# UCLA

**Title**

A System for Simulation, Emulation, and Deployment of Heterogeneous Sensor Networks

**Permalink**

https://escholarship.org/uc/item/5jz8f28r

**Authors**

Girod, Lewis
Stathopoulos, Thanos
Ramanathan, Nithya
et al.

**Publication Date**

2004-05-05

Peer reviewed

# A System for Simulation, Emulation, and Deployment of Heterogeneous Sensor Networks[*]

Lewis Girod, Thanos Stathopoulos, Nithya Ramanathan, Jeremy Elson,
Deborah Estrin, Eric Osterweil, and Tom Schoellhammer.
Center for Embedded Networked Sensing
University of California, Los Angeles

{girod,thanos,nithya,jelson,destrin,eoster,tschoell}@lecs.cs.ucla.edu

## ABSTRACT

Recently deployed Wireless Sensor Network systems (WSNs) are increasingly following *heterogeneous* designs, incorporating a mixture of elements with widely varying capabilities. The development and deployment of WSNs rides heavily on the availability of simulation, emulation, visualization and analysis support. In this work, we develop tools specifically to support *heterogeneous* systems, as well as to support the measurement and visualization of *operational* systems that is critical to addressing the inevitable problems that crop up in deployment. Our system differs from related systems in three key ways: in its ability to simulate and emulate *heterogeneous* systems in their entirety, in its extensive support for integration and interoperability between motes and microservers, and in its unified set of tools that capture, view, and analyze real time debugging information from simulations, emulations, and deployments.

## Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems

## General Terms

Design, Experimentation, Measurement

## Keywords

Sensor Networks, Real Code Simulation, EmStar, TinyOS

## 1. THE CASE FOR HETEROGENEOUS SYSTEMS

Recent controversy in the sensor network research community has questioned the selection of platforms for sensor network research: should the focus be on motes (8 bit microcontroller boards such as the Mica2) or microservers (32 bit systems such as the iPAQ and Stargate)?

As proponents of mote-based platforms, we argue that the hardware cost of a mote class device ten years hence will be in the \$1–\$5 range, [6] opening up new application domains that require small, low cost hardware. We also claim that these extremely low-cost platforms will have similar constraints to those of today's motes, and thus development of software for these scales of platforms is an important research direction. Some go on to suggest that new sensor network protocols will only be needed for relatively homogeneous networks of highly constrained devices, and that larger platforms will either run traditional IP applications, or run the same operating system as smaller nodes.

On the other hand, as proponents of larger platforms, we question whether system software for mote class platforms is really going to address all needs of future sensor networks. We question whether the low cost platform of ten years hence will necessarily be as constrained as today's mote (e.g. 4K of RAM): if a $1mm^2$ chip costs \$1 with 4K of RAM, what is the marginal cost of adding more RAM? Since cost is typically dominated more by production volume than functionality, the needs of high-volume applications will tend to drive the functionality of the low cost platforms. While some applications with minimal needs will push for the absolute lowest cost platform, high volume applications that need more functionality can just as easily drive down the price of more capable platforms. In either case, if production volume is the driving force toward lower prices, the first high-volume application (the "killer app") will need to be useful enough to support the higher initial engineering costs and risks – it can't just rely on low prices.

And so, as a community we have been converging on a model of heterogeneous systems. Given that the shape of future sensor network platforms is determined by a combination of future technological advances and future applications, making an accurate prediction seems particularly difficult. Gaining experience building systems *today* is probably the best path forward toward discovering the applications of the future. Today, heterogeneity is an important part of sensor network systems, and if history is a guide, it will be an important part of future systems as well:

- **Motes** and their descendants are the most scalable platform to support long lifetimes on small, single-use
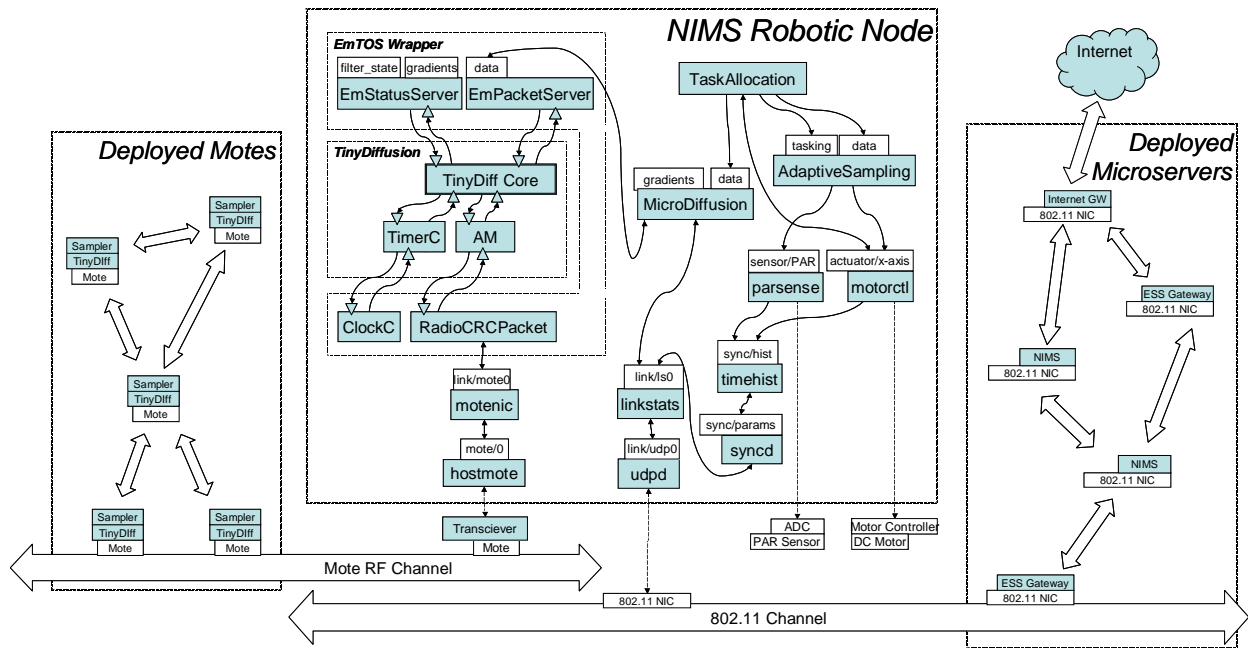
Figure 1: The NIMS/ESS deployment at the James Reserve is the target architecture in our discussion of Heterogeneous Systems. This diagram shows a complete picture of the NIMS/ESS system, composed of both motes and microservers. The box on the left labeled "Deployed Motes" shows a network of individual deployed microclimate sensor motes, each running a TinyOS stack including the TinyDiffusion routing/transport layer. The box on the right labeled "Deployed Microservers" shows a multihop network of deployed Stargate-class devices, a mixture of NIMS robotic nodes and ESS concentrators. The box in the middle details the software running on one of the NIMS nodes. Inside this box, we see both NesC/TinyOS code and EmStar code running. Different notation is used for the two frameworks: EmStar device interfaces are shown as a white box above a module, while NesC interfaces are shown as gray triangles. Arrows indicate client-server relationships. Note that the box labeled "TinyDiff Core" encapsulates 15 sub-modules that make up TinyDiffusion. The "EmTOS Wrapper" is a library that enables an entire NesC application to run as a single module in EmStar.

batteries and size-constrained energy harvesting. Because their memory architecture is designed for small footprints and there is no MMU, there is a need for an OS that is more oriented toward compile-time optimization and static verification, because memory protection between modules can't be enforced at run time.

- **Microservers** and their descendants are the most capable platforms that can be readily deployed. While they consume energy at a considerably higher rate than a mote, they are actually more efficient at many computational tasks, and they can support the larger memory footprint required for more complex tasks. Microservers are also more readily interfaced to high-bandwidth peripherals, such as high-rate ADCs and network interfaces. To leverage their capacity to tackle more complex tasks, microservers need an OS that supports standard features such as memory protection and dynamic configuration, but is also designed to address the problems special to sensor networks. While software and operating systems designed for motes can be run on microserver-class hardware, doing so will not leverage the hardware's capabilities—missing out on opportunities for increased robustness, flexibility, and performance, all of which are critical for deeply embedded systems and for which existing Internet software and protocols will not suffice.

