Etherware : Domainware for Wireless Control Networks * †

Girish Baliga, Scott Graham, Lui Sha, P. R. Kumar
University of Illinois at Urbana-Champaign
{gibaliga, srgraham, lrs, prkumar}@uiuc.edu

Abstract

The promise of middleware is to enable integration and evolution of complex systems dynamically. In demanding domains such as wireless control networks, fulfilling this promise while maintaining complete generality is extremely complicated. Understanding and exploiting the forcing functions *of a domain helps manage this complexity by avoiding redundant generalizations.* Domainware *exploits this technique and adopts a simpler architecture to support more important non-functional requirements effectively.*

This paper presents Etherware*, a Domainware for wireless control networks. Capitalizing on our development of a fairly complex control system testbed, commonly supported yet redundant generalizations are identified and eliminated. The resulting architecture is simple, and can support a wide range of trade-offs that can be manipulated easily at run-time. This is illustrated by showing how the performance of* Control Time Protocol *(CTP), an Etherware service, is optimized by the additional options available in Etherware.*

1. Introduction

The fundamental philosophy of middleware is flexibility. An application with a static configuration is suitably served by custom solutions. These include application frameworks that are tailored to the application, and represent fixed architectural trade-offs. Middleware is more general; it enables these trade-offs to be manipulated during system integration, and even at run-time. To illustrate, during the development of a large distributed business application, the security settings may be lowered to provide developers complete access to implement and test various parts of the application. During deployment, however, security settings may have to be upgraded. A good middleware should support such upgrades without having to reconfigure the system or restart it.

A key trade-off attends the development of middleware. General purpose middleware supports more functionality while sacrificing some domain specific and non-functional requirements. Support for more functionality tends to increase the complexity of the middleware, making it harder to support the specific needs of applications in demanding domains. For instance, in real-time control applications, the key non-functional requirements are real-time guarantees, fault tolerance and security. It is very hard to address these requirements with a complex general purpose middleware that also supports applications in demanding domains such as wireless control networks. It is appropriate to sacrifice some generality to refine the middleware for such domains. Real-Time CORBA [19] and Minimum CORBA [17] are good examples that exemplify this methodology.

Real-Time CORBA is the specialization of CORBA for real-time applications. An implicit assumption in the specification of RT CORBA, is that communication overheads are tolerable by applications. Hence, most of the specification deals with models for scheduling threads and providing real-time guarantees on a single node. For example, the notion of a distributable thread relies heavily on good and reliable communication to avoid deadlocks. In wireless networks, however, communication delays are unpredictable and packet losses are higher. Further, most control applications have controllers operating physical systems that need periodic

* This material is based upon work partially supported by USARO under Contract Nos. DAAD19-00-1-0466 and DAAD19-01010-465, DARPA under Contract Nos. N00014-01-1-0576 and F33615-01-C-1905, AFOSR under Contract No. F49620-02-1-0217, DARPA/AFOSR under Contract No. F49620-02-1-0325, and NSF under Contract Nos. NSF ANI 02-21357 and CCR-0325716. The views expressed in this article are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

† Please address all correspondence to: gibaliga@uiuc.edu

control updates. Hence, a controller cannot depend on remote sensor updates with unpredictable delays. In fact, controllers may use state estimation techniques that are robust to losses in sensor updates. Much more important for state estimation and robust control is low delay. Feedback must be available to controllers as soon as possible.

Wireless control applications need the ability to trade off reliability for low delay - an aspect not addressed in the RT CORBA specification. This is a key requirement, as the main determinant of good performance for such applications is communication delay. Support for this could be added as another feature in RT CORBA. But adding these features to RT CORBA would be an unnecessary overhead for most of its current applications. In fact, further support for key features such as fault tolerance becomes even more complicated, since implementations would then have to be efficient as well as easy to use for embedded, wired, and wireless control. Reference [8] considers some of the other issues that need to be addressed in RT CORBA.

We believe that the identification of *forcing functions* is the key to answer the above challenge of domain-based-specialization and extension of middleware. We define a forcing function as a characteristic of an application domain that must be addressed by any application level solution. For example, control applications operating on wireless networks are constrained by lossy links and unpredictable delays. Even with dedicated optimizations, real time guarantees are very hard if not impossible to support. Hence, the application itself is forced to address this issue by incorporating complex control laws and good state estimators that adapt to network conditions.

