A Real-Time Middleware for Networked Control Systems and Application to an Unstable System

Kyoung-Dae Kim and P. R. Kumar

Abstract—A well-designed software framework is important for the rapid implementation of a reliable and evolvable networked control applications, and to facilitate the proliferation of networked control by enhancing its ease of deployment. In this paper we address the problem of developing such a framework for networked control that is both real-time and extensible. We enhance Etherware, a middleware developed at the University of Illinois, so that it is suitable for time-critical networked control applications. We introduce a notion of Quality of Service (QoS) for the execution of a component. We also propose a real-time scheduling mechanism so that the execution of components can not only be concurrent but also be prioritized based on the specified QoS of each execution. We have implemented this framework in Etherware. We illustrate the applicability of this software framework by deploying it for the control of an unstable system, a networked version of an inverted pendulum control system, and verify the performance of the enhanced Etherware. We also exhibit sophisticated runtime functionalities such as runtime controller upgrade and migration, to demonstrate the flexible and temporally predictable capabilities of the enhanced Etherware. Overall, Etherware thus facilitates rapid development of control system applications with temporally predictable behavior so that physical properties such as stability are maintained.

Index Terms—Networked Control Systems, Middleware, Real-Time Systems, Unstable System.

I. INTRODUCTION

At a time when networked control systems are on the cusp of a takeoff, a software framework which enables the rapid implementation of a reliable and evolvable networked control application is indeed one of the most important factors for the proliferation of such networked control systems [1]. This motivates the development of a middleware specifically for networked control systems that can support component-based application development. Such a framework can facilitate an application to be developed easily through the composition of a set of relevant software components. Moreover, by providing service components, it can further simplify the development of networked applications which typically require expertise to address the issues caused by the existence of a not perfectly reliable communication network that mediates interactions between components.

Such a middleware, called Etherware, has been developed at the University of Illinois, which is also flexible in that it supports runtime system evolution through its component model. The ACE ORB (TAO) [2] is another software platform that can be used for networked control systems. Its architecture is based on RT-CORBA [3] which is a real-time extension of CORBA specification for distributed object computing from the Object Management Group (OMG). Also, Costa et al. [4] have proposed a component-based middleware, called RUNES middleware, for reconfigurable, ubiquitous, networked embedded systems. As in Etherware, one of its useful feature is that it supports runtime reconfiguration via a middleware service, called the Logical Mobility Service. ROS [5] is another development platform which is designed for rapid implementation of large-scale robot system applications. Even though there are several common concepts in ROS and Etherware such as message-based interaction and microkernel-based design, the fundamental design goals of ROS are quite different from those of Etherware. For example, while ROS is designed to be language neutral and to make it easy to reuse existing codes for robotic applications, Etherware is designed to support component-based application development for general purpose networked control systems.

In this paper, we address critical additional aspects necessary for networked control applications. Among them, we are especially interested in providing Etherware with real-time capability. This is especially important for a middleware for networked control systems, since many control systems are in fact time-critical systems, with the right action being required to be executed at the right time, for otherwise the performance of a control system can be degraded or even become unstable.

As an approach to real-time capability of Etherware, we first introduce a notion of the Quality of Service (QoS) for the execution of a component. More precisely, it is the QoS of a Message for a component that gets executed when it receives a Message. In general, the QoS of a component execution can contain any information, such as the period of the execution, or the deadline of each instance of execution, etc., all of which can potentially be used in scheduling decisions within Etherware. The scheduling mechanism of Etherware is enhanced so that multiple components can be executed concurrently while the preemption between executions is allowed based on the priority that is determined by the QoS of each component’s execution. Thus, the enhanced Etherware provides real-time performance through the prioritized and concurrent Message delivery based on the QoS of each Message.

We assess the performance of the real-time middleware for networked control by experimentation on a networked
inverted pendulum system. We also assess the flexibility of the framework through experiments including runtime controller upgrade and runtime controller migration. Together, these demonstrate the capabilities of the enhanced Etherware that are enabled by the combination of the flexibility that is provided by Etherware design and the temporal predictability which is provided through the enhancement in this paper. Thereby, we show how the incorporation of the real-time mechanisms with the other flexible mechanisms of Etherware leads to a powerful framework for networked control systems.