- **Middle ground?** While the future remains unclear, today *energy* is the most critical parameter in building deployable embedded systems. Energy requirements are typically driven by peripherals and memory footprint, both of which vastly affect system capability. As more energy is required to support an application, the marginal cost of providing additional energy drops relative to the increase in capability it enables. For example, once a system transits from being strictly battery powered to requiring a solar panel, the benefits of an increased RAM footprint and a more sophisticated operating systems often justify the incremental cost of making the panel larger. Given today's technology, this leads to a bi-modal distribution of system capabilities among general purpose, platforms suitable for development. As the field matures, emerging applications will drive the development of new devices that are highly optimized for specific applications. While these devices will fall into this category, their high degree of optimization constrains their use as development platforms and leads to different tools.

Thus, there will always be a lowest cost platform and applications that require that level of functionality. There will also always be applications that require more functionality, and can justify the additional costs. [1] Moreover, many applications will require a combination of both in order to com-

bine densely deployed lower end sensors with more sparsely deployed higher end nodes. The system designs we pursue must account for heterogeneity as a first order property, so that the unique properties of each part of the system can be exposed and leveraged.

In summary, there is long-term value in developing specialized system software for *both* motes and microservers, that is designed to meet the challenges of Wireless Sensor Networks: the implementation of complex distributed systems that robustly accomplish a task in a complex and dynamic environment. We can apply this software today in many heterogeneous systems that are currently deployed or under development [10] [18] [17] [7] [16] [11] [2]. In these systems, the mote portion of the implementation is done in NesC/TinyOS [5], while the microserver portion is done in EmStar/Linux [3].

## 1.1 Heterogeneous Target Architecture

Figure 1 shows the type of heterogeneous system architecture we are targeting. The figure diagrams the architecture of the NIMS/ESS deployment in the James Reserve [7] [17]. NIMS (Networked Info-Mechanical System) is a Stargate-based robotic node suspended on a cable between two trees. Using motors, the node can move between the two trees, and raise and lower a sensor package vertically. This enables the NIMS node to completely explore a 2-D plane through a habitat, as well as act as a "data mule" that visits collections of deployed sensor nodes. ESS is a dynamically taskable data collection system for the deployed sensor nodes. In ESS the microservers use TinyDiffusion to act as robust concentrators for a larger collection of sensor motes.

This architecture illustrates many of the ideas of heterogeneous systems. Because the NIMS node must move itself using motors, its energy budget is large, supported by a fixed infrastructure of solar panels and batteries. This greatly reduces the importance of the energy cost of the NIMS node's computational hardware, enabling more capable systems and qualitatively different sensor types to be used (e.g., high end imagers). On the other hand, the deployed sensor nodes must be powered independently, and are tasked to collect slowly time-varying microclimate data. For these components, motes are the best choice of platform.

In the figure, the large dashed box represents the software running on the NIMS node Stargate platform. This software is running within the EmStar [3] framework, over Linux. The Stargate connects to several hardware peripherals, including an 802.11 NIC, a mote, and local sensors and motor controllers. Via 802.11 the NIMS node can reach a multi-hop network of other NIMS nodes, other microservers such as ESS concentrators, and eventually the Internet. Via the mote network, the NIMS node can participate in a dynamically varying network of motes, tasking them and drawing out data.

The key point to capture from this target architecture is that *both* the motes *and* the microservers implement complex networks and distributed systems. Furthermore, the software *within* both motes and microservers is also complex. To develop these kinds of systems we need tools that can adequately represent and test these systems in their entirety, in a variety of degrees of "reality", before moving on to the costly and time-consuming business of deployment.

## 2. TOOLS FOR HETEROGENEOUS SYSTEMS

Following early experiences deploying wireless sensor systems, several research groups have independently concluded that deployable sensor network systems can only be developed with the help of "real code" simulation tools, that run identical code in simulation and deployment. The primary reason for this seems to be that sensor network systems often translate poorly from simulation to deployment, as a result of the difficulty of modeling the impact of the environment on the system. "Real code" simulation tools provide the critical capability to *quickly* iterate between simulation, deployment, and ideally somewhere in between.

Both the TinyOS/TOSSIM [9] and EmStar/EmSim [3] environments are "real code" simulators. TOSSIM simulates a mote application implemented within the NesC/TinyOS framework, while EmSim simulates a microserver application implemented within the EmStar/Linux framework. The next two sections describe these simulators in more detail.

Note that "real code" simulations are not the only important type of simulation tool—for many types of algorithm development they are neither practical nor necessary. Although this paper does not consider these issues, a useful extension would integrate a "real code" simulator and channel model with scalable, high-level simulations.

## 2.1 Simulating Motes: TOSSIM

TOSSIM is a discrete-event simulator (i.e. based on a virtual clock) that implements the lowest layer of components in the TinyOS API, and runs multiple copies of the complete TinyOS stack and NesC application above those components. TOSSIM uses special hooks in the NesC compiler to support the ability to run multiple instances of a TinyOS application in a single process, by reserving multiple instances of each TinyOS component's state.

TOSSIM provides several RF channel models and sensor models. The original version of TOSSIM simulated a component interface just above the hardware, so that the TinyOS MAC layer itself was part of the simulated application. This has the advantage that more details of the MAC layer are preserved in the simulation. To address scaling problems, TOSSIM also supports "packet mode", in which a packet-level interface is simulated. The TinyViz visualizer can connect to TOSSIM and view debugging information emitted by the simulated motes.

Because the same NesC application code runs on real motes and in the TOSSIM environment, it is easy to iterate between a set of real motes and a TOSSIM simulation. TOSSIM also has the capability to inject traffic into a simulated mote network, in order to simulate tasking and other stimuli from external sources. However, it is not always convenient to feed TOSSIM with a taskload generated by a complex external system.

While TOSSIM has many desirable features, it was not designed to simulate our heterogeneous target architecture. First, TOSSIM requires that an identical code base run on every mote. While there may be ways around this (e.g. combining applications and selecting based on node ID), this can make iteration between "real code" and simulation more cumbersome. Second, TOSSIM does not readily simulate systems composed of motes with differing capabilities or hardware, such as the case in ESS where different motes are loaded with different sensors.

Third, while TOSSIM can be used to simulate the "Deployed Motes" box of our target architecture, there aren't any tools that enable that simulation to link up to the rest of the picture in a unified simulation environment. The closest tool to that purpose is the "Tython" extension to TOSSIM, which enables easy scripting of packet injection into the simulated mote network. While in principle this can be used to simulate the behavior of external networks and systems, encoding the complete behavior of the outside world into Tython may prove difficult.

## 2.2 Simulating Microservers: EmSim

EmSim provides a "real code" simulation capability for software written in the EmStar framework. In EmStar, systems are composed of many Linux processes, generally one process per separable service or module. EmStar services can be written in any language, so long as they communicate using the EmStar IPC mechanisms.

Unlike TOSSIM, EmSim can't assume as much about the uniformity of the application, because EmStar code can be written in a variety of languages. Thus, rather than compiling everything into one process, EmSim runs multiple simulated nodes as separate process trees. Each simulated node communicates with other simulated nodes through standard EmStar IPC channels that are name-mangled to give each simulated node its own private namespace. This multi-process approach entails some scaling limitations that are characterized in Section 5.2.

The lowest layer of services is provided by "simulation components" that provide separate device interfaces for each simulated node. For instance, the RF channel model module provides a packet-level "Link Device" interface for each simulated node, and transfers packets among the simulated nodes according to a channel model. EmSim currently includes simulation components to support sensors and the RF channel. The RF channel simulator can assume one of several "personalities", including the standard TinyOS MAC, S-MAC [14], and 802.11, and supports a variety of propagation models tuned for different hardware types. EmSim does not attempt to model the MAC layer in detail, apart from gross aspects such as CSMA vs. TDMA and collisions.

One of the key features of EmSim is its ability to support "emulation" mode, where centrally simulated nodes interact using *real* radio or sensor hardware embedded in a target environment. Emulation often reveals bugs that don't appear in simulation, from aspects of the channel that aren't well modeled, to peculiarities of the MAC layer that are not represented in a packet-level simulator.

