Proceedings of the
44th IEEE Conference on Decision and Control, and
the European Control Conference 2005
Seville, Spain, December 12-15, 2005

A Middleware for Control over Networks[⋆]

Girish Baliga[∗] and P. R. Kumar[†]

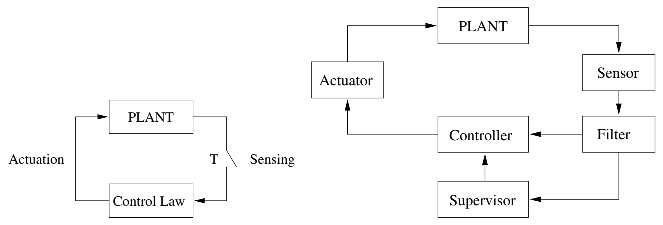
Abstract— **Our thesis is that a well-designed software architectural framework and middleware are critical for the widespread deployment and proliferation of networked control systems. We develop a list of key requirements for such middleware, and present Etherware, a message oriented component middleware for networked control. We describe the application models supported, and illustrate these vis-á-vis a vehicular control testbed. We conclude with experiments that demonstrate the support for system management and evolution in Etherware.**

I. INTRODUCTION

Much has changed in the last forty years since the advent of digital control. The convergence of communication and computing has resulted in pervasive technologies such as distributed systems, grid computing, and the Internet. However, most such systems do not yet interact with the real world, the domain of control systems. The convergence of *control* with communication and computation could accordingly be the next frontier in information technology.

Wireless networking is at a possible cusp of emerging as a key technology. It has the potential of supporting interactions between sensors, actuators, and even micro-controllers embedded in a plant, all without physical connections. With appropriate software, embedded devices may be able to automatically connect to each other, form control loops, and even self-assemble into fully functional applications on-the-fly. Further, the software in such devices can be upgraded, or even migrated, without having to physically access the associated devices. Together, these capabilities for very powerful computational hardware, wireless and wired networking, and powerful software, hold the promise of unleashing a new control revolution.

We contend that a well-designed software architectural framework is necessary to effectively leverage microprocessor and wireless networking technologies, and successfully address problems of complexity and scale in networked embedded control. Such a framework would provide a systematic approach to developing software that would be easy to integrate, manage, and reuse. It would also promote the development of libraries of "commodity" software for commonly used functionalities such as Kalman filtering and model-predictive control. A standardized architecture would then allow such software to be easily integrated into custom built systems.



(a) Periodic Digital Control (b) Software components

Fig. 1. Component based software for Digital Control

For instance, the TCP/IP protocol suite has given us the Internet [1]. Also, the use of commercial-off-the-shelf (COTS) software, and reuse of software modules and subsystems, is considered an integral part of control software development today; about 40% of the software for the Boeing 777 airplane was COTS [2]. The integration and management of such diverse software has become a major challenge to further proliferation.

Middleware is software infrastructure that has been used to successfully integrate and manage software for complex distributed systems [3]. Most middleware addresses a particular domain such as web services, and defines simple and uniform architectures for developing applications in the domain. Standard mechanisms for defining software interfaces and functionalities encourage the development of well-defined and reusable software. An appropriate middleware would allow software components such as a Kalman Filter to be integrated as easily as a standard PID controller. Further, middleware services can provide standard functionalities such as support for robustness and fault tolerance, which can be easily reused in most applications.

In this paper, we present *Etherware*, a message oriented component middleware for networked control. The application software in an Etherware based system is composed of *components*, which are autonomous software modules with well-defined functionalities. To the control engineer, Etherware also delivers the abstraction of *virtual collocation* so that she need not deal with the vagaries of a distributed system. In particular, it abstracts away details associated with network addresses, topologies, time synchronization, configuration, allocating computation to computers, optimization resources by reconfiguring and migrating software.

[⋆]This material is based upon work partially supported by NSF under Contract Nos. NSF ANI 02-21357 and CCR-0325716, USARO under Contract Nos. DAAD19-00-1-0466 and DAAD19-01010-465, DARPA/AFOSR under Contract No. F49620-02-1-0325, DARPA under Contact Nos. N00014-0-1-1-0576 and F33615-0-1-C-1905, and AFOSR under Contract No. F49620-02-1-0217.