In Section II, we address the characteristics of networked control systems and describe some requirements for a software framework. Then we introduce Etherware in Section III. Section IV discusses the design of Etherware mechanisms to support the timeliness requirement of the domain. Some implementation related issues are discussed in Section V. Then in Section VI, the deployment of Etherware on a networked inverted pendulum control system is described and the experiment results are presented.

II. NETWORKED CONTROL SYSTEMS

A. Domain Characteristics

There are many potential examples of networked control systems, including smart power grids, intelligent air/ground traffic control systems, automatic warehouse management systems. Even though these examples are in different application areas, they share some common characteristics as networked control systems:

1) Large-scale: Due to the existence of a communication network, the scale of networked control systems is typically larger than that of classical control systems. It is large not only because the number of entities to be controlled is large but also because the structure of networked control systems is complex due to multi-scale control objectives. As an example, in an automated warehouse system, each robot is controlled to move some objects from one place to another based on a given command that is determined to meet a high level control objective. In addition to these, there also should be another control layer for safety such as collision avoidance between robots while they are moving around, which adds complexity to the overall system.

2) Openness: In a networked environment, a system configuration which forms a control-loop of a system can easily be changed at runtime. More specifically, a new entity can join or an existing entity can leave the system at any point of time. Moreover, it is also possible that an existing entity can be replaced or even be migrated to another computing node at runtime depending on the system states.

3) Time-criticality: In most cases, a control system is a time-critical system in which given actions such as sensing and control are required to occur at the right time. Failure to do so can degrade the system’s performance or can even cause the system to become unstable.

4) Safety-criticality: A safety-critical system is one in which the cost of system failure is very expensive, causing severe damage or harm to people, equipment or the environment. It is easy to see that many control systems are indeed safety-critical systems since they are employed on physical systems.

B. Domain Requirements

The following are some of the requirements for a middleware framework for networked control systems.

1) Operational Requirements: The fact that the overall system is distributed over a communication network makes it potentially much harder and more time-consuming to develop an application. Also, clock differences between computing nodes accentuate difficulties in developing a networked control application. Therefore, it is necessary to have mechanisms in a middleware framework which can resolve the problems of both location difference and time discrepancy. Besides these two requirements, a mechanism which supports semantic addressing (or context-aware addressing) is also a desirable feature since it can significantly improve the portability and reusability of the application code.

2) Management Requirements: Owing to the openness feature, a networked control application is typically subject to change and evolution after its deployment. However it is not always possible to stop the whole system to implement these changes. Therefore, it is necessary that a middleware framework provides mechanisms for runtime system management which allow continuous system evolution.

3) Non-functional Requirements: The time-critical feature of a control system requires a middleware framework to provide mechanisms which guarantee timeliness of action. Also, the safety-criticality of a control system requires that the middleware framework itself be error-free, and also provide some mechanisms to tolerate faults of an application program, so as to achieve overall reliability.

III. ETHERWARE

We begin by describing the initial capabilities of Etherware [6], that was developed at the University of Illinois. In this section, we describe the basic features and see how they satisfy the domain requirements.

A. Etherware Architecture

From the viewpoint of its software architecture, Etherware can be decomposed into roughly two parts, Kernel and Components. Components can interact with each other by exchanging objects, called Messages, each of which is a well-defined XML document object. As shown in Fig. 1, a Message contains three XML elements. The name of the receiver component of a Message is specified in Profile, the time when a Message is created is specified in Time Stamp, and any information concerning the interaction semantics can be specified in Contents.

The Kernel is indeed the main part of Etherware since it provides essential functionalities for middleware operations. A component can be created by the Kernel and also be destroyed when it is necessary. Moreover, the Kernel is responsible for delivering Messages among components. When the Kernel delivers a Message from one component to another, it first
Etherware consists of a set of components. Components can be classified further into service components and application components. Service components are the components that need to be developed by a control application developer for an application, while service components are the components that are provided as parts of the Etherware infrastructure to make it easy to develop an application. Section III-C discusses service components in more detail.

B. Etherware Component Model

To make it easy to develop Etherware components, Etherware provides a template of components, in an Etherware Component Model, which is shown in Fig. 2. As shown in the figure, it consists of roughly three different parts, Shell, Component Logic, and Component State, with each of them being designed based on several software design patterns [7] such as Facade, Strategy, Memento, respectively. The Shell is a class object that provides an interface between the Etherware Kernel and the Component Logic that is a class object which implements a user defined application logic. The Component State is a class object where the runtime execution states of the Component Logic are stored and maintained.

