborative detection across several nodes.

This application demonstrates several of the attributes that are special to WSNs. First, the nodes in the system

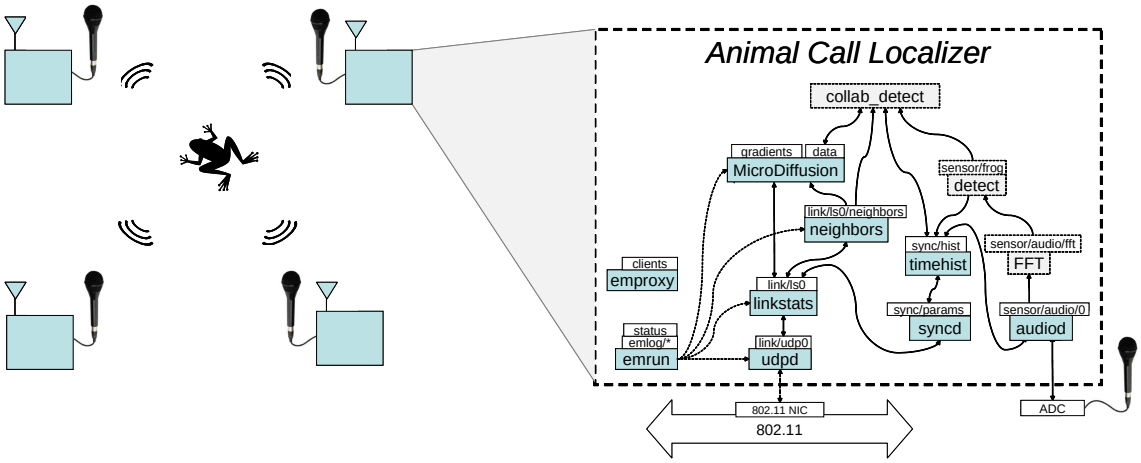

Figure 1: To motivate the EmStar design, we show a block diagram of a hypothetical WSN application to which EmStar is well suited. The diagram shows an improved version of our prototype animal call localization system described in [8]. In the new design, a network of "Localizer" nodes collaborate over a wireless network to localize an animal by its call. Each node detects the specific audio signature of the target animal and then collaboratively locates the target by comparing the arrival time of the signal at multiple points. The dashed box is an exploded view showing how EmStar components might be used to implement the Localizer nodes. The gray boxes represent existing EmStar modules, while the light gray dashed boxes represent hypothetical application specific modules. The white boxes represent various types of named device interface, including Sensor Devices, Link Devices, and Status Devices. Arrows indicate client-server relationships. Although all services have a control channel to EmRun, only four are shown, represented by dashed arcs.

have a higher probability of failure or disconnection than many Internet-based systems. Wireless connectivity and network topology can vary greatly, and systems deployed "in the wild" are also subject to hardware failures with higher probability. While Internet distributed systems often have low standards of client reliability, they typically assume a "core" of high reliability components that is not always present in a WSN.

Second, the digital signal processing (DSP) algorithms running on each node are complex and must work for a broad set of inputs that is difficult to characterize. In practice, this means that certain unexpected conditions may cause unforeseen error conditions. Fault tolerance and layers of filtering are needed to absorb these transients.

Third, energy considerations, along with aforementioned properties of wireless, influence the design of networking primitives. These issues favor soft state and hop-by-hop protocols over end-to-end abstractions. Energy considerations may also necessitate system-wide coordination to duty cycle the node. While many of the these issues are similar to those addressed by TinyOS [17], EmStar is better suited to applications built on higher performance platforms.

2 Tools and Services

EmStar incorporates many tools and services germane to the creation of WSN applications. In this section, we briefly describe these tools and services, without much implementation detail. In Section 3, we detail key building blocks used to implement these tools. Then, in Section 4 we show how the implementation makes use of the building blocks.

2.1 EmStar Tools

EmStar tools include support for deployment, simulation, emulation, and visualization of live systems, both real and simulated.

EmSim/EmCee Transparent simulation at varying levels of accuracy is crucial for building and deploying large systems [9] [11]. Together, EmSim and EmCee comprise several accuracy regimes. EmSim runs many virtual nodes in parallel, in a pure simulation environment that models radio and sensor channels. EmCee runs the EmSim core, but provides an interface to real low-power radios instead of a modeled channel. The array of radio transceivers used by EmCee is shown in Figure 2(b).

These 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. 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 also eliminates accidental code differences that can arise when running in simulation requires modifications. Other "real-code" simulation environments include TOSSim [11] and SimOS [20].

EmView/EmProxy EmView is a graphical visualizer for EmStar systems. Figure 2(a) shows a screen-shot of EmView displaying real-time state of a running emulation. Through an extensible design, developers can easily add "plugins" for new applications and services. EmView uses

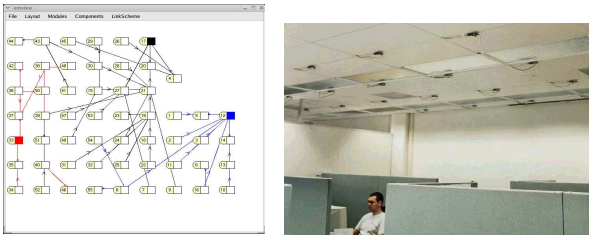

Figure 2: (a) EmView and (b) the Ceiling Array

a UDP protocol to request status updates from real or simulated nodes. Although the protocol is only best-effort, the responses are delivered with low latency, such that EmView captures real-time system dynamics. EmProxy is a server that runs on a node or as part of a simulation, and handles requests from EmView. Based on the request, EmProxy will monitor node status and report report back changes in real time.

EmRun EmRun starts, stops, and manages running services in EmStar It processes a config file that specifies how the EmStar services are "wired" together, and starts the system up in dependency order, maximizing parallelism. EmRun also maintains a control channel to each child process that enables it to monitor process health (respawn dead or stuck processes), initiate graceful shutdown, and receive notification when starting up that initialization is complete. Log messages emitted by EmStar services are processed centrally by EmRun and exposed to interactive clients as in-memory log rings with runtime-configurable loglevels.

2.2 EmStar Services

EmStar services include support for networking, sensing, and time synchronization.

Link and Neighborhood Estimation Wireless channels have a significant "gray zone" where connectivity is unreliable and highly time-varying [6]. Node failures are also common. Therefore, applications are brittle when they assume the topology is pre-configured. Dynamic neighbor discovery is a basic service needed by all collaborative applications if they are to be robust. Potential collaborators must be discovered at *run-time*.