[∗]CSL and Dept. of CS, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA. email: gibaliga@uiuc.edu.

[†]CSL and Dept. of ECE, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA. email: prkumar@uiuc.edu.

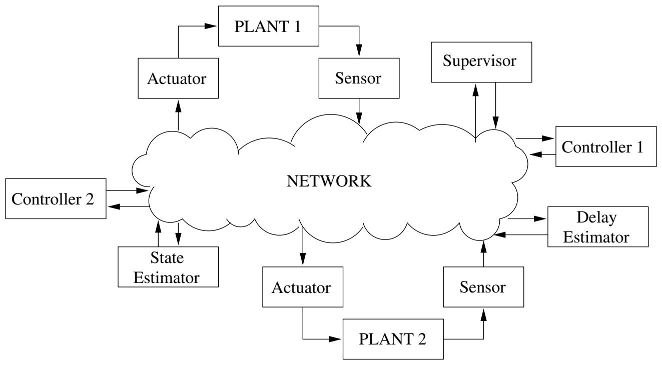


Fig. 2. A Networked Control System

II. Networked Control Software

We begin by addressing what are the basic abstractions for networked control, and discuss the requirements that they impose on middleware.

A. Abstractions for Networked Control

Figure 1(a) shows a standard representation of a control system. A simple implementation of this system would be a modular software program implementing the control laws and plant interfaces. However, established practice in software engineering [4], [5], [6] advocates architecting software based on *components*, as shown in Figure 1(b), A *component* is an autonomous software module with well-defined functionality, e.g., Sensor, Actuator, Controller, Filter, and Supervisor.

A component based software architecture has several benefits. Since components are well-defined, they can be replaced without affecting the rest of the system. For instance, a zero-order hold filter in Figure 1(b) can be dynamically replaced by a Kalman filter without having to change the remaining software or restart an operational system. A component architecture also allows individual components to be developed separately and easily integrated later, which is very important for the development of large systems. Further, such architecture promotes software reuse, since a well designed component such as a control algorithm, tested for one system, can be easily transplanted into another similar system.

As illustrated in Figure 2, control loops may be distributed. Networked control systems have software components executing on multiple computers connected over a network. Unlike in traditional digital control, it is practically impossible to guarantee periodic communication with hard real-time deadlines over best effort wireless or IP networks. Hence, it is a very important goal to provide appropriate software abstractions that allow digital control theory to be usable in such systems.

Virtual collocation [7] is the abstraction that all software components execute on a single computer. It hides details about network locations, topology, and communication protocols, and the control software does not have to distinguish between local and remote components. Such an abstraction has several additional benefits. Since no assumptions are made about network locations, the same components can be reused in different systems and over different networks. In fact, components can even be dynamically migrated for optimizing processor and network usage. Consequently, virtual collocation is not only a simplifying abstraction for control engineers, but also a very useful construct for improving system integration, management, and reuse.

Local temporal autonomy [7] is a method for enhancing robustness and reliability. It consists of ensuring that a component can execute for a small interval of time even after another connected component has failed. For instance, using a state estimator as a buffer can help a controller tolerate delayed or lost updates from a sensor. Upon a software failure, if the sensor is restarted quickly enough, then the controller can continue executing without being aware of this. These and other mechanisms [7] allow network delays and losses to be tolerated, and help enhance the abstraction of virtual collocation.

B. Architectural considerations

Supporting virtual collocation and reusable software imposes many requirements on the middleware.

1) Modularity: Most good systems are modular, and system software is composed of well-defined components. This promotes software reuse. Components and their replacements have therefore to be compatible, and interact with similar components through compatible protocols.

2) Location independence: We believe that to enable future proliferation of networked control systems it is important to reduce design-cycle time by reducing the time spent by human designers. Assigning computation to processors is a low-level task that requires detailed information about system configurations. Such allocation and optimization must be done automatically by the middleware since it has access to configuration details, To enable such location independence, it must support components being allocated to different computers in different scenarios.

