Design and Experimental Verification of Real-Time Mechanisms for Middleware for Networked Control

Kyoung-Dae Kim and P. R. Kumar

***Abstract*— Networked control systems give rise to several new problems which need to be resolved for successful development of such control systems. One of the most important is to develop a software framework with appropriate architecture and mechanisms for rapid, reliable, and evolvable distributed control applications. In particular, timeliness in the interaction among entities which constitute the overall control loop is one of the crucial requirements that must be provided by the platform. We discuss the issues related to the design and implementation of such mechanisms to support timeliness requirements. We describe the design of real-time enhancements to *Etherware*, a middleware for distributed control systems that has been developed in the Information Technology Convergence Laboratory at the University of Illinois. We further evaluate the performance of the designed middleware through a distributed inverted pendulum control application. We demonstrate the satisfactory performance of the unstable system even under several sophisticated run-time functionalities such as component upgrade and migration. We demonstrate that stability is maintained even under each of these tests as well as a computational stress test.**

I. INTRODUCTION

A software framework which enables rapid, reliable, and evolvable distributed system application development is critically important for the proliferation of networked control systems ([1], [2]). This is the motivation for our development of *Etherware*, a middleware for the distributed control system domain [3], which supports several essential functional requirements identified in [1] and [2]. This is deployed in the Information Technology Convergence Laboratory at the University of Illinois, which features as an exemplar a small-scale automatic traffic control system.

In this paper, we address the design of several critical enhancements with respect to the *timeliness* requirement since many control systems are time-critical systems, with the right action being required to be executed at the right time. Typically, control engineers design a control system by implicitly assuming the timely behavior of a system, but if the timeliness assumption is violated then either the stability of the system could be violated or the performance of the system could be degraded. Thus, timeliness is an essential requirement which should be supported by the platform on which the control system is implemented.

This material is based upon work partially supported by NSF under Contract No. ECCS-0701604, Contract No. CNS-07-21992, Contract No. NSF CNS 05-19535, AFOSR under Contract No. FA9550-09-0121, and USARO under Contract No. W911NF-08-1-0238 and Contract No. W-911-NF-0710287.

K.D. Kim and P.R. Kumar are with the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA {kkim50, prkumar}@illinois.edu

When we combine the temporal predictability provided by our real-time enhancement with the flexibility provided by the Etherware design, we generate rather powerful capabilities. We demonstrate two such capabilities in this paper, controller upgrade at run-time, and controller migration at run-time. Controller upgrade refers to the capability to change the control law while a system is running. We demonstrate the capability on an unstable system. Controller migration refers to the capability to relocate where a control law is being computed, even while the system is running. Etherware makes such flexible design easy, since its mechanisms are designed to be inherently flexible. They follow the principle of policy and mechanism separation, so that they can be used under various real-time usage models.

However, in order to reliably make use of these flexible mechanisms, one needs to ensure that system stability, a property that depends on time-critical actions, is preserved even under upgrade and migration. Our real-time enhancements make possible such stability preservation, as we demonstrate. Thus, the incorporation of these real-time mechanisms with the other flexibility providing functionalities provided by Etherware leads to a more powerful framework for distributed control system design.

In Section II, we introduce Etherware to describe how our middleware provides flexibility. The main contribution of this paper begins with Section III, where we discuss the design and implementation of Etherware mechanisms to support the timeliness requirement of the domain. To evaluate the performance of the real-time mechanisms in a demanding environment, we not only show that stability is maintained under sophisticated run-time functionalities such as component upgrade and migration, but that the real-time mechanisms preserve stability even when the system is subject to computational stress. The details about the system and experimental results are provided in Section IV.