Emulation mode is made possible in part by the fact that EmSim natively runs in "real time" rather than according to a virtual clock. The advantage of running in real time is that it enables interaction with external hardware or whole systems that also run in real time. The primary disadvantage is scalability: a large EmSim simulation may miss timing deadlines if the load is too high. Conversely, a lightly loaded system can't easily be sped up. An experimental addition to EmStar called "TimeWarp" addresses these problems by using a kernel module to directly scale the impact of POSIX timing calls such as `select()`, `sleep()` and `gettimeofday()`. As a side effect to time scaling, TimeWarp also enables simulation "pause".

While EmSim has many desirable features for simulating systems of microservers, it suffers from a similar problem to

TOSSIM in terms of the "big picture" from our target architecture. EmSim only simulates "real code" written within the EmStar framework, with no convenient way to link up to mote software written in NesC/TinyOS. Thus, it can simulate the "Microservers" box of our target architecture, but no tools exist that can integrate that simulation with a simulation of motes.

## 2.3 Simulating Heterogeneous Systems

The previous sections detailed the capabilities of TOSSIM and EmSim, and showed that both of these projects lacked the capacity to easily interoperate with the other in order to address heterogeneous systems. An effective heterogeneous simulation environment implies several key requirements:

- **"Real Code":** The capacity to readily move between simulation and real deployment. A large part of debugging is tied up in configuration files and other state describing the high-level structure of the system. In addition to running real application code, a fully integrated simulation environment would enable these configurations to be used unmodified.

- **Integrated World Model:** An important part of simulating heterogeneous systems is enabling all parts of the system to interact through the simulated world. This requires a single integrated world model, rather than a loose coupling of independent simulation environments and independent channel simulators.

- **Integrated Tools:** The utility of a simulation environment is closely tied to its ease of use. Ease of use is greatly improved with an integrated set of visualization and analysis tools, that works for all of the cases we have identified in our target architecture.

Although both TOSSIM and EmSim each satisfy these requirements within their respective domains, neither adequately addresses our heterogeneous target architecture in which Linux-based microservers share a world with TinyOS based motes. To bridge this gap, we extended EmSim's simulation infrastructure to support different types of platform within a single integrated environment.

As the first instantiation of this approach, we developed an extension to EmStar called "EmTOS" that enables mote-based NesC/TinyOS applications to run alongside EmStar applications within a single EmSim environment. Section 3 describes EmTOS in more detail. Section 4 describes extensions to EmSim and EmView, the EmStar visualizer. Section 5 presents a performance analysis of EmTOS, both with an example application and with comparative metrics vs. native TinyOS.

While this paper only discusses how TinyOS applications can be integrated into EmSim, we believe that the same approach can be applied to other types of application environment and hardware platform as they arise. Some of these efforts are described in Section 7 on future work.

## 3. EMTOS

EmTOS is an extension to EmStar that enables an entire NesC/TinyOS application to run as a single module in an EmStar system. As shown in Figure 2, EmTOS works by providing a wrapper library and a set of stub components that enable existing NesC/TinyOS applications to operate
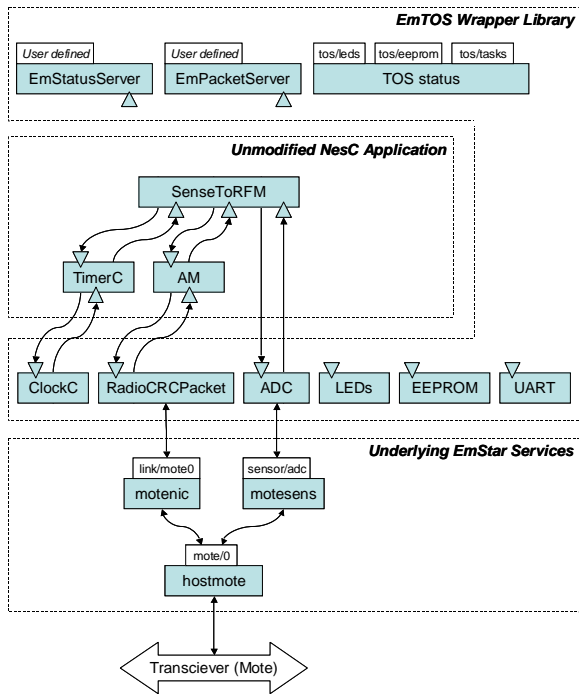
**Figure 2: An unmodified NesC Application encapsulated within the EmTOS wrapper library. EmStar device interfaces are denoted by white boxes containing the name of the device. TinyOS component interfaces are denoted by gray triangles. EmPacketServer and EmStatusServer are TinyOS components that enable a NesC application to *export* one or more EmStar device interfaces, enabling other EmStar modules to connect to it. The EmTOS wrapper itself exports several EmStar devices representing the internal state of EmTOS, including the state of the task queue, the LEDs, and the EEPROM.**

using existing services provided by the EmStar system. Effectively, this enables NesC/TinyOS applications to compile and run on microservers running EmStar, and therefore enables EmSim to simulate them.

The EmTOS wrapper library works in a similar way to TOSSIM itself. Like TOSSIM, EmTOS is used by building a NesC application for a special platform target. Whereas the `pc` platform is used to build TOSSIM, the `emstar` platform is used to build an EmTOS module. The `emstar` platform implements all of the lowest layer component interfaces, that then link to EmStar services and interfaces. Referring to Figure 2, the low layer components defined by the EmTOS platform are shown in the lower tier under the "NesC Application" box: ClockC, RadioCRCPacket, ADC, etc. The NesC application links to these in the normal way, and the EmTOS library implements the required adaptation layer to connect to existing EmStar services such as Link Devices. Note that this adaptation layer is configurable. While Figure 2 shows RadioCRCPacket connecting to the MoteNIC[1]

---

[1] The MoteNIC is an EmStar interface that proxies packets over serial to a mote for transmission. The term *MoteNIC* generally refers either to the microserver-side software, or to the whole system (mote + microserver software). Transceiver is the name of the packet proxy application that

Link Device, it could just as easily be connected to a Link Device running over 802.11.

EmTOS has a dual significance: first, as a mechanism enabling microservers to readily participate in mote networks, and second as a mechanism that enables certain forms of heterogeneous simulation within EmSim.

## 3.1 Microserver Participation in Mote Nets

There are many examples of systems in which microservers must participate in a mote network: the NIMS node and ESS concentrator are two such examples. While in some cases the interactions are simple, in others they are complex enough that the interfacing effort incurs a significant cost. There are several ways to address this problem, and each may be the ideal choice, depending on the particular details of the application:

1. **Serial Gateway**: Devise a serial protocol that encodes the message traffic to and from the mote as needed, and write software on the microserver side to process that traffic from a microserver-side program such as SerialForwarder, or using the EmStar Host-Mote protocol.

2. **Re-implement or Port**: Port the NesC application that runs on the mote to run on the microserver, for example using SerialForwarder to send and receive radio packets, and link it up with a proxy mote application such as GenericBase.

3. **Use EmTOS**: Compile the NesC application for the `emstar` platform, generating a single EmStar module that implements the NesC application and binds to existing EmStar services. Then, extend the NesC code to provide new interfaces for debugging and for interfacing to other EmStar components. Then run the new module, along with existing EmStar components MoteNIC, HostMote, and others.

If the application requires precise timing or interrupt handling, option (1) may be the only possibility. Because of the additional latency of the serial port and the I/O latency on standard Linux kernels, interrupts can only be handled with precise timing on the mote itself (see Section 5 for a quantitative discussion of timing issues). The drawback of option (1) is that it can be a more inconvenient implementation, because it requires exposing specific new functionality through a serial protocol. EmStar supports this mode of integration via the HostMote protocol and HostMoteM TinyOS component, which provide a robust solution to this problem, described in more detail in Section 3.3.

If precise timing and access to specific hardware on the mote is not a factor, then both options (2) and (3) are possibilities. Option (2) has the drawback that it requires the maintenance of independent implementations. While multiple implementations allows for more flexibility, they can easily get out of sync. However, for some cases, for instance if the code must run on Windows, this may be the only viable alternative.

