Experimental Platform for Studying Distributed Embedded Control Applications

Miklos Maroti, Ken Frampton, Gabor Karsai, Stefan Bartok, Akos Ledeczi
Vanderbilt University
akos.ledeczi@vanderbilt.edu

***Abstract**. Networked embedded systems are highly distributed systems with limited resources and communication capabilities tightly coupled to physical processes with sensors and actuators. These constraints do not allow the deployment of existing heavyweight middleware layers to support the distributed control algorithms in the different application domains of these systems. The paper describes a simulation framework that enables the timing accurate simulation of these systems including the physical system, the computing nodes and the communication network. The tool serves two distinct purposes by 1) allowing experimentation with different application domains and distributed control algorithms in the presence of communication delays, clock drift and faults and 2) providing a platform for distributed middleware research.*

Introduction

Networked Embedded Systems (NEST) constitute a new category of systems that necessitate the development and application of novel systems and software engineering techniques. These systems are tightly coupled to physical processes, and distributed across a relatively large number of processing nodes often having limited resources and communication capabilities. Each node has one or more sensors and actuators directly attached to it that interacts with the physical process at the node's location.

As an example for a NEST application, consider the problem of active control of aircraft interior noise. Aircraft interior noise is a result of two primary sources: engine noise and turbulent boundary layer noise. Engine noise, which itself originates from engine vibration, combustion and turbulent engine flow, is transmitted to the aircraft cabin through the airframe. Turbulent boundary layer noise, which consists of stochastically varying acoustic pressures acting over the entire aircraft exterior, enters the cabin by being transmitted through the aircraft fuselage [4]. In each case, there is a bottleneck in the path by which acoustic energy enters the cabin; this is in the fuselage panels.

Active noise control is the reduction of unwanted sound through the use of "active" sources such as loud speakers or vibrational excitation. Some type of sensor, such as an accelerometer or microphone, measures the unwanted sound. A controller is implemented, typically via a digital signal processor, which takes the sensor signals and uses them to calculate an optimal control signal. This control signal is used to drive the active source resulting in the cancellation of the unwanted noise. In an aircraft interior noise control application, the sensors would be a combination of microphones to measure the interior noise and vibration sensors to measure the aircraft fuselage vibration.

Previous investigations [5-6] produced promising results, having reduced the transmitted acoustic energy by an order of magnitude. However there are some hurdles to be overcome before a practical, large-scale implementation can be realized. First, while transmission control on single panels has met with great success, the idea of configuring all fuselage panels with the appropriate sensors and actuators for control is rather daunting. Perhaps the greatest hurdle in this regard is the need for communications between the sensors, actuators and a centralized controller. The weight of the wires alone is enough to prohibit the application on an aircraft. Thus, a decentralized control scheme is more appealing wherein each panel has an independent controller with its own set of sensors and actuators and only communicates with neighboring controllers.

Large-scale active noise control is only one possible application of NEST technology. Other domains include active flow control, micro-satellite constellations, autonomous vehicles, smart civil structures, and cooperative robotics. The common characteristics of these applications introduce very specific constraints on the solution. There needs to be a large number of resource-limited computing nodes. Resource limitations include, for instance, limits on CPU performance, memory size, available power, and others. Communication between nodes is limited in terms of bandwidth and connection topology. In the most restrictive situation, nodes may be able to communicate with their immediate neighbors only.

The computing nodes and communication links are unreliable. Node failures must be detected and application-specific reconfiguration must be performed to compensate for the loss.

The highly distributed nature of computing in a NEST application implies that each node should be equipped with sophisticated middleware —a kind of distributed operating system that provides global services for the applications (in addition to the local operating system that supports local resource management). This middleware layer is a key ingredient of NEST applications: it encapsulates services that are reusable across a number of specific problems, yet independent of the underlying hardware infrastructure (which is managed by the local OS).

The middleware is expected to support various coordination services beyond basic communication protocols. Coordination services range from simple event- and time-based coordination to complex algorithms for leader election, spanning tree formation, protocols for distributed consensus and mutual exclusion, distributed transactions, group communication services, clock synchronization and others [7]. These services go beyond the usual capabilities provided by networking protocols. Additionally, because of the inherent unreliability of the nodes and communication links, aspects of fault tolerance must also be addressed by the middleware.

