IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922 © 2004 Published by the IEEE Computer Society
Vol. 5, No. 9; September 2004

Service Continuity in Networked Control Using Etherware

Girish Baliga, *University of Illinois at Urbana-Champaign*
Scott Graham, *University of Illinois at Urbana-Champaign*
Lui Sha, *University of Illinois at Urbana-Champaign*
P.R. Kumar, *University of Illinois at Urbana-Champaign*

Safety requirements and the ability to tolerate changes make service continuity crucial for networked control systems. The authors describe how their middleware for networked control, Etherware, addresses these issues.

The ability to tolerate change is a fundamental quality of a sustainable dynamic system. A system's ability to respond to change in its operating conditions determines, to a large extent, its useful lifetime. However, it's impractical to anticipate all changes before you deploy a system. Instead, systems are often designed to be able to evolve and adapt dynamically. Software systems are particularly malleable because of the "program as data" concept that the von Neumann architecture introduced.[1] Indeed, developers have created systems software such as operating systems and middleware to manage application software. This is the basis of most dynamically evolvable software systems today.

Our focus is on networked control systems comprising sensors, actuators, and computers that coordinate over a network to manage distributed real-time systems. Because these systems directly interact with the real world, they're subject to drastic and often unpredictable changes. Hence, maintaining such systems' operational integrity requires containing the changes' effects. Further, developers usually build these systems as networks of components that interact by providing and consuming services. Therefore, service continuity is a basic requirement that must be addressed for the viability of such applications.

In this article, we first develop the notion of operational integrity for networked control and identify the main challenges of maintaining it. Second, we identify several middleware issues involved in operational integrity, consider their influence on the design of Etherware—our middleware for networked control—and describe how Etherware supports service continuity. A key feature of Etherware is the ability to maintain communication channels during component restarts and upgrades.

Operational integrity in networked control

Networked control software interacts with a distributed real-time system, usually called a *plant*. Sensors provide feedback about the plant behavior. Computers running control programs use this sensor feedback to generate actuator commands that accomplish specified goals. Actuators implement these commands to control the plant. Sensors and actuators constitute the interface between the software and the real world (see Figure 1). Also, the controller is typically implemented as a set of software components operating over a network of computers.

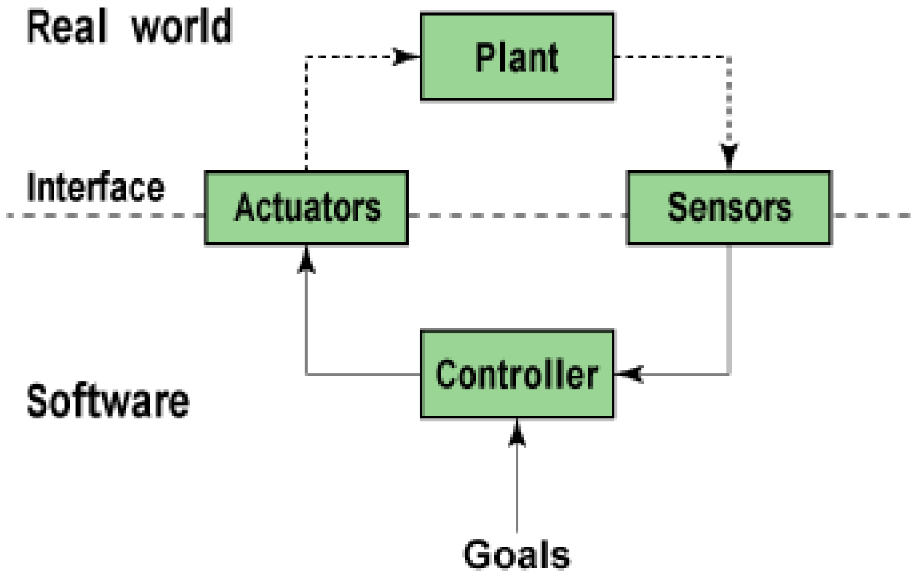

Figure 1. Schematic of a networked control system.

