Abstractions, Architecture, Mechanisms, and a Middleware for Networked Control

Scott Graham, Girish Baliga, and P. R. Kumar

***Abstract*—We focus on the mechanism half of the policy-mechanism divide for networked control systems, and address the issue of what are the appropriate abstractions and architecture to facilitate their development and deployment. We propose an abstraction of "virtual collocation" and its realization by the software infrastructure of middleware. Control applications are to be developed as a collection of software components that communicate with each other through the middleware, called *Etherware*. The middleware handles the complexities of network operation, such as addressing, start-up, configuration and interfaces, by encapsulating application components in "Shells" which mediate component interactions with the rest of the system. The middleware also provides mechanisms to alleviate the effects of uncertain delays and packet losses over wireless channels, component failures, and distributed clocks. This is done through externalization of component state, with primitives to capture and reuse it for component restarts, upgrades, and migration, and through services such as clock synchronization. We further propose an accompanying use of local temporal autonomy for reliability, and describe the implementation as well as some experimental results over a traffic control testbed.**

***Index Terms*—Abstractions, architecture, mechanisms, middleware, networked control, networked embedded control systems, third generation control.**

I. Introduction

A. A Historical Perspective

The first generation of platforms for control systems in the electronic age that began with the vacuum tube was based on analog computing[1]. This created a need for theories to use the technology of operational amplifiers. The challenge was met by Black, Bode, Nyquist and others, who developed frequency domain based design methods. A second era of digital control systems commenced around 1960, based on the technology of digital computers. It too created a need for new theories for exploiting the capabilities of digital computing, which was met by the explosion of work on state-space based design by Kalman, Pontryagin, and others.

Now, about fifty years later, we may be on the threshold of a third technological revolution in control platforms. This is driven by advances in VLSI hardware, wireline and wireless communication networking, and advanced software such as middleware. There has been tremendous growth in embedded computers, with 98% of microprocessors now sold being embedded [2]. In communication, besides the Internet, we may be on the cusp of a wireless revolution with Wi-Fi (IEEE 802.11x) experiencing double-digit growth since 2000 [3].

Less recognized is that software engineering has also evolved significantly in the last two decades. Although many basic ideas such as object oriented programming and distributed computing were mooted early on, they have been brought to fruition only recently due to better understanding of software management and advances in hardware. In particular, experience in design of large and complex software programs has been well codified and widely reused through design patterns such as *Strategy* and *Memento* [4], software frameworks such as Model-View-Controller (MVC) [4], and software development processes such as eXtreme Programming (XP) and Rational Unified Process (RUP) [5]. The development and widespread use of software libraries and infrastructure such as middleware has drastically reduced development cycle times, and resulted in better software. A good indicator of advances made is that the level of abstractions, particularly in programming languages, is much higher today. Software design primitives have evolved from registers and variables to libraries, components, messages, and remote procedure calls. A typical mid-range automobile has about 45 micro-controllers connected by Controller Area Networks (CAN) and uses complex software [6]. This has thrust software engineering into a central role [7] since the development and testing of complex software is expensive and fraught with numerous problems. There is much impetus to reuse previously developed and tested software, since it not only reduces the effort and expenses involved, but also enables sharing of valuable experience. About 40% of the software for the Boeing 777 airplane was of the commercial-off-the-shelf (COTS) variety [8].

These developments present new opportunities for control, as well as several challenges. Properly harnessed, they can potentially revolutionize control system platforms and lead to a new era of system building.

B. Importance of Abstractions and Architecture

In order for networked control systems to proliferate, they will need to be made easy to design and easy to deploy. This is

Manuscript received June 04, 2007; revised October 26, 2008. Current version published July 09, 2009. This work was supported in part by USARO Contracts W911NF-08-1-0238 and W-911-NF-0710287, NSF Contracts ECCS-0701604, CNS-07-21992, CNS-0626584, CNS-05-19535, and CCR-0325716, DARPA/AFOSR Contract F49620-02-1-0325, DARPA Contact N00014-0-1-1-0576, and AFOSR Contract F49620-02-1-0217. Recommended by Associate Editor J. Hespanha.

S. Graham is with the Air Force Institute of Technology (AFIT), Wright Patterson Air Force Base, OH 45433-7765 USA (e-mail: scott.graham@afit.edu).

G. Baliga is with Google, Mountain View, CA 94043 USA (e-mail: baliga@google.com).

P. R. Kumar is with the CSL and Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL 61801 USA (e-mail: prkumar@illinois.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TAC.2009.2022094

[1]Mindell [1] provides a detailed account of how control, communication and computation were intertwined in the first half of the twentieth century.

the objective in this paper. The goal addressed is how to reduce the design and development time. This also requires alleviation of the burden on the control system designer to be an expert in issues which lie outside the domain of control, by minimizing the attention needing to be paid to extraneous details. An analogy is general purpose computing, where a computational platform is easily usable by multiple users with different applications. Another analogy is general purpose networking where computers can be easily interconnected to communicate with each other regardless of the envisaged application.

These objectives require the identification of well designed *abstractions*, the design of a matching *architecture*, and the development of a supporting *middleware*. It is also important to provide *services* that render design easy and quick, as well as *design principles* that enhance reliability. This paper can be regarded as focusing on the "mechanism" half of the *mechanism-policy* divide. The development of a holistic and systematic *theory*, that has been the grand project of control systems research since the 1950s, can be regarded as the other "policy" half of the mechanism-policy divide.

An analogy with the developments leading to modern computation and computer science is appropriate. The work of Alonzo Church and Alan Turing laid the theoretical foundations of sequential computing. They independently developed formal systems to define and model the notion of computable functions [9]. However, it was John von Neumann's subsequent ideas about organizing these concepts into an architecture that is the basis of today's sequential computers. It led to the practical realization of the theoretical ideas. In particular, the stored program concept [10] was a significant breakthrough. It has led to the development of simple and uniform mechanisms to write, test, and maintain complex programs today. The von Neumann architecture had a wide influence on the first generation of digital computer engineers in the 1950s. Till the mid 1960s, commercial computers produced by IBM, Burroughs, UNIVAC, and others, were still custom built machines. However, the IBM 360, first shipped in June 1966, changed all that. It was the first commercially successful general purpose computer, with the name 360 (degrees) intended to market its "all round" general purpose capabilities. The 360/370 series introduced the notion of compatibility in computers [11]. All computer hardware was built to a common set of abstractions such as instruction sets, memory models, etc. Consequently, the same software could run on all the different machines in this series. This standardization tremendously reduced software development costs. More importantly, it solved problems due to changing customer requirements by supporting seamless upgrades of hardware and software. This interface between hardware and software is what today allows hardware and software designers to develop their products separately, resulting in the proliferation of serial computation. In contrast, parallel computation lacks such a "von Neumann bridge" [12], which, arguably, has stifled its proliferation.

The development of appropriate abstractions and matching architecture has been fundamental to the proliferation of other successful technologies too. The reason for the success of *networked communication* is, arguably, fundamentally its architecture, and, only secondarily, the algorithms involved, though of course they too are very important. In the Open Systems Interconnection (OSI) architecture [13], each layer provides a service to the layer above it, and in turn can be oblivious to the details of layers beneath it. These capabilities make networks robust and evolvable, and give longevity to the basic design. For example, over the course of the years, different versions of the transport layer protocol, TCP, have been proposed and deployed, without necessitating a change in other layers. The overall design allows heterogeneous systems to be composed in a plug and play fashion, making them amenable to massive proliferation, and has resulted in the Internet. The massive proliferation has led to reduced cost per unit over the long run, and resulted in minimizing deployment costs for new networks, further stimulating proliferation. A similar role has been played in *digital communication* by Shannon's source/channel separation, one of the rare architectural principles resulting from mathematical theory. Source coding is often performed in software by algorithms such as JPEG, while channel coding techniques such as QPSK are performed in hardware by network interface cards [14]. Closer home, in *control*, it is a standard abstraction to view the plant separately from its controller. This is however obvious only in retrospect, and, even now, is not routine in other fields; for example, it is difficult to routinely simulate new policies in manufacturing system simulation software where plant and controller are intermixed. Another architectural result, also one with a mathematical foundation, is the separation of estimation and control, which considerably simplifies design of control systems.