II. ETHERWARE

A. *Domainware for Distributed Control Systems*

Our earlier generation of middleware [3] for distributed control systems, called *Etherware*, was developed to support *component-based application development*. One major benefit of component-based programming is that an application can be developed easily owing to the composability of components. Etherware uses the message exchange mechanism for component interaction. In Etherware, *Message* is a well-defined XML document object and it is the root class in the hierarchy of the Message class. A new message type for applications can be easily defined by extending the Message class. Listing 1 shows the XML structure of the Message class.

```
<EtherMsg type=... rel=... >
  <profile name=... ></profile>
  <content> ... </content>
  <ts value=... ></ts>
</EtherMsg>
```

Listing 1. XML structure of an Etherware Message

A name of the message can be specified in the `type` attribute of the `EtherMsg` element. In the `profile` element, the name of the recipient component is specified. In the `content` element, any information concerning the interaction semantics can be specified. The clock time when the message is created is specified in the `ts` element.

B. *Etherware Architecture*

Roughly, Etherware consists of a *Kernel* and *components*. Components can be classified further into *service components* and *application components*. The Kernel provides a set of fundamental functionalities for middleware operations, such as component life-cycle management and message delivery among components. To deliver a message from one component to another, Kernel creates and uses a *job*. A job is a scheduling entity in Etherware which contains the message to be delivered and the address of recipient component. When a new message arrives in the Kernel, the Kernel encapsulates the message into a job and enqueues it into a job queue. The jobs in a job queue are processed one by one by a job processing software module, called *Dispatcher*. The other functionalities which are required to be implemented in a middleware framework are provided as several service components. Section II-D discusses services in more detail.

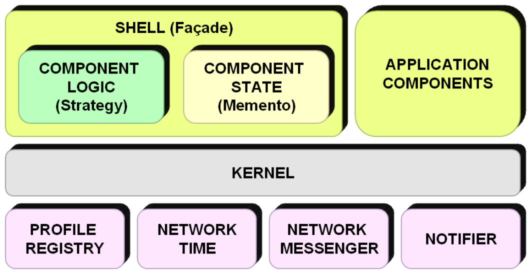

Fig. 1. Etherware architecture and component model

C. *Etherware Component Model*

In Etherware, as noted above, an application can be developed as a set of components. In designing the component model, several *software design patterns* [4] are used. *Shell*, a class object where design is based on the *Facade* software design pattern, encapsulates a user defined class object which implements the application logic. It manages the life-cycle of the encapsulated class object and provides an interface which allows the encapsulated class object to interact with the other components. The *Strategy* design pattern is used to design a uniform interface between Shell and the class object encapsulated by it. Due to this Strategy design pattern, Shell can be used to perform runtime *component replacement*. For *component migration*, the execution state of a component should be smoothly continued after the migration to avoid disrupting performance. The *Memento* design pattern was adopted to support this feature.

D. *Etherware Services*

Etherware supports several functionalities, that are commonly required for distributed control applications, as Etherware services. *ProfileRegistry* is a naming service which maps a semantic name to a physical deliverable address of a component. All the details about the network are hidden from the user by the *NetworkMessenger* service which is responsible for sending and receiving messages over the network. The *NetworkTime* service performs automatic timestamp translation. It translates the time stamp from the clock of the remote computing node to that of the local machine, for every message which is received by NetworkMessenger. Basically, Etherware is an *event-driven system* such that a component gets executed only when it receives a message. However, in many cases, control actions need to be taken based on *time*. In such situations, the *Notifier* service enables a component to execute at the time when it has to, by sending a notification message to that component.

III. ETHERWARE MECHANISMS FOR REAL-TIME GUARANTEES

A. *Issues for Real-Time Properties*

A real-time system is not a system which is fast, but is rather a system which is predictable. In fact, *predictability* is one of the most fundamental attributes of any real-time system [5]. In general, predictability depends on every aspect of the system, including H/W platform, communication network, operating system, programming language, etc. A middleware is a software framework interposed between the operating system and application programs. Therefore, in our following discussion, we assume that the hardware and software over which a middleware is executed, comprises a predictable real-time platform. Examples of such platforms are computer systems running a real-time operating such as VxWorks [6], QNX [7], and Real-Time Linux [8].