3) Virtual collocation: In a networked system, software components execute on different computers with potentially unsynchronized clocks [8]. Also, component network locations must be tracked. Such mechanisms must be transparent to component software.

4) Robustness: Key middleware mechanisms such as efficient component restarts are necessary to complement techniques such as local temporal autonomy.

5) Manageability: Middleware must support proper initialization upon *startup*, and management of components.

6) System evolution: Support for *system evolution* is necessary to promote the longevity of networked control systems. For instance, a plant upgrade may require an associated controller to be *updated* as well.

7) Component migration: Component *migrations* can automatically optimize the system configuration while it is running, by reducing or balancing communication and computational loads. The designer should not have to worry about details of resource allocation and optimization.

In the rest of this paper, we discuss how these features are supported in Etherware.

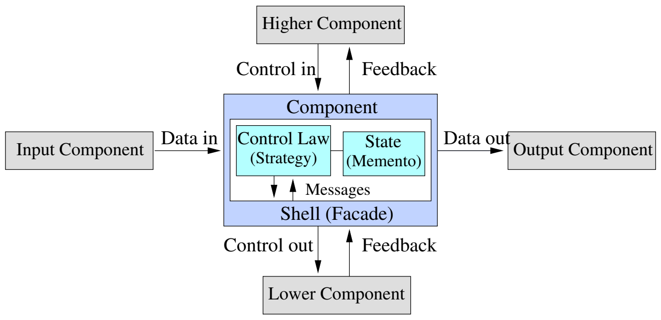


Fig. 3. Generic component in an Etherware based application

III. ETHERWARE

This section describes the application model supported by Etherware, the basic architecture and services provided, and a description of how middleware requirements are addressed.

A. Application Model

Etherware is a message-oriented component middleware for networked control systems implemented in Java [9]. Etherware based applications are composed of components that interact with each other by exchanging messages delivered through Etherware. A queue of messages is maintained for each component. A component is usually passive and is activated when messages arrive in its queue.

A generic Etherware based component is illustrated in Figure 3. A component such as a Controller may participate in a control hierarchy receiving set-points from a higher-level Supervisor and sending controls to a lower-level Actuator.

As shown in Figure 3, a component typically encapsulates a control law, called a *Strategy*, which represents its functionality. This could be equations for a control law, a plant model, or a state estimator, or device drivers that interact with physical sensors and actuators. A component also has a well-defined *Memento* that represents its state at any given time. Strategies and Mementos constitute the application software that an engineer would implement for a component. Also, each component is encapsulated by a *Shell*, which acts as a *Facade* and provides a simple interface to interact with the rest of the system. We note that Strategy, Memento, and Facade are well known and widely used software design patterns [10].

Components are identified by one or more descriptions called *Profiles*. Etherware assigns a unique profile, called a *Binding*, to each component by default. A component may also register additional profiles with Etherware describing its various functions. Other components can then identify and communicate with this component using any of its registered profiles, allowing addressing by attributes rather than IP addresses. For instance, a vision based sensor could register a profile specifying that it is a gray-scale camera covering a region between the points (0,0) and (100,50). Etherware can match requested and available profiles and automatically forward the connection request.

Messages are well-formed XML [11] documents with the following constituents:

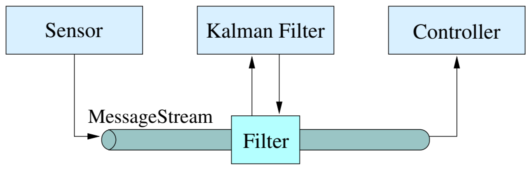


Fig. 4. MessageStreams and Filters

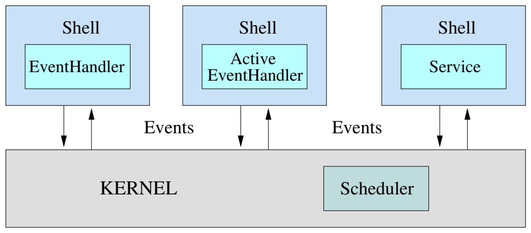


Fig. 5. Architecture of Etherware

- *Profile:* This identifies the message recipient.
- *Content:* This contains the application related content.
- *Time-stamp:* This records the time of message creation.