If EmStar can run on the microserver, then option (3) is a very convenient way to integrate motes and microservers. We have already discussed how the EmTOS wrapper enables a NesC application to run on the microserver. What makes

---

runs on the mote as part of the MoteNIC.

this integration path particularly smooth is the ability to extend the NesC application, both to use services provided by EmStar that don't exist in TinyOS, and to *provide* new EmStar services on the microserver. This means that the developer can choose exactly where to draw the boundary, deciding how much of the work should be done in NesC and how much should be done in new or existing EmStar components and application code.

The new release of ESS deployed at the James Reserve uses EmTOS to implement the concentrator in exactly this way. Rather than try to interface to a mote through a special serial protocol, ESS runs the EssSink NesC application inside EmTOS, sending traffic to the radio via the EmStar MoteNIC service and its mote side packet proxy, Transceiver (see Section 3.3 for more details). In addition, EssSink is extended with numerous helpful debugging outputs describing the current state of its queues and neighbor link estimators. EssSink also provides server interfaces that other EmStar modules can use to task the network and receive data back.

One might ask, given EmTOS, why not write the whole ESS application in NesC? While in principle much of the system *could* be implemented in NesC alone, leveraging the larger EmStar framework brings many advantages that are consistent with the less constrained context of microserver applications:

- **Support for Dynamics:** Rather than requiring the dependency graph to be statically determined at build time, EmStar allows clients to dynamically connect to servers, and allows services to be dynamically created at run time.

- **Support for Concurrency:** EmStar modules can use threads and multiple processes to simplify implementations that don't fit well into an event-based form, such as long-lived computation tasks.

- **Resiliency and Robustness:** EmStar leverages the memory protection features of 32 bit processors by running each service as a separate process. Resiliency against cascading failures is also enabled by libraries that automatically recover from failed connections and by soft state protocols between modules.

- **Diverse Language Support:** EmStar fully supports any language that can link to C libraries, and to a lesser extent supports any language that can make POSIX system calls. Currently most modules are written in C and C++, and a few are now being developed in Java. In addition, many EmStar functions can be accessed from shell scripts and other scripting tools.

## 3.2  Heterogeneous Simulation using EmTOS

In the previous section we presented ESS as an example of a system that uses EmTOS in a deployment to enable mote–microserver integration. Since the concentrator is a microserver running EmStar, EmSim can naturally be used, for example enabling simulation of a network of concentrators. However, that simulation would leave out an important component: the mote networks that sense and feed data to the ESS concentrators via Tiny Diffusion.

Simulation of "stand-alone" motes can also be achieved using EmTOS, in the same set of circumstances where EmTOS can be used for deployment (i.e. cases in which the target code does not require access to precise timing or interrupts). This only requires compiling the stand-alone mote application for the `emstar` platform, and setting up configuration files that describe each mote's hardware and software configuration.

For each node, whether mote or microserver, the simulation config file defines the hardware configuration by specifying the location and orientation of the node, along with which radio and sensor channels the node can access. The radio and sensor channel models use the location information to generate the appropriate channel behavior. The simulation config also defines which software should run on the node. In the case of a simulated microserver, this is usually done by specifying a custom EmRun configuration file that expresses the dependency graph of EmStar services and application components. In the case of a simulated mote, a generic EmRun configuration can often be used, specifying the name of the EmTOS build as a command line argument.

This mechanism enables heterogeneous simulations, whenever precise timing is not required and all peripherals used by the application are supported by the simulation. In addition to pure simulation, there are three other modes of operation that can be useful in testing and in cross-validating the simulation. These modes are described in Section 4.1. Further discussion of timing issues and possible workarounds can be found in Section 5.

## 3.3  The HostMote Protocol

While EmTOS can be used to run NesC code without having a mote involved at all (e.g. connecting to an 802.11 radio), the common case thus far has been to proxy EmTOS activations back and forth across the serial port to a mote. For this we use the HostMote serial protocol, and a set of applications that run over it.

The HostMote protocol is a simple framed wire protocol with variable length packets, a type field, and a CRC. On both the mote and microserver sides, the HostMote protocol is demultiplexed to clients according to type field. On the mote side, the HostMoteM module implements the serial protocol and enables client applications to register for particular types. On the microserver side, the `hostmoted` daemon performs a similar function, enabling independent client applications to register for specific types.

Currently, HostMote is used to support integration of motes and microservers in a number of ways:

- **Packet Proxy:** The MoteNIC service and Transceiver mote application proxies packets from the microserver to the RF channel and back over HostMote, similar to GenericBase. It reports ACKs from the MAC layer, along with a variety of statistics about the channel.

- **Debug:** The debug logger enables NesC code running on a real mote to efficiently report debug messages via serial to a microserver. The debug facility emits typed messages of variable length with two bytes of header overhead per message, and can aggregate multiple debug messages into a single HostMote message. On the microserver side, an extensible logging facility parses the messages and either prints them or exposes them as Status Devices.

- **Configuration:** The HostMote configuration protocol enables an extensible set of attributes to be configured, and enables status information to be returned,
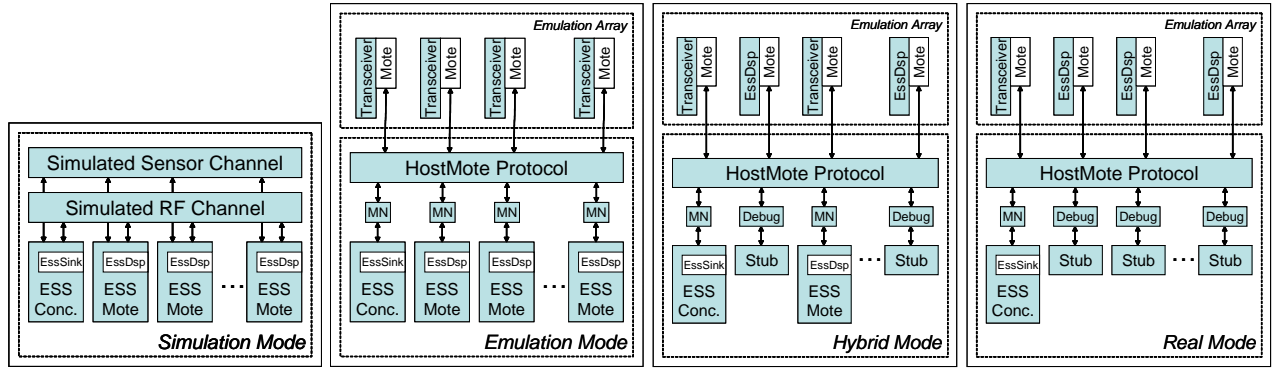
**Figure 3: The four modes of operation for our heterogeneous simulation, showing ESS with one concentrator and *N* motes as an example. In a deployed system, the concentrator would run on an iPAQ and the motes would run on Mica2 hardware. The small "MN" boxes represent the microserver side of the MoteNIC, and the "Debug" boxes represent an service that logs and proxies debugging information collected over serial.**

analogous to the UNIX utility `ifconfig`. The returned status structure includes packet statistics (RX, TX, errors, serial CRC errors), and the current configuration (power setting, address, etc.)

Debug mode is used to collect debugging information from real motes that are participating in networks with emulated motes (described further in Section 4.1). This enables a unified set of visualization and analysis tools to operate over hybrid networks (described further in Section 4.2.2).

The MoteNIC is particularly critical to EmStar mote–microserver integration, and has been extensively debugged and hardened. It is immune to a wide variety of failures, from hot-swapping the mote during operation to many failure modes that might cause the mote to reset or lose power. The HostMote software on the microserver side retains the current desired configuration of the mote (e.g. power level, address, etc.) and periodically requests a replay of the current configuration. In the event that the reported configuration does not match the requested configuration, it will automatically reconfigure the mote.

An interesting an unexpected use of this feature occurred recently at the ESS deployment, when there seemed to be a lot of collisions in the middle of the network but there was no way to measure them. A quick solution was to place a battery-powered mote running Transceiver in the area in question. After letting the Transceiver mote collect statistics, we hot-swapped it for the Transceiver mote connected to the microserver, logged in and retrieved the statistics. While this was not its intended purpose, this freedom to hot-swap components of the system was only possible because most of the disconnection, reconnection, and power loss cases have been handled.