Because of resource limitations, a complex, monolithic middleware layer that contains all services for all applications is not feasible. The middleware layer for NEST applications needs to be thin, application-specific and high-performance. Developing the technology to support these requirements is our current research focus. However, what we quickly realized is that there is no readily available platform to experiment with NEST applications and middleware. Therefore, we developed a modular configurable simulation environment, called the **S**imple **NEST A**pplication Simulator (SIESTA) that enables the experimentation with different distributed control applications and middleware technology.

The next section describes the architecture and unique characteristics of SIESTA. The middleware layer is discussed in a separate section. Finally, an application example is detailed, the structural damping of a vibrating beam. Preliminary results are also presented on how message delay affects the control quality in this particular application.

SIESTA

The **S**imple N**EST A**pplication Simulator (SIESTA) provides a simple abstract platform for experimentation with typical NEST applications such as active vibration control. It has a modular, layered architecture, so that the physical system simulator, the simulator of the distributed computing platform including the communication network, the actual middleware layer and the control application are all pluggable components. As illustrated in Figure 1, component interface with each other through well-defined, simple APIs.

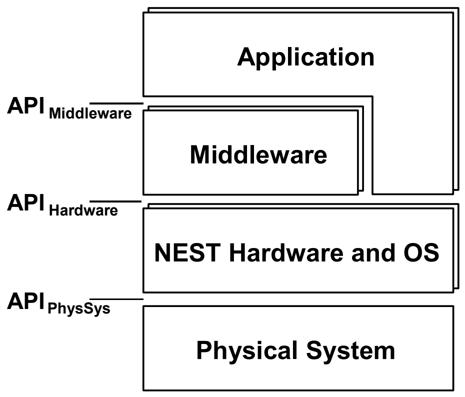

Figure 1: SIESTA Architecture

Because of its modular architecture and pluggable components, SIESTA is in fact a family of simulators. In the remainder of this paper we refer to one specific instance of it that simulates the structural damping of a vibrating beam. The last section of this paper describes the application and preliminary simulation results in detail.

Physical System Simulation

At the bottom of the architecture is the physical system simulator. It's API is extremely simple:

```
double sense(int sensor_id);
void actuate(int actuator_id, double value);
```

This allows a high degree of flexibility in plugging in external physical system simulators. Even existing third party tools can be interfaced with using a small adaptation layer.

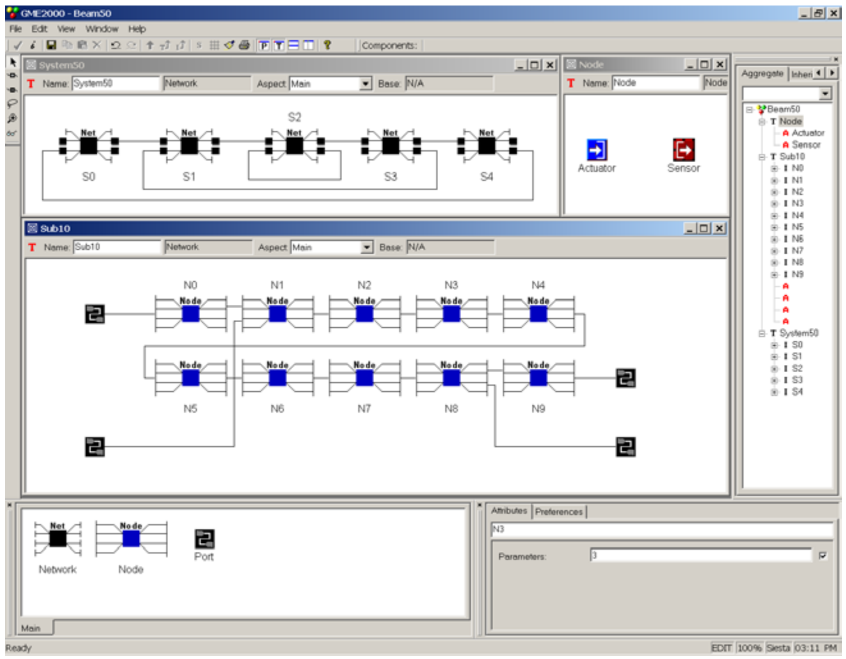

Figure 2: Siesta Configuration

The biggest constraint in interfacing, however, is that SIESTA uses a synchronous model, i.e. the different simulator modules need to work in lockstep. That's why the API does not contain timestamps or any other indication of time. The SIESTA scheduler keeps track of global time (the individual computational node simulators keep track of their own local time). It is the task of the scheduler to make sure that the whole system is simulated in a timing accurate manner.