Etherware delivers messages reliably and in order, by default. However, different classes of messages may need to be delivered using different QoS specifications. To address such requirements, Etherware supports the notion of a *MessageStream*, which identifies a stream of messages from one component to another, as in Figure 4.

Further, it may be necessary to modify messages in a MessageStream during system operation. To address this, Etherware also supports the addition of one or more *Filters* to MessageStreams. Filters are basically components that process messages in the MessageStream before they are delivered to the recipient, see Figure 4.

Finally, components in Etherware can be passive or active. *Passive components* are only activated by incoming messages. *Active components* execute even when no messages have been received, and in particular, can pro-actively generate messages during such activity. For instance, an acoustic sensor component that polls on a physical sensor device could be implemented as an active component.

B. Etherware Architecture

The architecture of Etherware is based on the micro-kernel concept [12], as shown in Figure 5. Each Etherware based software process [12] has a simple *Kernel*, which represents the minimum invariant in Etherware. The Kernel manages all components in the process and delivers messages between them. All other middleware functionality is implemented as service components. This allows such services to be dynamically restarted, upgraded, and managed just as application components in the system.

The Kernel also has a *Scheduler* as shown in Figure 5. This schedules all the messages delivered by the Kernel. The Scheduler also manages additional threads of control [12] for active components. In addition, the Scheduler also provides a notification service that is very useful for passive components. In particular, components can register to receive the following two types of notification messages:

- *Tick:* A tick is part of a stream of messages that are generated at periodic intervals. It is very useful for standard periodic sampling, for instance.
- *Alarm:* An alarm is a one-time notification message, and is generated after a pre-specified delay. For instance, a sensor component waiting for the initialization of a physical device can register to be woken up by an alarm after a required interval of time. This is useful for time driven activity.

This notification service allows most components to be implemented passively, perhaps in response to a tick stream. This has significant benefit as it simplifies the implementation of components that would otherwise have to have complex multi-threaded design. Passive components are easier to manage since their operation is sequential and their state is well-defined.

Finally, all components are encapsulated in Shells that maintain their information. In particular, Shells maintain check-pointed states of components that are then used to reinitialize replacements during component restarts and upgrades. Shells are also responsible for creating and maintaining MessageStreams between components.

C. Etherware Services

Most of the Etherware functionality is implemented in service components, shown in Figure 5, described below.

1) Profile registry: Each Etherware process has a *Local* Profile Registry that registers profiles of components in the process. If a profile cannot be matched in a Local Registry, then the message is forwarded to a *Global* Profile Registry that registers all profiles in the system.

2) Network messenger: The Network Messenger delivers messages that are addressed to components on remote computers. It encapsulates the network from the rest of Etherware, so that none of its other constituents need be aware of details such as network addresses, ports, and transport layer protocols used for communications.

3) Message delivery: Message delivery in Etherware is a composite service achieved by the collaboration of the Kernel, Profile Registry and Network Messenger services. First, components are identified by *delivery addresses* in Etherware that consist of two parts:

- Unique component *Binding* created by Etherware.
- Network address used by the network messenger to communicate over the network.

Second, a message always has the delivery address of the sender appended by the sender's Shell, which is then buffered by the recipient's Shell. Also, delivery addresses are registered by Shells when MessageStreams are established between their encapsulated components. So, if a Shell knows the receiver's delivery address then that is added to the message as well. Third, the Kernel only understands delivery addresses with the local (default) network address. So, if a message has a receiver's delivery address with a local network address *and* the binding of a local component, then it is forwarded to that component. But, if the network address is not local then the Kernel forwards it to the Network Messenger which in turn forwards the message over the network. However, if the message has no receiver address, then the message is forwarded by default to the Profile Registry. Finally, since the network is encapsulated by the Network Messenger, neither the Kernel or any of the other components understand network addresses.