EmStar's Neighbors service monitors links and provides applications with a list of active, reliable nodes. Applications are notified when the list changes so that they can take action in response to environmental changes. The LinkStats service goes one step further: in exchange for slightly more packet overhead, it provides much finer-grained reliability statistics. This can be useful, for example, to a routing algorithm that weights its path choices by link reliability.

Time Synchronization The ability to relate the times of events on different nodes is critical to most distributed sensing applications, especially those interested in correlation of high-frequency phenomena. The TimeSync service provides a mechanism for converting among CPU clocks (i.e. gettimeofday()) on neighboring nodes. Rather than attempt to synchronize the clocks to a specific "master", TimeSync estimates conversion parameters that enable a timestamp from one node to be interpreted on another node. Timesync can also compute relations between the local CPU clock and other clocks in the system, such as sample indices from an ADC or the clocks of other processor modules [3].

Routing EmStar supports several types of routing: Flooding, Geographical, Quad-Tree, and Diffusion. One of the founding principles of EmStar is that innovation in routing and hybrid transport/routing protocols are key research areas in the development of wireless sensor network systems. EmStar "supports" several routing protocols, but it also makes it easy to invent your own. For example, the authors of Directed Diffusion [16] [18] have ported diffusion to run on top of EmStar.

2.3 EmStar Device Support

EmStar includes native support for a number of devices, including sensors and radio hardware.

HostMote and MoteNIC EmStar systems often need to act as a gateway to a network of low-energy platforms such as Mica Motes running TinyOS. The HostMote service implements a serial line protocol between a Mote and an EmStar node. HostMote provides an interface to configure the attached Mote and an interface that demultiplexes Mote traffic to multiple clients. MoteNIC is a packet relay service built over HostMote. MoteNIC provides a standard EmStar data link interface, and pipes the traffic to software on the attached Mote that relays those packets onto the air.

Audio Server The Audio service provides buffered and continuous streaming interfaces to audio data sampled by sound hardware. Applications can use the Audio service to acquire historical data from specific times, or to receive a stream of data as it arrives. Through integration with the TimeSync service, an application can relate a specific series of samples on one node to a series taken at the same time on another node. The ability to acquire historical data is crucial to implementing triggering and collaboration algorithms where there may be a significant nondeterministic delay in communication due to channel contention, multihop communication, duty cycling, and other sources of delay.

3 Building Blocks

In this section, we will describe in more detail the building blocks that enabled us to construct the EmStar suite of tools and services. EmStar systems encapsulate logically separable modules within individual processes, and enable communication among these modules through message passing via device files. This structure provides for fault isolation and independence of implementation among services and applications.

In principle, EmStar does not specify anything about the implementation of its modules, apart from the POSIX system call interface required to access device files. For example, most EmStar device interfaces can be used interactively from the shell, and EmStar servers could be implemented in any language that supports the system call interface.

In practice, there is much to be gained from using and creating standard libraries. In the case of EmStar we have implemented these libraries in C, and we have adopted the GLib event framework to manage select() and to support timers. Using the event framework we encapsulate complex protocol mechanisms in libraries, and integrate them without explicit coordination. The decision to use C, GLib, and the POSIX interface was designed to minimize the effort required to integrate EmStar with arbitrary languages, implementation styles, and legacy codebases.

We will now describe some key building blocks in more detail: the EmStar IPC mechanisms and associated libraries. We will explain them in terms of what they do, how they work, and how they are used.

3.1 FUSD

FUSD, the Framework for User-Space Devices, is essentially a microkernel extension to Linux. FUSD allows device-file callbacks to be proxied into user-space and implemented by user-space programs instead of kernel code. Though implemented in userspace, FUSD drivers can create device files that are semantically indistinguishable from kernel-implemented /dev files, from the point of view of the processes that use them. FUSD follows in the tradition of microkernel operating systems that implement POSIX interfaces, such as QNX [29] and GNU HURD [25].

As we will describe in later sections, this capability is used by EmStar modules for both communication with other modules and with users. Of course, many other IPC methods exist in Linux, including sockets, message queues, and named pipes. We have found a number of compelling advantages in using using user-space device drivers for IPC among EmStar processes. For example, system call return values come from the EmStar processes themselves, not the kernel; a successful write() guarantees that the data has reached the application. Traditional IPC has much weaker semantics, where a successful write() means only that the data has been accepted into a kernel buffer, not that it has been read or acknowledged by an application. FUSD-based IPC obviates the need for explicit application-level acknowledgment schemes built on top of sockets or named pipes.

FUSD-driven devices are a convenient way for applications to transport data, expose state, or be configured in a convenient, browseable, named hierarchy—just as the kernel itself uses the /proc filesystem. These devices can respond to system calls using custom semantics. For example, a read from a packet-interface device (Section 3.2.2) will always begin at a packet boundary. The customization of system call semantics is a particularly powerful feature, allowing surprisingly expressive APIs to be constructed. We will explore this feature further in Section 3.2.

3.1.1 FUSD Implementation

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

FUSD drivers are conceptually similar to kernel drivers: a set of callback functions called in response to system calls made on file descriptors by user programs. In addition to the device name, fusd_register() accepts a structure full of pointers to callback functions, used in response to client system calls—for example, when another process tries to open, close, read from, or write to the driver's device. The callback functions are generally written to conform to the standard definitions of POSIX system call behavior. In many ways, the user-space FUSD callback functions are identical to their kernel counterparts.

When a client executes a system call on a FUSD-managed device (e.g., open() or read()), the kernel activates a callback in the FUSD kernel module. The module blocks the calling process, marshals the arguments of the system call, and sends a message to the user-space driver managing the target device. In user-space, the library half of FUSD unmarshals the message and calls the user-space callback that the FUSD driver passed to fusd_register(). When that user-space callback returns a value, the process happens in reverse: the return value and its side-effects are marshaled by the library and sent to the kernel. The FUSD kernel module unmarshals the message, matches it with the corresponding outstanding request, and completes the system call. The calling process is completely unaware of this trickery; it simply enters the kernel once, blocks, unblocks, and returns from the system call—just as it would for a system call to a kernel-managed device.

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

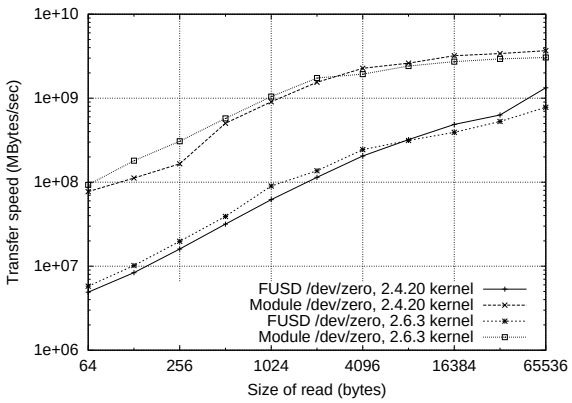

Figure 3: Throughput comparison of FUSD and in-kernel implementations of /dev/zero. The test timed a read of 1GB of data from each test device on a 2.8 GHz Xeon, for both 2.4 and 2.6 kernels. We tested read() sizes ranging from 64 bytes to 64 Kbytes. Larger read sizes are higher throughput because the cost of a system call is amortized over more data.