An Etherware component can easily be developed with the Etherware Component Model since the Shell is already implemented and provided as part of the Etherware infrastructure. Moreover, a template class is provided so that an object of the Component State can easily be created. Hence, an application developer needs to implement only an interface for the Component Logic, called Component, to develop an application Etherware component. One additional notable feature of the Etherware Component Model is that, due to the software design patterns used in designing the component model, it indeed supports runtime system management, making possible capabilities such as runtime component replacement and runtime component migration. We illustrate these two capabilities in Section VI.

C. Etherware Services

There are several functionalities that can commonly be useful in any networked control application. Etherware provides such functionalities as Etherware service components as shown in Fig. 2.

ProfileRegistry provides a naming service which enables a component to send a Message to other components without knowing the physical address of a recipient component. Instead, a sender component can specify the Profile, a semantic name of a component such as ‘controller of car 1’.

NetworkMessenger maintains the network connections between Etherwares over a communication network. It is also responsible for sending and receiving Messages over the network. Thus, all the details about the network are hidden from application components by the NetworkMessenger.

In general, every computer in a distributed system has a different time clock. To address this issue, Etherware provides the NetworkTime service component which estimates the time difference between computing nodes. It also translates the time stamp of a Message from the clock of the remote computing node to that of the local machine whenever a new Message arrives over a network from a remote computing node. In many control systems, some activities such as sensing and control action have to be taken based on time. For such situations, Etherware provides the Notifier which generates and sends time-driven Messages so that a component can be activated at the time that it is supposed to.

D. Domain Requirement Analysis

Table I summarizes the domain requirements and the functionalities supported by the existing Etherware. As shown in the table, Etherware satisfies many important domain requirements, especially the requirements relevant to the operation of a distributed application and flexible system management. However, it still needs to satisfy some critical requirements which are essential for control systems. In this paper, we focus our attention on the real-time related issues. We now discuss in more detail the design and implementation of Etherware’s mechanisms for real-time guarantees.

IV. DESIGN OF ETHERWARE MECHANISMS FOR REAL-TIME GUARANTEES

A real-time system is not a system that is fast, but is rather a system which is temporally predictable [8]. Here, what we
TABLE I

Domain Requirements vs. Etherware Implementation

<table>
<thead>
<tr>
<th>Domain Requirements</th>
<th>Etherware Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operational</td>
<td></td>
</tr>
<tr>
<td>Location transparency</td>
<td>NetworkMessenger</td>
</tr>
<tr>
<td>Hiding time discrepancy</td>
<td>NetworkTime</td>
</tr>
<tr>
<td>Semantic addressing</td>
<td>ProfileRegistry</td>
</tr>
<tr>
<td>Management</td>
<td></td>
</tr>
<tr>
<td>System evolution</td>
<td>Component model</td>
</tr>
<tr>
<td>Non-functional</td>
<td></td>
</tr>
<tr>
<td>Timeliness</td>
<td>Reliability</td>
</tr>
</tbody>
</table>

mean by the temporal predictability\(^2\) of a system is that if a task is specified with some information relevant to its execution such as release time, finish time, and so on, then a system should provide a guarantee of supporting such specifications when it executes a task. In general, the temporal predictability of an entire system relies on the temporal predictability of every subsystem, including H/W platform, communication network, operating system, etc. However, in the following discussion, we only consider the problem of how to improve the temporal predictability at the level of middleware, especially in Etherware, which is a software framework interposed between the operating system and application programs. In particular, we do not consider issues such as packet delay, jitter, and omission in the communication layer. Hence, the overall temporal predictability of Etherware-based systems is restricted by the features supported by the underlying platforms such as operating systems and communication networks.

A. Quality of Service (QoS) of Message Delivery

For the temporal predictability of the Etherware, we first need to develop a mechanism that can be used to specify the execution relevant information. In this paper, we call a collection of such information as Quality of Service (QoS). Then the first design decision is: Where does this QoS specification need to be embedded so that it can be used in the Etherware Scheduler? Notice that the Etherware is an event-driven system. Which means that a component is to be executed by a Dispatcher only when it receives a message. Hence, it turns out that the Message class object is the right place to specify QoS. Now, the second design decision is: What information is to be specified in QoS? A particular choice of a set of information that is used in defining the QoS is shown in Fig. 3. Once the QoS of a Message is specified by a component when the Message is created, then the Etherware scheduler can utilize the QoS when it makes a scheduling decision for Message delivery.