B. *Quality of Service (QoS) of Message Delivery*

In our work, we define QoS as a collection of attributes of an application which are used in scheduling for execution. The QoS specification can contain arbitrary types of attributes which affect the execution of an application. As shown below, it could be the period, the relative deadline, or the worst case execution time (wcet) of an application execution, or it could be an attribute related to the application's importance in the task set.

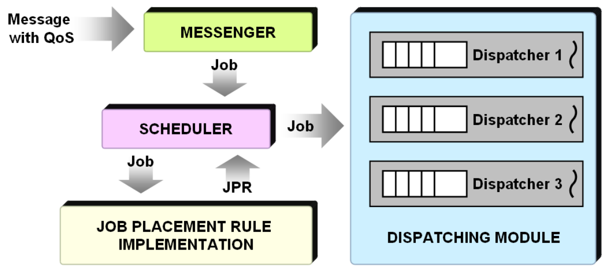

Fig. 2. Real-time scheduling mechanism with three Dispatchers

```
<EtherMsg type=... rel=... >
  <profile name=... ></profile>
  <content> ... </content>
  <ts value=... ></ts>
  <QoS crit=...  period=...  deadline
      =... wcet=...>
  </QoS>
</EtherMsg>
```

Listing 2. QoS specification in Message

Once we define a set of attributes as a QoS requirement, then the question is: Where does this QoS specification need to be embedded so that it can be used in Etherware's scheduling action? Considering the fact that Etherware is an event-driven system, as described in Section II-D, the Message class object is the right place to put QoS. Therefore, the XML document of a Message class object is modified to contain an element called `QoS`, which has attributes of `crit`, `period`, `deadline`, and `wcet` as illustrated above. Now, an application component can specify its QoS information about message delivery whenever it creates and sends a message to other components. Then Etherware uses the specified QoS information when it makes a scheduling decision for message delivery.

C. Priority-based & Concurrent Scheduling

Most real-time operating systems support priority-based scheduling ([6], [7], [8]). They provide predictable behavior with respect to the priority of a process (or a thread in some cases). Etherware utilizes this priority-based scheduling mechanism of the underlying platform to render it a real-time middleware.

As explained in Section II-B, Dispatcher is a software module inside Etherware Kernel for processing a job. For concurrent processing, we have therefore added to Etherware Kernel a *dispatching module* consisting of multiple Dispatchers, as shown in Fig. 2. The Etherware mechanism for the dispatching module is designed such that the decision about the number of Dispatchers and the priority[1] of each Dispatcher can be specified by an Etherware user. This specification (or *policy* in more formal terminology) is called a *Thread Scheduling Rule* (*TSR*) in Etherware. Each Dispatcher has its own job queue to hold jobs to be processed by the Dispatcher, as shown in Fig. 2. Now, the job queue is modified to become a prioritized queue so that jobs in the queue can be ordered based on a specified attribute of each job.

Typically, a scheduler makes a decision about execution order among tasks. However, the Etherware Scheduler shown in Fig. 2 operates a little differently from such typical scheduling actions. Instead of deciding an execution order among tasks, it determines the right place where a job (a scheduling entity in Etherware Kernel) should go in the dispatching module. When the Etherware Scheduler makes a decision, it refers to and implements the rules specified by an Etherware user. This user-specified rule is called the *Job Placement Rule* (*JPR*). To find a right place in the dispatching module, Etherware Scheduler needs two pieces of information, one to select a Dispatcher in dispatching module, and the other to find the right position in the job queue of the selected Dispatcher. Thus, a JPR returned from a user-implemented software module as shown in Listing 3 should contain such information.

```
/* QoS : criticality(C), period(P) */
JPR queryJPR(Job job) {
 JPR jpr;
 now=current_time();
 if(job.C=1) jpr=(Disp#1, now+P);
 else if(job.C=2) jpr=(Disp#2, now+P);
 else if(job.C=3) jpr=(Disp#3, now+P);
 return jpr;
}
```

Listing 3. A pseudo-code example of JPR implementation

D. Issues concerning Implementation

For better predictability, the concurrent message processing mechanism is adopted, as explained in Section III-C. One of the major concerns that needs to be addressed in any concurrent system is *synchronization*. Without having proper synchronization among multiple concurrent processing entities[2] in a software program, the program might fall into a *deadlock* situation or produce some unexpected execution outcomes due to a *race condition* [9]. Therefore, as a first step toward a concurrent real-time software system, it is always an important procedure, from the implementation point of view, to analyze and modify the software system to make sure that it has the right synchronization at every required place. This is indeed the source of difficulty for concurrent programming.