The situation in control systems today is similar to that of computers in the early 1960s, with the critical present requirement being a well designed organizing architecture that will make them amenable to massive proliferation. Complex systems design can involve a basic trade-off between architecture and performance. A system designed with a more *extensible* architecture has redundancies to improve design flexibility, while a system optimized for performance has tighter coupling between sub-systems at the cost of lower design flexibility. While the latter approach is better in the short term, the former is better in the long term. Improved design flexibility implies easier system extensibility, and modular design. Additional functionality to address changing requirements can be incorporated much more easily into systems which have an architecture that allows for evolution. Modular design allows different sub-systems to be developed separately and then integrated into operational systems. Further, these sub-systems can be changed, or even reused in other systems, with minimal impact on the rest of the system. All this results in systems that are more reliable and have a longer useful lifetime. Moreover, more general architecture allows recovery or even enhancement of system performance by facilitating sophisticated self-optimization capabilities. As an example, we will show how one may deploy mechanisms such as automated software migration to optimize the use of resources and loop delays at run-time, by designing a sufficiently rich software infrastructure that supports such mechanisms.

There are several key challenges. Distributed operation introduces problems due to concurrency and non-determinism in program execution. The network complicates the system fur-

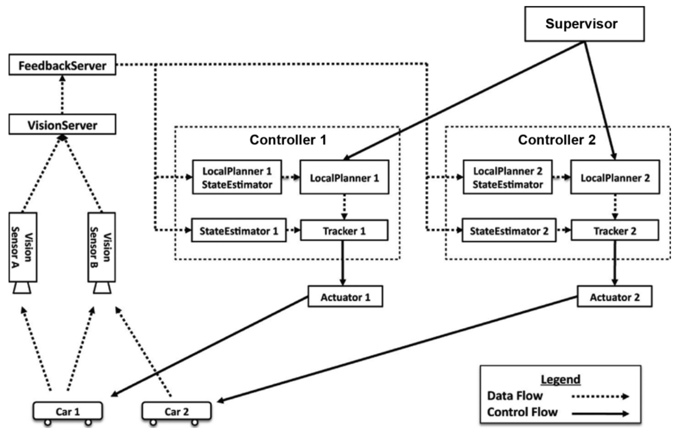

Fig. 1. Networked control system: Software component architecture.

ther with configuration, operational, and maintenance problems. While present networking technology provides some network abstractions, there remain numerous configuration details, such as determining network addresses and protocols, which need to be resolved in each application. These details occupy substantial portions of design time and effort, motivating the need for even higher-level abstractions. Another issue is that wireless links are prone to unpredictable delays and high losses. Hence, support for methodologies to facilitate control system design in the presence of such characteristics will have to be provided. What is required is an infrastructure that supports reusable solutions for common problems.

One caveat is in order. Just as general purpose computing is not targeted at high performance problems such as scientific computing, so also "general purpose" control may be inappropriate for situations where only a highly coupled solution, highly customized in hardware and software, can meet stringent requirements. However, even then, in an extensible architecture, one could embed lower level custom code in key subsystems to enhance performance.

C. Summary of Contributions

Rather than implementing a classical control system as simply a software program involving the control laws and plant interfaces, it is preferable to use a *component architecture* [15]–[17], which consists of decomposing it into *components*, which are autonomous software modules with well-defined functionalities. As an example, `StateEstimator`, `Supervisor`, `Controller`, `Sensor`, and `Actuator`, are all possible components in Fig. 1. A component architecture has several benefits. It allows individual components to be developed separately and integrated later, which is important for development of large systems. Since components are well-defined, they can be replaced without affecting the rest of the system. For instance, a zero-order hold filter can be dynamically replaced by a Kalman filter without having to change the remaining software or restarting an operational system. Software reuse is promoted, since a component such as a control algorithm tested on one system can be transplanted into others. Mechanisms that support component interface specification, including sequence and types of component interactions and format and contents of messages exchanged, and explicit specification of design assumptions, can simplify integration problems while reusing components.

Component based design is based on primitives such as components and messages, which are at a higher level of abstraction than sockets and processes supported by operating systems. These abstractions can be created by software infrastructure such as *middleware*, which has emerged as a preeminent framework to develop large distributed applications. Since systems integration and testing constitute the most expensive development activities in the engineering of complex software [5], a lot of effort has been directed at the development of software tools and architectures. Many successful applications have been developed using a CORBA based architecture [18]. Currently, IBM [19], Microsoft [20], and SUN [21] promote their middleware based technologies as platforms for software engineering. We anticipate that the next generation of control applications will also follow this path, and present such a proposal.

The middleware we have developed is called *Etherware*. It requires that the control system designer only write the logic of the control law in components, without being burdened with how it will be executed. The information flow between components is routed through the middleware, without a component needing to be aware of where a destination component is executing, thus providing *location independence* of components. The middleware also supports *semantic addressing*. It further supports facilities such as *migration*, whereby a Kalman Filter `Component` can migrate to a more computationally powerful node or a node with lesser latency between it and the sensor, or between it and the actuator. The middleware also provides features to enhance reliability, a key performance requirement for control systems. It allows *automatic restart* upon failure of a component, through mechanisms such as checkpointing of the state.

The middleware employs several *Design Patterns* [4] that codify good design solutions, capturing best practices, thus leading to much better reuse of design. We have incorporated the design patterns of *Strategy*, *Memento*, and *Facade* in Etherware. The *Strategy* pattern has led to replaceable control laws, *Memento* has resulted in component state check-pointing mechanisms, and *Facade* has allowed a simple and uniform middleware interface.

The major abstraction that we propose and which the middleware realizes is a *Virtually Collocated System*, which allows control engineers to design control laws for networked systems using the same techniques as for centralized systems, i.e., as though the distributed application were executing on a single computer. The control system designer need not concern herself with extraneous details of a networked system, and needs to focus only on the details relevant to the control law. It can potentially significantly reduce design time for networked control.

Architecturally, we propose that this abstraction be regarded as a *Virtually Collocated System Layer* (VCL) that resides above the Transport Layer. It can be regarded as a replacement for the Presentation and Session Layers, which are anyway absent in the TCP-IP architecture.

We propose an accompanying principle of *Local Temporal Autonomy* for design of reliable networked control systems, where the goal is to design components so that they can function for a limited period of time even when neighboring components fail, thus allowing them to migrate, or for a neighbor to restart. This facilitates robustness, reliable system evolution, development and debugging. We show how Kalman Filters, Receding Horizon Control, and Actuator Buffers, can all be used to provide Local Temporal Autonomy.

We identify *services* that are useful across a range of control systems and show how they can be supported. One example is distributed time. This is useful since networked control systems employ time-driven computing in addition to the usual event-driven computing. We show how such services are supported through a middleware feature called "Filters."

We have implemented the middleware on a laboratory *testbed*[2] featuring a networked system of cars, vision sensors, wireless and wired networks, and the full range of control hierarchy including regulation, tracking, trajectory generation, planning and scheduling, along with a discrete event system for liveness and safety.

This paper does *not* address the issue of *theories* needed for networked control. It only focuses on the *mechanism* half of the mechanism-policy division of research. Continuing theoretical effort, conducted on the *policy* front, is needed to fully realize the potential of networked control. The current situation is that technology has overtaken theory in some respects, in that we do not know how to optimally utilize the capabilities already supported in the middleware. We conclude by providing some examples of theoretical challenges for future research.

II. Description of Implementation

While describing the middleware, how the networked control system is to be implemented, and what functionalities the middleware can provide, it is helpful to refer to specific illustratory examples. Hence we provide a brief account of the networked control testbed, see Fig. 2, on which all of the above has been implemented. It consists of a set of remote controlled cars with no on-board computational capabilities. Each car is controlled by radio transmitter over a dedicated radio frequency channel connected to the serial port of a laptop through a micro-controller, which converts discrete commands from the laptop into analog controls specifying the speed and steering angle in discrete steps. Commands can be sent at a rate of up to 50 Hz, i.e., one command every 20 ms. Each car has a chassis top with uniquely coded color patches that are used to identify its position and orientation. The cars are monitored using a pair of ceiling mounted cameras. The video feed from each camera is processed by an image processing algorithm executing on a dedicated desktop computer. This feedback is available at the rate of 10 Hz, i.e., one update every 100 ms. All computers in the testbed are connected optionally by a wired Ethernet or by an ad hoc wireless network with IEEE 802.11 [23] PCMCIA cards.

The *architecture of the control application*[3] is illustrated in Fig. 1 where the control software components in the testbed are shown. The camera feed is processed in `VisionSensor` components. A data fusion component, called the `VisionServer`,

[2]Movies are available at [22].

[3]It should be specifically noted that this is distinct from the primary focus of this paper, the *architecture of the middleware*, the *infrastructure* over which the application is run.