The goals that the controller receives specify the objectives to be achieved during plant operation. An important part of these goals is a set of safety criteria that ensures that the plant operates in a "safe" region. For example, in a traffic control system, a safety criterion would be to maintain a given distance between any two cars, ensuring against collisions. In general, safety criteria depend on specific applications and are usually part of the system specification. In this context, *operational integrity* is the property that the system always meets its safety criteria while still maintaining its operational capability, which is its ability to meet specified goals.

Operational networked control systems are subject to many changes, classified as *voluntary* or *involuntary*. Voluntary changes such as component upgrades and configuration changes are intentionally introduced by a system operator and might affect one or more of the following:

- *Syntax.* An upgraded component might require syntactic changes such as additional functions or parameters in a service interface. For example, an upgraded controller might require additional information in the updates it gets from the sensors.

- *Semantics.* Changes in operating conditions or upgrades might trigger changes in operational semantics. For example, on detecting a safety violation between two coupled subsystems, the respective controller components might communicate directly to avoid it. You might need to upgrade such fault avoidance algorithms as the system evolves.

- *Communication.* You might need to add components to established communication channels at runtime. For example, if updates from a sensor are too noisy, then you might need to add a filter to reduce this noise. Components might also want to change communication channels' quality-of-service parameters. For example, components in wireless networks might want to trade reliability for lower delay, as we've shown in earlier work.[2]

- *Timing.* Changes in operating conditions could cause related changes in timing requirements. For example, the controller might need updates at a higher frequency when approaching critical conditions.

- *Location.* You can migrate components to better use resources. For example, if feedback from a sensor is a lot more frequent than any other communication involving a controller component, you can migrate the component to the same node as the sensor to reduce network traffic.

Environmental events beyond operator control cause involuntary changes, such as

- *Passive failures.* These include component crashes owing to exceptions, node faults, and failure in communication links.

- *Active failures.* These are usually caused by misconfigured or erroneous components. Also, system managers might make mistakes during system reconfiguration. Malicious components cause *Byzantine failures*, which are usually the hardest to cope with.

To maintain operational integrity, you must handle these changes dynamically and minimize their impact on the system's operational capability. Notably, active and Byzantine failures require semantic information and must be handled by application-specific mechanisms.

Maintaining service continuity

A system designer typically creates middleware-based systems as a set of coordinating components that interact by providing and consuming services. For networked control systems in particular, some of these services might be critical for a component's operation. For example, the controllers in Figure 1 depend on getting feedback from the sensors. Controls are usually discrete and calibration is imperfect. Therefore, any changes in this feedback service could result in serious faults in controller operation. Hence, the feedback service's continuity is imperative to the controller's operation. Similarly, the actuators depend on receiving controls from the controller.

Networked control systems have fairly strict safety requirements, so components must respond to changes as soon as possible. For example, on detecting a safety violation, a controller component might not be able to wait for an acknowledgment from another remote component before it decides to take a safe action. On a wireless channel in particular, delays can be fairly large because of interference and fluctuating channel conditions. To maintain operational integrity, components must be able to operate asynchronously.

Another important consideration is the presence of dependencies in push-based communication channels. For example, a controller can't wait for updates from the sensor before sending controls to the actuator. In particular, updates might be delayed or lost because of communication failures in a wireless link. Synchronous communication would require such components to be multithreaded or to use a fairly complex design involving poller objects for each sensor. Asynchronous operation, on the other hand, eliminates this source of complexity. Based on these considerations, we've developed Etherware as an asynchronous message-based middleware.

Key components might terminate owing to voluntary upgrades or involuntary failures. To maintain operational integrity, however, components must be restarted and operational within application-specified deadlines. For example, if a controller is restarted, it should know the plant's current state, as this information might not be entirely available from sensor feedback. Checkpointing is a common technique for addressing this requirement. Necessary state is checkpointed so components can be restarted appropriately. To support this, we've closely integrated checkpointing with Etherware design and provided it as a basic service.