B. Priority-based and Concurrent Scheduling

In terms of temporal predictability, there are two main issues with the prior existing Etherware [6] that need to be enhanced. The first is to support a true notion of ordering among Messages when they are scheduled for delivery, rather than simply scheduling them in a first in first out (FIFO) order for delivery within a Dispatcher. For temporal correctness of behavior of the overall system, as well as to give priority to properties such as stability over other second order objectives, it is necessary to order Messages based on extra informations specified within a Message in making scheduling decision for delivery. In Section IV-A, we already have described how such information can be specified in a Message. Thus an application designer has to confront the question of how to order Messages based on the QoS specifications, i.e., what real-time scheduling policy\(^3\) is to be used. The particular choice of a real-time scheduling policy depends critically on the application that is being developed over the Etherware platform. Hence, it is desirable to design a Etherware scheduling mechanism that is independent of a specific real-time scheduling policy. That is, we separate mechanism from policy, and design Etherware specifically to provide mechanisms for real-time control, leaving the schedulability analysis to the application designer.

A second feature to support in Etherware is concurrent Message delivery. Concurrency is in fact an essential feature of any real-time scheduler since it allows preemption so that an important (with respect to some notion of ordering) Message can be delivered first before other less important Messages.

To address these issues, a real-time scheduling mechanism, called a hierarchical scheduling mechanism, is proposed as shown in Fig. 4. As shown in the figure, concurrency is supported through a module, called Dispatching Module, consisting of multiple Dispatchers with each of them executing independently to deliver Messages. Preemption between Dispatchers occurs based on the priority assigned to each of

---

\(^2\)We use the terms ‘real-time’ and ‘temporally predictable’ interchangeably.

\(^3\)In general, a real-time scheduling policy is a rule governing how to order the executions of multiple tasks that are running concurrently.
the Dispatchers if the underlying platform of the Etherware provides a priority-based thread\(^4\) scheduling. Notice that most real-time operating systems do indeed support priority-based scheduling. Moreover, the job queue in each of the Dispatchers is also modified to become a prioritized job queue so that jobs in each queue are automatically ordered based on an attribute of a job. An example of such attribute can be the absolute deadline of a Message. Thus, Messages within a Dispatching Module can be ordered by the priority of a Dispatcher and the attribute of a job. To determine the right position for a Message within a Dispatching Module, the Etherware Scheduler uses an object, called Job Placement Rule (JPR), that is comprised of the ID of a Dispatcher and the value for an attribute that is used within a job queue of a Dispatcher.

![Algorithm 1: A pseudo-code example of Job Placement Rule implementation.](image)

As mentioned above, it is the scheduling policy that decides what QoS specification of a Message to use, and how to use it, to determine a corresponding JPR. To make the scheduling mechanism independent of a specific scheduling policy, an interface is defined to communicate between the Scheduler and a module, called JPR Implementation, which is an implementation of a scheduling policy. Thus a scheduling policy can be chosen and its implementation can be provided to the Etherware’s overall scheduling process as a separate module. We note that there are many scheduling policies that can be supported via the hierarchical scheduling mechanism. For example, the rate-monotonic (RM) static scheduling policy, or a non-preemptive version of the earliest-deadline-first (EDF), or the combination of these two can all be supported. Algorithm 1 is an example of such scheduling policy implemented as a JPR implementation. In addition to the flexibility with respect to the choice of a scheduling policy, the proposed real-time scheduling mechanism also supports flexibility with respect to the configuration of the Dispatching Module. In fact, the number of Dispatchers and the priority for each Dispatcher\(^5\) can be specified as a separate rule, called Thread Scheduling Rule (TSR).

---

\(^4\) A thread of execution (or thread in short) is the smallest unit of processing that can be scheduled by an operating system [9].

\(^5\) The specific priority set is given by the underlying software platform.