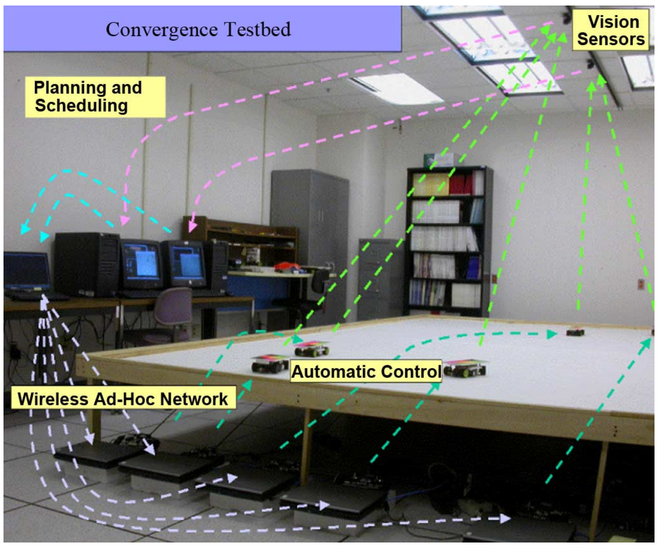

Fig. 2. Networked control system: traffic control testbed.

combines the sensor data from both vision systems and distributes the position and orientation information for each car via the communication network. Individual cars are operated by corresponding `Controllers` that implement Receding Horizon Model Predictive Control. Incoming sensor data is passed to a `StateEstimator` module which implements a Kalman Filter. Its prediction of future states is fed to the `Controller`, thus buffering it against delays and jitter of the communication system. A software module, called the `Actuator`, feeds commands to the radio-control system. The `Actuator` is designed to buffer a small horizon of future commands, so that it can tolerate brief control outages, and stopping the car if the `Controller` is inoperative for too long. A `Supervisor` resides above the `Controllers` and oversees the operation of the testbed. The `Controllers` need to be given trajectories to operate the respective cars. They are generated by the `Supervisor`, which also ensures global properties such as safety, the avoidance of car collisions, and liveness, the elimination of traffic gridlocks. The `Supervisor` generates trajectories along a pre-specified network of roads, using algorithms described in [24]. Briefly, blocks of the road network are modeled as bins in a corresponding graph. The planning problem is formulated as finding shortest paths in a graph, and the scheduling problem is modeled as assigning cars to bins while avoiding collisions and gridlocks. Finally, the `Supervisor` also receives feedback from the `VisionServer`, forming a higher-level control loop. Due to hardware constraints, the `Actuator` component for each car, and the `VisionSensor` component for each camera, must be executed on respective computers. All other components can execute on any computer in the testbed.

We have tested the system with up to eight cars operating simultaneously, and closely following pre-specified trajectories. Examples of traffic scenarios implemented are *pursuit-evasion* where a set of software controlled cars follow a manually controlled car, and *automatic collision avoidance*; see the videos at [22].

III. Networked Control System Requirements

We begin by identifying requirements for common functionalities needed for general networked control applications that should be part of a control-specific middleware. They may be grouped into *Operational Requirements*, *Non-functional Requirements*, and *Management Requirements*.

A. Operational Requirements

1) Distributed Operation: Connecting diverse components executing on different nodes on a network leads to numerous problems, including the need to locate and connect related components across a network, and support the exchange of messages between connected components. Examples of the problems that arise are initializing the components, connecting them, developing protocols for their interaction, and synchronizing their operation to resolve conflicts. In the traffic control testbed, the `Supervisor` and the various `Controllers` execute on different computers.

2) Location Independence: This is an abstraction that provides an addressing scheme for components that is independent of their actual location in the network, important for design of distributed operations. It allows components to communicate without distinguishing between local and remote components. It also allows integrating components uniformly in different network configurations. For example, in the traffic testbed, the ability to easily switch between wired and wireless networks, which use different addressing schemes, requires such details to be abstracted away from the components. The abstraction also supports the future use of other networking technologies such as Bluetooth [25], without having to update component code.

3) Service Description: Connecting components in a network involves determining if appropriate components are executing in the system. If several such components are available, then the most suitable component has to be selected. This requires mechanisms to specify and discover services provided by components. A car controller that knows the geographic location of its car should be able to connect directly to a sensor covering that location, without having to know about sensor locations in the network.

4) Interface Compatibility: Integrating independently developed software components can lead to interface incompatibilities; e.g., the set of functions defined in the interface of a sensor module may not be compatible with the functions required by a controller component. This can be eliminated if standard interfaces are specified and supported for compatibility.

5) Semantics: Incongruent assumptions in the implementation of components can lead to problems. The interaction between different software components is usually based on an implicit finite state machine (see Fig. 6), with exchanged messages assumed to be in specific formats. However, current interface description languages such as the CORBA IDL [26] do not provide a mechanism to specify these assumptions properly. For example, suppose the `Controller` in Fig. 1 is implemented so that it checks for an update at the `VisionServer` before computing controls. This assumes that `VisionServer` responds immediately, returning an update if available, and none otherwise. However, if `VisionServer` is implemented so that it checks with `VisionSensors` for updates before responding, the additional delay may lead to failure as `Controller` is also waiting for an update. Hence, `Controller` and `VisionServer` may have consistent interfaces, but the interaction semantics may still be incompatible. Consequently, there must be additional provision for specifying interaction semantics in interface descriptions.

6) Distributed Time: Components executing on different computers do not share common clocks, and need a mechanism to correctly translate time. For example, time-stamps on remote sensor updates must be properly translated to local time to avoid collisions.

B. Non-Functional Requirements

These are features that are not necessarily required for the correct operation of the system, but support for which would significantly improve system performance.[4]

1) Robustness: Robustness being a fundamental requirement for the viability of networked control, component failures must be contained and their effect on the overall system minimized. For instance, the failure of a faulty sensor module should not immediately cause a connected controller to fail as well. This requires dependencies between components to be eliminated or replaced by *use-only if available* relationships as much as possible.

2) Delay-Reliability Trade-Off: Reliable delivery of data over a network introduces additional delay due to retransmission of lost or re-ordered packets. However, some components may not require reliable delivery, and should therefore be able to trade-off reliability for lower average delay. For example, time-stamped sensor updates are more useful when delays are smaller, even though a few updates may be lost over the network.

3) Other Requirements: Algorithms should have good *scalability*. *Security* is also a key requirement and the system must be protected from misbehaving or malignant components.

C. Management Requirements

These are required for starting, managing, and updating components in an operational system.

1) Startup: Programmable interfaces to *startup* procedures that allow the specification of dependencies between components help ensure correct overall system initialization. For instance, a controller component may need to be started up before a related actuator component.

2) System Evolution: It is necessary to be able to migrate or update components at run-time. Component migration is required to optimize the configuration of software components to balance or reduce communication and computational loads. For example, a `Controller` in the testbed can be migrated to another computer, where it is closer to the corresponding `Actuator` or less loaded computationally, thus reducing loop delay. Supporting such run time optimization allows changing controllers to address evolving plant goals, without having to restart the whole system.

[4]Strange as the terminology may seem, these may actually be more important than functional requirements.

IV. ETHERWARE

We now present *Etherware*, a middleware for networked control. We begin with the design considerations, followed by the usage model and architecture.

1) Application Design: Control loops have to be closed over communication channels that are unreliable, especially in the case of a wireless network. We can mitigate this by employing various application design strategies, particularly the principle of *local temporal autonomy*. As an example, noisy feedback to the controller is filtered using `State Estimator`, which implements a Kalman Filter. However, what is of more interest to us here is that, architecturally, the interposed State `Estimator` can continue to provide position estimates of the cars, even if sensor measurement updates are lost or delayed, thus enhancing the *local temporal autonomy* of the downstream consumer of sensor information, `Controller`, from the producer, `Sensor`.

We further enhance reliability by using receding horizon control where the controller sends a *sequence* of future controls that can be used to control the car for a longer period, rather than sending just one control input [27]. Future controls are stored in an *Actuator Buffer* at the actuator, and used in case updates are delayed or lost. A similar buffer at the controller holds future way-points from Supervisor.

By providing support for such strategies in the middleware, the cost of application design complexity can be reduced. Since critical components can now endure some delays, we have been able to implement the entire system using only soft real time control[5].

2) Stability Considerations: Robustness requires the ability to restart application components efficiently to maintain system stability. For example, if `Controller` in Fig. 1 fails due to a software error, then restarting it should not require reestablishing communication with `VisionSensor`, reinitializing `Actuator`, or re-establishing `Controller` state, as these delays could cause system instabilities and result in car collisions.