Components typically maintain several communication channels with other components. For service continuity, restarting or updating such components shouldn't require channels to be reestablished. Consequently, Etherware also supports maintaining communication channels across these changes. In particular, you can save identifiers for communication channels as part of checkpointed state. This lets restarted or upgraded components continue using previously established channels. It also provides communication continuity to other components during these changes.

The need to support efficient components restarts has motivated another basic design choice in Etherware. A single-kernel process manages all components on a given node. Components can have separate threads if necessary. Furthermore, services the middleware provides also must be easily restartable and upgradable. Also, invariant aspects of the middleware that can't be changed dynamically must be minimized for maximum flexibility. This motivated us to adopt a microkernel-based design for Etherware.[3] This philosophy of flexibility has also resulted in the development of a bare minimum, functional interface for components to interact with the middleware. For flexibility and uniformity, all interaction with middleware services is message based.

Etherware

Etherware is messaging middleware implemented in Java for portability. In an Etherware-based application, components communicate by exchanging messages, which are XML documents.[4]

Etherware provides a hierarchy of Java classes that application components can use to manipulate these documents. The hierarchy's root is the Message class that provides various primitives to manipulate the underlying XML document. Application-defined messages are required to be Message subclasses.

Figure 2a shows a generic component's interactions in an Etherware-based network control application. Components participate in control hierarchies such as between a supervisor and a controller (see Figure 2a) and in data flows, such as from sensors to controllers. We based the design in Figure 2a on design patterns that include[5]

- *Memento.* Support for restarts and upgrades requires the ability to capture component state. The Memento pattern addresses this, wherein component state can be checkpointed and restored on reinitialization.

- *Strategy.* Service continuity requires the ability to replace components without disrupting service. In particular, if the functional interface the system uses to communicate with the component is invariant, Etherware can replace components dynamically. In this case, it uses the Strategy pattern along with the Memento pattern.

- *Facade.* Interaction with various services in middleware usually requires a component to invoke different subsystems. This may lead to unnecessary dependencies between the component and middleware subsystems. Using the Facade pattern to provide a uniform middleware service interface for the components eliminates this.