Etherware was originally developed using the *Java* programming language [10], for several reasons. First, it is easy to develop a software program over a well designed Object-Oriented Programming (OOP) language. Java has many built-in class packages for networking, data structures, multiprocessing, and so on. Also, it releases the burden

[1]The specific priority set is given by the underlying software platform.

[2]This processing entity is typically called a *process* or a *thread* in operating systems.

of memory management from a programmer, through its garbage collection mechanism. Second, it supports platform independent application development. Therefore, from the software engineering point of view, Java is a good candidate for developing a software program for a distributed system such as Etherware.

However, there are a couple of caveats concerning Java, with respect to real-time performance. Java was not originally designed for real-time applications. It is designed and optimized for performance in terms of overall throughput rather than predictability. Specifically, the dynamic loading, linking and initialization of classes or interfaces of Java Virtual Machine (JVM) could cause unpredictable delays in program execution. Even worse, unpredictability comes from the runtime memory management, i.e., garbage collection. Thus, even though Java is well suited for a distributed application, it is not well suited for a distributed control application. However, responding to the recent increased demands for real-time embedded systems, there have been several efforts to expand Java's application domain into the real-time computing areas over the past decade. Toward this goal, the first version of the Real-Time Specification for Java (RTSJ) [11] was released in 2000 through the Java Community Process (JCP). In 2005, Sun Microsystems released its first version of the RTSJ implementation, called *Sun Java Real-Time System* (*Sun JavaRTS*) [12].

At the heart of the specifications defined in RTSJ for better predictability, are specifications for thread scheduling/dispatching and memory management. For real-time execution, two new classes of threads are defined in RTSJ. `RealtimeThread` (RTT) extends the standard `java.lang.Thread` class and also implements the `Schedulable` interface, which is also newly defined in RTSJ. In terms of predictability, RTT can only provide soft real-time performance. A thread which is an instance of RTT can still be preempted by some internal behaviors of JVM, such as garbage collection, dynamic object loading and so on. For better predictability, RTSJ recommends using `NoHeapRealtimeThread` (NHRTT), which is extended from RTT. Based on RTSJ, NHRTT can provide hard real-time performance under some strict restrictions on memory usage[3] to avoid preemption by a garbage collector. However, this requires significant change to the programming model, due to the restrictions on memory usage, making it much harder to write a program in Java. Therefore, NHRTT is not yet a practical solution for hard real-time performance, especially for a large-scale software program like Etherware. Since we believe that the easy-to-use programming model of current Java is one of its biggest benefits, we do not employ NHRTT to achieve hard real-time performance.

Besides RTSJ, there have been several research works on real-time garbage collection (RTGC) ([13], [14]) to improve predictability of Java programs. In this line of research, the basic objective is to develop a garbage collection mechanism which performs automatic memory management without sacrificing the predictability of RTT. However, even though there has been some significant improvement in terms of predictability with RTGC mechanism, it is still necessary to use the NHRTT mechanism of RTSJ for hard real-time guarantee.

[3]For details about memory management in RTSJ, we refer the reader to [11].

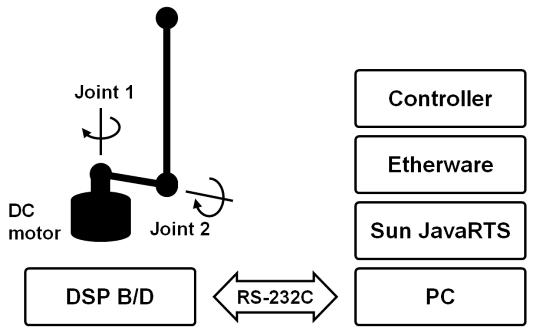

Fig. 3. Schematic of the inverted pendulum control system

To implement Etherware's real-time mechanisms, we use the second version of Sun JavaRTS which includes an RTGC enhanced from the work in [13]. With this RTGC, an RTT in Sun JavaRTS provides quite good performance in terms of predictability in most cases, as we show below.