This motivates a simple and uniform design to *externalize component state*, so that it can be reinitialized with this state, and restarted at the same node or even migrated to a different location and restarted there. Etherware enforces component state externalization as a basic architectural precept. Each component is instantiated with an initial state, and is required to support a state check-pointing mechanism. On a check-point request, it has to return a state object that can be used to reinitialize it upon restart, update, or migration.

Maintenance of communication channels across such changes is also supported in Etherware. Identifiers for communication channels can be saved as part of the check-pointed state, allowing restarted or upgraded components to continue using previously established channels. It also provides communication continuity to other components during such changes.

3) Message Based Communication: Components in networked control systems have to respond to changes as soon as possible. Upon detecting a safety violation, `Controller` may not be able to wait for an acknowledgment from `Supervisor` before it takes action. Over a wireless channel in particular, delays can be large due to deep fading and queued packets. Another important consideration is the presence of dependencies in push-based communication channels. For example, the controller cannot wait for the updates from the sensor before sending controls to `Actuator` since updates may be delayed or lost. Conventional middlewares for transaction based systems such as CORBA [26] use synchronous communication where a component sending a message is blocked until it receives a reply. This leads to complex design with each component requiring multiple threads of control, since a separate receiving thread is required for blocking on each channel. On the other hand, asynchronous operation eliminates this source of complexity since a component is not blocked when sending a message.

Based on these considerations, Etherware has been developed as a *message-oriented middleware*. Message based communication requires a specification of message formats. Support for interface and semantic compatibility during changes and component reuse requires this specification to be flexible, extensible, and backward compatible. *Flexibility* is the ability to easily incorporate changes in the interface and semantics of a component, and *extensibility* supports ease of adding specifications for new functionality, while still honoring the original specifications used by older components to communicate with it. Based on these requirements, we have used *XML* [29] as the language for messages, with all communication in Etherware through well-formed XML documents with appropriately defined and extensible formats. For platform independence, and due to availability of support for XML, Etherware has been implemented using the Java programming language [30]. While these choices incur some additional processing overhead, we believe that the use of open standards and advances in computer hardware compensate for this.

4) Architecture: An important architecture design decision was to choose an execution model for *active* components, such as sensors, which need their own threads of control. We decided to go with thread-per-active-component instead of process-per-active-component model for the following reasons. First, support for Thread management is much better than support for Process management in Java. For instance, support for thread prioritization and thread interruption is available in a standardized and platform-independent interface. Second, since Threads are much lighter-weight entities than Processes, their creation and update time is much smaller. This is well illustrated in Section VIII-A where we see that a Process restart is on the order of seconds, while a Thread restart is on the order of tens of milliseconds. Finally, having one Etherware process per computer makes it easier to define and use resources such as published network ports for remote communication. However, individual active components can create and manage their own processes if necessary. For instance, due to programming language and platform constraints, the `VisionSensor` components for cars in the testbed create and manage processes to collect data from the frame grabbers connected to the camera outputs.

[5]We are presently incorporating real-time support into Etherware [28].

Services provided by the middleware need to be easily restartable and upgradeable. Supporting this allows basic versions of services to be deployed initially for resource savings, and subsequently updated based on changing application requirements, without having to restart the system. Hence, invariant aspects of the middleware, which cannot be changed dynamically, have to be minimized to maximize flexibility.

These considerations have motivated us to adopt a *microkernel* [31] based design. This consists of using a simple and efficient kernel, which has only the most basic functionality needed for minimal operation, and which will not need to be changed during system operation. This consists of functionality to deliver messages between components, and primitives for creating new components. All other middleware functionality is implemented as components, just as the application is; hence these can be changed without having to restart the system. This philosophy of maximizing flexibility has also resulted in the development of a bare minimum functional interface for components to interact with the middleware. Flexibility is increased since the above design minimizes the number of aspects in the functional interface that need to be changed or upgraded during system operation. For uniformity, application components interact with middleware services with messages, just as though they were other application components.

A. *Etherware Usage Model*

We next turn to the programming or usage model for Etherware, that is, the abstractions provided to control engineers and software programmers using Etherware.

A *component* in Etherware can participate in control hierarchies. For example, Controller in Fig. 1 takes goals from the higher-level `Supervisor` and sends commands to the lower-level `Actuator`. The `VisionServer` of Fig. 1 takes data inputs from `VisionSensors` and provides data to `Controllers`.

Etherware is an asynchronous message-based middleware. Components communicate by exchanging messages, which are well formed XML documents [29]. XML documents can be directly manipulated as large strings. However, Etherware also provides a hierarchy of Java classes which provide various primitives to manipulate the underlying XML document across a Java based interface. This hierarchy can be extended to support additional messages with user-defined XML formats, in addition to the predefined messages with specific XML formats that are encoded in these classes.

Two basic problems attending message delivery in distributed systems are discovery and identification of destination components. The identification problem is solved in Etherware by associating a globally unique id, called a *Binding*, to each component. The discovery problem is solved by associating semantic profiles to addressable components. Each component that needs to be addressed registers one or more profiles with Etherware. A profile describes the set of services that a component provides. For example, a profile for a vision sensor could specify the type of camera and the geographic region covered by it, and a car controller could use this information to connect to a relevant vision sensor.

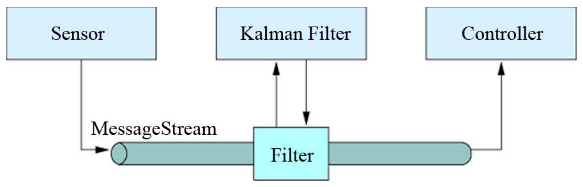

Fig. 3. Filters for *MessageStreams*.

All messages have the following three XML tags or constituents:

Recipient Profile: This identifies the recipient of the message. A profile can be a service description as above, or the globally unique Binding of a component.

Content: This represents the contents of the message including application specific information.

Time-stamp: Each message has a time-stamp associated with it which is automatically translated to the local clock on that computer as a message moves from one computer to another.

By default, messages are delivered reliably and in order. However, there may be streams of messages that need to be delivered using other specifications as well. To identify and manipulate a stream of messages as a separate entity, Etherware supports the notion of a *MessageStream*. A component can open a *MessageStream* to another component and send messages through it. *MessageStreams* have settings that can be used to specify how messages are delivered through them. For example, `Controller` can tolerate a few lost sensor updates for lower delay, but has no use for old updates. Accordingly, `VisionSensor` could open a *MessageStream* to `Controller` requesting unreliable in-order delivery, as shown in Fig. 3. This means that messages along this pipe will not be retransmitted if lost, and messages arriving late will be discarded. In addition, Etherware provides *MulticastStreams* that support efficient multi-source multi-cast of messages.

Etherware provides the capability to add, at run time, *Filters* that can intercept all messages sent to, or received by, a component, since it may be necessary to modify messages in a *MessageStream* in response to changes in operating conditions. For example, updates from VisionSensor could get noisy due to bad lighting conditions, and it is desirable to have the capability to filter out this noise without having to change Sensor or the Controller. Fig. 3 shows the effective configuration after a Kalman filter has been added to the message pipe between `VisionSensor` and `Controller`.

The design of a generic Etherware component is shown in Fig. 4. It is based on several well known “design patterns” [4]. In software development, many design problems have a common recurring theme, and design patterns represent solutions that exploit the theme. We now consider the various design problems for Etherware components, and describe how these are addressed using appropriate design patterns.

Memento: Support for restarts and upgrades requires the ability to externalize and capture application state. This is solved by the *Memento* pattern, wherein component state can be checkpointed and restored on re-initialization.

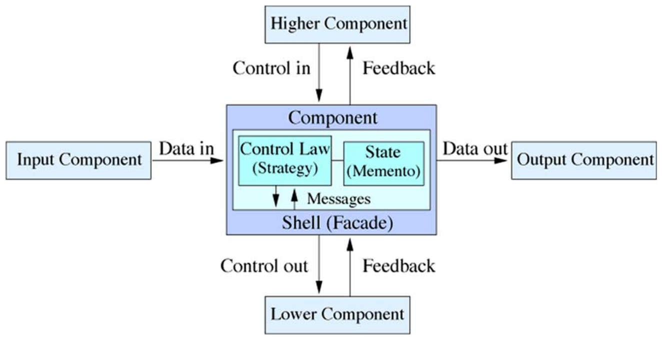

Fig. 4. Programming model for a `Component` in Etherware.