The simulation of the vibrating beam is done using a state-space model. It is implemented as a Java component running in its own thread. Details will be provided in the last section.

Hardware Component

The heart of SIESTA is the Hardware component. Its tasks include 1) simulating the individual NEST nodes and the local operating system by providing an isolated environment for the application code to run in and 2) simulating the whole network by providing point-to-point communication capability between neighboring nodes using dedicated communication. The network size and topology, as well as sensor and actuator allocation, are configurable from graphical models.
Figure 2 shows the graphical configuration tool of Siesta. It is based on the Generic Modeling Environment [3]. The small window in the top right corner shows the model of a single node. It contains a sensor and an actuator. Each of them has a unique identifier as a textual attribute. The network is modeled as a two-layer hierarchy to simplify the models. Each subsystem called Sub10 contains ten nodes connected in a linear array. Some of the node interfaces are exposed to the outside world via ports. The top window called System50 shows the entire network consisting of ten subsystems connected so that they form a linear array with three additional links to decrease the diameter of the network. The basic linear topology is a good fit of the problem since the model of the beam is one-dimensional.

The graphical models are used to automatically generate a Java class containing all the necessary configuration information for Siesta.

Timing Accurate Simulation

SIESTA is not a real-time simulator, in fact, it will run orders of magnitude slower than the real system. (It is called SIESTA for a reason.) Two reasons for this are that it is a single processor application simulating hundreds of NEST processors and it is implemented in Java for simplicity and portability. However, the single most significant contributing factor is the requirement to simulate "logical" timing as accurately as possible. How is this accomplished?

In SIESTA, every NEST node has its own Java thread. An internal scheduler gives each of the nodes an N-millisecond timeslice in a round robin fashion. 1 millisecond is the absolute minimum that can be scheduled somewhat reliably in Java (depending on the JVM). It is assumed that this represents only T microseconds of real-time because the current control applications need to run control loops in the KHz range. To simulate the effects of clock drift on the quality of the control, the physical system simulator needs to be run at an even higher rate. However, if the application code does not have anything to do in the given timeslice, as it is the case most of the time, the middleware engine of the given node executes a number of "steps" (e.g. message routing operations) between a predefined minimum and maximum number and then the node yields the processor. This ensures that 1) idle nodes do not waste CPU cycles 2) the middleware always advances even if the host OS (e.g. Windows) takes away the CPU and the timeslice expires for the current node by the time SIESTA gets back the control. Note that the scheduler is a pluggable component itself. In fact, we currently have three different schedulers available in Siesta.

Since each simulated node keeps track of its own local time, it is fairly easy to introduce asynchronous clocks and hence, clock drift, making experimentation with different clock synchronization algorithms possible. Also, communication links can insert a configurable explicit time delay before message delivery. This delay can depend on the simulated hardware capabilities, network traffic, and other factors, or include a random component. We are planning to include fault injection capabilities to SIESTA in the near future to allow experimentation with different classes of faults, including different node, link, sensor, and actuator failures.

The hardware (and local operating system) API is also very simple:

```
int getNodeID();
double sense(int sensor_index);
void actuate(int actuator_index, double value);
int numberOfChannels();
void sendMessage(int channel, Object message);
Object receiveMessage(int channel);
```

It allows the application program and/or middleware to access the unique id of the hardware node it is running on and provides access to the sensor(s) and actuator(s) connected to the current node. Notice that it no longer the globally unique identifier of sensors and actuators that is used by the physical system simulator that is needed here, only a local one. Finally, the API assumes a simple message passing communication protocol allowing direct communication with neighboring nodes. The number of neighbors can also be accessed.