# 4. UNIFIED VISUALIZATION AND ANALYSIS

One of the key design goals for EmTOS and EmStar is to build a unified set of visualization and analysis tools that can operate on any mode of operation of the system. These modes include Pure Simulation, Emulation, and Deployment. In the context of EmTOS, there are several new Emulation modes, shown in Figure 3, that are specific to the case of motes in an Emulation Array.

## 4.1 EmTOS Emulation Modes

One of the novel features of EmStar has been its ability to run in "Emulation Mode", in which a simulation on a centralized server uses real deployed motes as radios, in place of a computed radio channel model [3]. Using EmTOS, the same concept applies: just as some nodes in a simulation can be simulated "stand-alone" motes while others are microservers, the exact same configuration can be run in emulation mode on an "emulation array" by a simple change on the command line that starts the simulation.

In addition to working in emulation mode, EmTOS extends this idea to support two new modes of operation: "Real Mode" and "Hybrid Mode". Figure 3 diagrams these new modes, in addition to "Emulation" and pure simulation.

### 4.1.1 Emulation Mode

In the original "Emulation Mode", all of the physical motes in the array are programmed with the Transceiver application, enabling packet proxy functionality via the Host-Mote protocol described in Section 3.3. On the server side, each node runs an instance of MoteNIC (labeled "MN" in the diagrams) to provide the EmStar Link Device interfaces.

Then, the different emulated nodes run different software on the centralized server. The first node in the diagram represents the microserver in this system, running the "ESS Concentrator" software. The TinyDiffusion protocol is implemented in NesC by the EssSink application, and run inside an EmTOS wrapper. EssSink communicates with other modules in the EmStar world to implement the rest of the "Concentrator" functionality, such as issuing tasking requests and processing the results.

The remainder of the nodes are emulating motes in the ESS system. They run the "ESS Mote" software, which uses EmTOS to emulate a NesC application called "EssDsp" that responds to tasking requests and reports data back via TinyDiffusion.

### 4.1.2 Real Mode

In "Real Mode", all of the motes in the ESS system are run natively rather than emulated, and use the emulation array as a serial debugging backchannel, while the ESS concentrator software still runs as before on the simulation machine. Thus, while the mote corresponding to the ESS concentrator

is still programmed with Transceiver, all of the other motes in the array are programmed directly with the EssDsp application.

To support collection of data for debugging, visualization and analysis, special "Stub" software is run on the simulation machine, corresponding to each natively running mote. This stub logs information from the Debug type of the Host-Mote protocol, and proxies that data into Status Devices. These Status Devices provide an interface that matches the interface provided by EssDsp when running in emulation mode, enabling the same visualization and analysis tools to operate in both cases.

### 4.1.3 Hybrid Mode

"Hybrid Mode" is a mixture of "Real" and "Emulated". In "Hybrid Mode", selected motes in the system are run natively rather than emulated. As shown in the diagram, some of the motes are programmed with EssDsp and correspond to "Stub" software on the simulation machine, while the rest are programmed with Transceiver and correspond to an EmTOS emulation of EssDsp.

## 4.2 Unified Support for Data Gathering

In order to visualize or analyze data, there needs to be a framework for capturing it. If this mechanism is uniform across the different modes of operation, then the visualization and analysis tools will work uniformly.

There have been similar efforts in the TinyOS community, notably Message Interface Generator (MIG), which forms the basis of data interpretation for TinyViz/Tython [9] and MoteLAB [13]. MIG translates a TOS message into a Java class that can parse it. The Java class can then be integrated into TinyViz and MoteLAB. EmStar instead relies on both sides sharing a C structure definition. This has the advantage of being divorced from TOS messages, which are very TinyOS specific, and are less useful outside the world of motes.

EmStar defines two standards through which data may be gathered. The first is the EmProxy protocol, which is layered over UDP. The protocol issues requests to one or more nodes or simulation servers, and receives responses back on a best-effort basis, but with low latency. The request encodes a set of EmStar device files to watch, and each response represents one or more updates from the devices being monitored at a given node. The EmProxy server is an EmStar module that speaks this protocol. EmProxy runs on both real and simulated nodes, enabling the same visualization software to work in both cases. By conforming to the EmProxy protocol, other server implementations can also be visible to the EmStar analysis tools.

The second standard is the device file hierarchy itself. If EmProxy is already running on a node or simulation server, then by creating the appropriate set of device files, those devices and the data they expose will become visible to the EmStar analysis tools. The standard meaning of particular files in the device file hierarchy is generally determined by the software that exposes them, and any standards that are in place. For example, the standard format for neighbor discovery or link quality outputs is defined by the `neighbor_t` structure in the `link/neighbor.h` header file. Thus, the easiest way to expose state about neighbor lists or link quality to the visualizer is to expose it as a Status Device that exports data in that format. Once exposed, the Link/Neighbors module of the visualizer can pick it up

```
bash-2.05b$ cat tinydiff/filter_state
NeighborStore Filter State
--------------------------
Node ID  Qual InL  OutL Miss Seq  Cnt  Low  EWMA
-------- ---- ---- ---- ---- ---- ---- ---- ----
      2    0    0   50    5   33   30   33   10
      5    0    0   0*    5   33   23   33    7
      3    0   50   0*    3   31   23   31   21
      6    0    0   0*    3   77   54   77   12
      4    1   67   0*    0   0*   0*   0*   47
--------------------------
Calc Index:      0
Lambda:          20
Low Water Mark:  20
Hi Water Mark:   40
```

**Figure 4: Human readable "Filter State" output from Tiny Diffusion's link estimator. The device can also be "watched" using the `echocat` program, and a new report will be generated whenever the state changes.**

and correctly display it.

These two standards provide two different opportunities to hook into the visualizer. We will now detail several examples of how elements of our heterogeneous simulation link up to the analysis tools.

### 4.2.1 Gathering Data from EmTOS Code

When the EmTOS wrappers are used to integrate a NesC application into an EmStar system, the most convenient method of integrating to the EmStar analysis tools is to expose EmStar Status Devices from the NesC application. Exposing Status Devices essentially hooks into the device file hierarchy so that existing analysis tools can access the new application's state via EmProxy.

The EmStatusServer and EmPacketServer modules implement NesC interfaces that can create EmStar Status Devices. The interface is similar to the interface used in EmStar [3]. To create a Status Device, the service first specifies the name of the device in an initialization call, and then implements several Events that are triggered in response to a request from a client.

Each of these Events is responsible for reporting the state that the module is exposing. The "Printable" Event reports the state in a human-readable format, for instance a table of filter values such as the one shown in Figure 4. While this is usually not critical to supporting analysis tools, it is very convenient for interactive debugging.

The "Binary" Event typically reports the same state as a vector of binary structures. As a rule, the visualization and analysis tools will interpret this binary state, and it usually must conform to a standard format in order to be parsed correctly. Existing standard formats are defined as C structures in header files, e.g. Neighbor and Link Quality (`neighbor_t`), and Routing (`routing_entry_t`).

Note that while the operation of these Events can be costly in terms of memory and code space, both of these Events only apply to NesC code running inside EmTOS (i.e. on a microserver). When this code is compiled for mote platforms such as `mica`, the EmStatusServer module turns into a stub module that won't call those Events. In addition, the implementations of the Events call a function `bufprintf()` that turns into a NOP when not compiled for platform `emstar`.

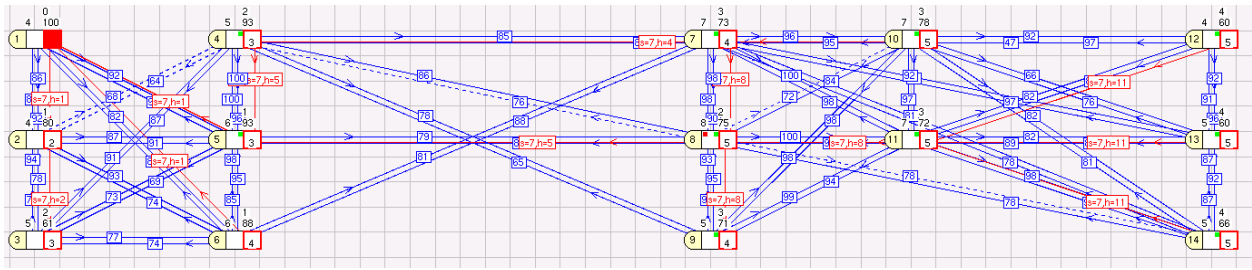In addition to the Events that report the state when re-

Figure 5: EmView screenshot of ESS running in Emulation Mode. In this case, Node 1 is the sink, and all other nodes are EmTOS nodes running the EssDsp application. Links with small boxes represent current link quality estimates from the Link/Neighbors module. Lighter colored links with larger boxes represent current gradient state from the Routing/Trees module. The positions of the nodes are approximately correct, with 1 meter grid lines.

quested, the EmStatusServer interface supports a Command that triggers notification on the Status Device. This means that a module that exports debugging state can add a Notify Command that gets called whenever the state has changed in a significant way, in turn causing the clients to request the new state.