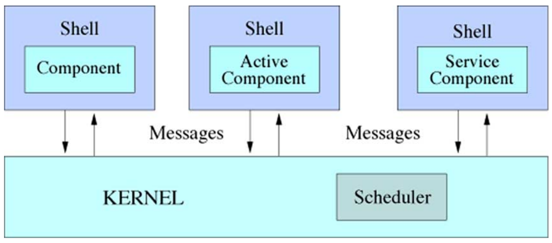

Fig. 5. Architecture of Etherware.

Strategy: System stability is enhanced by the ability to replace components without disrupting service. In particular, if the functional (syntactic) interface used to communicate with the component is invariant, then components can be replaced dynamically. In this case, the *Strategy* pattern is used in conjunction with the *Memento* pattern.

Facade: Interaction with various services in the middleware usually requires a component to send messages using function calls. If the component has to directly call functions on the various sub-systems, the structure of these subsystems needs to be programmed in it. This introduces unnecessary dependencies and makes the component and the middleware hard to evolve, as changing the middleware architecture will require the software of all the components to be updated as well. This is eliminated by using the *Facade* pattern to provide a uniform functional interface that is independent of the actual middleware architecture.

Components can be active or passive. *Passive* components do not have any active threads of control. They only respond to incoming messages by processing them appropriately and generating resulting messages if any. *Active* components have one or more active threads of control. They can generate messages based on activities in their individual threads of control. For example, VisionSensor can be implemented as an active component with a separate thread to block on sensor hardware updates, and generate appropriate messages when new sensor data is available.

B. Etherware Architecture

The architecture of Etherware is based on the micro-kernel concept shown in Fig. 5. As motivated in Section IV, the *Kernel* represents the minimum invariant in Etherware. All other services are implemented as components. The Kernel manages all components on a given computer. The function of the Kernel is to create new components, and deliver messages between its (local) components. We allow one Etherware process per node in the current implementation.

The Kernel has a *Scheduler* that is responsible for scheduling all messages and threads. The Scheduler can be replaced dynamically if more complex scheduling algorithms are necessary as the application evolves. In addition, the Scheduler also provides a notification service, whereby a component can request to be notified with *alarms* after a given delay. This allows components to wait, without having to create separate threads of control for this purpose. Components can also register to receive periodic notifications or *ticks*. This has been used to implement all soft real time control in the testbed. The car `Controller` operates at 10 Hz, and has been implemented as a passive component. For periodic activation, it registers with `Scheduler` to receive periodic tick messages every 100 ms.

Each component is encapsulated in its own Shell as shown in Fig. 5. 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 configurations of *MessageStreams*. Activities involved in component restart, upgrade, and migration have also been implemented in Shells.

All other functionality in Etherware is provided by service components. An instance of each service component executes on each computer. The following basic services are used:

ProfileRegistry: The *ProfileRegistry* is used to register and look up profiles of components. Profiles of newly created components are registered and recipients of messages addressed using profiles are determined by the registry. Each node has a *Local ProfileRegistry* for components on that node. A network also has a *Global ProfileRegistry* for components on all nodes in the network.

NetworkMessenger: This sends and receives remote messages, i.e., messages between a local component and a remote component operating on another computer. Hence, it encapsulates all communication with remote nodes over the network, including details such as IP addresses, ports, and transport layer protocols. By default, all messages addressed to remote components are forwarded by the Kernel to this service. The *NetworkMessenger* is an active component with separate threads to receive messages from remote nodes.

NetworkTimeService: This service translates time-stamps of messages as they are transmitted from one node to another. To implement this, a `NetworkTimeService` component is added as a *Filter* to the *NetworkMessenger*, so that it intercepts all messages that are sent to and received from other nodes. (It should be noted how easy it is to time-stamp information once we have the notion of Filters). Time translation is based on computing clock offsets using the Control Time Protocol [32].

C. Etherware Capabilities

We now describe how the requirements listed in Section III are addressed in Etherware.

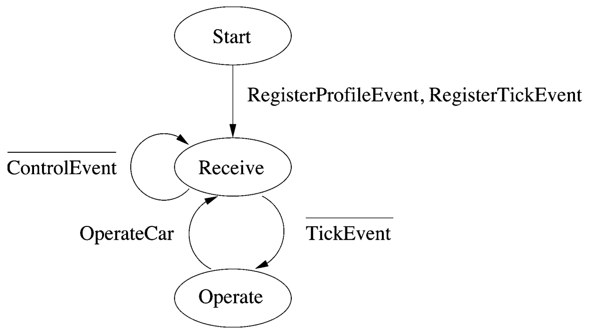

Fig. 6. Finite state automata for `Actuator` semantics.

1) Operational Requirements: These requirements are addressed primarily by the Etherware programming model.

Distributed Operation: Components executing on different computers can communicate with each other using the same primitives as used for local interactions. The following mechanisms collectively implement *location independence* in Etherware. Each component is assigned a globally unique-id called a Binding. This addressing scheme is independent of network location and topology. Each Binding is mapped into a network specific address, such as an IP address, and a component id on a given computer. This mapping can change as components migrate. Local routing of messages on a single computer is performed by the Kernel. Routing of messages between computers is done by the *NetworkMessenger*.

Service Description: This is a consequence of addressable components being able to register service profiles. A component that wishes to access a given service can just address messages using the appropriate profile. The profile is then matched with registered profiles by the *ProfileRegistry* as explained in Section IV-B. If a match is found, then the message is directly forwarded to the appropriate component. If not, an appropriate exception message is returned.

Interface Compatibility: This is primarily achieved by use of XML documents for communication between components. Besides, components use a simple and uniform functional interface provided by the Etherware Shell, and as suggested by the *Facade* pattern in Section IV-A.

Semantics: This requirement can be properly addressed by an interface description language (IDL), which is a language to describe the functional and interaction interface exposed by a component. The Etherware programming model mandates message based interactions between components. This requires components to incorporate interaction semantics that can be formally specified using a finite state machine [33]. It includes specification of the messages that a component can send or receive, and conditions under which it can send or receive specific messages. Fig. 6 represents the FSA that specifies Actuator semantics. The labels on the arrows represent messages and actions, with overbars indicating received messages. Etherware component semantics can be formally specified in an executable fashion in the rewriting logic based formal language Maude [34], as

```
mod TESTBED is
  pr RAT .
  pr QID .
  inc CONFIGURATION .

  subsort Qid < Oid .

  sorts ActuatorState State .
  subsort ActuatorState < State .

  sorts Control Controls .
  subsort Control < Controls < Attribute .

  ops startA receiveA : -> ActuatorState .
  op state :_ : State -> Attribute .
  op car :_ : String -> Attribute .

  op Actuator : -> Cid .
  op registerMsg(_,_) : Oid Qid -> Msg .
  op controlMsg(_,_,_) : Oid Int Rat -> Msg .
  op actionMsg(_,_) : Int Rat -> Msg .
  op tickMsg(_) : Oid -> Msg .

  op control(_,_) : Int Rat -> Control .
  op nil : -> Controls .
  op _ _ : Controls Controls -> Controls [assoc id: nil] .

  op newActuator(_,_) : Qid String -> Object .

  var A : Oid .
  var S : String .
  var I : Int .
  var R : Rat .
  var AS : AttributeSet .
  var C : Control .
  var CS : Controls .

  eq newActuator(A,S) = < A : Actuator | state : startA , car : S, nil > .

  rl [a-init] : < A : Actuator | state : startA , AS >
    => < A : Actuator | state : receiveA , AS > registerMsg(A,A) .

  rl [a-recv] : < A : Actuator | state : receiveA , AS , CS > controlMsg(A,I,R)
    => < A : Actuator | state : receiveA , AS , CS control(I,R) > .

  rl [a-send] : < A : Actuator | AS , control(I,R) CS > tickMsg(A)
    => < A : Actuator | AS , CS > actionMsg(I,R) .
endm
```

Components and systems specified in this fashion can be verified for validity and correctness using theorem proving tools in Maude. XML provides the representation for strongly typed messages, where the type and format of all possible messages can be defined as part of the interface.

Note that the IDL provided by Etherware is only a language for specifying component interface and interaction semantics. The actual semantics for a given component IDL specification are highly dependent on its domain of operation. For instance, the IDL specification of a `VisionSensor` component could specify the refresh rate using an XML tag called "RefreshRate," but for other components to correctly use its sensory output, they must understand that the "RefreshRate" tag identifies the specification of the refresh rate of the vision sensor, i.e., they must have a common shared vocabulary with consistent meanings.