V. DISCUSSION ON IMPLEMENTATION

While Etherware was originally developed using the Java programming language [10], we implement the proposed Etherware’s real-time mechanisms in the Sun Java Real-Time System (Sun Java RTS) [11]. Java provides several important advantages over other programming languages, especially when developing a software framework for distributed system applications such as Etherware. First, it is a well-designed Object-Oriented Programming (OOP) language with many useful built-in class packages for networking, data structures, multiprocessing. Second, it provides a garbage collection mechanism for automatic memory management during the execution of a program. Third, it supports platform independent application development. Thus it is easy to write a distributed software program in Java.

However, Java was originally designed and optimized for performance in terms of overall throughput rather than for temporal predictability. Specifically, the dynamic loading, linking, and initialization of classes or interfaces, just-in-time (JIT) compilation, and the garbage collection of Java Virtual Machine (JVM) could cause unpredictable delays in program execution. As stated in Section IV, the temporal predictability of the underlying platform is necessary to provide real-time performance of the overall Etherware-based networked control system. Therefore, to implement the proposed Etherware’s real-time mechanisms, we use a special version of Java Virtual Machine (JVM), called Sun Java Real-Time System (Sun Java RTS) [11], which is an implementation of the Real-Time Specification of Java (RTSJ) [12] that has been developed for better predictability of Java programs. One notable enhancement of RTSJ compared to the standard JVM specification is that it supports hard real-time thread scheduling via a fixed-priority scheduling mechanism. Hence, unlike in the standard JVM, in Sun Java RTS, a control task can be configured to run with a priority higher than those of others such as garbage collector so that it’s execution is not affected at runtime.

VI. NETWORKED INVERTED PENDULUM CONTROL SYSTEM

We demonstrate the flexibility along with the temporal predictability of the enhanced Etherware on an unstable system, the inverted pendulum control system shown in Fig. 5, when it is controlled over a network.

A. Inverted Pendulum Control System

An inverted pendulum system is chosen to illustrate the real-time performance as well as flexibility in application development made possible by Etherware, since due to its open loop instability it is representative of control system which requires strict predictability with respect to its sensing and control action. We first describe the configuration of the system.

1) System Configuration: Fig. 6 shows the schematic of the inverted pendulum control system. As shown in the figure, the inverted pendulum has two joints, and correspondingly two links attached to each joint. Joint 1 is at the base of the system, and is actuated by an attached DC motor, while joint
2 is passive. Thus, the DC motor has to be controlled in an appropriate manner to keep link 2 upright, i.e., regulate \( \theta \) close to zero based on our selected coordinate frame. A DSP board is used to read encoder values from two joints upon request by a controller. It is also used to implement a PWM signal to the DC motor based on the control value delivered from a controller.

To control such an inverted pendulum, a controller is developed as a component running on Etherware, which in turn runs on a separate PC that has a serial communication with the DSP board. In our implementation of the controller, the controller component is activated every 15 ms by the Etherware. In Etherware, a component can be periodically activated when it receives a periodic time-driven message, call a Tick, from the Notifier service. Hence, the Tick for the controller is set to a 15 ms period. At each activation, the controller first sends a request to the DSP board to read the angles of the two joints. Once it receives the data, it computes a control output value and sends it back to the DSP board. Then the DSP board delivers the control action right after it receives the control command from a controller.

Table II shows the TSR specification that is used to configure Etherware’s real-time scheduler. Dispatcher 0 is the default Dispatcher for Etherware’s operations that are not necessary to be temporally predictable, while the others are configured for real-time execution. The range of the priority levels is based on the priorities provided by the underlying Sun JavaRTS platform. The higher the number is, the higher the priority. Thus, a task executed by Dispatcher 1 has the highest priority and can preempt any other tasks executed by the other Dispatchers. In terms of the real-time scheduling policy that is used in this experiment, we use the JPR implementation as shown in Algorithm 1.