4) Network time service: The Network Time Service (NTS) translates time-stamps, as messages move between different computers with distinct clocks [8]. The mechanism used for the actual translation of message time-stamps may be of some interest. The local NTS component is simply added as a Filter for all messages sent and received by the Network Messenger component.

D. How Etherware addresses middleware requirements

1) Modularity: Etherware based applications are composed of sets of components, ensuring component level modularity. The requirement of Mementos for components ensures that component state is well represented. The message passing component model in Etherware allows component interactions and communicating protocols to be easily represented using formalisms such as communicating finite state machines [13].

2) Location independence: Network related details are abstracted away by the Network Messenger Service. Hence, all other components can be potentially located on any computer in a system. In fact, reuse of components in different systems does not require any software changes with respect to network related information.

3) Virtual collocation: In Etherware, the application models do not distinguish between local and remote components. Consequently, all components operate under the abstraction that they are virtually collocated.

4) Robustness: Shells encapsulate components and handle all exceptions caused by their software failures. Upon such a failure, a component can be restarted using its current state. MessageStreams are maintained across restarts. Hence, component failures are effectively isolated in Etherware.

5) Manageability, evolution, and migration: The Kernel provides a service interface which accepts messages for component creation, upgrade, and migration.

IV. ETHERWARE BASED DESIGN

Through our vehicular control testbed we illustrate how networked controlled systems can be implemented using Etherware.

A. Vehicular control testbed

The testbed consists of a set of remote RC controlled cars. Each radio transmitter is connected to the serial port of a dedicated laptop through a micro-controller. Commands can be updated every 40ms. The cars are monitored using a pair of ceiling mounted cameras. The image processing computers are connected by a wired Ethernet. The laptops

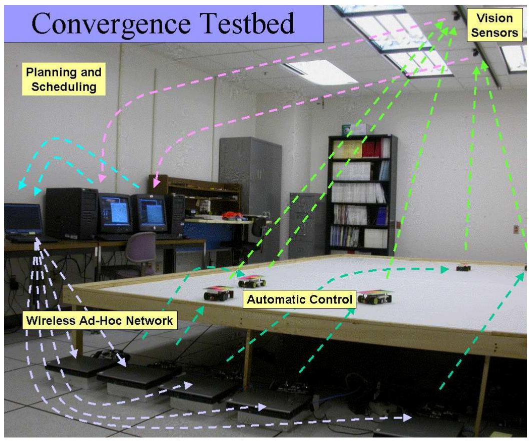


Fig. 6. Traffic Control Testbed

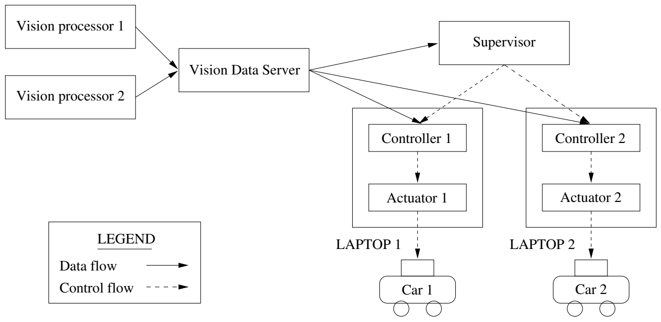


Fig. 7. Software architecture of the testbed

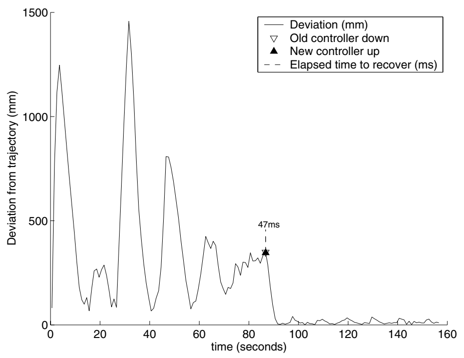


Fig. 8. Error in car trajectory due to controller upgrade

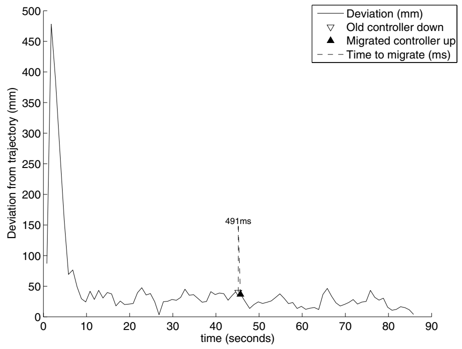


Fig. 9. Error in car trajectory due to controller migration

are also connected by an ad hoc wireless network using IEEE 802.11 [14] PCMCIA cards. Videos documenting several scenarios of interest can be viewed at [15].