In addition, an Etherware proxy service has also been implemented, which allows formal component specifications in Maude [34] to interact with regular components in Etherware. This enables rapid prototyping, where formal specifications can be checked for both logical correctness and operational stability. For instance, the `Actuator` module specified above can be plugged into the actual testbed, and could emulate a real car in the system.

Distributed Time: This is provided by the *NetworkTimeService*.

2) Non-functional Requirements: These requirements have been addressed by various architectural features and services in Etherware.

Robustness: This is facilitated primarily by the use of the *Memento* pattern. It allows the effect of component failures to be contained by efficient check-point and restart mechanisms in Etherware. The efficiency of these techniques is considered in Section VIII.

Delay-reliability Trade-off: MessageStreams support trading off reliability for lower delays. They also provide a simple mechanism to incorporate support for other QoS requirements.

Other Requirements: The current algorithms for various services scale well for the testbed requirements. We have tested them by operating up to eight cars at a time, which represents the scenario with the maximum load in the system. Further, component Shells provide a uniform location to potentially incorporate security features.

The Etherware architecture essentially trades-off system performance (verbose protocols, java vms, soft real-time operation, etc.) to support complex functionality and interactions between components. However, high performance critical control loops can still be implemented using a locally optimized language and platform, and these can then be "wrapped" with an Etherware component to plug it into a larger system. In this design, Etherware acts as an integration platform for complex networked control systems, with higher-level supervisory control implemented for soft real-time operation, connected to lower-level critical control loops implemented on native platforms for hard-real time operation.

3) Management Requirements: These are addressed using configuration support and design:

Startup: Start-up configurations and dependencies are specified to Etherware using a configuration file for each computer. Etherware ensures that components are initialized in the correct order. Currently, the absolute order in which components need to be initialized on a given node is specified, but support for more complicated dependencies can be easily supported using a richer dependency evaluation system such as `ApacheAnt` [35].

System Evolution: System evolution through component update and migration, is supported by the use of *Memento* pattern. Application state is maintained across both operations using *Mementos*. Connections using *MessageStreams* are maintained across component updates. As noted in Section IV-A, *MessageStreams* and Filters also provide a mechanism for system evolution without having to change any existing connections between components.

V. Services

In addition to the Etherware infrastructure itself, many control applications have many common sub-problems whose solutions could be reused. We facilitate such reuse into *services* bundled with the middleware. Some of the services that we have considered are:

1) Distributed Time Management: We have implemented a clock synchronization algorithm.

2) Plant Delay Estimation: A dynamic delay estimation service is of interest for closing loops since delays may change with time.

3) Component Repositories: Software component repositories, from where new control software can be downloaded and installed dynamically into operational systems would be useful in promoting use of new theories and algorithms.

4) Dynamic System Optimization: This is a useful service to facilitate automatic determination of optimal system configuration at run-time. For instance, whether to process all the pixels from the camera at a certain node, thus stressing its computational resource, or to transfer them to another node, thus stressing the communication network, is a low level decision that depends on the power of the processor at a node as well as the current communication traffic load, and should be automatically done. The control designer's role can thus move to the higher plane of specifying high level rules or algorithms for such migration.

VI. A Layer for Networked Control

The Open Systems Interconnections (OSI) [13] architecture is a standard reference model for networked systems. Our proposed architecture is based on this model, and the mapping between the different layers is shown in Fig. 7. Central to it is the abstraction of a *virtually collocated system*, where information exchange between application components is supported at the level of messages exchanged between seemingly local components. This abstraction is provided by the middleware, which hides the details of a networked system such as IP addresses, network protocols, data formats, and computational resources. Components can be identified using application specific content instead of network addresses and ports. They can exchange information on a regular basis, as information push whenever

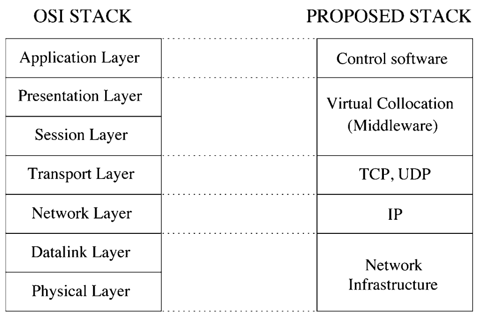

Fig. 7. Mapping from OSI stack to proposed architecture.

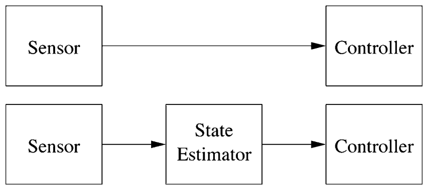

Fig. 8. A `StateEstimator` separating `Sensor` and `Controller` can reduce execution and timing dependencies between them.

updates are available, or as pull on demand by an information consumer. Hardware or software can be added or removed while the system is running, and the designer need not deal with issues like starting up computers in an appropriate order. Also, clocks are aligned and all sensor and other information is automatically time-stamped. These features correspond to the Session and Presentation layers shown in Fig. 7. Utilizing the abstraction of a virtually collocated system provided by middleware, a system designer can focus on algorithmic code for planning, scheduling, control, adaptation, estimation, or identification.

VII. Design Based on Local Temporal Autonomy

Networked control systems will be subject to communication and node failures as well as component failures. We suggest the *local temporal autonomy* design principle: Design components to tolerate, with graceful degradation, failures in connections to, and operation of, other connected components. We provide three illustratory examples.

1) State Estimator: To match the unpredictable characteristics of the communication network with the predictability required by control, we insert a buffer, a `StateEstimator`, before controllers in the system, see Fig. 8. It removes the *execution* and *timing* dependencies of Controller on the inherently unreliable communication channel , since it can be used by Controller to interpolate between lost sensor updates. Such a design can take aperiodic sensor data as input, and provide continuous or periodic outputs to the Controller, the format preferred by digital control design methodology. This improves local temporal autonomy of Controller, since it buffers it from failures in the sensor and the communication channel.

2) Actuator Buffer: Similarly, at the actuator end, instead of providing just one current control input at each update to an actuator, by using receding horizon control, a `Controller` can provide a window of control inputs up to a time horizon. This again improves the local temporal autonomy of `Actuator` as it now has fall-back controls in case subsequent controller updates do not arrive.

By weakening the dependence between components, these approaches also facilitate dynamic component upgrade and migration, in addition to simplifying design and testing.

3) Migration for Self-optimization and Reliability: Components in the networked control system can execute at any location. Their optimal physical locations depend upon timing constraints, communication delays, computational loads, etc., which may vary during operation. Hence it would be desirable for the execution of a component to be able to *migrate* to a better location within the system. To accomplish migration, the middleware incorporates several supporting functions such as the components being able to continue to communicate after the move. This involves updating the communication information at each of the nodes or components which communicate with the migrating component, as well as updating the communication information at the component itself. To help an application optimize decisions on when and where to migrate a component, the middleware must be able to estimate the loads on the physical resources which will exist before and after the migration, which can be provided as a middleware service.

VIII. Evaluation

This section presents three experiments evaluating the performance of Etherware mechanisms.

A. Controller Restart

In the first experiment, `Controller` was restarted several times. Faults were injected at random by performing an illegal operation (divide by zero) in `Controller`, which caused `Controller` to raise exceptions and be restarted by Etherware. The deviation of the actual car positions from the desired trajectory, as a function of time, is shown in Fig. 9(a). Restarts are indicated by pointers, and the accompanying numbers indicate, in milliseconds, the time for each restart. These are time-stamps at `Observer`, and include communication and synchronization times between the restarted `Controller` and `Observer`. The first restart occurred at about 70 seconds into the experiment, and was followed by two other restarts in the next 20 seconds. The last three faults were also handled by the restart mechanisms. We see that the error in the car position during these restarts was within the system error bounds during normal operation prior to restart. Two of the Etherware mechanisms described in Section IV contributed to the quick recoveries. First, the Shell intercepted exceptions due to the `Controller` faults, and restarted it without affecting the *MessageStream* connections to the other components. Second, before termination, the `Controller` state was check-pointed according to the *Memento* pattern, and then used for re-initialization. To illustrate the performance of these two mechanisms, we also tested the alternative mechanism of restarting the Etherware *process* managing the `Controller` at about 100 seconds after the start. We see that the restart of Etherware and the `Controller` took about three seconds, during which the car position accumulated a large error of about 0.8 meters. This illustrates the necessity for efficient restarts. Furthermore, even though the `Controller` restarted after three seconds,