3.1.2 FUSD Performance

While FUSD has many advantages, the performance of drivers written using FUSD suffers relative to an in-kernel implementation. To quantify the costs of FUSD, we compared the performance of FUSD and in-kernel implementations of the /dev/zero device in Linux. To implement /dev/zero using FUSD, we implemented a server with a read() handler that returned a zeroed buffer of the requested length. The in-kernel implementation implemented the same read() handler directly in the kernel.

Figure 3 shows the results of our experiment, running on a 2.8 GHz Xeon. The figure shows that for small reads, FUSD is about 17x slower than an in-kernel implementation, while for long reads, FUSD is only about 3x slower. This reduction in performance is a combination of two independent sources of overhead.

The first source of overhead is the additional system call overhead and scheduling latency incurred when FUSD proxies the client's system call out to the user-space server. For each read() call by a client process, the user-space server first be scheduled, and then must itself call read() once to retrieve the marshalled system call, and must call writev() once to return the response with the filled data buffer. This additional per-call latency dominates for small data transfers.

The second source of overhead is an additional data copy. Where the native implementation only copies the response data back to the client, FUSD copies the response data twice: once to copy it from the user-space server, and again to copy it back to the client. This cost dominates for large data transfers.

In our experiments, we tested both the 2.6 and 2.4 kernels, and found that 2.6 kernels yielded an improvement for smaller transfer sizes. The 2.6 kernel has a more significant impact when many processes are running in parallel, as shown in the results of our tests of EmStar simulations in Section 4.1.4. Further performance analysis of specific EmStar FUSD-based interfaces appears in Section 3.3.2.

3.2 Device Patterns

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 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.

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, while the Status Device library defines the mechanism of handling each read(), it calls back to the application to represent its current "status" as data.

Relative to other approaches such as log files and status files, a key property of EmStar device patterns is their active nature. For example, the Logring Device pattern creates a device that appears to be a regular log file, but always contains only the most recent log messages, followed by a stream of new messages as they arrive. The Status Device pattern appears to be a file that always contains the most recent state of the service providing it. However, most status devices also support poll()-based notification of changes to the state.

The following sections will describe the Device Patterns 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.

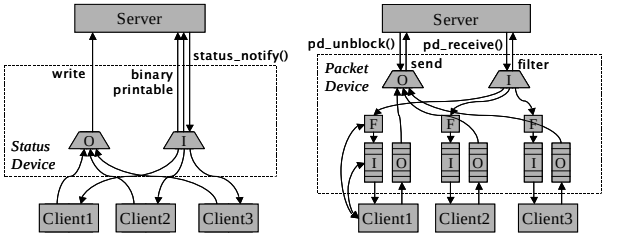

Figure 4: Block diagram of the (a) Status and (b) Packet Device patterns. In the Packet Device diagram, the "F" boxes are client-configurable filters, and the curved arrows from Client1 represent ioctl() based configuration of queue lengths and message filtering. Trapezoid boxes represent multiplexing of clients.

3.2.1 Status Device

The Status Device pattern provides a device that reports the current state of a module. 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 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 both convenience and compactness.

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 break 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 the 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).

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 status device that was client-configurable to select only one of the trees.

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 input and output 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 Command Device

The Command Device pattern provides an interface similar to the writable entries in the Linux /proc filesystem, which enable user processes to modify configurations and trigger actions. In response to a write(), the provider of the device processes and executes the command, and indicates any problem with the command by returning an error code. Command Device does not support any form of delayed or asynchronous return to the client.

While Command Devices can accept arbitrary binary data, they typically parse a simple ASCII command format. Using ASCII enables interactivity from the shell and often makes client code more readable. Using a binary structure might be slightly more efficient, but performance is not a concern for low-rate configuration changes.

The Command Device pattern also includes a read() handler, which is typically used to report "usage" information. Thus, an interactive user can get a command summary using cat and then issue the command using echo. Alternatively, the Command Device may report state information in response to a read. This behavior would be more in keeping with the style used in the /proc filesystem, and is explicitly implemented in a specialization of Command Device called the Options Device pattern.

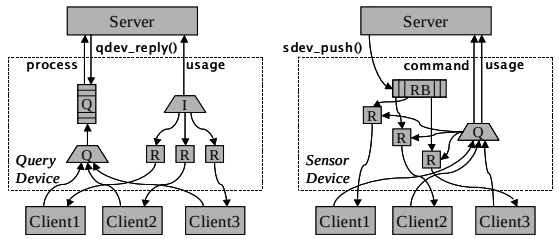

Figure 5: Block diagram of the (a) Query and (b) Sensor Device patterns. In the Query Device, queries from the clients are queued and "process" is called serially. The "R" boxes represent a buffer per client to hold the response to the last query from that client. In the Sensor Device, the server submits new samples by calling sdev_push(). These are stored in the ring buffer (RB), and streamed to clients with relevant requests. The "R" boxes represent each client's pending request.

3.2.4 Query Device

The Device Patterns we have covered up to now provide useful semantics, but none of them really provides the semantics of RPC. To address this, the Query Device pattern implements a transactional, request/response semantics. To execute a transaction, a client first opens the device and writes the request data. Then, the client uses poll() to wait for the file to become readable, and reads back the response in the same way as reading a Status Device. For those services that provide human-readable interfaces, we use a universal client called echocat that performs these steps and reports the output.

It is interesting to note that the Query Device was not one of the first device types implemented; rather, most configuration interfaces in EmStar have been implemented by separate Status and Command devices. In practice, any given configurable service will have many clients that need to be apprised of its current configuration, independent of whether they need to change the configuration. This is exacerbated by the high level of dynamics in sensor network applications. Furthermore, to build more robust systems we often use soft-state to store configurations. The current configuration is periodically read and then modified if necessary. The asynchronous Command/Status approach achieves these objectives while addressing a wide range of potential faults.

To the service implementing a Query Device, this pattern offers a simple, transaction-oriented interface. The service defines a callback to handle new transactions. Queries from the client are queued and are passed serially to the transaction processing callback, similar to the way the output queues are handled in a Packet Device. If the transaction is not complete when the callback returns, it can be completed asynchronously. At the time of completion, a response is reported to the device library, which it then makes available to the client. The service may also optionally provide a callback to provide usage information, in the event that the client reads the device before any query has been submitted.