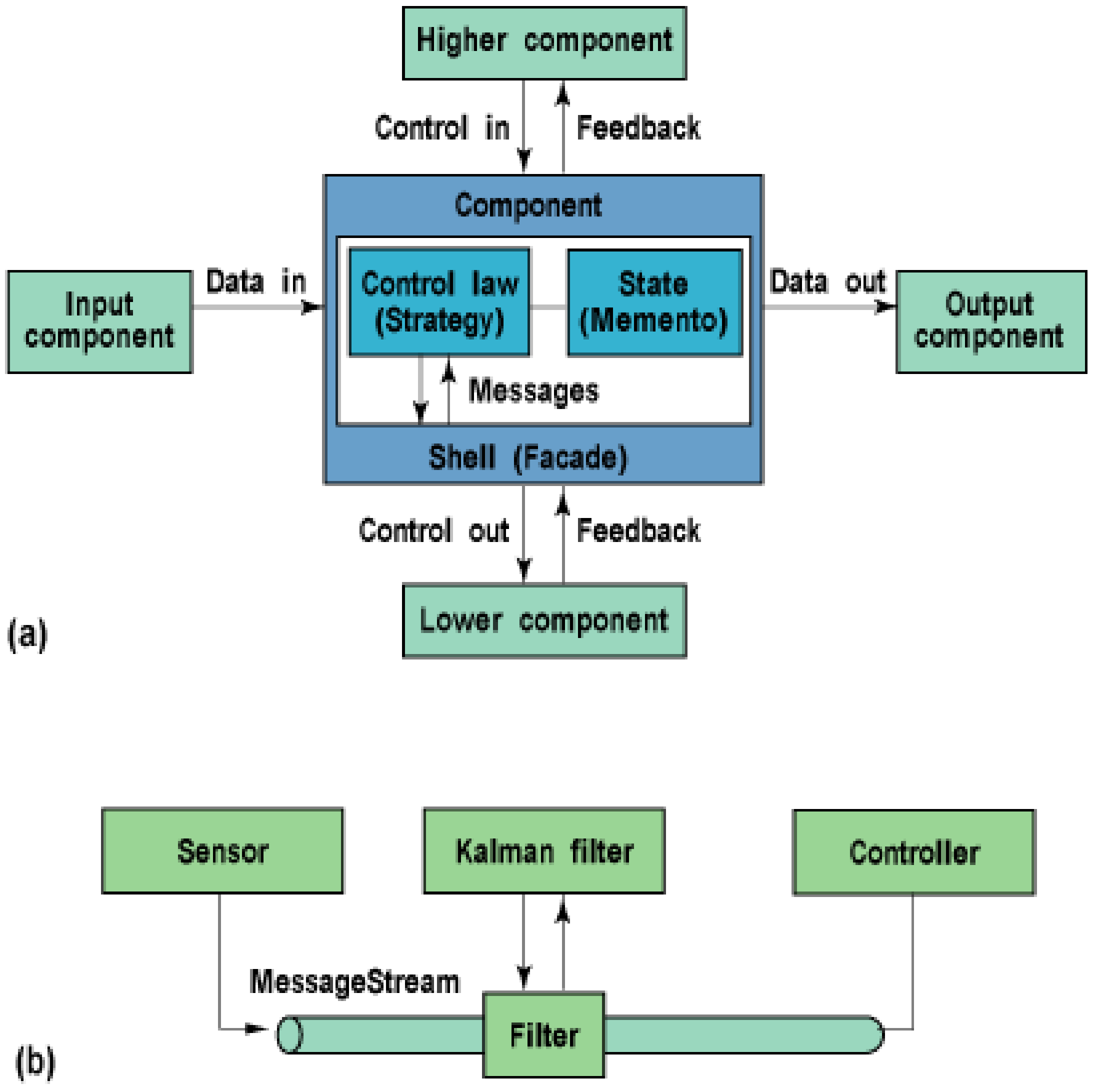

Figure 2. An Etherware programming model: (a) programming model of an Etherware component and (b) filters for MessageStreams.

Etherware components can be *active* or *passive*. Passive components don't have active threads of control. They only respond to incoming messages by processing them appropriately and generating resulting messages, if any. Active components, on the other hand, can have one or more active threads of control. They can generate messages based on activities in their individual threads of control.

Message delivery in distributed systems face two basic problems: discovering and identifying destination components. Associating profiles to addressable components solves the discovery problem. Each component that must be addressed registers a profile with the middleware. We solve the identification problem in Etherware by associating a globally unique ID, called a *binding*, to each component.

A profile is an XML-based description of a service a component provides. For example, a vision sensor profile could specify that the sensor is a grayscale camera covering the region between the points (0,0) and (100,50) in an appropriate coordinate space. Suppose a controller is only interested in the location between coordinates (25,37) and (45,52). It could use this information to connect to a relevant vision sensor, using an appropriately defined profile request. Etherware would then match this profile with the sensor's profile and forward the connection request to it. In general, we've defined suitable matching rules to match profiles.

Incorporating the features we've mentioned, all messages in Etherware are XML documents and have three constituent XML tags. *Profile* identifies the message's recipient and can be a service description (as mentioned) or a component's globally unique binding. *Content* represents the message's contents and contains all application-specific information. A *time stamp* is associated with each message. As a message is transferred from one node to another, the time stamp automatically translates to the destination's local time.