In the following section, we illustrate the real-time performance of our real-time Etherware through a distributed inverted pendulum control application.

IV. DISTRIBUTED INVERTED PENDULUM CONTROL SYSTEM

A. Inverted Pendulum Control System

To illustrate the application of our real-time Etherware for distributed control, an inverted pendulum system was chosen, since it is an inherently unstable system which requires strict predictability of the computing system on which the controller is executed. Fig. 3 shows the schematic of the inverted pendulum system that is used in our experiment. As illustrated in Fig. 3, it has two links: one is on the base which is actuated by a DC motor attached to it, and the other is a passive link. Also, it is equipped with a DSP board for both measuring the angles of both joints, and for applying the PWM signal to a DC motor. The controller actually runs on a PC which is connected with the DSP through RS-232C serial communication. The controller is developed as an Etherware component running on Etherware, which in turn runs on Sun JavaRTS for real-time performance.

In our implementation of the controller, the controller is activated every 15 ms. In each period, the controller first requests the angle data from the DSP. Once it receives the measured angle data, it then computes a control output value and sends it back to the DSP. Then the DSP board delivers the control action right after it receives the control command from a PC over the serial port.

We now proceed to further test the real-time performance under a "stress" test on the Etherware and operating system.

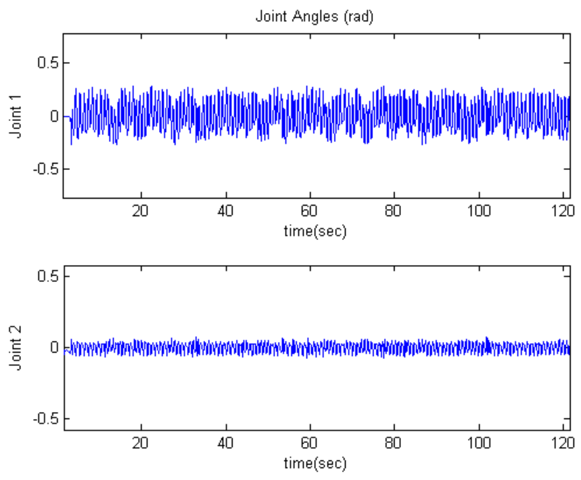

Fig. 4. Periodic control of an inverted pendulum under stress

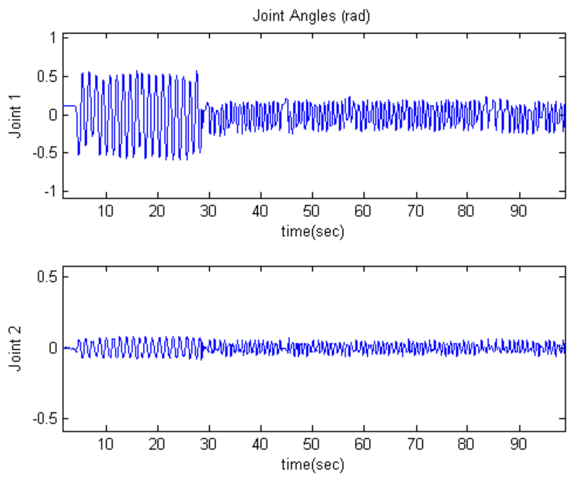

Fig. 5. Joint angles of the inverted pendulum under runtime controller upgrade

B. Periodic Control under Stress

To verify the timeliness guarantee of the real-time Etherware, the inverted pendulum is controlled by a periodic controller under a stress condition. In this experiment, to stress the computing node where a periodic controller is running, an extra periodic task which consumes approximately 20 percent of the CPU time is made to run concurrently with the periodic control task. The stress task has a period of 5 seconds, and consumes 1 second to execute its computational task. Clearly, the 1 second execution of the stress task is long enough to make the inverted pendulum system unstable if the real-time performance is not supported by Etherware.