### 4.2.2 Gathering Data over Serial from Real Motes

Unlike the case of EmTOS, gathering data from real motes must be done carefully to avoid consuming too many resources. The standard binary structures used in EmStar tend to be larger than necessary, in part to cover more cases, and in part because memory is not a scarce resource. To address this problem, the interface to EmStatusServer also includes a "CompressedBinary" Event, that should be implemented to produce a more compact version.

When compiled for platforms other than `emstar`, the EmStatusServer ignores the "Printable" and "Binary" Events, and instead issues the "CompressedBinary" Event. The CompressedBinary Event provides a buffer to fill, and the client should fill it with a compact binary output, if the buffer has enough space. The EmStatusServer component integrates with the HostMote component to forward the data back over the serial port, as Debug type data.

On the microserver side, the `hm_logger` module receives the Debug data from HostMote and proxies it into EmStar Status Devices according to the type of each Debug message. This logger is modular, making it is easy to add a new handler for a new Debug type. Since `hm_logger` understands how to convert data from the compressed Debug formats to the standard EmStar formats, the Status Devices it exposes can be picked up directly by the analysis tools via EmProxy. This `hm_logger` infrastructure is part of the "Stub" shown in the diagrams of Hybrid Mode and Real Mode.

### 4.2.3 Gathering Data from Deployed Nodes

When systems are deployed in the wild, there is usually no debugging backchannel. Without a backchannel, all debugging data must be forwarded back multiple hops over the wireless network, and collected for analysis or visualization. In these cases, it is critical to ensure that the data being collected does not interfere with the operation of the system itself.

Rather than try to invent a generalized way to retrieve this kind of data, we take the approach of providing a way to gateway the information retrieved from the network to a visualization and analysis infrastructure. For example, a program very similar to the `hm_logger` program could process messages in some compressed form and translate them into the standard EmStar formats, sending them back via the EmProxy protocol.

Our initial approach to this has been to link up to EmProxy the same way that `hm_logger` does, that is, by creating Status Devices that EmProxy itself can see. A better long-term approach, for future work, might be to implement a gateway that speaks the EmProxy protocol directly.

## 4.3 Visualization and Analysis Tools

Layered above our unified data gathering mechanisms, we use EmStar's visualization and analysis tool, EmView. To support ESS, we added a number of extensions, including support for display and analysis of routing trees, and support for heterogeneous networks. Figure 5 is a screenshot of EmView, showing an emulated run of ESS.

EmView is an extensible visualization engine, written in C using the GTK toolkit. EmView presents a plugin interface that enables new modules to be written to visualize new kinds of state. For every new application or service, an associated visualization plugin can be written to represent that application's state. The details of acquiring the data and drawing the nodes and decorations are handled by the EmView core, leaving it up to the plugin to parse the C structures that arrive from the deployed systems or simulations, and to represent that data back to the core in an abstract form as links or data values.

The original version of EmView was configured from the GUI to show or hide various decorations and attributes of the nodes in the viewer. We extended EmView to instead retrieve the configuration for each node from the node itself, along with all the other data about that node. This approach has numerous advantages:

- The user operating the GUI does not need to understand the details of the software running on the nodes.

- Complex per-node configuration can be represented more readily, by encoding it into the config files that "run" the node.

- The GUI can more readily adapt to different node configurations. For instance, and EmTOS node has attributes such as the state of the LEDs and the EEPROM, that a non-EmTOS node does not.
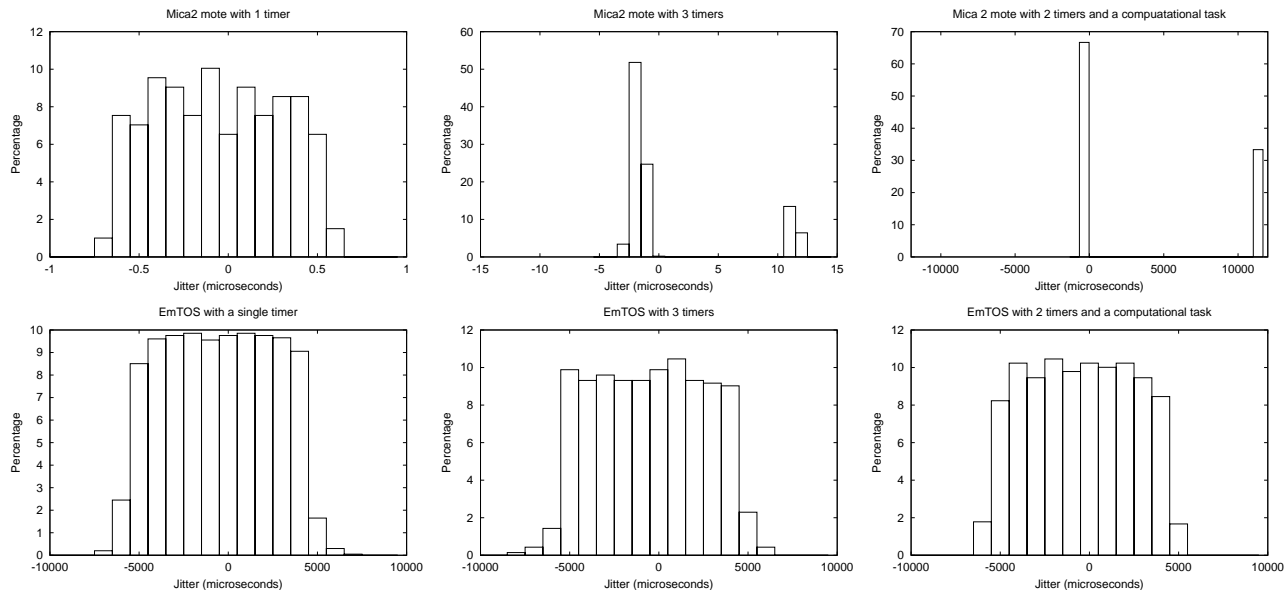
**Figure 6: Comparative analysis of timer jitter, between EmTOS and a Mica2. Each graph shows the distribution of timer values, centered at the median value. The top row are results from a Mica2, and the bottom row are the results from EmTOS.**

EmView is also a logical place to put analysis tools, since EmView is already gathering all the necessary data in one place. We extended our new "Routing/Tree" EmView module to include some tree analysis code that can output data to a log file for later analysis. Other features, such as disabling the EmView GUI and increased logging / replay capabilities, are left to future work.

## 5. PERFORMANCE ANALYSIS OF EMTOS

In this Section, we will analyze the performance of EmTOS. Previously, we have discussed the ways in which the performance of EmTOS emulation differs from the performance of real motes running the same software. Determining whether these differences matter is a prerequisite for drawing conclusions about the results of tests in emulation, and for deciding whether or not to use EmTOS at all.

To measure the effect of EmTOS on performance, we perform two experiments. The first experiment measures timer jitter for three different NesC programs, taking measurements on both a Mica2 and on EmTOS. Then, the same tests are run in an EmTOS simulation to measure the effect of scaling on timing jitter. The second experiment measures jitter and latency in packet transmission through the MoteNIC, and compares those results to packet transmissions for native code on a Mica2.

After showing these basic performance metrics, we present some measurements from a case study of ESS that compare results from different emulation modes.