Clients of a Query Device are normally serviced in round-robin order. However, some applications need to allow a client to "lock" the device and perform several back-to-back transactions. The service may choose to give a current client the "lock", with an optional timeout. The lock will be broken if the timeout expires, or if the client with the lock closes its file descriptor.

3.3 Domain-Specific Interfaces

In Section 3.2 we described several device patterns, generally useful primitives that can be applied to a wide variety of purposes. In this section, we will describe a few examples of more domain-specific interfaces, that are composed from device patterns, but are designed to support the implementation of specific types of services.

3.3.1 Data Link Interface

The Data Link interface is a specification of a standard interface for network stack modules. The Data Link interface is composed of three device files: data, command, and status. These three interfaces appear together in a directory named for the specific stack module.

The data device is a Packet Device interface that is used to exchange packets with the network. All packets transmitted on this interface begin with a standard link header that specifies common fields. This link header masks certain cosmetic differences in the actual over-the-air headers used by different MAC layers, such as the Berkeley MAC [17] and SMAC [22] layers supported on Mica Motes.

The command and status devices provide asynchronous access to the configuration of a stack module. The status device reports the current configuration of the module (such as its channel, sleep state, link address, etc.) as well as the latest packet transfer and error statistics. The command device is used to issue configuration commands, for example to set the channel, sleep state, etc. The set of valid commands and the set of values reported in status varies with the underlying capabilities of the hardware. However, the binary format of the status output is standard across all modules (currently, the union of all features).

Several "link drivers" have been implemented in EmStar, to provide 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. Two versions of the Mote driver exist, one that supports the standard Berkeley MAC and one that supports SMAC. Because all of these drivers conform to the link interface spec, some applications can work more or less transparently over different physical radio hardware. In the event that an application needs information about the radio layer (e.g. the nominal link capacity), that information is available from the status device.

In addition to providing support for multiple underlying

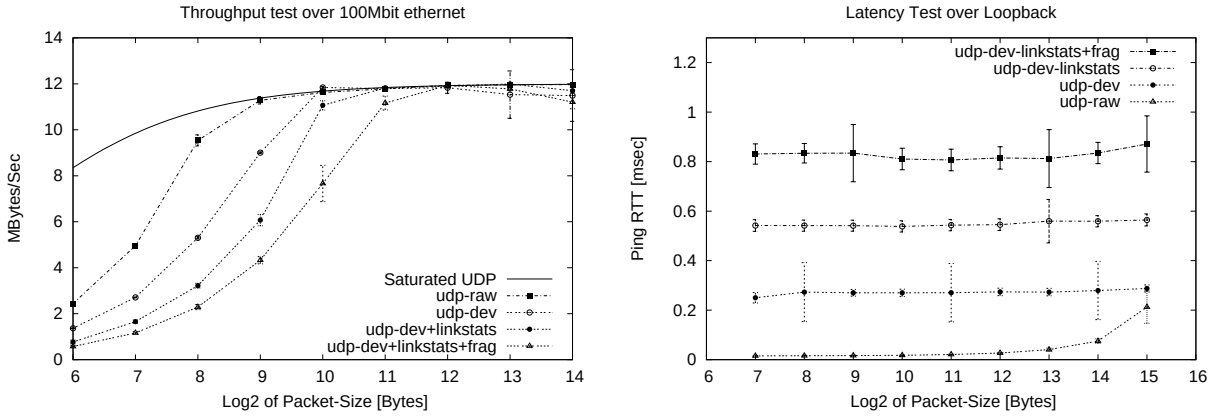

Figure 6: Measurements of the EmStar stack on a 700 MHz Pentium III running the 2.4.20 kernel. The throughput graph shows the performance of a single process sending at maximum rate over a 100Mbit Ethernet, as a function of packet length, through different EmStar stacks. The solid curve represents link saturation, while the other curves compare the performance of sending directly to a socket with that of sending through additional layers. The error bars are 95% confi dence intervals. The latency graph 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, consistent with previous results about FUSD.

radio types, the standard Data Link interface enables a variety of useful "pass-through" stack modules and routing modules. Two standard modules in EmStar network stacks are LinkStats and Fragmentation. Both of these sit between a client and an underlying radio driver module, transparently to the client. In addition to passing data through, they proxy and modify status information, for example updating the MTU specification.

3.3.2 Cost Analysis of the Data Link Interface

Our discussion up to this point has yet to address the cost of this architecture. In order to quantify some of these costs, we performed a series of experiments, the results of which are shown in Figure 6. We found that while our architecture introduces a measurable increase in latency and decrease in throughput relative to a highly integrated and optimized solution, these costs have a negligible impact when applied to a low bandwidth communications channel. This is an important case, since EmStar is intended for WSN applications which typically are designed to have a high ratio of CPU to communication.

To assess the costs of EmStar, we measured the costs incurred by layering additional modules over an EmStar link device. 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 LinkStats module, represented by the +linkstats curves. For a three-layer stack, we added a Fragmentation module over LinkStats, shown by the +frag curves.

Our first experiment characterized the cost of EmStar in terms of throughput. In Figure 6(a), our test application sent UDP packets as quickly as possible over a 100Mbit Ethernet channel. We ran this application over our four configurations, comparing direct sends to a socket with three EmStar configurations. For each configuration, the time required to send 1000 packets was measured, and the results of 10 such trials were averaged. The graph shows that per-packet overhead prevents the application from saturating the link until larger packet sizes sufficiently amortize the per-packet costs. Per-packet costs include scheduling latency and system call overhead, while message-passing across the user-kernel boundary results in additional per-byte costs.

Our second experiment characterized the cost of EmStar in terms of latency. In Figure 6(b), our test application sent UDP "ping" packets over the loopback interface to a ping replier on the same machine. 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. 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.

While these experiments show definite costs to the EmStar architecture, these costs are less critical for WSN applications where communications channels have lower bandwidths and higher latency relative to the rate of local processing. For example, many of our applications use a Mote as a radio interface, which has a maximum bandwidth of about 19.2Kbit/sec and incurs a latency of 125 milliseconds to transmit a 200 byte packet over serial to the Mote and then over the channel. Given this type of interface, the

additional latency and bandwidth costs of EmStar are negligible.

3.3.3 Sensor Device

Two of the applications that drove the development of EmStar centered around acquisition and processing of audio data. One application, a ranging and localization system [15], extracts and processes audio clips from a specific time in the past. The other, a continuous frog call detection and localization system [8], receives data in a continuous stream. Both applications needed to be able to correlate time series data captured on a distributed set of nodes, thus timing relationships among the nodes needed to be maintained.