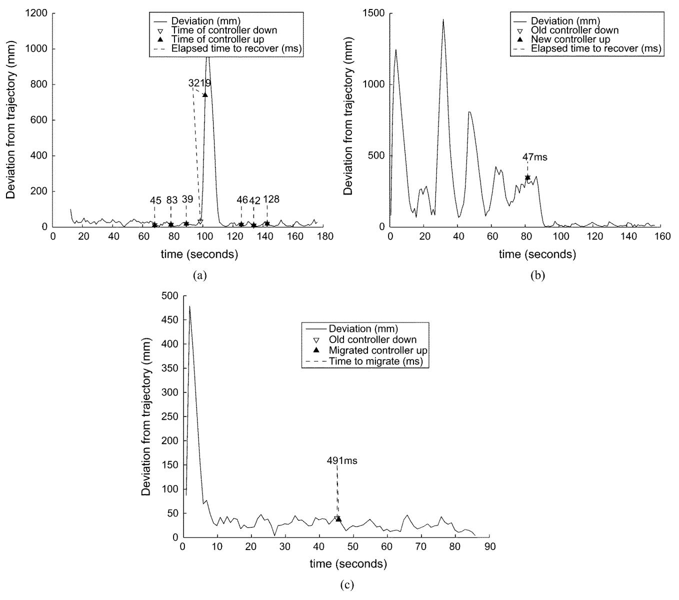

Fig. 9. Performances of controller restart, upgrade and migration. (a) error in car trajectory due to controller restarts. (b) Error in car trajectory due to eontroller upgrade. (c) error in car trajectory due to controller migration.

additional error was accumulated before recovery. This was because the `Controller` had to reconnect to the other components, rebuild the state of the car, and bring it back on track. These avoidable delays are eliminated through efficient restart mechanisms in Etherware.

B. Controller Upgrade

In the second experiment, we tested the performance of software upgrade mechanisms in Etherware. The car is initially controlled by a coarse `Controller` that operates myopically. Etherware is then commanded, at about 90 seconds, to upgrade the coarse `Controller` to a better model predictive `Controller`. From Fig. 9(b), we see the improvement in the car operation after update. The involved transients are within the system error bounds as well. This functionality is due to three key Etherware mechanisms from Section IV. First, the *Strategy* pattern allows 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 *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 other components and the car operation.

C. Component Migration

In the third experiment, the support for migration in Etherware was tested. As before, the error in the car trajectory is shown in Fig. 9(c). The large spike at the beginning of the graph was the error due to the car trying to catch up with its trajectory. This is achieved at about 10 seconds into the experiment, after which the car follows the trajectory within an error of 50 mm. In particular, the error introduced due to migration is well within the operational error of the car. Two Etherware mechanisms enable migration. First, the *Memento* pattern allows the current state of `Controller` to be captured upon its termination. Second, the primitives in the Kernel on the remote computer allow a new controller to be started there with the check-pointed state of `Controller`.

IX. Related Work

This section presents an overview of related earlier work in this area. CORBA [26] is probably the most popular middleware and has been used in a variety of different domains. It was primarily intended for transactions based business and enterprise systems. Hence, the trade-offs incorporated in CORBA are not all compatible with requirements for control systems. For example, the specification mandates the use of TCP [36] for reliable delivery. This can be a serious limitation if lower delay is more important than reliability, as such a trade-off cannot be supported. Popular versions of middleware for control applications are based on various flavors of CORBA, such as Real time CORBA [37] and Minimum CORBA [38]. For example, OCP [39] is based on Real Time CORBA and has been used to control unmanned aerial vehicles. ROFES [40] implements Real Time CORBA and Minimum CORBA, and is targeted for real-time applications in embedded systems. Fault-tolerant CORBA (FT-CORBA) [26] is the primary OMG specification that addresses fault tolerance in distributed systems. A key problem in using CORBA based middleware is that the CORBA interface description language (IDL) is not descriptive enough to specify key assumptions in component design. In particular, issues involved in semantic compatibility, such as description of data validity and interaction semantics, cannot be specified. This leads to numerous problems while integrating independently developed, yet functionally compatible components. Other interesting approaches include Giotto [41], real-time framework [42] for robotics and automation, and OSACA [43] for automation. A good overview of research and technology that has been developed for implementing reusable, distributed control systems is provided in [44].

X. Theoretical Challenges

Exploiting the capabilities of Etherware poses theoretical challenges. At the tactical level, we need to be able to optimally exploit middleware capabilities already present, such as migration, restart, and the services provided. Two examples of problems that arise are the following. Since the middleware provides to the controller the actual delay that has already been incurred by a packet containing a measurement, i.e., random but known delay, how should one design a controller that is robust to plant uncertainties? Another is how to optimize the "migration controller" as empirical delay profiles between various origin-destination node pairs over a networked system change.

At the strategic level, there is a need for a cross-domain theory that is holistic and includes real-time scheduling, plant uncertainties, and hybrid systems. We present an illustrative example. Consider the car control loop model shown in Fig. 10. The Controller and the Actuator are collocated, see [45], and there is negligible delay between a `Controller` issuing a control, and the `Actuator` effecting it in the car. Suppose the feedback channel

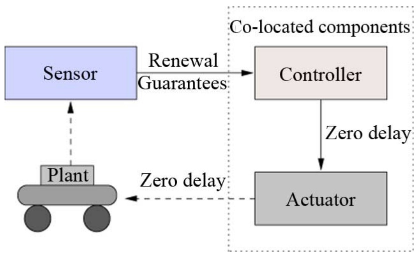

Fig. 10. Control loop model.

from the Sensor to the Controller is scheduled according to the *renewal task* model, [46] where a task is a sequence of jobs that execute on a specified resource, and where each job has a fixed computation time C and a relative deadline D, with the renewal referring to the fact that a new job is released immediately after a previous job is completed. A renewal task is characterized by its task *density* $\rho = C/D$. In our setting, each job is a computation of car controls, speed and orientation, to be applied for a specific time interval.

We model a car itself as an oriented point in a lane, with state (d, θ), where d is the absolute distance of the car from the center of the lane, and θ is the angle of the car to the median of the lane. As a convention, the direction of traversal of the lane is assumed to be from left to right in what follows. We assume that a car has a single non-zero speed s, and four steering controls: two in anti-clockwise direction, and two in clockwise direction, both allowing motion along circles of radii $\underline{R}$ and $\bar{R}$ with $\underline{R} < \bar{R}$. Based on the renewal task model for sensory feedback, a car can move a bounded distance before it is observed again. Since the car moves at a constant speed s, this corresponds to a deadline of $\underline{R}\alpha/s$ in the renewal task model. Finally, a control is sent immediately after the car has been observed.

Theorem 10.1: A set of cars can be driven along pre-specified routes without collisions (*safety* guarantee) or gridlocks (*liveness* guarantee), if the road network has straight single-lane roads of length at least $L = 2\gamma\bar{R}\underline{R}/\bar{R} - \underline{R}$, and lane-width at least $2(W + \underline{R}(1-\cos\gamma))$ with $W = \underline{R}(1-\cos\beta(2\cos\alpha - 1))$, $\alpha \leq \pi/3$, and $\beta = \cos^{-1}(2\cos\alpha - 1)$.

Proof: We provide a brief outline; see [47] for details. First, it can be proved that a car can be controlled so that it stays within a road of width $W = \underline{R}(1 - \cos\beta(2\cos\alpha - 1))$. Then it can be proved that the length of the road is sufficient to keep the car within its lane when moving from one lane to another at an intersection. Third, it can be shown that a set of cars in a road network with single lane roads can be cleared by a *supervisory algorithm* described in [24]. This algorithm models the traffic system as a discrete graph, where a lane is modeled as a sequence of bins. The algorithm then moves cars between bins in discrete time. This establishes the liveness guarantee.

Next, it can be shown that the cars can be scheduled without collisions by ensuring that at most one car can be in a bin at a given time. This is enforced during bin transitions by keeping the bin in front of each car empty. This, in turn, is accomplished by associating an additional "virtual" car always in front of each real car. This final step establishes the safety guarantee for the system. □

As we build more complex systems, a challenge is to be able to automate such proofs of overall system correctness.

XI. Conclusion

This paper has focused on the mechanism half of the policy-mechanism divide, and proposed an abstraction and a middleware based approach to support the design, development, and deployment of networked control systems. The Etherware design supports the development of applications as collections of components. The support for standardized interfaces and seamless integration enables the reuse of components in other applications, which allows the building a library of components embodying various control algorithms and designs that can be assembled into working applications in deployed systems. Application development consists of specifying the interconnection of components and their individual configurations. Using Etherware primitives, system optimization can be supported by developing algorithms to automatically determine optimal component interconnection configurations, and migrate components accordingly.