### 5.1 Comparative Timer Jitter in EmTOS

For our jitter measurements, we wrote three NesC applications, that are summarized in the table below. In each case, the 200ms timer records a timestamp using SysTime and schedules it to be sent back over the serial port.

| Case 1 | One timer, set for 200ms |
| Case 2 | Three timers: 100ms, 200ms, 500ms |
| Case 3 | One 200ms timer, and a 120ms timer that runs a 20ms compute task |

We selected these tasks partly to demonstrate that, while TinyOS timers on the Mica2 can be quite accurate, they can also have orders of magnitude variation, depending on what else is going on in the system. This means that in a truly modular design (i.e. one in which modules can be developed independently), making assumptions about low timer jitter may not be possible.

The results of these tests are shown in Figure 6. The first thing to observe is that the EmTOS graphs (the bottom row) are all uniformly distributed with 10ms of jitter. This is caused by the Linux 10ms jiffy clock, which defines the minimum granularity with which a process can wake for a timer. Newer kernels support faster jiffy timers, and this data suggests that a faster timer might improve EmTOS timer performance.

The Mica2 experiments show a much wider range of performance characteristics. In the single timer case, the timer is extremely precise, hitting the target time as precisely as we could measure. However, few applications other than "Blink" run only one timer. In the three timer case, we see more variation, with occasional fires late by $10\mu s$. Based on proportion, these late fires are most likely the result of the 500ms timer being posted ahead of the 200ms timer, incurring the latency of posting and running an empty task. In the compute task case, when the timer is posted behind the lengthy compute task, it fires a full 10ms late, giving it similar jitter bounds to EmTOS. While posting long tasks might cause many other problems, the data demonstrates the possibility that real applications that post numerous tasks might see timer performance approaching EmTOS's jitter bounds.
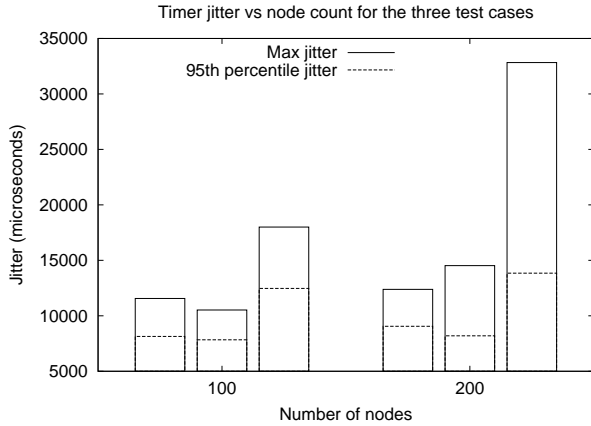
Figure 7: Timer jitter for simulations of 100 and 200 EmTOS nodes. The three bars correspond to our three applications, i.e. starting from the left: one timer, three timers, and compute task.
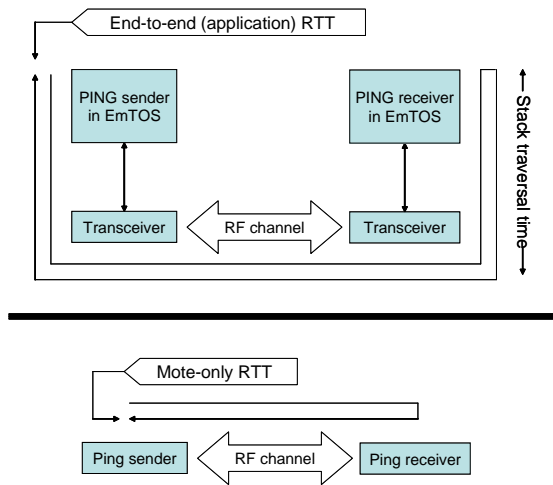


Figure 8: Diagram of two "Ping" experiments, one running a "Ping" application in an EmTOS emulation, the other running "Ping" natively on Mica2.

## 5.2 Timer Jitter Scaling in EmSim

Given our performance results on jitter, the next question to address is how that jitter is affected by scaling in a simulation. Figure 7 shows the maximum and $95^{th}$ percentile jitter values for 100 and 200 node simulations running our three test applications. The choice of the $95^{th}$ percentile was based on our determination that the jitter is uniformly distributed with outliers; thus 95% of the time the jitter is uniformly distributed within the smaller range.

The experiments were run on a 2.7GHz dual Xeon with 512MB RAM. The results show that for lightly-loaded simulations of up to 200 nodes, there is not a noticeable decrease in performance. Both the one and three timer cases showed a slight increase in the maximum jitter but the $95^{th}$ percentile was not significantly changed. However, for the compute task case the performance clearly starts to get worse in the 200 nodes case.

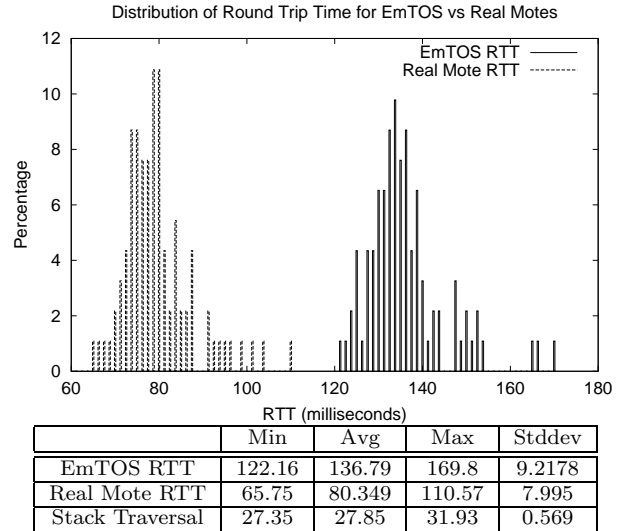Currently, there are some hard-coded limits in EmSim



Figure 9: Distribution and statistics of latency (in ms) for the EmTOS RTT and Real Mote RTT. Statistics are also provided for the EmTOS stack traversal time, i.e. the time required for a packet to be sent from Transceiver to EmTOS and back.

that prevent simulations larger than 200 nodes. These limits are not fundamentally difficult to fix, but there has been little motivation to address them because most of the prior use of EmSim has involved smaller numbers of nodes. While extending the limit is not difficult, we anticipate that other scaling issues may arise, such as increased scheduler latency due to large numbers of processes. Most likely, scaling to more complex simulations, and to more than 500 nodes, will require splitting the load onto several machines and linking their simulated channel models via a fast network.

## 5.3 Comparative Packet Latency in EmTOS

Along with timer performance, the other critical element of a typical EmTOS application is the performance characteristics of radio packet events. Relative to real motes, what kind of latency and jitter can be expected using EmTOS?

To answer this question, we performed an experiment in which we timestamped two independent "Ping" experiments, one running between two real motes, and one running between two EmTOS-based emulated motes. We then took timestamps at different points in the path of the packet. The experimental setup is shown in Figure 8. For these experiments, the RF packets were 47 bytes, and the serial port was running at 57,600 baud. The HostMote protocol added an overhead of 8 bytes per packet to the serial transfer. In the case of Ping running natively on motes, SysTime is used to capture timestamps at the sender and receiver. In the case of EmTOS, SysTime maps to `gettimeofday()`, and timestamps are captured at the EmTOS sender and receiver, as well as within Transceiver.

The results of the experiment are shown in Figure 9. The two distributions in the top graph are the distribution of RTT for real motes and for EmTOS. As might be expected, EmTOS shows and additional 60ms of latency. Interestingly, the two distributions look almost identical, suggesting that very little additional jitter is added to the timing
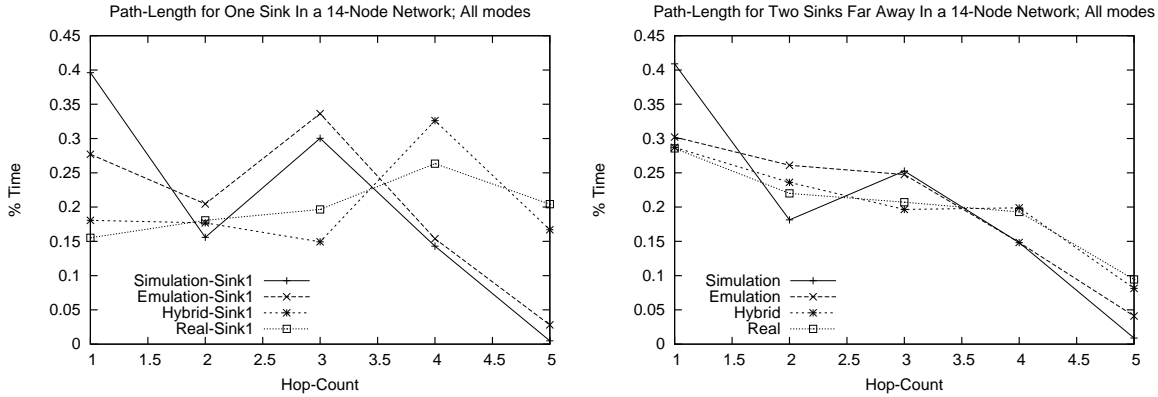
**Figure 10: Distributions of path length over 14 nodes and 90 minute runs. The graph on the left shows the distribution of hopcounts for runs with one sink at node 1 (refer to Figure 5 for the topology). The graph on the right shows the distribution for runs with one sink at node 1 and a second sink at node 14. In each case, the system was run for 90 minutes in each of the four modes: Simulation, Emulation, Hybrid, and Real.**

of the packet. This makes sense considering the scale of the distribution, and considering our previous results that showed that even in simulation environments with hundreds of nodes the jitter was still dominated by the jiffy clock.