The Sensor Device interface encapsulates a ring buffer that stores a history of sampled data, and integrates with the EmStar Time Synch service to enable clients to relate local sensor data to sensor data from other nodes. A client of the sensor device can open the device and issue a request for a range of samples. When the sample data is captured, the client is notified and the data is streamed back to the client as it continues to arrive.

Keeping a history of recent sensor data and being able to relate the sample timing across the network is critical to many sensor network applications. By retaining a history of sampled data, it is much easier to implement applications where an event detected on one node triggers further investigation and sensing at other nodes. Without local buffering, the variance in multi-hop communications times makes it difficult to abstract the triggered application from the communications stack.

3.4 EmStar Events and Client APIs

One of the benefits of the EmStar design is that services and applications are separate processes and communicate through POSIX system calls. As such, EmStar clients and applications can be implemented in a wide variety of languages and styles. However, a large part of the convenience of EmStar as a development environment comes from a set of helper libraries that improve the elegance and simplicity of building robust applications.

In Section 3.2 we noted that an important part of device patterns is the library that implements them on the service side. Most device patterns also include a client-side "API" library, that provides basic utility functions, GLib compatible notification interfaces, and a *crashproofing* feature intended to prevent cascading failures.

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.

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. 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.

4 Examples

The last section enumerated a number of building blocks that are the foundation for the EmStar environment. In this Section, we will describe how we have used them to construct several key EmStar tools and services.

4.1 EmSim and EmCee

EmSim and EmCee are tools designed to simulate unmodified EmStar systems at varying points on the continuum from simulation to deployment. EmSim is a pure simulation environment, in which many virtual nodes are run in parallel, interacting with a simulated environment and radio channel. EmCee is a slightly modified version of EmSim that provides an interface to real low-power radios in place of a simulated channel.

EmSim itself is made up of modules. The main EmSim module maintains a central repository for node information, initially sourced from a configuration file, and exposed as a Status Device. EmSim then launches other modules that are responsible for implementing the simulated "world model" based on the node configuration. After the world is in place, EmSim begins the simulation, starting up and shutting down virtual nodes at the appropriate times.

4.1.1 Running Virtual Nodes

The uniform use of the /dev filesystem for all of our I/O and IPC leads to a very elegant mechanism for transparency between simulation, various levels of reality, and real deployments. The mechanism relies on name mangling to cause all references to /dev/* to be redirected deeper into the hierarchy, to /dev/sim/groupX/nodeY/*. This is achieved through two simple conventions.

First, all EmStar modules must include the call to misc_init() early in their main() function. This function checks for certain environment variables to determine whether the module is running in "simulation mode", and what its group and node IDs are. The second convention is to wrap every instance of a device file name with sim_path(). This macro will perform name-mangling based on the information discovered in misc_init(). For simplicity, we typically include the sim_path() wrapper at the definition of device names in interface header files.

This approach enables easy and transparent simulation of many nodes on the same machine. This is not the case for many other network software implementations. Whenever the system being developed relies on mechanisms inside the kernel that can't readily be partitioned into virtual machines, it will be difficult to implement a transparent

simulation.

For example, ad-hoc routing code that directly configures the network interfaces and kernel routing table is very difficult to simulate transparently. While a simulation environment such as *ns-2* [27] does attempt to run much of the same algorithmic code as the real system, it does so in a very intrusive, #ifdef-heavy way. This makes it cumbersome to keep the live system in sync with the *ns-2* version.

In contrast, EmStar modules don't even need to be recompiled to switch from simulation to reality, and the EmStar device hierarchy provides transparency into the workings of each simulated EmStar node. However, this flexibility comes at a cost in performance. An ad-hoc routing algorithm that dragged every packet to user-space would likely suffer poorer performance.

4.1.2 Simulated World Models

The capability to transparently redirect EmStar IPC channels enables us to provide a world for the simulated nodes to see, and in some cases, affect. There are many examples of network simulation environments in the networking community, some of which support radio channel modeling [27][28]. In addition, the robotics community has devoted much effort to creating world models [12]. For sensor networks, the robotic simulations are often more appropriate, because they are designed to model a system sensing the environment, and intended to test and debug control systems and behaviors that must be reactive and resilient.

The existence of EmStar device patterns simplifies the construction of simulated devices, because all of the complexity of the interface behavior can be reused. Even more important, by using the same libraries, the chances of subtle behavior differences are reduced. Typically, a "simulation module" reads the node configuration from EmSim's Status Device and then exposes perhaps hundreds of devices, one for each node. Requests to each exposed device are processed according to a simulation of the effects of the environment, or in some cases in accordance with traces of real data.

The notification channel in EmStar status devices enables EmSim to easily support configurations changes during a simulation. Updates to the central node configuration—such as changes in the position of nodes—trigger notification in the simulation modules. The modules can then read the new configuration and update their models appropriately. In addition, we can close the loop by creating a simulation module that provides an actuation interface—for example enabling the node to move itself. In response to a request to move, this module could issue a command to EmSim to update that node's position and notify all clients.

4.1.3 Using Real Channels in the Lab

EmCee is a variant of EmSim that integrates a set of virtual nodes to a set of real radio interfaces, positioned out in the world. We have two EmCee-compatible testbeds: the ceiling array and the portable array. The ceiling array is composed of 55 Crossbow Mica1 Motes, permanently attached to the ceiling of our lab on a 4 foot grid. Serial cabling runs back to two 32-port ethernet to serial multiplexers. The portable array is composed of 16 Crossbow Mica2 Motes and a 16-port serial multiplexer, that can be taken out to the field [6].

The serial multiplexers are configured so that their serial ports appear to be normal serial devices on a Linux server (or laptop in the portable case). To support EmCee, the HostMote and MoteNIC services support an "EmCee mode" where they open a set of serial ports specified in a config file and expose their devices within the appropriate virtual node spaces.

Thus, the difference between EmSim and EmCee is minimal. Where EmSim would start up a radio channel simulator to provide virtual radio link devices, EmCee starts up the MoteNIC service in "EmCee mode", which creates real radio link devices that map to multiplexer serial ports and thus to real Motes.

Our experience with EmCee has shown it is well worth the infrastructure investment. Users have consistently observed that using real radios is substantially different from our best efforts at creating a modeled radio channel [2][6]. Even channels driven by empirical data captured using the ceiling array don't seem to adequately capture the real dynamics. Although testing with EmCee is still not the same as a real deployment, the reduction in effort relative to a deployment far outweighs the reduction in reality for a large part of the development and testing process.