2) Controller Design: A rotating type inverted pendulum system can generally be modeled as a dynamic system in the following form [13].

\[
M\ddot{q} + C\dot{q} + G = \tau \tag{1}
\]

\[
M = \begin{bmatrix}
J_1 + m(l_1^2 + l_2^2 \sin^2 \theta_2) & ml_1 l_2 \cos \theta_2 \\
ml_1 l_2 \cos \theta_2 & J_2 + m l_2^2
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
ml_2^2 \dot{\theta}_2 \sin \theta_2 \cos \theta_2 & ml_2^2 \dot{\theta}_1 \sin \theta_2 \cos \theta_2 \\
-ml_1 l_2 \dot{\theta}_2 \sin \theta_2 & -ml_1 l_2 \dot{\theta}_2 \sin \theta_2
\end{bmatrix}
\]

\[
G = \begin{bmatrix}
0 \\
-ml_2 g \sin \theta_2
\end{bmatrix}
\]

where \( q = [\theta_1, \theta_2]^T \), \( \tau = [u, 0]^T \), \( u \) is a control input, \( J_i \) is the moment of inertia of link \( i \) about its center of mass. The values for the parameters of the inverted pendulum are \( m = 0.2(kg) \), \( l_1 = 8(cm) \), \( l_2 = 20(cm) \), \( J_1 \approx 0.0064(kg \cdot m^2) \), and \( J_2 \approx 0.0002(kg \cdot m^2) \).

Let \( x = [x_1, x_2, x_3, x_4]^T = [\theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2]^T \) be the state vector. Then the dynamics of the inverted pendulum in (1) can be represented by a nonlinear differential equation in the state space form of \( \dot{x} = f(x, u) \). Linearizing the nonlinear equations about the equilibrium point \( x_0 := [\theta_1^0, 0, 0, 0] \) for some \( \theta_1^0 \in [0, 2\pi) \) that is a set position for \( \theta_1 \), gives us the following linear differential equation,

\[
\dot{x} = Ax + Bu, \tag{2}
\]

where

\[
A = \left. \frac{\partial f}{\partial x} \right|_{x=x_0, u=u_0} \quad \text{and} \quad B = \left. \frac{\partial f}{\partial u} \right|_{x=x_0, u=u_0}.
\]

Notice that \( u_0 = 0 \) at the equilibrium state \( x_0 \), \( A \in \mathbb{R}^{4 \times 4} \) and \( B \in \mathbb{R}^{4 \times 1} \). To stabilize the linearized system (2), the control input \( u \) is designed as a full state feedback controller in the form of \( u = -Kx \). Here, the gain \( K \) is determined through the pole placement technique.

B. Periodic Control Under Computational Stress

To verify that the enhanced Etherware indeed supports temporal predictability, we control an inverted pendulum under a stress condition on the computational system. Fig. 7
illustrates how the system is configured in this experiment. As shown in the figure, we run an extra periodic task, called a stressing task component, along with a periodic control component. In our experiment, the stressing task is activated every 5 seconds and it executes its computation which takes about 1 second to finish at each activation. Furthermore, the stressing task is activated upon receiving a Tick that has QoS with low criticality, while Tick for a controller has high criticality in its QoS. Thus the system is configured so that the controller is executed in higher priority than the stressing task. However, it is important to notice that even though the system is so configured, if the executions of two periodic tasks are not scheduled appropriately by Etherware, then the inverted pendulum cannot be controlled successfully since the 15 ms period of a controller is too short compared to the 1 second average execution time of the stressing task.

Fig. 7. System configuration for the stress test where C and S represent the controller component and stressing task component, respectively.

Fig. 8 shows the serial communication signal between the PC and the DSP board that is captured while the inverted pendulum is being controlled by the controller under the computational stress condition. In each cycle of sensing and control action, the first signal from the PC is sent by the controller to request joint angles, and this request is immediately followed by a response from the DSP board to return the measured joint angles to the controller. Then the controller sends a control value that is the second signal from the PC to the DSP board. As shown in the figure, this sensing and control action cycle is indeed periodic, which implies that the execution of the controller is not affected at all by the execution of the stressing task in the PC, which in turn demonstrates that the controller is successfully scheduled by Etherware to execute at each of its execution periods regardless of the execution of the stressing task. The measured joint angles are plotted in Fig. 9. As shown in the result, the inverted pendulum is successfully stabilized at its upright position, \( \theta_2 = 0 \), even under the stressing condition. This shows that real-time temporal predictability is indeed achieved by Etherware.

Fig. 8. Periodic sensing and control action over RS-232C communication.

Fig. 9. Periodic control of an inverted pendulum under stress.