As a sanity check, the table also shows that the difference between the EmTOS RTT minus twice the stack traversal time is almost exactly equal to the real motes RTT.

While these tests address a single node, they do not tell us for certain how a whole Emulation Array might behave. If most of the additional jitter seen by an emulation server is still under 10ms, then it shouldn't be a problem, given that the distribution of packet latencies on native motes is about 30ms. However, we will need to conduct additional experiments to verify this for certain.

### 5.4 Case Study: ESS

Our final results are culled from some experiments we conducted in our recent work developing ESS. The purpose of these experiments is to show some results drawn from a real application that is currently under development. We ran several experiments using our ceiling-mounted testbed of 14 Mica2's with sensor boards. The testbed spans an area about 50m by 10m, with a topology as shown in Figure 5.

In these tests, we test two sink configurations. The single sink case is run with a sink at node 1, in the upper-left corner of the grid. The two-sink case places one sink at node 1, and another at node 14, on the other side of the network.

For each configuration, we run ESS for 90 minutes, in one of our four modes (Simulation, Emulation, Hybrid, and Real). Using our visualization and analysis framework, we collect data about each node once per second. For each node, we record which sink it has currently selected as "best", and the current number of hops to that sink, as computed according to the currently reported gradient state.

The graphs of the results are shown in Figure 10. There are a few notable points about the graphs. First, it appears that the overall path length performance is better for simulation and emulation, in that there are fewer paths with many hops. We suspect that this is caused by the increased latency seen in simulation and emulation modes, that slows down the diffusion interest flood.

This version of TinyDiffusion schedules packets with a 100ms minimum delay, and adds an additional 200ms of jitter. It also defines its gradient based on the first interest packet it receives, subject to a minimum link quality, but without regard to hop count or variations in link quality. It is possible that these parameters rely on separation between each round of flooding, but don't provide enough separation to work effectively.

The second point to observe is that two sinks has less of an impact on hopcount than one might have imagined. While there is a marginal improvement, it is not the 50% gain that we had hoped for. Using our tools, we observed several phenomena that might cause this problem. First, we observed that limited neighbor table space often causes paths that are much longer than what appears to be needed. This affects both the single sink case and the dual sink case, because a node can be close to a sink but still have a long path to it, and thus have no short path to either sink. Second, we observed some glitches in the hop count estimator implemented by ESS that may be introducing additional problems.

While we may have identified several problems, fixing ESS is not the subject of this paper. Rather, we use ESS as an example to show how the tools help in the identification, diagnosis and analysis of these kinds of problems.

## 6. RELATED WORK

Throughout the text, we have referred to the work most related to this paper. In this section we will discuss a few additional references that we didn't mention in the text.

The *ns-2* network simulator [19] has been used for many years for simulations of networks, mainly in the context of the Internet. While it has elements of the "real code" approach, maintaining real code in *ns-2* is not as transparent as it is in TOSSIM or EmSim. Many implementations of network protocols use hard-to-maintain `#ifdef`'s to support NS and link to its event model.

SensorSim [12] is a simulation framework for sensor networks that has many features in common with EmSim, including the capability to run Hybrid simulations with real nodes alongside simulated nodes. SensorSim has more well developed sensor models and power modeling capabilities. However, it is heavily integrated with the SensorWare middleware layer, and does not integrate easily to simulate sys-

tems running TinyOS or EmStar.

`atemu` [15] is a hardware-level simulator for motes. While this would be helpful for debugging problems that required very precise timing, it does not address the problem of heterogeneous systems.

# 7. CONCLUSIONS AND FUTURE WORK

In this paper, we demonstrated a new capability to simulate a system of motes and microservers in its entirety, in a variety of emulation modes. We showed how EmTOS allows a microserver to participate in a mote network, and we quantified some limits of its application. We also identified areas of future work:

- **Generality:** We plan to provide support for Click [8] and SNACK [4], as well as a modified version of EmTOS to support SOS-based[2] mote applications.

- **Scalability:** We plan to use the EmSim TimeWarp facility to implement a slower clock that can improve timing accuracy as well as increase the scalability of our simulations. Further improvements in scalability may also result from improvements in the 2.6 Linux kernel, including the constant time scheduler and the faster jiffy timer. Supporting parallel simulations, as suggested in Section 5.2, is more difficult but could significantly increase scalability.

- **Visibility:** Support for funneling live data from a deployment without requiring a backchannel to our analysis and visualization tools.

# 8. REFERENCES

[1] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In *Proceedings of the SIGCOMM Workshop on Communications in Latin America and the Carribean*, Costa Rica, Apr. 2001.

[2] L. Girod, V. Bychkovskiy, J. Elson, and D. Estrin. Locating tiny sensors in time and space: A case study. In *in Proceedings of ICCD 2002 (invited paper)*, Freiburg, Germany, September 2002. `http://lecs.cs.ucla.edu/Publications`.

[3] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. In *Proc. of the 2004 USENIX Tech. Conf.*, Boston, MA, 2004. USENIX Association.

[4] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (snack). In *Proc. of the 2nd Intl. Conf. on Embedded Networked Sensor Systems*. ACM Press, 2004.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Arch. Support for Prog. Lang. and Oper. Sys. (ASPLOS-IX)*, pages 93–104, Cambridge, MA, USA, November 2000. ACM.

[6] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for smart dust.

In *Proc. of the 5th ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99)*, N.Y., August " 15–20" 1999. ACM.

[7] W. J. Kaiser, G. J. Pottie, M. Srivastava, G. S. Sukhatme, J. Villasenor, and D. Estrin. Networked infomechanical systems (nims) for ambient intelligence. Technical Report CENS Technical Report 0031, CENS/UCLA, Dec. 2003.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. on Comp. Sys.*, 18(3):263–297, August 2000.

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

[10] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *WSNA*, 2002.

[11] W. Merrill, L. Girod, J. Elson, K. Sohrabi, F. Newberg, and W. Kaiser. Autonomous position location in distributed, embedded, wireless systems. In *the IEEE CAS Workshop on Wireless Comm. and Networking*, Pasadena, CA, 2002.

[12] A. S. S. Park and M. B. Srivastava. Sensorsim: a simulation framework for sensor networks. In *Proc. of the 3rd ACM Intl. Workshop on Modeling, Analysis and Sim. of Wireless and Mobile Sys.*, pages 104–111, Boston, MA USA, 2000.

[13] G. Werner-Allen and M. Welsh. Motelab: A web-enabled sensor network testbed. `http://motelab.eecs.harvard.edu`.

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

[15] atemu - sensor network emulator / simulator / debugger. `http://www.cshcn.umd.edu/research/atemu/`.

[16] Cens seismic array. `http://www.cens.ucla.edu/Project-Descriptions/Seismology/index.html`.

[17] Extensible sensing system: An advanced network design for microclimate sensing. `http://www.cens.ucla.edu/`.

[18] James reserve. `http://cens.ucla.edu/Research/Applications`.

[19] The network simulator ns-2. `http://www.isi.edu/nsnam/ns/ns-documentation.html`.

---

[2]SOS is a new mote OS in development at NESL/UCLA.