By default, messages are delivered reliably and in order. However, there might be streams of messages that must be delivered using other specifications. For example, a controller might tolerate a few lost sensor updates for lower delays and need to trade some reliability for lower delays. To identify and manipulate a stream of messages as a separate entity, Etherware introduces the notion of a MessageStream. A component can open a MessageStream to another component and send messages through it. MessageStreams have settings that you can use to specify how messages are delivered through them.

Changes in operating conditions might also require modifying messages in a MessageStream. For example, updates from the vision sensor could get noisy because of bad lighting conditions. We should be able to filter out this noise without having to modify the sensor or the controller. Etherware supports this by adding Filters to MessageStreams dynamically. Figure 2b shows the effective configuration after adding a Kalman filter to the MessageStream between a sensor and a controller. You can also add filters to intercept all messages sent to or received by a component.

Etherware's architecture is based on the microkernel concept (see Figure 3). The kernel manages all components in a single process and represents the minimum invariant in Etherware. In the current implementation, we have one Etherware process per node. The kernel's basic function is to deliver messages between its (local) components. It also exposes a service interface, which can be used to manipulate the components it manages. The kernel has a scheduler that's responsible for scheduling all messages and threads and can be replaced dynamically.

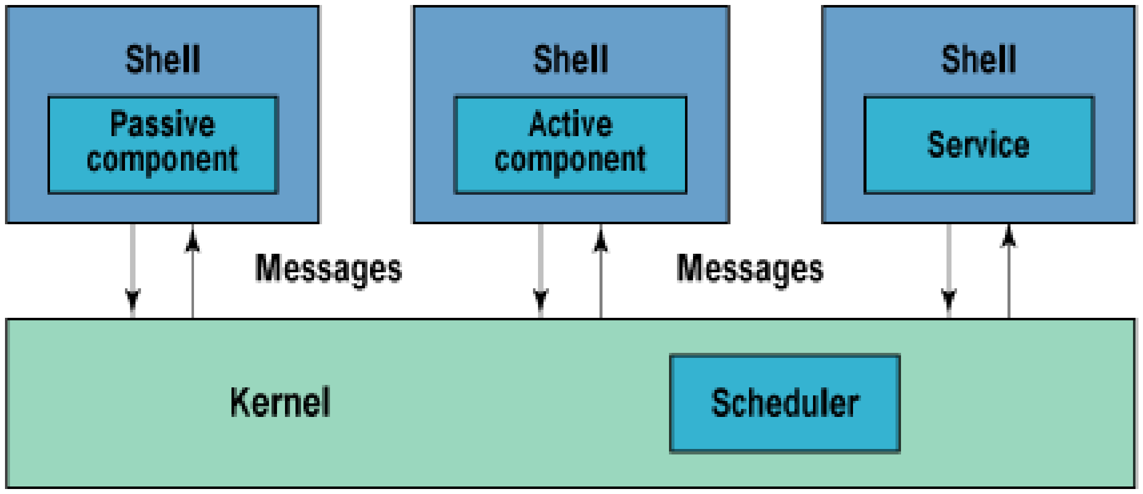

Figure 3. An Etherware process configuration.

As Figure 3 illustrates, each component is encapsulated in its own shell. A shell presents a facade to the component and provides a uniform interface for it to interact with the rest of the system. Shells also encapsulate component-specific information, such as MessageStream configuration, and handle activities involved in component restart, upgrade, and migration.

Service components provide all other functionality in Etherware. The following basic services are used during Etherware's normal operation:

- *ProfilerRegistry*. This service is used to register and look up components' profiles. It's equivalent to a name service.

- *NetworkMessenger*. The NetworkMessenger encapsulates all communication with remote nodes over the network, including details such as IP addresses, ports, and transport layer protocols. The kernel forwards all messages addressed to remote components to the NetworkMessenger. This is an active component with separate threads to receive components from remote nodes.