Middleware

The middleware layer of Siesta is a pluggable component, which allows different middleware frameworks to experiment with. The set of implemented services in each of the frameworks is not fixed either; it depends solely on the needs of the application. Currently, we are experimenting with two middleware technologies. The one used in our beam experiment is based on the asynchronous input output automata model as described in the "Distributed Algorithms" text by Nancy Lynch [7]. The other provides an event driven environment that is modeled after TinyOS [9]. There is extensive literature on both middleware technologies.

I/O automaton

Using the well-developed methodology of asynchronous I/O automaton we gain immediate access to a wealth of published and verified distributed algorithms. To facilitate the rapid implementation of these algorithms we designed our middleware engine around a core set of classes that closely resemble the basic concepts and notions used by the distributed algorithms community (see Figure 3).

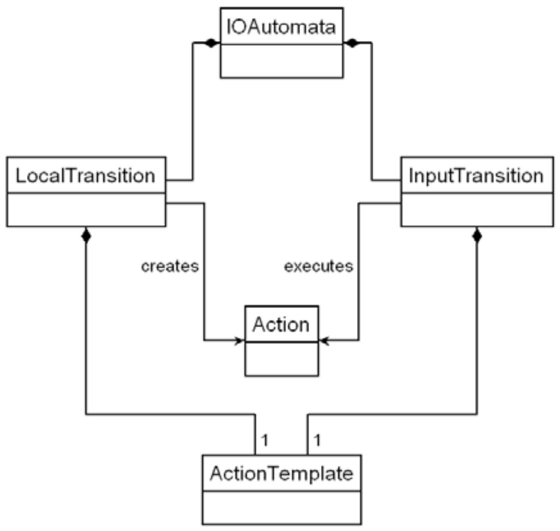

Figure 3. I/O Automaton implementation

IOAutomata is a base class that each middleware service needs to extend by defining member variables to hold state information, and by adding embedded classes to hold code fragments that describe state transitions. There are two types of state transitions: local (initiated by this service) and input (initiated from outside), which correspond to classes LocalTransition and InputTransition, respectively. A

LocalTransition is called enabled if the described state transition is possible or desirable in the current state of the automaton. The model has a very simple way of composing several automata into a single automaton by passing the “output” of one automaton to the “input” of (perhaps several) others, where the “output” is a list of parameters encapsulated in the class Action. Thus the middleware component is a single IOAutomata that is composed of several IOAutomata implementing different services, by merging all LocalTransitions and all InputTransitions. A single invocation (or step) of the IOAutomata starts by selecting an enabled LocalTransition randomly and letting it create an Action. Then all InputTransitions are selected and executed whose signature matches the created Action. Each step is executed atomically, and the IOAutomata needs to be invoked periodically. The ActionTemplate is a helper class that specifies the signature of the Action a LocalTransition creates and an InputTransition executes. An ActionTemplate has a set of parameters some of which can be null specifying a wildcard. This parameter list can also include an unlimited number of wildcards at the end of the list (tail).

For example, the Broadcast middleware service has a SendMessage LocalTransition with the following ActionTemplate parameter list: "send", null, "broadcast", "message" and a tail. When creating a specific Action, the second parameter will hold an integer, the channel number, while the tail contains the message. The created Action will be matched by the Send InputTransition of the Channel IOAutomata whose ActionTemplate parameter list is: "send", null and a tail. The execution of this InputTransition will cause a call to the underlying simulated hardware channel send primitive (OS service). Similarly, the Channel Receive LocalTransition generates an Action that will be matched by the Broadcast ReceiveMessage InputTransition (on the node at the other end of the hardware channel). Notice how the Action matching technique supports the composition of different I/O automata. The calling graph for each LocalTransition can be computed form the ActionTemplates before the simulation starts.

Our representation of the basic concepts of I/O automata might seem unnatural at first, but in fact it is almost the only choice. For example, in an execution environment actions cannot be classified to input, output and internal actions, because the same action can participate as an input action of one automaton, an output action of another automaton, and an internal action of the composition of the two automata. Similarly, one cannot separate output and internal transitions, which is why we used the term local transitions. Moreover, after analyzing the rules of composition, we can observe that no action can be matched by more than one local transition. This is why we can say that the local transition “creates” the action, while input transitions “execute” it.