4.1.4 Performance of EmSim/EmCee

Currently, an important limitation of our simulator is that it can only run in real-time, using real timers and interrupts from the underlying operating system. In contrast, a discrete-event simulator such as *ns-2* runs in its own virtual time, and therefore can run for as long as necessary to complete the simulation without affecting the results. Discrete-event simulations can also be made completely deterministic, allowing the developer to more easily reproduce an intermittent bug.

The real-time nature of EmSim/EmCee makes performance an important consideration. With perfect efficiency, the simulator platform would need the aggregate computational power of all simulated nodes. In reality, extra headroom is needed for nonlinear costs of running many processes on a single computer.

To test the actual efficiency, we ran test simulations on two SMP-enabled servers. One had 4 700MHz Pentium-III processors, running Linux kernel 2.4.20. The other had 2 2.8GHz Xeon processors, with hyperthreading disabled, running Linux 2.6.3. We tested both kernels because Linux 2.6 has a "O(1) scheduler"—i.e., the 2.6 scheduler performs constant work per context switch regardless of run-queue size. 2.6 kernels also have much finer-grained locking, thus better kernel parallelism. The FUSD kernel

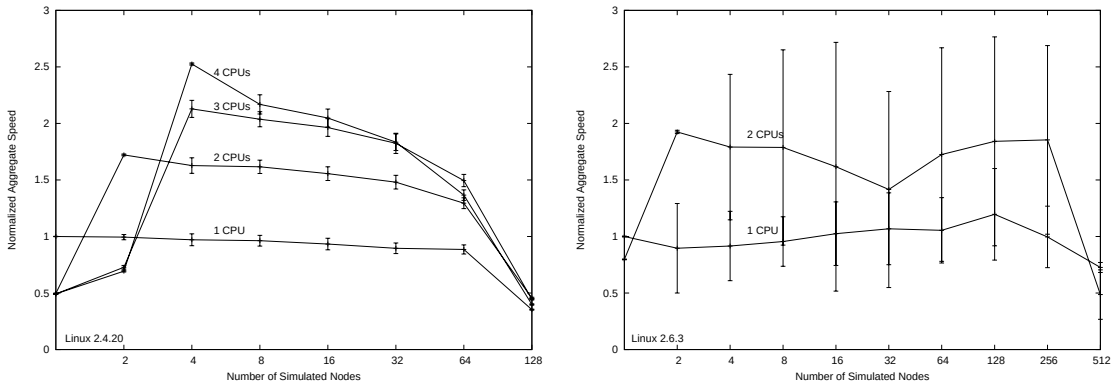

Figure 7: Performance of a simple EmSim simulation, varying the number of nodes simulated, and the number of CPUs available on the simulator platform. Linux kernels 2.4.20 (*left*) and 2.6.3 (*right*) were tested. Each "node" is two processes that continuously exchange data via a EmStar Status Interface. We plot the aggregate transfer rate summed across all simulated nodes. Results are normalized so that $y = 1$ corresponds to the speed achieved by a single-node simulation (2 processes) running on a single CPU.

module also has fine-grained locking.

In our initial testing, the default Linux scheduler was used; no explicit assignment of processes to CPUs was made. Each "node" consisted of two processes that exchanged data at maximum possible rate via a EmStar Status Device. The results are in Figure 7.

We draw several conclusions from the data. First, the Linux 2.6 scheduler does seem to be a win. Even with differences in CPU speed factored out, it supported much larger simulations than the 2.4 scheduler (512 vs. 128 nodes). In addition, it supported better parallelism: Linux 2.6 with 2 CPUs had, on average, 1.7 times more throughput than for a single CPU, compared to 1.5 times for Linux 2.4. However, Linux 2.6 simulations suffered much higher jitter, i.e. differences in performance from node to node. The cause of this unfairness is still under investigation.

The data also emphasize the high cost of FUSD interprocess communication across processes not running on the same CPU. This can be seen in that a single-node (2-process) simulation ran on a single-CPU platform at nearly at nearly *twice the speed* as on 2-, 3- or 4-CPU platform. The Linux scheduler, by default, places the Status Device Client and Server processes on separate CPUs if available. In applications that have a very high communication-to-computation ratio, as in our test workload, the overhead of extra CPUs is a much higher cost than the benefit of extra cycles. However, many EmStar applications (and WSN applications in general) strive to do as much computation as possible per unit of communication, making these limitations of SMP model a virtual non-issue in "real" simulations.

4.2 EmRun

EmRun starts up, maintains, and shuts down an EmStar system according to the policy specified in a config file. There are three key points in its design: process respawn, in-memory logging, and fast startup, graceful shutdown.

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 3.4). When an EmStar process crashes and restarts, Crashproofing prevents a ripple effect, and the system operates correctly when the process is respawned.

In-Memory Logs EmRun saves each process' output to in-memory log rings that are available interactively from the /dev/emlog/* hierarchy. These illustrate the power of FUSD devices relative to traditional logfiles. Unlike rotating logs, EmStar log rings never need to be switched, never grow beyond a maximum size, and always contain only recent data.

Fast Startup EmRun's fast startup and graceful shutdown is critical for a system that needs to duty cycle to conserve energy. The implementation depends on a control channel that EmStar services establish back to EmRun when they start up. EmStar services notify EmRun when their initialization is complete, signaling that they are now ready to respond to requests. The emrun_init() library function, called by the service, communicates with EmRun by writing a message to /dev/emrun/.int/control. EmRun then launches other processes waiting for that service, based on a configured dependency graph.

This feedback enables EmRun to start independent processes with maximal parallelism, and to wait *exactly* as long as it needs to wait before starting dependent processes. This scheme is far superior to the naive approach of waiting between daemon starts for pre-determined times, i.e., the ubiquitous "sleep 2" statements found in *NIX boot scripts.

Various factors can make startup times difficult to predict and high in variance, such as flash filesystem garbage collection. On each boot, a static sleep value will either be too long, causing slow startup, or too short, causing services to fail when their prerequisites are not yet available.

Graceful Shutdown The control channel is also critical to supporting graceful shutdown. EmRun can send a message through that channel, requesting that the service shut down, saving state if needed. EmRun then waits for SIGCHLD to indicate that the service has terminated. If the process is unresponsive, it will be killed by a signal.

An interesting property of the EmRun control channel is one that differentiates FUSD from other approaches. When proxying system calls to a service, FUSD includes the PID, UID, and GID of the client along with the marshalled system call. This means that EmRun can implictly match up the client connections on the control channel to the child processes it has spawned, and reject connections from non-child processes. This property is not yet used much in EmStar but it provides an interesting vector for customizing device behavior.