- *NetworkTimeService*. This service is used to translate time stamps of messages as they're transmitted from one node to another. To implement this, the NetworkTimeService is added as a filter for all messages that are sent to and received from the NetworkMessenger. Time translation is based on computing clock offsets using the Control Time Protocol.[6]

- *Ticker*. Mostly used by passive components for periodic activations, the Ticker can also send one-time alarm messages. It's also an active component.

Because Etherware has these services implemented as components, it can also restart or update them dynamically.

Service continuity using Etherware

Etherware supports service continuity during most of the changes we've listed in the following ways:

- *Restarts, upgrades, and migration.* Externalization of the component state is supported using the Memento pattern in Etherware. Before Etherware restarts, upgrades, or migrates a component, it captures the component's state as a memento. It then uses the memento to initialize the new component. This also lets the component bring the plant to a safe state before any changes.

- *Syntax changes.* Components interact by exchanging messages directly with their shells (see Figure 2a). This involves a simple and uniform functional interface, which isn't expected to change during most operation. So, the key sources of syntactic change during component upgrade are the formats of messages it consumes and produces. However, because message formats are XML document specifications, Etherware easily supports format changes and backward compatibility.

- *Semantic changes.* Support for explicitly stating interface semantics of components promotes service continuity during semantic changes. A formal specification of components' application semantics can further enhance this notion.

- *Communication changes.* Inserting filters, upgrading based on checkpoints, and migrating components support changes in connection topologies. The system can intercept messages by defining appropriate filters. The addition and deletion of filters doesn't require the messages' sender or receiver to be involved.

- *Timing changes.* Using MessageStreams supports changes in message delivery's timing behavior. However, Etherware doesn't yet support hard real-time deadlines.

- *Location changes.* The provision for globally unique bindings of components supports location independence. Migrating components is supported as a basic primitive.

- *Component failures.* Checkpointing and restart mechanisms in shells support passive failures such as component exceptions, as we mentioned earlier. A node failure, on the other hand, triggers appropriate exception messages informing remote components connected to components on the failed node. The component can then use these exceptions to handle node failures in application code. For example, all Etherware services we described in the " Etherware" section are designed to tolerate node restarts.

- *Other failures.* Active and Byzantine failures involve application-specific mechanisms, and we haven't yet provided comprehensive support for such changes in Etherware.

We've tested the performance of these mechanisms through experiments on a prototype traffic-control application.[7]

Related research

Fault-tolerant CORBA is the primary Object Management Group specification that addresses fault tolerance in distributed systems.[8] The key mechanism in FT-CORBA is supporting fault tolerance through redundancy of entities. However, a chief problem with this model is that, because replicas execute the same algorithms and have the same inputs, they'll have similar failures owing to application errors. Thus, safe component restarts are also necessary to support such failures. Researchers have also considered other problems that future work must address before we can use FT-CORBA for distributed real-time systems.[9]

Software frameworks and middleware for networked control in general are areas of active research. Open Control Platform[10] is a Real-Time CORBA -based middleware[11] for reconfigurable control systems. Although OCP supports service continuity during component and service reconfiguration, it doesn't provide mechanisms to tolerate faults in application software. Detailed surveys address related efforts.[12,13]

Conclusion

Etherware is the basis for much ongoing and future research. For example, Simplex is an elegant architecture that supports safe, dynamic upgrades of control software.[14] However, component restarts or upgrades still require the application to reestablish communication channels, and this might affect operational integrity. Because Etherware can support this, we're working to incorporate the Simplex architecture into it. We're also developing an interface description language to support message type specification and component interaction semantics in Etherware.

We've also implemented a traffic control testbed using Etherware.[7] On the basis of state estimation and buffering techniques,[2] we were able to develop the system with soft real-time control. However, Etherware doesn't yet support hard real-time deadlines, so incorporating this is another aspect of our current research.

References

1. J. Von Neumann, , *First Draft of a Report on the EDVAC*, tech. report, Moore School of Electrical Eng., Univ. of Pennsylvania, 1945.