TinyOS programming model

TinyOS [9] is an event based operating environment designed for use with embedded networked systems. It is designed to support the concurrency intensive operations required in such systems with minimal hardware requirements [8]. TinyOS mandates its own programming model on how components are written. It presents three abstractions: events, commands and tasks. The application is composed of a hierarchical set of components by wiring the events and commands to their handlers. Events represent function calls; they are executed synchronously. It is critical that events be as short as possible, although their execution might involve posting tasks. Tasks represent asynchronous, longer-running computations. Commands are very similar to events, except for their restriction on the calling graphs. Commands can only call lower level commands, while events can fire higher-level events and call lower level commands. Events are often the result of hardware interrupts, and they can preempt tasks.

The middleware framework developed for Siesta does not emulate any of the hardware or system components of TinyOS. It merely allows the development of middleware services using the programming and composition methodology of TinyOS. We have the following set of core classes (see Figure 4).

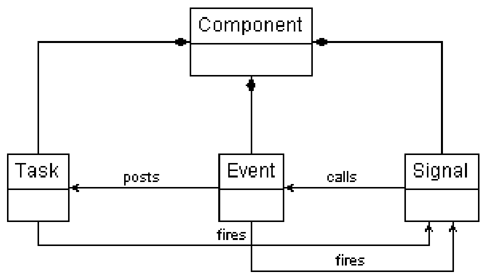

Figure 4. The implementation of the TinyOS methodology

In our current implementation we replace all commands with events and put no restriction on the calling graphs of events. Component is a base class that each middleware service needs to extend by defining member variables to hold state information, and by adding embedded classes to hold code fragments that describe commands, events and tasks. The embedded classes must be derived from Event or Task. Signals are implemented by declaring member variables of type Signal. Signals maintain a list of Events that must be called when the Signal is fired by Tasks and Events. Parameters passed between Events and Signals are encapsulated in the class Action. The Component maintains a list of posted Tasks that it executes one at a time.

For example, the Broadcast middleware service has a ReceiveMessage Event which calls the SendMessage Signal with the following Action parameter list: channel number and message. The SendMessage Signal is wired to the SendMessage Event of the Channel middleware service. The execution of this Event will cause a call to the underlying simulated hardware channel send primitive (OS service). On the node at the other end of the hardware channel the ReceiveMessage Event is executed by a simulated hardware interrupt, which saves the message and posts the ProcessMessage Task. When the ProcessMessage Task executes, it calls the DeliverMessage Signal, which is wired to the ReceiveMessage Event of Broadcast. Notice how the calling graphs are broken by posting Tasks and by the simulation of the hardware layer.

Example Application

The physical system under consideration is a simply supported beam subject to a random, point force disturbance. The beam is also subjected to the control inputs of each node and the response is measured at each node by a point velocity sensor. The beam dynamics are modeled using Galerkin's technique to discretize the partial differential equations. The result is a set of simultaneous ordinary differential equations of the form:

$$0 = M_k \frac{d^2}{dt^2}(q_k(t)) + K_k q(t) + Q_d(t) + \sum_{n=1}^{N} Q_n(t)$$

where M_k and K_k are the modal mass and stiffness, q_k is the displacement of the kth mode, Q_n are the control generalized forces and Q_d is the disturbance generalized force. In this case, the system was configured with 50 nodes distributed evenly along the length of the beam. For convenience in simulation, this set of equations was cast in discrete time, state variable form as follows [1]:

$$\mathbf{x}(\mathbf{k}+\mathbf{1}) = \mathbf{A}\mathbf{x}(\mathbf{k}) + \mathbf{B}\mathbf{u}(\mathbf{k})$$
$$\mathbf{y}(\mathbf{k}) = \mathbf{C}\mathbf{x}(\mathbf{k})$$

where k is the time step, **x** is the state vector containing the beam mode displacements, $q_k(t)$ and their derivatives, **u** is a vector of the control and disturbance forces, f_n and f_d, and **y** is the beam vibration velocity at node.

Although this paper focuses on the beam system described above, any system that can be cast in state variable form can be incorporated into SIESTA without any code modification. As an example, a physical model of a launch vehicle payload fairing has already been incorporated into one version of SIESTA.