Domainware is middleware that exploits application functionality already imposed by the forcing functions of their domain. In wireless control networks, delay and bandwidth characteristics of communication channels have high variability. Applications are forced to use good state estimation to cope with this. In fact, they are also forced to have default safe states to cope with the loss of a large sequence of updates due to bad channel conditions. For example, airplanes flying in auto-pilot mode are controlled by GPS and a local Inertial Navigation System (INS). INS is used for controlling the airplane, but its reference point drifts and GPS is used to correct it periodically. However, if communication with the GPS system is lost, then the default mode is to fly using INS. The key point is that airplanes, and in general most controllers using wireless networks, are designed to handle bad channel conditions. Domainware exploits this by relaxing its own requirements and has a much simpler architecture. As

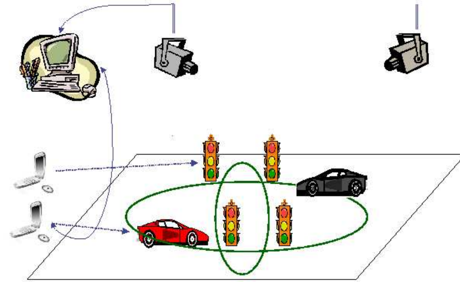


Figure 1. A Wireless Control Network Testbed

individual components are fail-safe, non-functional requirements such as fault tolerance and system evolution are addressed in a much simpler way.

We have developed a wireless control network testbed in the IT Convergence Lab [2] at the University of Illinois, and based on this experience we have developed Etherware, a Domainware for wireless control networks. Section 2 presents the wireless control network testbed that we have studied, and lists the requirements that a middleware for this domain must support. Section 3 presents Etherware and describes how this supports these requirements. We illustrate the use of forcing functions on Control Time Protocol (CTP), an Etherware service, and a typical application for wireless control networks. CTP and its operation are presented in detail in Section 4, and Section 5 demonstrates how the method of forcing functions has been applied in Etherware effectively. Section 6 presents some related work and Section 7 concludes.

2. The Testbed and its Requirements

This section describes the traffic control system that we are developing as an example application of the convergence of control with communication and computation [2]. Figure 1 illustrates the system consisting of a fleet cars and Figure 2 shows a preliminary implementation of this system.

Each car is controlled by a model predictive controller [9] operating on its dedicated laptop issuing up to 20 controls per second. The controls used to drive the car are speed and direction. The controller gets its trajectory from a Traffic Scheduler that runs a path planning algorithm called "Trajector" [11]. The controllers may also track neighboring cars to avoid collisions, which the Trajector cannot handle due to lower operating frequency.

Feedback is provided by ceiling mounted cameras that track the cars by their color coded tops. Vision

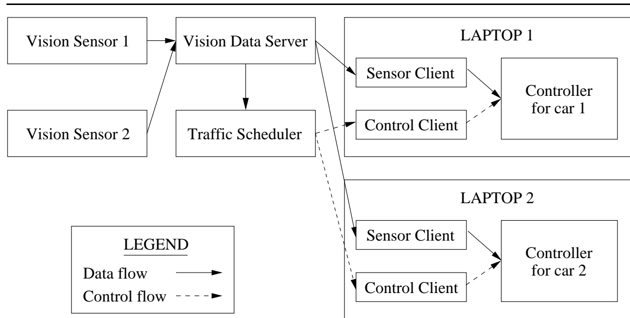


Figure 2. A preliminary implementation of the Traffic system

feed from the cameras is processed by image processors to determine the position and orientation of cars. A data server running on another laptop collects this information and serves it as feedback to the car controllers and the trajector.

We have tested scenarios with up to eight cars operating simultaneously on the track and closely following pre-specified trajectories in well behaved traffic scenarios. Other demonstrated scenarios include pursuit evasion, a leader follower configuration where a set of software controlled cars follow a manually controlled car, and an automatic collision avoidance system.

Based on this implementation and scenarios studied, we have developed the following list of requirements for middleware supporting wireless control networks.