To achieve timely execution behavior of the periodic control task, a higher execution priority is assigned to the periodic control task than that assigned to the stress task. The experimental result is shown in Fig. 4. In this experiment, the stress task starts to execute its periodic task around 20 seconds after the system starts. As shown in the result, the stability of the inverted pendulum has not been affected at all by the execution of the stress task.

C. Runtime System Management

The experiments to be described in this section emphasize the necessity of a real-time middleware framework for distributed control systems. The specific capabilities that we aim to provide are controller upgrade and controller migration.

It is important to have predictability concerning the behavior of a distributed control system that is subject to such change of its application configuration at runtime, and to determine a priori whether the real-time mechanisms can support the runtime management features in Etherware. In some applications, we can safely perform the system reconfiguration at runtime. However, in some other applications, which are typically time-critical control systems like an inverted pendulum system, it is necessary to have a guaranteed time bound for the time of reconfiguration. Without having such a time bound, the control system may fail to maintain its stability. In the sequel, we detail experimental results concerning the two above mentioned important runtime system management tasks - controller upgrade and migration.

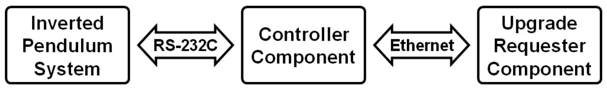

Fig. 6. System configuration for controller upgrade

1) Controller Upgrade: Here the goal is to upgrade a controller at run-time to change a control law while the system is running. Fig. 6 shows the configuration of a distributed application for such runtime controller upgrade. The controller runs on the PC that is directly connected to the inverted pendulum system through a serial communication. However the component which requests the controller upgrade runs on the other PC, and sends its request to Etherware around 30 seconds after the system is started, for better control performance. The angles (0, 0) are the reference positions for the two joints of the inverted pendulum. The joint angles are measured during the controller upgrade process and plotted in Fig. 5.

As seen in the results, the second link of inverted pendulum maintains its vertical upright position even while the controller is upgraded. Also, we can see the difference in control performance before and after controller upgrade.

2) Controller Migration: Here the goal is to move the location where the control law computation is performed from one node to another, while the system is running. Such a capability can be important in optimizing the behavior of control systems. For example, if network delays cause congestion, one may want to relocate where the control law is computed. More generally, one may want to optimize control loop performance with respect to delays. To migrate a component from one PC to another PC, Etherware performs several steps of actions internally, such as the externalization of the component's runtime state, creation of a new instance of component at the destination node, restoring the externalized state in a new component instance, and more. For *safe* component migration, all these actions are required to be performed in a timely manner in time-critical control systems. As stated in the previous section, this is an important situation in which real-time mechanisms are essential.

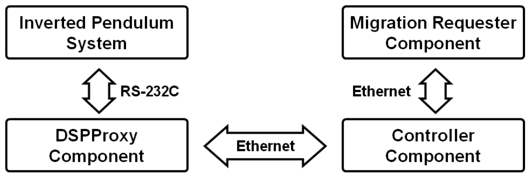

Fig. 7. System configuration for controller migration

Fig. 7 illustrates the system configuration for a controller migration application. The inverted pendulum is initially controlled by a controller running at a remote computing node in this application. Since the controller cannot directly communicate with the inverted pendulum system, another component, the DSPProxy component, is developed to mediate the interaction between the inverted pendulum system and the periodic remote controller. Yet another component which requests controller migration to Etherware runs at the third computing node. In the experiment, the requester component sends a controller migration request around 40 seconds after the system is started. Then the controller is migrated from the remote node to the node where the inverted pendulum is directly connected.

Fig. 8 shows the measured joint angles during the controller migration process. As shown in the plot, there is no loss of the stability of the inverted pendulum control system during the migration.