4.3 Time-Synchronized Sampling in EmStar

Several of the driving applications for EmStar have involved distributed processing of high-rate audio: audible acoustic ranging, acoustic beamforming, and animal call detection are a few of the applications. We used earlier versions of EmStar to tackle a few of these problems [10][15][8]. Referring back to the animal call localization application of Figure 1, we see how the "syncd" and "audiod" services collaborate so that "collab_detect" can correlate events detected on nodes across the network. In this section, we will describe these services in more detail.

TimeSync Between Nodes The TimeSync service uses Reference Broadcast Synchronization (RBS) [3] to compute relationships among the CPU clocks on nodes in a given broadcast domain. This technique correlates the arrival times of broadcast packets at different nodes and uses linear regression to estimate conversion parameters among clocks that receive broadcasts in common. We chose RBS because techniques based on measuring send times, such as TPSN [14], are not generally applicable without support at the MAC layer. Requiring this support would rule out many possible radios, including 802.11 cards.

A key insight in RBS is that it is better to enable conversion than to attempt to train a clock to follow some remote "master" clock. Training a clock has many negative repercussions for the design of a sampling system caused by clock discontinuities and distortions.

Thus, TimeSync is really a "time conversion" service. The output of the regression is reported through the /dev/sync/params/ticker device, in a complete listing of all known pairwise conversions. Clients of TimeSync read this device to get the latest conversion parameters, then convert times from one timebase to another. The code for reading the device and converting among clocks is implemented in a library.

TimeSync within a Node Many systems have more than one clock. For example, a Stargate board, with an attached Mote and an audio card has three independent clocks. Thus to compare audio time series from two independent nodes, an index in a time series must be converted first to local CPU time, then to remote CPU time, and finally to a remote audio sample index.

The TimeSync service provides an interface for other services to supply pair-wise observations to it, i.e. a CPU timestamp and a clock-X timestamp. This interface uses a Directory device to enable clients to create a new clock, and associate it with a numeric identifier. The client then writes periodic observations of that clock to the timesync command device /dev/sync/params/command. The observations are fit using linear regression to compute a relationship between the two local clocks.

The Audio Server The Audio service provides a Sensor Device output It defines a "sample clock", which is the index of samples in a stream, and submits observations relating the sample clock to the CPU time to TimeSync.

A client of the Audio service can extract a sequence of data from a specific time period by first using TimeSync to convert the begin and end times to sample indices and then placing a request to the Audio service for that sample range. Conversely, a feature detected in the streaming output at a particular sample offset can be converted to a CPU time. These clock relations can also be used to compute and correct the skew in sample rates between devices, which can otherwise cause significant problems.

Generating the synch observations requires minor changes to the audio driver in the kernel. We have made patches for two audio drivers: the iPAQ built-in audio driver and the Crystal cs4281. In both cases, incoming DMA interrupts are timestamped and retrieved by the Audio service via ioctl(). While this approach makes the system harder to port to new platforms and hardware, it is a better solution for building sensing platforms.

The more common solution, the "synchronized start" feature of many sound cards, has numerous drawbacks. First, it only gives you one data point for the run, where our technique gives you a continous stream of points to average. Second, it is subject to drift, and since the end is not timestamped there is no way to accurately determine the actual sample rate. Third, it forces the system to coordinate use of the audio hardware, whereas the Audio server runs continuously and allows access by multiple clients.

5 Design Philosophy and Aesthetics

In this section, we will describe some of the ideas behind the choices we made in the design of EmStar. These choices were motivated by the issues faced by WSNs, which have much in common with traditional distributed systems.

5.1 No Local/Remote Transparency

One of the disadvantages of FUSD relative to sockets is that connections to FUSD services are always local, whereas sockets provide transparency between local and remote connections. Nonetheless, we elected to base EmStar on FUSD because we felt that the advantages outweighed the disadvantages.

The primary reason for giving up remote transparency in EmStar is that remote access is rarely transparent in WSNs. Communications links in WSNs are characterized by high or variable latency, varying link quality, evolving topologies, and generally low bandwidth. In addition, the energy cost of communication in WSNs motivates innovative protocols that minimize communications, make use of broadcast channels, tolerate high latency, and make tradeoffs explicit to the system designer. Remote communication in WSNs is demonstrably different than local communication, and very little is achieved by masking that fact.

In abandoning remote transparency, the client gains the benefit of knowing that each synchronous call will be received and processed by the server with low latency. While an improperly implemented server can introduce delays, there is never a need to worry that the network might introduce unexpected delay. Requests that are known to be time consuming can be explicitly implemented so that the results are returned asynchronously via notification (e.g. Query Device).

5.2 Intra-Node Fault Tolerance

Tolerance of node and communications failures is important to the design of all distributed systems. In WSNs, node robustness takes on an even greater importance. First, the cost of replacing or repairing embedded nodes can be much higher, especially when network access to the node is unreliable or a physical journey is required—in extreme cases, nodes may be physically irretrievable. Second, many scientific applications of WSNs intend to discover new properties of their environment, which may expose the system to new inputs and exercise new bugs.

We address fault tolerance within a node in several ways: EmRun respawn, "crashproofing", soft-state refresh, and transactional interface design. We discussed EmRun respawn and crashproofing in Sections 4.2 and 3.4, as a means of keeping the EmStar services running and preventing cascading failures when an underlying service fails.

While soft-state and transactional design are standard techniques in distributed systems, in EmStar we apply these techniques to IPC as well. Status devices are typically used in a soft-state mode. Rather than reporting more economical "diffs", every status update reports the complete current state, leaving the client to decide how to respond based on its own state. To limit the damage caused by a missing notification signal, clients periodically request a refresh in the absence of notification. When the aggregate update rate is low it is usually easy to make the case for trading efficiency for robustness and simplicity.

Similar considerations hold in the reverse direction. Clients that push state *to* a service typically use transactional semantics with a soft-state refresh. Rather than allowing the client and server to get out of synch (e.g. in the event of a server restart), the client periodically resubmits its *complete* state to the service, enabling the service to make appropriate corrections if there is a discrepancy. Where the state in question is very large, there may be reason to implement a more complex scheme, but for small amounts of state, simplicity and robustness carry the day. While trading off efficiency for robustness may not be the right approach for all applications and hardware platforms, it has worked well for the applications we have built.

5.3 Code Reuse

Code reuse and modularity were major design goals of EmStar. EmStar achieves reusabililty through disciplined design, driven by factoring useful components from existing implementations. For example, each device pattern was originally implemented as a part of several different services, and then factored out into a unified solution to a class of problems. Table 1 shows a quantitative picture of reuse.

The design of EmStar services has followed the dictum "encapsulate mechanism, not policy". This approach encourages reuse, and reduces system complexity while maintaining simple interfaces between modules. EmStar implements modules as independent processes rather than as libraries, eliminating a wide variety of unanticipated interactions, thus better controlling complexity as the number of modules increases.