- **Operational Requirements:** The middleware must support basic requirements such as *distributed operation* and *location independence*. Most of the communication is from sensors to controllers, and between higher level controllers and lower level controllers. All such communication is *asynchronous* and can tolerate some loss of updates. *Semantic profiling* of addressable components saves a lot of system configuration effort. For example, if a car knows its location to be (24,48), it should be able to directly connect to a vision server covering these coordinates. Support for implementing components as *single threaded code* is very useful for ease of development, testing, and upgrades.
- **Management Requirements:** Detailed, programmable interfaces are required during *startup* to ensure that all parts of the system are brought up correctly. *System evolution* requires the ability to update or migrate components at run-time. *Run time updates* allow changing controllers to respond to varying system conditions, and *component migration* optimizes the system to reduce communication and computational loads.
- **Non-functional Requirements:** *Robustness is a fundamental requirement.* Component failures and restarts must be contained and their effect on the overall system should be minimized. As shown in Section 4, *path delay characterization* is useful for delay critical control loops, and *minimal delay* for delivery of updates is essential. Algorithms should have good *scalability*. *Security* is a key requirement and the system must be protected from misbehaving or malignant components.

In the following section, we illustrate how we seek to address these requirements in the current version of Etherware.

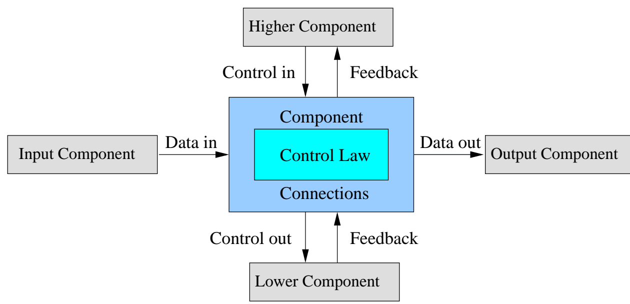
3. Etherware

This section presents Etherware, our Domainware for wireless control networks.

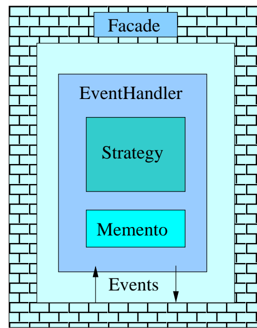
3.1. Architecture design considerations

Based on the preliminary implementation of our testbed, we have determined that the key forcing functions for our system, and wireless control networks in general, are lossy channels and instability due to delays. TCP connections tend to back-off due to losses and sometimes one connection ends up capturing the channel while others are blocked out. In the preliminary implementation we used UDP connections to communicate sensor updates. Link level retransmissions provide sufficient reliability, and UDP optimizes delay as best as can be done at the application level. Further, our controllers read in updates only between successive iterations of control loops, and any update received during an iteration is simply buffered. In fact, we can use this feature to make the controllers single-threaded and buffer updates in the middleware. The key point is that controllers operate best with asynchronous event based communication semantics.

Another key point is that controllers in wireless control networks are forced to use state estimators, and have default modes when a large sequence of updates are lost. However, there may still be need for communication such as infrequent path planning updates, where reliable in-order delivery is necessary. Hence, controllers must be able to trade off reliability for low delay. Since this is a key feature that affects system



(a) Typical component in an FCS



(b) Design patterns in an EventHandler

Figure 3. Programming Model in Etherware

performance, Etherware incorporates it as a basic design principle. In fact, the common case of almost reliable low delay communication is the optimized mode.

Fault tolerance requires the ability to restart application components when they fail. System evolution requires the ability to update and migrate application components. Both of these require a simple and uniform way to externalize application state so that it can be check-pointed for the above operations. Also, this must be done in an application aware fashion so that the instabilities are minimized. Etherware supports this as a basic architectural precept.

3.2. Programming Model

Most applications on a wireless control network are developed as a set of communicating components. A typical component has connections as shown in Figure 3(a). A component encapsulates a *control law* that models some aspect of the system, such as sensing, actuation or control. A component such as a car controller participates in a *control hierarchy*. It takes inputs from a "higher level" component such as a trajector, and controls a "lower level" component such as an actuator. A component also participates in a *data flow*. For example, car controllers obtain vision inputs from sensors. As noted in Section 2, most of this communication is asynchronous, and requires minimal delay even at the cost of some loss of updates.