V. CONCLUSIONS

In this paper, we show that it is important to develop a real-time middleware framework for realization of flexible distributed control systems. Since networked control systems are typically open and large-scale systems, we argue that the framework should be designed with an appropriate architecture and mechanisms to support the fundamental requirements of distributed control systems. In view of domain requirements identified in [15], we have enhanced Etherware [3] to support time-critical nature of the domain requirements. In this paper, we have addressed the issues related to timeliness. We have proposed Etherware mechanisms to support real-time performance and discussed issues related to the implementation of the mechanisms. To evaluate the real-time performance of the designed middleware, a distributed inverted pendulum control system is investigated as described in Section IV. Through the experiments involving sophisticated runtime functionalities such as runtime controller upgrade and migration, we have demonstrated the temporal predictability of our real-time Etherware.

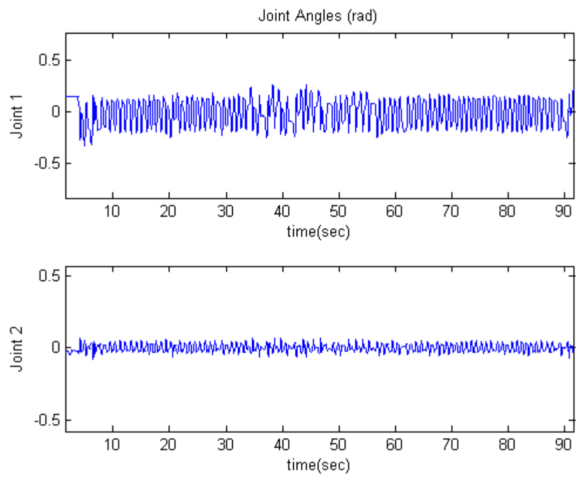

Fig. 8. Joint angles of the inverted pendulum under runtime controller migration

VI. ACKNOWLEDGMENTS

The authors gratefully acknowledge the contribution of Daniel J. Block for his generous experimental support.

REFERENCES

[1] S. Graham, G. Baliga, and P. R. Kumar, "Issues in the convergence of control with communication and computing: proliferation, architecture, design, services, and middleware," in *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, vol. 2, 2004, pp. 1466–1471.

[2] ——, "Abstractions, architecture, mechanisms, and a middleware for networked control." *IEEE Transactions on Automatic Control*, vol. 54, no. 7, pp. 1490–1503, July 2009.

[3] G. Baliga, "A middleware framework for networked control systems," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[4] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, November 1994.

[5] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd ed. Springer, October 2004.

[6] "VxWorks," Wind River, http://www.windriver.com/.

[7] "QNX RTOS," QNX Software Systems, http://www.qnx.com/.

[8] "Real-Time Linux," FSM Labs, http://www.fsmlabs.com/.

[9] A. S. Tanenbaum, *Modern Operating Systems (3rd Edition)*, 3rd ed. Prentice Hall, December 2007.

[10] K. Arnold, J. Gosling, and D. Holmes, *Java (TM) Programming Language*, 4th ed. Prentice Hall PTR, August 2005.

[11] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull, *The Real-Time Specification for Java*, 1st ed. Addison Wesley Longman, January 2000.

[12] "Sun Java Real-Tim Systems," Sun Microsystems, http://www.sun.com/.

[13] R. Henriksson, "Scheduling garbage collection in embedded systems," Ph.D. dissertation, Lund University, 1998.

[14] D. F. Bacon, P. Cheng, and V. Rajan, "The metronome: A simpler approach to garbage collection in real-time systems," in *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*, 2003, pp. 466–478.

[15] K.-D. Kim and P. R. Kumar, "Architecture and mechanism design for real-time and fault-tolerant Etherware for networked control," in *Proceedings of the 17th IFAC World Congress*, July 2008, pp. 9421–9426.

[16] T. Lindholm and F. Yellin, *Java(TM) Virtual Machine Specification*, 2nd ed. Prentice Hall PTR, April 1999.