Building Block	Server Uses	Client Uses
Status Device and derivatives	40	22
Command Device	17	N/A
Packet Device	10	5
Data Link Interface	12	32

Table 1: Reuse statistics culled from LXR.

5.4 Reactivity

Reactivity is one of the most interesting characteristics of WSNs. They must react to hard-to-predict changes in their environment in order to operate as designed. Often the tasks themselves require a reaction, for example a distributed control system or a distributed sensing application that triggers other sensing activities. EmStar supports reactivity through notification interfaces in EmStar devices. Most EmStar services and applications are written in an event-driven style that lends itself to reactive design.

5.5 High Visibility

While the decision to stress visibility in the EmStar design was partly motivated by aesthetics, it has paid off handsomely in overall ease of use, development, and debugging. The ability to browse the IPC interfaces in the shell, to see human-readable outputs of internal state, and in many

cases to manually trigger actions makes for very convenient development of a system that could otherwise be quite cumbersome. Tools like EmView also benefit greatly from stack transparency, because EmView can snoop on traffic travelling in the middle of the stack in real time, without modifying the stack itself.

6 Related Work

In addition to related work we mentioned throughout this paper, in this section we highlight the most related systems.

The closest system to EmStar is TinyOS [17]. 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.

Click [19] is a modular software system designed to support implementation of routers. While Click is designed for a different application space, there are many similarities, including an emphasis on modularity. A key difference is that like TinyOS, Click leverages language properties and static configuration to perform global optimizations. EmStar instead supports dynamic configuration and provides greater levels of fault isolation between modules.

Player/Stage [12] is a software system designed for robotics that supports "real-code" simulation. Player is based on sockets protocols, which have the advantage of remote transparency but are not browseable.

7 Conclusion and Future Work

We have found EmStar to be a very useful development environment for WSNs. We use EmStar at CENS in several current development efforts, including a 50-node seismic deployment and the ESS microclimate sensing system. We also support other groups using EmStar, including the NIMS [13] robotic ecology project and ISI ILENSE.

Our current platform focus is the Crossbow/Intel Stargate platform, an inexpensive Linux platform based on the XScale processor. Stargates are much easier to customize than other COTS platforms such as iPAQs.

We plan several extensions to EmStar, including: better integration between Motes and Microservers based on a TinyOS "VM", virtualization of EmSim's clock to enable "simulation pause" and larger simulations, remote device acess over local networks via sockets, and efficient support for high-bandwidth sensor interfaces such as audio, image data, and DSPs using a shared-memory data channel.

Acknowledgements

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. Additional support from the DARPA NEST program (the "GALORE" project, grant F33615-01-C-1906).

References

[1] In *Proc. of the First Intl. Conf. on Embedded Networked Sensor Systems (ACM Sensys)*, Los Angeles, CA, November 2003.
[2] H. Dubois-Ferriere. Using voronoi clusters to scope fbods from multiple sinks. Lecture, CENS Systems Seminar, October 2003.
[3] J. Elson, L. Girod, and D. Estrin. Fine–grained network time synchronization using reference broadcasts. In *OSDI*, pages 147–163, Boston, MA, December 2002.
[4] D. Estrin, W. Michener, G. Bonito, and Workshop Participants. Environmental Cyberinfrastructure Needs for Distributed Sensor Networks: A Report From an NSF Workshop. Workshop Report, Scripps Institute of Oceanography, La Jolla, CA, August 2003.
[5] A. Cerpa et. al. Habitat Monitoring: Application Driver for Wireless Communications Technology. In *2001 ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, San Jose, Costa Rica, April 2001.
[6] A. Cerpa et. al. SCALE: A tool for Simple Connectivity Assessment in Lossy Environments. *CENS TR 0021*, September 2003.
[7] A. Mainwaring et. al. Wireless Sensor Networks for Habitat Monitoring. In *WSNA*, number 1, 2002.
[8] H. Wang et. al. Target Classifi cation and Localization in Habitat Monitoring. In *In ICASSP 2003*, Hong Kong, China, April 2003.
[9] J. Elson et. al. EmStar: An Environment for Developing Wireless Embedded Systems Software. *CENS TR 0009*, March 2003.
[10] J.C. Chen et. al. Coherent Acoustic Array Processing and Localization on Wireless Sensor Networks. *Proc. of the IEEE*, 91(8), August 2003.
[11] P. Levis et. al. TOSSIM: Accurate and Scalable Simulations of Entire TinyOS Applications. In *Sensys*, 2003.
[12] R. T. Vaughan et. al. On Device Abstractions For Portable, Resuable Robot Code. In *In IEEE/RSJ IROS 2003*, Las Vegas, USA, October 2003.
[13] W. J. Kaiser et.al. Networked infomechanical systems (nims) for ambient intelligence. Technical Report CENS TR 0031, Center for Embedded Networked Sensing, UC, Los Angeles, December 2003.
[14] S. Ganeriwal, R. Kumar, and M. Srivastava. Timing Sync Protocol for Sensor Networks. In *Sensys*, 2003.
[15] L. Girod, V. Bychkovskiy, J. Elson, and D. Estrin. Locating tiny sensors in time and space: a case study. In *In ICCD 2002*, Freiburg, Germany, September 2002.
[16] J. Heidemann, F. Silva, and D. Estrin. Matching Data Dissemination Algorithms to Application Requirements. In *Sensys*, 2003.
[17] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *ASPLOS 2000*, Cambridge, USA, November 2000.
[18] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *In MobiCom 2000*, Boston, USA, August 2000.
[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Frans Kaashoek. The click modular router. *ACM TOCS*, 18(3):263–297, 2000.
[20] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the simos machine simulator to study complex computer systems. *ACM TOMACS Speical Issue on Computer Simulation*, 1997.
[21] H. Wang, D. Estrin, and L. Girod. Preprocessing in a Tiered Sensor Network for Habitat Monitoring. In *in EURASIP JASP Special Issue on Sensor Networks*, number 4, pages 392–401, 2003.
[22] W. Ye, J. Heidemann, and D. Estrin. An energy-effi cient MAC protocol for wireless sensor networks. In *Proceedings of IEEE INFOCOM*, 2002.
[23] Center for embedded networked sensing. www.cens.ucla.edu.
[24] Crossbow mica2. http://www.xbow.com.
[25] Gnu hurd. http://www.gnu.org/software/hurd/docs.html.
[26] James reserve. http://cens.ucla.edu/Research/Applications.
[27] Ns-2. http://www.isi.edu/nsnam/ns/ns-documentation.html.
[28] Parsec. http://pcl.cs.ucla.edu/projects/parsec/manual.
[29] Qnx. http://www.qnx.com.