The programming model for components in Etherware is based on the above description. A component in the Etherware is called an *EventHandler*. Every EventHandler has the template shown in Figure 3(b). This template is composed of a set of design patterns from [10]. As prescribed by the forcing functions discussed in Section 3.1, an EventHandler communicates with the rest of the system through asynchronous messages called *Events*. The *Facade* is a uniform interface for exchange of events. For system evolution, we need the ability to replace one control law in an EventHandler by another, at runtime. For this, objects implementing control laws must operate across the same interface. This employs the *Strategy* pattern. While replacing control laws, we must be able to communicate the accumulated state between objects, which is a problem addressed by the *Memento* pattern.

An EventHandler can be active or passive. A passive EventHandler blocks to receive events from other EventHandlers. Actuators are passive. Most sensors and controllers can also be implemented as passive EventHandlers and activated using periodic "tick" events. Active EventHandlers are multi-threaded, with one thread blocking to receive events, and other threads processing and generating events asynchronously.

Events are well formed XML [21] documents that can be created and manipulated using a hierarchy of Event classes. Each event has a *profile* and a *content* tag. The profile is a semantic description used to address the recipient of the event. EventHandlers have globally unique ids which can also be used for addressing. The content tag has the contents of the event.

By default, events are delivered reliably and in order. Since a lot of the application code usually addresses system setup, configuration, and management, reliable in-order delivery is key for such communication. However, the application components that perform most of the communication, benefit greatly from the ability to trade-off between reliability and low delays. To support this, EventHandlers can set up *EventPipes* to send a stream of events. These EventPipes have globally unique ids and properties that specify how the events in the pipe are to be delivered. Possible specifications for delivery of events include best effort, in-order delivery with some losses, delivery with statistical guarantees, etc. EventPipes can have additional *Taps* that allow other EventHandlers to "filter"

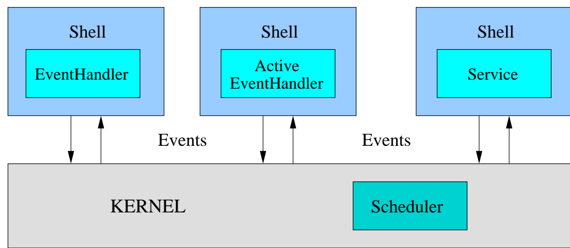


Figure 4. Per-node configuration of Etherware

events. For example, if there is an EventPipe between a sensor and a controller, but the data in the stream of events generated by the sensor has a lot of noise, then a filter can be added at runtime, using a Tap, to reduce this noise.

3.3. Etherware Architecture

Figure 4 illustrates the basic architecture of Etherware. Each EventHandler is encapsulated in a *Shell* that maintains EventHandler specific information and is its interface to the rest of the system.

The design is based on the micro-kernel concept used in operating systems [7]. The *Kernel* implements basic functionality for delivering events, managing EventHandlers, and has a *Scheduler* to schedule their operation. The *Kernel* delivers local events between the EventHandlers that it manages. All other Etherware services are implemented as EventHandlers. This permits these services to also be replaced, restarted and manipulated as EventHandlers. The *Kernel* also exposes a service management interface that provides reflection [15].

We consider three of the basic services provided in Etherware, and useful in most applications.

- The **Profiler** is like a name service and maintains registered profiles of EventHandlers.
- The **RemoteBus** sends and receives events from remote EventHandlers. It hides network specific information such as IP addresses from the rest of Etherware.
- The **Ticker** is intended to have the only active thread in Etherware, apart from threads that block on sockets in the RemoteBus. Other EventHandlers that need to be awakened periodically, such as fixed rate controllers, can register with the Ticker to receive periodic "tick" events.

We now consider how the requirements outlined in Section 2 are addressed in Etherware. Operational requirements such as distributed operation, location independence, and semantic profiling, are supported by profiles and globally unique ids of EventHandlers. All events are sent and received asynchronously. The delay vs. reliability trade-off can be tuned at run-time using configurable EventPipes. Path delay characterization is provided as an Etherware service. Management requirements are addressed by migration of EventHandlers, and reflection in the Etherware Kernel. Architectures and services for non-functional requirements such as robustness and security are tied to the ongoing implementation of our traffic system using Etherware.

3.4. Implementation issues

Etherware has been implemented on the Java 2 platform Standard Edition [4], for platform independence. As noted in Section 3.2, events are XML documents. This enables the definition of the *Service Interface* of an EventHandler that describes the set of events it accepts, and specifies the formats of these events using document type declarations (DTDs). This also allows EventHandlers to "learn" to talk to each other at run time by looking up respective *Service Interfaces* that are defined using a common ontology. Taps and filters can be used to implement the kind of functionality provided by Aspect-Oriented Software Development [1].