Control Architecture Design

The decentralized control architecture employed here is referred to as a horizontal hierarchy. A horizontal hierarchy is one that consists of only a single layer of nodes that communicate only with neighboring nodes as shown in Figure 5. The degree of information sharing among nodes is defined by the “reach” of the system. A node that has a reach of R will have access to 2R+1 sensors; R sensors to the left, R sensors to the right in addition to its own sensor. Nodes located near the edge of the domain will simply have non-existent sensors omitted. Each node then creates a single control output based on the available sensor signals. This output is used to command the actuator associated with that node.

Each node employs a simple control law of the form

$$Q_n = \sum_{i=n-R}^{n+R} a_i \frac{d}{dt} x_i$$

In words, each node produces a control force that equals the weighted sum of all available sensor signals. The weight applied to each sensor signal, ai, was determined from a well-established LQR, optimal output feedback algorithm [2].

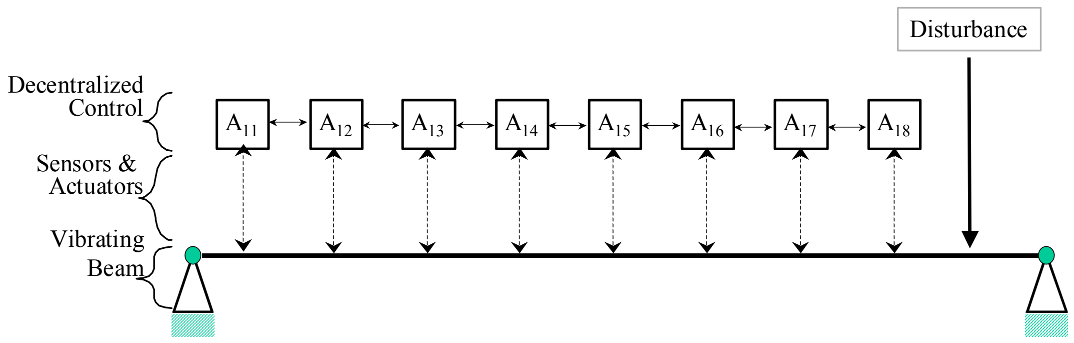

Figure 5 Schematic of the decentralized control architecture

Physical System Simulation

In order to allow SIESTA to simulate clock drift among nodes, it was necessary to permit different nodes to send their control signals to the beam at different times. To allow for this feature the physical system state variable equations were discretized at a sampling rate of 10 kHz (i.e. it updates once every 100 µsec). However, each node only updated its control signal at a rate of 2 kHz (every 500 µsec). Therefore, clock drift as reflected in the time at which a node applies its control signal, was simulated in increments of 100 µsec.

With this in mind, the physical system simulation evolves as follows: the beam model is given the control input for the previous time step and returns the sensor output for the current time step; then the control algorithm calculates the current control signal based on the appropriate sensor signals and which will be applied to the beam at the next time step. As noted previously, although the physical system is poled every 100 µsec, the control inputs are only changed once every 500 µsec.

Simulation Results

One of the most important questions that SIESTA was designed to answer was the effect of network communication delays on control performance. From a control system perspective, one would expect that the system would become unstable once the delay became sufficiently large. However, network communication delays are not simple fixed delays. Rather, they vary considerably with distance traveled, network traffic volume, etc. The effects of various amounts of delay are shown in Figure 6. This figure shows the beam transfer function (i.e. frequency domain ratio of output signal to input signal) between the disturbance input and the 23rd sensor output. Included in the plot are the transfer functions without control and with a reach=5 control system subject to fixed communication delays of 10 µsec, 100 µsec and 1000 µsec. Initial inspection shows that the performance under delays is not significantly affected. Although there are some differences, overall the reduction of the vibration amplitude relative to the no control case is similar. However, note that the highest frequency peak under 1000 µsec delay is significantly larger than that for no control. This certainly indicates a reduction in performance and may indicate impending instability in the system. A 1000 µsec delay corresponds to 2 samples at the control system update rate of 2000 Hz. Since the delay is per channel, reach=5 results in a maximum of 5000 µsec delay from the neighbor the furthest away. From a control system design perspective one would expect this to cause problems in performance. However, the highest frequency dynamics of the beam are about 200 Hz corresponding to a period of 5000 µsec. The delay in this case is just inside the highest frequency dynamics. We plan to run more experiments to see what effects even longer delays might cause.