2. G. Baliga , et al., "Etherware: Domainware for Wireless Control Networks,"*Proc. 7th IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing* (ISORC 04), IEEE CS Press, 2004, pp. 155–162, http://csdl.computer.org/comp/proceedings/isorc/2004/2124/00/21240155abs.htm.

3. A. Silberschatz , P.B. Galvin, and G. Gagne, , *Operating System Concepts,* 6th ed., Wiley & Sons, 2001.

4. "Extensible Markup Language (XML) 1.0 (3rd ed.)," World Wide Web Consortium (W3C) recommendation, Feb. 2004; www.w3.org/TR/REC-xml.

5. E. Gamma , et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995.

6. S. Graham and P.R. Kumar, , "Time in General-Purpose Control Systems: The Control

Time Protocol and an Experimental Evaluation," submitted for publication to *Proc. 43rd IEEE Conf. Decision and Control*, IEEE CS Press.

7. Information Technology Convergence Lab of Professor P.R. Kumar, http://decision.csl.uiuc.edu/~testbed.

8. "Fault Tolerant CORBA Specification, V1.0," Object Management Group, 2000.

9. A. Gokhale , et al., "DOORS : Towards High-performance Fault-Tolerant CORBA ,"*Proc. 2nd Int'l Symp. Distributed Objects and Applications* (DOA 00), IEEE CS Press, 2000,p. 39, http://csdl.computer.org/comp/proceedings/doa/2000/0819/00/08190039abs.htm.

10. L. Wills , et al., "An Open Control Platform for Reconfigurable, Distributed, Hierarchical Control Systems,"*Proc. Digital Avionics Systems Conf.,* IEEE Standards Office, vol. 1, 2000,pp. 4D2/1-4D2/8.

11. "Real-Time CORBA Specification Version 2.0," Object Management Group, Nov. 2003.

12. T. Samad and G. Balas, , eds., *Software-Enabled Control: Information Technology for Dynamical Systems*, IEEE Press, 2003.

13. 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, 2003, pp. 21–35.

14. D. Seto , et al., "Dynamic Control System Upgrade Using the Simplex Architecture,"*IEEE Control Systems Magazine,* vol. 18, no. 4, 1998, pp. 72–80.

15. P.R. Kumar and P. Varaiya, , *Stochastic Systems: Estimation, Identification and Adaptive Control,* Prentice-Hall, 1986.

Girish Baliga is a PhD student in the Computer Science Department at the University of Illinois at Urbana-Champaign. His research interests include software architecture and networked control systems. Contact him at Coordinated Science Laboratory, Univ. of Illinois, Urbana-Champaign, 1308 W. Main St., Urbana, IL 61801; gibaliga@uiuc.edu.

Scott Graham is an assistant professor of electrical and computer engineering at the Air Force Institute of Technology. His research interests include networking, architecture, control, and system integration. Contact him at Air Force Institute of Technology, 2950 Hobson Way, Bldg. 641, Rm. 210, Wright-Patterson AFB, OH 45433-7765; scott.graham@afit.edu.

Lui Sha is a professor of computer science at the University of Illinois at Urbana-Champaign. He is active in dependable real-time and embedded systems, and he served on the National Academy of Science's study group on software dependability and certification. He is a fellow of the IEEE. Contact him at Dept. of Computer Science, 201 N. Goodwin, Urbana, IL 61801; lrs@uiuc.edu.

P.R. Kumar is the Franklin W. Woeltge Professor of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. He received his PhD from Washington University in St. Louis, Missouri. He is a fellow of the IEEE. Contact him at Coordinated Science Laboratory, Univ. of Illinois, Urbana-Champaign, 1308 W. Main St., Urbana, IL 61801; prkumar@uiuc.edu

Cite this article: G. Baliga, S. Graham, L. Sha, and P.R. Kumar, "Service Continuity in Networked Control Using Etherware," *IEEE Distributed Systems Online*, vol. 5, no. 9, 2004.