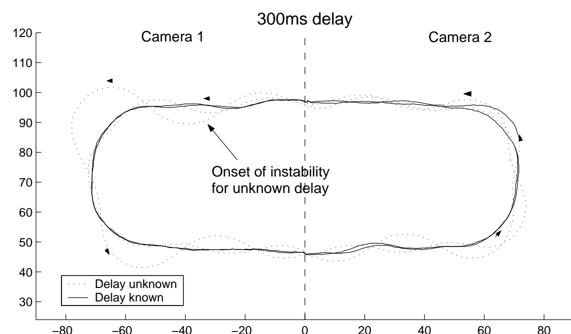
4. Control Time Protocol

This section describes the Control Time Protocol (CTP), a time-stamp translation service we have developed in Etherware.

4.1. Need for time-stamping

In networked control systems, updates convey the sampled state of a system as feedback from sensors to controllers. Delaying delivery of feedback samples in a control loop can cause an otherwise stable system to become unstable. However, when updates are timestamped, a controller can still use state estimation techniques to extrapolate the current state of the system.

Figure 5 illustrates how delays affect the stability of a car controller. The dotted line indicates the car's trajectory when the delay was unknown to the controller. The solid line indicates the recovery and the resulting stability of the trajectory when the delay was made known through timestamps. When the delay is not known, we see the onset of instability at a delay of about 300ms. Instability onset in the known delay case did not occur until the delay reached 800ms. By time-stamping updates, we see that this system can remain stable for an additional 500ms of delay.



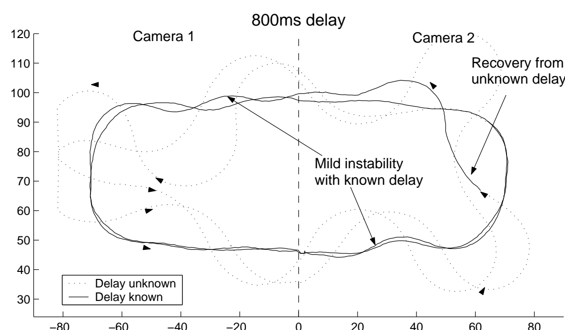


Figure 5. Stability effects of known vs unknown additional delay

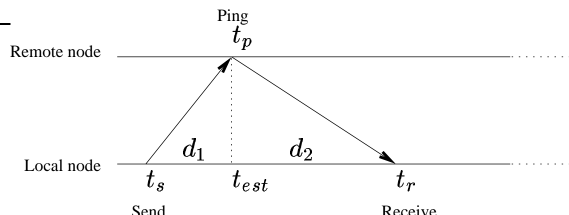


Figure 6. Message exchanges in Control Time Protocol algorithm

4.2. Control Time Protocol

Time-stamping poses several challenges in a distributed environment. Distributed sensors and controllers usually do not have access to the same clock. Hence, timestamps created at a sensor must be translated somehow into the time reference of the controller. CTP is an Etherware service that automatically timestamps all events and performs necessary translations.

Figure 6 shows the periodically exchanged messages in CTP. Assuming symmetric delays, as in NTP [5], the remote ping time t_p is aligned with the midpoint between the send time t_s, and receive time t_r on the local node. If the delays are indeed symmetric, the algorithm gives exact estimates. If the delays are asymmetric or noisy, the error is still less than half the round-trip delay time.

CTP uses a windowed least-squares approach [12], to compute a best-fit line representing the skew and offset between the nodes. The relationship is given by $t_{remote} = \alpha\ t_{local} + \beta$. Note that this is done on a pairwise basis between all communicating nodes.

4.3. Effects of delay on CTP

From the above discussion, we can see that the offset computations used in CTP may be in error when the delays are asymmetric. However, the error is bounded by the round trip time, and hence low round trip times are desirable. Because clocks are almost linear, CTP is hardly affected by a few lost updates. As seen in the next section, CTP is better served by a lower average delay at the expense of a few lost updates.

5. Evaluation

This section demonstrates how the forcing functions of wireless control networks have been effectively exploited in Etherware. We present a comparison of the performance of Etherware with two widely available implementations of CORBA - ROFES 0.3b [6] and JacORB 2.0[3]. ROFES supports part of the Real-Time CORBA 1.1 specification [18] and the Minimum CORBA specification [17]. JacORB supports the CORBA 3.0 specification.