B. Testbed design using Etherware

The Etherware based software architecture of the traffic control testbed is illustrated in Figure 7. There are two Vision Processor components. Car positions and orientation information is accumulated and merged in a Vision Data Server component. The Controller and Actuator components for each car typically execute on the corresponding laptop. All other components usually execute on separate computers and communicate over the network.

There are two control loops in the system depicted. The lower level loop involves the Controller which takes a trajectory as input, and computes a corresponding sequence of controls which are sent to the Actuator. In the higher level control loop, the Supervisor computes the desired car trajectories and sends them to the Controller.

All the components shown have been implemented as individual passive components that subscribe to the notification service for periodic operation. Arrows shown correspond to MessageStreams. All components other than the Vision Processors and Actuators can be executed on any of the machines without having to change a single line of component code. All software failures of all components are contained through restarts, and components can be dynamically upgraded and migrated as demonstrated next.

V. EXPERIMENTS

A. Controller Upgrade

In the first experiment, we evaluate the performance of software upgrade mechanisms in Etherware. The car is initially controlled by a coarse Controller, which is then dynamically upgraded to a better Controller while the car continues to operate. Videos documenting these experiments can be viewed at [15]. The deviation of the car from the desired trajectory, as a function of time, is shown in Figure 8. The controller is upgraded at about 90s into the experiment, and the car subsequently tracks the trajectory with very small error. The upgrade took about 47ms, well within the allowed 100ms. All MessageStreams to the controller are maintained during this upgrade.

Such efficient component upgrade is due to three key Etherware mechanisms. First, the component based design and the Strategy pattern allow one controller to be replaced by another without any changes to the rest of the system. Second, the Shell is able to upgrade the Controller without affecting the connections to the other components. Finally, the state check-pointing mechanism due to the Memento pattern allows the coarse Controller to check-point its state before termination. This is then used to initialize the new Controller. The first mechanism allows for simple upgrades, while the other two mechanisms minimize the impact of the upgrade on the rest of the system.

B. Component Migration

In the second experiment, the support for migration in Etherware is evaluated. The car Controller is initially operating on a different computer from the Actuator. During the experiment, the Controller is dynamically migrated to the same computer as the Actuator, while the car continues to operate. The error in the car trajectory is shown in Figure 9. The error introduced due to controller migration, which occurs about 45s into the experiment, is well within the operational error permitted for the car.

Two Etherware mechanisms enable Migration. First, the Memento pattern allows the current state of the controller to be captured upon its termination. Second, the primitives in the Kernel on the second computer allow a new controller to be created there with the captured state of the old controller.

These two experiments demonstrate the effectiveness of Etherware mechanisms that support robust software management and evolution in networked control systems.

VI. RELATED WORK

CORBA [16] is probably the most popular middleware and has been used in a variety of different domains. In particular, popular versions of middleware for control applications are based on various flavors of CORBA such as Real time CORBA [17] and Minimum CORBA [18]. For example, OCP [19] is based on Real Time CORBA and has been used to control unmanned aerial vehicles. ROFES [20] implements Real Time CORBA and Minimum CORBA, and is targeted for real-time applications in embedded systems.

A key problem in using CORBA based middleware is that the CORBA interface description language (IDL) is not descriptive enough to sufficiently specify component interfaces. In particular, component interaction protocols cannot be specified. This leads to numerous problems while integrating independently developed, yet functionally compatible components.