C. Runtime System Management

Now we show how Etherware makes possible the ready development of rather sophisticated application functionalities. In specific we will show that it is feasible to do runtime migration which allows real-time reconfiguration of a control system. Such a functionality can be very useful, for example, in enhancing reliability of a control systems when some component fail under operation. This is but one example of higher level functionalities which Etherware makes readily feasible. The networked inverted pendulum control system is implemented as shown in Fig. 10 to demonstrate the suitability of the enhanced Etherware as a software platform for networked control systems. More specifically, we demonstrate the Etherware’s capabilities to support runtime reconfiguration.
of a control system, such as controller upgrade and controller migration, which typically require both flexible and temporally predictable execution behavior of the underlying execution platform. One notable difference from the system configuration in Section VI-B is that a component, called DSPProxy, is run at the computational node that has a serial communication with an inverted pendulum system, to mediate the interaction between a DSP board and a controller that is typically running at a remote computing node. In fact, it is necessary to have a component such as a DSPProxy component to allow a controller to run at a remote computer.

1) Controller Upgrade: In this experiment, the goal is to upgrade a controller while an inverted pendulum is being controlled. More precisely, while an inverted pendulum is being controlled by a networked controller at a computing node 2, another component, called Requester, requests the Etherware at node 2 to upgrade the running controller component. Upon receiving such a request, the Etherware at node 2 first terminates the running controller and then initializes a new controller component to continue to control the inverted pendulum over a network. The inverted pendulum is an inherently open-loop unstable system, and it is controlled with 15 ms period. Hence, if Etherware cannot complete this upgrading procedure in a timely manner, then the link 2 of the inverted pendulum cannot be controlled to be at its upright position during this controller transition process.

Fig. 11 shows the results of the experiment. In this experiment, a Requester component initiates the controller upgrade procedure at around 30 seconds elapsed time by sending a request message. Moreover, the request message is specified with a QoS that has high criticality so that the controller can be upgraded at the highest priority level by the Etherware. As shown in the figure, it is easy to see that the controller is indeed replaced around 30 seconds, and that the inverted pendulum is successfully controlled around its upright position during this controller upgrade. Notice that, in this experiment, to clearly visualize the effect of runtime controller upgrade, the control gains of the controller before and after upgrade are intentionally tuned to have different control performance. As we can see in Fig. 11, the control performances of the replacing controller is better than that of the replaced one.

2) Controller Migration: In this next experiment, the goal is to move a controller from one computing node to another at runtime. Such a capability can be important in optimizing the behavior of control systems. For example, if network delays cause congestion, one may want to change the node where the control law is computed. As in Section VI-C1, a Requester component sends a request message to the Etherware at node 2 where a controller is running to control an inverted pendulum. However, in this experiment, the request message contains the information that requests the Etherware to migrate a controller from node 2 to node 1. To complete this migration procedure, two Etherware processes at node 1 and 2 have to interact with each other in (i) initializing a new controller at node 1, (ii) terminating a currently running controller at node 2, and (iii) migrating the runtime state of the terminated controller from node 2 to a new controller at node 1. Thus, migration is a more complicated procedure than upgrade. Hence it is more difficult to preserve the stability of the inverted pendulum while a controller is being migrated.

Fig. 12 shows the results of this experiment. In this experiment, a migration request message is sent by a Requester component at around 40 seconds elapsed time. At the moment when the migration request is received by Etherware, the controller running at node 2 is automatically terminated, migrated from node 2 to node 1, and then restarted at node 1. As shown in the figure, there are no noticeable changes in the motion of the inverted pendulum due to the timely controller migration. Thus the stability of the system is well preserved during the migration of a control component through Etherware, which in turn demonstrates that Etherware indeed provides both temporal predictability and flexibility in complex application development.

VII. Conclusion

In this paper, we have demonstrated the importance of a well-designed software framework for the development of networked control applications. We have enhanced Etherware [6] so that it can support the temporally predictable execution behavior that is necessary for safety critical applications. A notion of Quality of Service (QoS) of Message delivery is introduced, and the scheduling mechanisms of the Etherware are enhanced in such a way that a Message with a QoS specification can be delivered with the priority that is determined by the QoS of a Message.

We have also implemented a networked inverted pendulum control system to experimentally verify the real-time performance as well as the application development flexibility of the enhanced Etherware. Through experiments including the runtime controller upgrade and migration, which necessarily require both flexibility and temporal predictability of the software platform, we have demonstrated that the enhanced Etherware indeed exhibits satisfactory performance in both respects.
Fig. 12. Joint angles of the inverted pendulum during a runtime controller migration.

REFERENCES