We implemented and executed CTP on Etherware, ROFES and JacORB. A reference implementation in Java using sockets was used as the baseline. The configuration was basically a client pinging a server at a frequency of 10 Hz. The time line for this is as shown in Figure 6. The client periodically sent pings to the server which responded with a time-stamp t_p. Each ping packet and its response had about 1500 bytes of "payload" data to simulate actual packets in the system. At run time, only time stamps were collected and recorded in log files. The CTP estimation algorithm was run offline on the accumulated log files. Hence the delays in the experiments were mainly due to overhead of the middleware and its communication protocols.

We considered three scenarios for connecting the client and server - a wired network, an ad hoc wireless network (single hop), and a wireless network in base-station mode (two hops). The base-station mode is a fairly standard configuration for IEEE 802.11 wireless networks. The wired network was a 10 Mbps Ethernet.

	Min	Mean	Std Dev	Median
Java	0	0.2258	1.1026	0
JacORB	1	2.6532	2.1488	2
ROFES	0	5.3765	4332	3
Etherware	14	17.4990	1.3575	18

Table 1. Round trip time statistics for wired link

	Min	Mean	Std Dev	Median
Java	1	13.1169	12.9551	8
JacORB	16	29.6037	9.3428	28
ROFES	16	38.7757	11.9378	38
Etherware	17	28.9885	11.0453	25

Table 2. Round trip time statistics for wireless link (one hop)

The IEEE 802.11 wireless networks used Cisco Aironet 350 series cards. The base station was an ORiNOCO AP-1000 access point. All code was executed on Pentium III machines with Red Hat Linux 9.0 as the operating system. The wired networks were isolated LANs, but each wireless network scenario had cross traffic from the other scenario, neighboring labs, and the campus network.

Tables 1, 2, and 3 show round-trip time statistics for the three scenarios considered. According to Figure 6, the round-trip time for each ping is given by $rtt = t_r - t_s$. Table 1 illustrates the overhead in the implementations of the middleware. The high mean delay for Etherware is mainly due to a preliminary sub-optimal implementation. This also includes overheads due to multi-threading and processing of events which are essentially XML documents. Hence the remaining comparisons clearly favor the CORBA implementations. In the one-hop wireless link case of Table 2, we see that the three implementations have comparable minimum delays, while Etherware has the lowest mean delay. This is further accentuated in the base-station case of Table 3. These statistics show how the delay-reliability trade off is affected by different protocols and interaction semantics of these platforms.

	Min	Mean	Std Dev	Median
Java	2	26.5024	23.9759	18
JacORB	14	45.8100	25.2337	36
ROFES	14	50.1266	23.7878	44
Etherware	18	39.8066	17.7430	35

Table 3. Round trip time statistics for Base station mode (two hop)

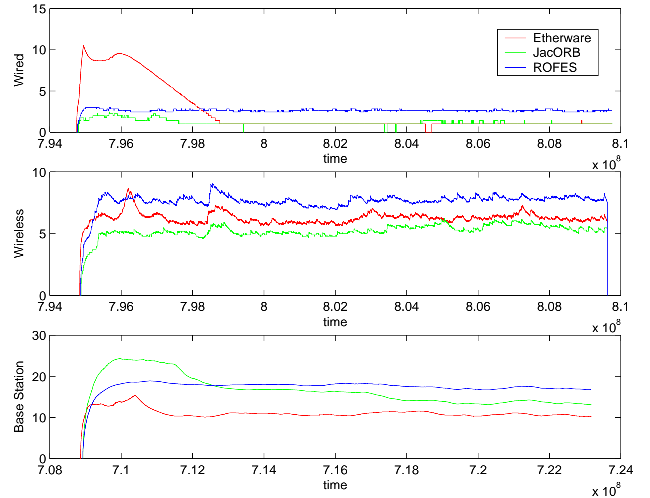


Figure 7. Comparison of mean error for the three scenarios

We now consider the impact of these statistics on the estimation algorithms of CTP. We verify the accuracy of CTP's estimate, assuming symmetric delays. In Figure 6, this implies $d_1 = d_2 = d$. t_{est} is CTP's estimate of the remote time stamp t_p based on d. The difference $err = t_p - t_{est}$ can be positive or negative. To consider the long-term behavior we plot the exponentially weighted moving average e, of err, computed iteratively as follows:

$$e_t = \sqrt{\lambda e_{t-1}^2 + (1-\lambda) err_t^2}$$

Here, err_t^2 is the squared-error at time t and e_t is the average up to time t.