SIESTA simulations have produced other interesting results as well. Although these are not detailed here, a summary is worthwhile. One conclusion stemmed from the comparison of performance between decentralized controllers of varying reach. One would expect that the larger the reach the better the performance. At the time of writing the results did not support this hypothesis, however, further analysis is required to make a firm conclusion.

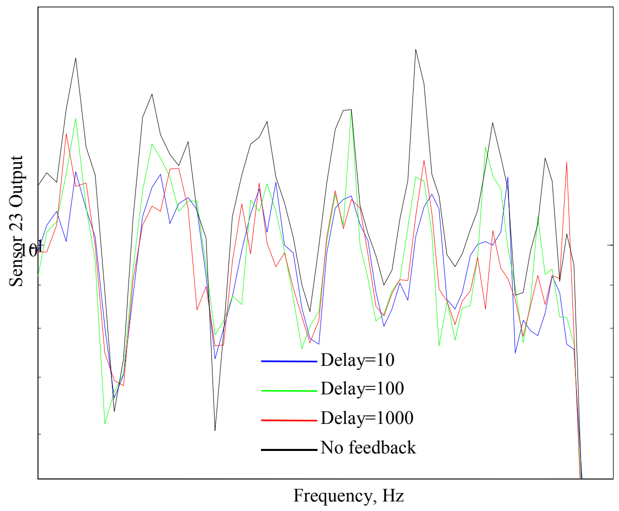

Figure 6 Beam transfer functions demonstrating the effects of communication delay on control performance

Another conclusion was that the effect of inter-node clock drift was negligible over relatively short time periods. The current version of SIESTA is limited to an initially synchronized state among modes. The time required to run the simulation long enough for significant drift effects was beyond the patience of the investigators. In order to investigate this further, future versions of the software will permit an initially unsynchronized state.

Conclusions

SIESTA has proven to be a valuable tool for networked embedded systems research. It serves two different purposes equally well. First, it provides a simple target platform for distributed middleware research. Second, it provides a realistic simulation of NEST systems in different application domains enabling the study of different distributed control schemes in highly distributed, resource constrained embedded systems. We plan to extend SIESTA to support other application domains. We want to add fault injection capabilities as well. Last, but not least, we shall optimize the tool to speed up the simulations.

Acknowledgement

The DARPA/ITO NEST program (F33615-01-C-1903) has supported, in part, the activities described in this paper.

References

[1] Clark, R., Saunders, W., and Gibbs, G *Adaptive Structures: Dynamics and Controls*, John Wiley & Sons, New York, New York, 1998.

[2] Levine, W. and Athans, M. "On the Determination of the Optimal Constant Output Feedback Gains for Linear Multivariable Systems," *IEEE Transactions on Automatic Controls*, Vol. AC-15, No. 1, 1970, pp 44-48.

[3] Ledeczi, A. et al. "Composing Domain-Specific Design Environments," *Computer*, pp. 44-51, November, 2001.

[4] 11. Frampton, K. D. and R. L. Clark, "Control of Sound Transmission through a Convected Fluid Loaded Plate with Piezoelectric Sensoriactuators," *Journal of Intelligent Materials Systems and Structures*, Vol. 8, No. 8, pp. 686-696, August 1997.

[5] Frampton, K. D. and R. L. Clark, "Improved Control of TBL Pressure Transmission through Aeroelastic Plates with LQG Compensation," presented at the 39th Structures, Structural Dynamics, and Materials Conference, Adaptive Structures Forum, Long Beach, CA, April 1998.

[6] Henry, J and R. L. Clark, "Active Control of Sound Transmission Through a Curved Panel into a Cylindrical Enclosure" to appear in the Journal of Sound and Vibration

[7] Lynch, N. A. *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996

[8] Hill, J. et al. "System Architecture Directions for Networked Sensors," *Proceedings of ASPLOS*, 2000

[9] http://webs.cs.berkeley.edu/tos