Acknowledgment

The authors are grateful to L. Sha for enthusiastic discussions and much wisdom over several years.

References

[1] D. A. Mindell, *Between Human and Machine: Feedback, Control, and Computing Before Cybernetics*. Baltimore, MD: JHU Press, 2004.
[2] J. Stankovic, Vest: A Toolset for Constructing and Analyzing Component Based Operating Systems for Embedded and Real-time Systems Univ. of Virginia, Tech. Rep. TRCS-2000-19, 2000.
[3] A. Lahjouji, K. Carter, and N. McNeil, Unlicensed and Unshackled: A Joint OSP-OET White Paper on Unlicensed Devices and Their Regulatory Issues May 2003 [Online]. Available: http://hraunfoss.fcc.gov/edocs:_public/attachmatch/DOC-234741A1.pdf
[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, ser. Professional Computing Series*. Reading, MA: Addison-Wesley, 1995.
[5] I. Sommerville, *Software Engineering*. Reading, MA: Addison-Wesley, 2000.
[6] R. Bannatyne, "Microcontrollers for the automobile," *Micro Control J.*, 2003 [Online]. Available: http://www.mcjournal.com/articles/arc105/arc105.htm
[7] R. Sanz and K.-E. Arzen, "Trends in software and control," *IEEE Control Syst. Mag.*, pp. 12–15, Jun. 2003.
[8] R. J. Pehrson, Software Development for the Boeing 777 The Boeing Company, Tech. Rep., 1996.
[9] H. P. Barendregt and E. Barendsen, "Introduction to lambda calculus," in *Proc. Aspenæs Workshop Implementation Functional Languages*, Göteborg, Sweden, 1988, [CD ROM].
[10] J. von Neumann, First Draft of a Report on the EDVAC Tech. Rep., 1945.
[11] C. H. Ferguson and C. R. Morris, *Computer Wars—The Fall of IBM and the Future of Global Technology*. New York: Three Rivers Press, 1994.
[12] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, Aug. 1990.
[13] W. R. Stevens, TCP/IP Illustrated, unpublished.
[14] V. Kawadia and P. R. Kumar, "A cautionary perspective on cross layer design," *IEEE Wireless Commun. Mag.*, vol. 12, no. 1, pp. 3–11, Feb. 2005.
[15] CORBA Components, Version 3.0 Object Management Group, 2002.
[16] Microsoft.NET Microsoft Inc [Online]. Available: http://www.microsoft.com/net/
[17] V. Matena and B. Stearns, *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform, ser. Java Series*. Santa Clara, CA: Sun Microsystems Press, 2001.
[18] CORBA Success Stories OMG Inc. [Online]. Available: http://www.corba.org/success.htm

[19] Middleware is Everywhere IBM [Online]. Available: http://www-306.ibm.com/software/

[20] Microsoft.NET Microsoft Inc [Online]. Available: http://www.microsoft.com/net/

[21] Jave 2 Platform, Enterprise Edition (J2EE) Sun Microsystems [Online]. Available: http://java.sun.com/j2ee/

[22] IT Convergence Lab [Online]. Available: http://decision.csl.illinois.edu/~prkumar/testbed

[23] *IEEE 802 LAN/MAN Standards Committee, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, IEEE Standard 802.11, 1999 Edition*, , 1999.

[24] A. Giridhar and P. R. Kumar, "Scheduling traffic on a network of roads," *IEEE Trans. Veh. Technol.*, vol. 55, no. 5, pp. 1467–1474, Sep. 2006.

[25] The Official Bluetooth [Online]. Available: http://www.bluetooth.com/

[26] CORBA: Core Specification Object Management Group, 2002.

[27] S. Kowshik, G. Baliga, S. Graham, and L. Sha, "Co-design based approach to improve robustness in networked control systems," in *Proc. Int. Conf. Dependable Syst. Networks*, Yokohama, Japan, Jun. 2005, pp. 454–463.

[28] K.-D. Kim and P. R. Kumar, "Architecture and mechanism design for real-time and fault-tolerant etherware for networked control," in *Proc 17th IFAC World Congress*, Seoul, Korea, Jul. 2008, pp. 9421–9426.

[29] XML W3C—World Wide Web Consortium, 2000.

[30] Java 2 Platform (j2se) [Online]. Available: http://java.sun.com/j2se/

[31] *Applied Operating System Concepts*. New York: Wiley, 2000.

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

[33] A. V. Aho and J. D. Ullman, *Foundations of Computer Science*. New York: W. H. Freeman and Co, 1995.

[34] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott, Maude Manual (Version 2.1) Mar. 2004 [Online]. Available: http://maude.cs.uiuc.edu/maude2-manual

[35] Apache ant [Online]. Available: http://ant.apache.org

[36] T. Socolofsky and C. Kale, "RFC 1180—TCP/IP Tutorial," Jan. 1991 [Online]. Available: tools.ietf.org/html/rfc1180

[37] Real-Time CORBA Specification Version 2.0 OMG, Inc, 2003.

[38] Minimum Corba Specification OMG, Inc, 2002.

[39] L. Wills, S. Sander, S. Kannan, A. Kahn, J. V. R. Prasad, and D. Schrage, "An open control platform for reconfigurable, distributed, hierarchical control systems," in *Proc. Digital Avionics Syst. Conf.*, Oct. 2000, pp. 4D2/1–4D2/8.

[40] Rofes: Real-time corba for embedded systems [Online]. Available: http://www.lfbs.rwth-aachen.de/users/stefan/rofes/

[41] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proc. IEEE*, vol. 91, no. 1, pp. 84–99, Jan. 2003.

[42] A. Traub and R. Schraft, "An object-oriented realtime framework for distributed control systems," in *Proc. IEEE Conf. Robot. Automat.*, Detroit, MI, May 1999, vol. 4, pp. 3115–3121.

[43] P. Lutz, W. Sperling, D. Fichtner, and R. Mackay, "OSACA—the vendor neutral control architecture," in *Proc. Eur. Conf. Integr. Manufact.*, Dresden, Germany, Sep. 1997, pp. 247–256.

[44] B. S. Heck, L. M. Wills, and G. J. Vachtsevanos, "Software technology for implementing reusable, distributed control systems," *IEEE Control Syst. Mag.*, vol. 23, no. 1, pp. 21–25, Feb. 2003.

[45] C. Robinson and P. R. Kumar, "Optimizing controller location in networked control systems with packet drops," *IEEE J. Selected Areas Commun., Special Issue Control Commun.*, 2008.

[46] K.-J. L. C.-C. Han and C.-J. Hou, "Distance-constrained scheduling and its applications to real-time systems," *IEEE Trans. Computers*, vol. 45, no. 7, pp. 814–826, Jul. 1996.

[47] G. Baliga, "A Middleware Framework for Networked Control Systems," Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, Urbana, 2005.

Scott Graham received the M.S. degree in electrical engineering from the Air Force Institute of Technology (AFIT), Wright Patterson Air Force Base, OH, in 1999 and athe Ph.D. degree in electrical engineering from the University of Illinois at Urbana-Champaign in 2004.

He is an Adjunct Professor at the AFIt. His research interests include directional networks, mobile networks, and networked control systems.

Girish Baliga received the B.Tech. degree in computer engineering from the National Institute of Technology, Surathkal, India and the M.S. degree in computer science, the M.S. degree in mathematics, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign.

He developed Etherware, a middleware for networked control system. He is currently a Software Engineer in the System Infrastructure Group, Google, Inc., Mountain View, CA.

P. R. Kumar received the B.Tech. degree in electrical engineering from the Indian Institute of Technology (I.I.T.), Madras, India, in 1973 and the D.Sc. degree in systems science and mathematics from Washington University, St. Louis, MO, in 1977.

He was with the Department of Mathematics, UMBC, from 1977 to 1984. Since 1985, he has been with the University of Illinois, Urbana-Champaign. He is the Franklin Woeltge Professor of Electrical and Computer Engineering (ECE). His current research interests are in wireless networks, sensor networks, and networked embedded control systems.

Dr. Kumar received the Eckman Award of AACC, IEEE Control Systems Field Award, Ellersick Prize of the IEEE Communications Society, a Doctor Honoris Causa from ETH, Zurich, and is an NAE Member.