Figure 7 show plots with $\lambda = 0.9995$. The first plot shows the performance of the estimator in the wired network case. In the steady state, the errors are similar and quite small (less than 3 ms). The primary cause for error in this plot is jitter. However the error is bounded by round-trip time. Thus we see that the small jitter in Etherware allows it to have a tight error bound in spite of a larger average round-trip time. The initial transient seen in the Etherware plot is due to startup contentions. The second and third plots show corresponding performances for the one-hop and two hop wireless cases, respectively. We see that performance degrades in all cases, but the degradation for Etherware is lesser than for the CORBA implementations.

One explanation for these results is that the CORBA implementations use TCP for all communication as the specifications for CORBA and Real-Time CORBA do

not mention UDP as a candidate protocol. On the other hand, Etherware allows the application to opt for UDP if necessary. This choice does not matter much for embedded systems and wired networks, as seen in Table 1, and more importantly, in the success of Real-Time CORBA for various control applications. However, in wireless networks, basic TCP is known to perform badly. As we have argued in this paper, wireless control applications do not benefit from the reliability offered by TCP, especially if this is at the cost of a higher average delay.

6. Related work

Popular approaches to developing application frameworks and middleware for networked control applications are based on various flavors of CORBA, such as Real time CORBA[19] and Minimum CORBA [17]. For example, OCP [22] is based on Real Time CORBA and has been used to control unmanned aerial vehicles. In Section 5, we considered ROFES [6], which is targeted for real-time applications in embedded systems.

Other interesting approaches include Giotto [14], real-time framework [20] for robotics and automation, and OSACA [16] for automation. A good overview of research and technology that has been developed for implementing reusable, distributed control systems is provided in [13].

Regarding time translation, standard synchronization tools such as NTP [5] directly change the system clock. Hence, they may not always be applicable, especially between computers operating in different administrative domains. However, since CTP does not manipulate clocks, it is still useful even in such scenarios.

7. Conclusions and Future Work

In this paper, we have considered some of the key challenges facing middleware for control applications. We have presented the approach of using forcing functions to address the challenges of domain-specific extension and specialization of middleware. Domainware is middleware that exploits application functionalities imposed by the forcing functions of its domain. We have presented Etherware as a Domainware for wireless networked-control applications. We have also compared Etherware to two CORBA implementations and demonstrated how it leverages the ability of its applications to tolerate losses. In future work, we intend to address issues such as fault tolerance and security in wireless control networks.

References

[1] Aspect oriented software development. http://aosd.net/.

[2] Information technology convergence lab, csl. http://decision.csl.uiuc.edu/ testbed/.

[3] Jacorb. http://www.jacorb.org.

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

[5] Ntp: The network time protocol. http://www.ntp.org/.

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

[7] *Applied Operating System Concepts.* John Wiley and Sons Inc, first edition, 2000.

[8] The Boeing Company. *Real-Time CORBA Trade Study*, 1999.

[9] E. F. Camacho and C. Bordons. *Model Predictive Control.* Springer Verlag, June 1999.

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

[11] A. Giridhar. Scheduling traffic on a road network. Master's thesis, University of Illinois at Urbana-Champaign, December 2002.

[12] G. C. Goodwin and K. S. Sin. *Adaptive Filtering: Prediction and Control.* Prentice Hall, N. Y., 1984.

[13] B. S. Heck, L. M. Wills, and G. J. Vachtsevanos. Software technology for implementing reusable, distributed control systems. *IEEE Control Systems Magazine*, 23(1), February 2003.

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

[15] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6), June 2002.

[16] P. Lutz, W. Sperling, D. Fichtner, and R. Mackay. Osaca - the vendor neutral control architecture. In *Proc. European Conference on Integration in Manufacturing*, pages 247–256, Dresden, Germany, Sept 1997.

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

[18] OMG, Inc. *Real-Time CORBA Specification Version 1.1*, Aug 2002.

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

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

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

[22] 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 *Proceedings of the Digital Avionics Systems Conference*, Oct 2000.