Another issue is that the CORBA was primarily intended for transaction-based business and enterprise systems. Hence, the trade-offs incorporated in the design of CORBA are not all compatible with requirements for control systems. For example, the specification mandates the use of TCP [21] for reliable delivery. This can be a serious limitation if lower delay is more important than reliability, since such a trade-off cannot be supported.

Other interesting approaches include Giotto [22] for time triggered systems and real-time framework [23] for robotics and automation. A good overview of research and technology that has been developed for implementing reusable distributed control systems software is provided in [24].

VII. CONCLUSIONS AND FUTURE WORK

We have made the case that middleware is a crucial technology for the further proliferation of networked embedded control systems. We have argued for the importance of good software architecture in such systems, and presented some simplifying abstractions for such architecture. We have developed key requirements in middleware for these abstractions, and presented Etherware as a message-oriented, component architecture-based middleware that addresses these requirements. We have also illustrated Etherware based application design through our vehicular control testbed.

More services such as control-loop delay estimation, system monitoring services, etc, commodity components such as standard controllers and state estimators, real-time scheduling algorithms, as well as formalisms that will allow us to model, verify, and deploy prototypes and systems, etc. are needed. Some of these are on the anvil in the Convergence Lab at the University of Illinois.

REFERENCES

[1] D. E. Comer, *Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture*, 4th ed. Prentice Hall, Jan 2000.

[2] R. J. Pehrson, "Software development for the boeing 777," The Boeing Company, Tech. Rep., 1996.

[3] *CORBA Success Stories*, OMG Inc, http://www.corba.org/success.htm.

[4] *CORBA Components, Version 3.0*, Object Management Group Inc, June 2002.

[5] V. Matena and B. Stearns, *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform*, ser. Java Series. Sun Microsystems Press, Jan 2001.

[6] J. Lowy, *Programming .NET Components*, 1st ed. O'Reilly, Apr 2003.

[7] S. Graham, "Issues in the convergence of control with communication and computation," Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, 2004.

[8] S. Graham and P. R. Kumar, "Time in general-purpose control systems: The control time protocol and an experimental evaluation," in *Proc. of the 43rd IEEE Conference on Decision and Control*, Dec 2004, pp. 4004–4009.

[9] "Java 2 platform, standard edition (j2se 5.0)," http://java.sun.com/j2se/.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, ser. Professional Computing Series. Addison-Wesley, 1995.

[11] *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C - World Wide Web Consortium, October 2000.

[12] *Applied Operating System Concepts*, 1st ed. John Wiley and Sons Inc, 2000.

[13] D. Brand and P. Zafiropulo, "On communicating finite-state machines," *J. ACM*, vol. 30, no. 2, pp. 323–342, 1983.

[14] IEEE 802 LAN/MAN Standards Committee, "Wireless LAN medium access control (MAC) and physical layer (PHY) specifications," IEEE Standard 802.11, 1999 edition, 1999.

[15] "It convergence lab, csl, uiuc," http://decision.csl.uiuc.edu/ testbed/.

[16] *Common Object Request Broker Architecture: Core Specification, Version 3.0.3*, Object Management Group Inc, March 2004.

[17] *Real-Time CORBA Specification Version 2.0*, OMG, Inc, Nov 2003.

[18] *Minimum Corba Specification*, OMG, Inc, Aug 2002.

[19] L. Wills and et. al., "An open platform for reconfigurable control," *IEEE Control Systems Magazine*, June 2001.

[20] "Rofes: Real-time corba for embedded systems," http://www.lfbs.rwth-aachen.de/users/stefan/rofes/.

[21] T. Socolofsky and C. Kale, *RFC 1180 - TCP/IP Tutorial*.

[22] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Proc. 1st Int. Workshop Embedded Software (DMSOFT '01)*, Tahoe City, Ca, 2001.

[23] A. Traub and R. Schraft, "An object-oriented realtime framework for distributed control systems," in *Proc. IEEE Conference on Robotics and Automation*, Detroit, Mi, May 1999.

[24] B. S. Heck, L. M. Wills, and G. J. Vachtsevanos, "Software technology for implementing reusable, distributed control systems," *IEEE Control Systems Magazine*, vol. 23, no. 1, February 2003.