Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

IFAC

Architecture and Mechanism Design for Real-Time and Fault-Tolerant Etherware for Networked Control ⋆

Kyoung-Dae Kim and P.R. Kumar

Department of Electrical and Computer Engineering and Coordinated Science Laboratory
University of Illinois, Urbana-Champaign, USA
(E-mail: kkim50, prkumar@uiuc.edu).

Abstract: We believe that a standard control software framework which enables rapid, reliable and evolvable application development is the key for the proliferation of the networked control systems. Accordingly, we have been working on developing a domainware for general purpose control system, called Etherware. Even though Etherware supports many of the distributed system domain requirements, it still needs further advances to be more suitable as a middleware for control systems. In this paper, we present an architecture and mechanisms for an enhanced middleware that supports networked control system design through enabling temporally correct interactions. We also propose an architecture and mechanisms for enhancing the robustness of networked control systems to faults, that we are currently implementing.

Keywords: Middleware, real-time systems, fault-tolerant systems, networked control systems

1. INTRODUCTION

Advances in wireless communication, embedded computing, sensor and actuator technologies, are leading to a third generation of control systems. This next generation of systems however raises several issues that we need to overcome to take advantage of the opportunities. In particular, developing a general purpose *domainware*, an application domain specific middleware, for distributed control systems, which enables rapid, reliable and evolvable application development, is critical for the large scale development of (wireless) networked control systems (Graham et al. [2004]).

An early version of such a middleware, called Etherware (Baliga [2005]), has been developed and implemented in the Information Technology Convergence laboratory at the University of Illinois. To enhance the usefulness of Etherware as a domainware for control systems, it is important to extend it to provide *real-time* guarantees and *fault-tolerance*. In this paper we extensively address middleware architecture and mechanisms that can provide such real-time and fault-tolerant capabilities to Etherware.

There have been several research works aimed at developing a middleware for distributed real-time embedded (DRE) control systems (Brinkschulte et al. [2001], Arzen et al. [2007]). Even though they are targeted at the domain of control systems, most of them focus on the issue of real-time. In recent years, some research has been aimed at developing a more reliable real-time middleware system, by combining Real-Time CORBA and Fault-Tolerant CORBA (OMG [2003, 2004]). However, there are several issues with conventional Fault-Tolerant CORBA which are not quite compatible with real-time systems. To resolve these problems, (Gokhale et al. [2004]) and (Balasubramanian et al. [2007]) propose a SEMI-ACTIVE replication strategy and a resource utilization based replication selection algorithm, respectively. However, there are still several issues which need to be considered, such as additional communication overhead for fault-tolerant operation and unpredictable delays for timely fault management.

Therefore, in this paper, we take a different approach to provide both real-time and fault-tolerant capabilities to Etherware. We believe that our approach can achieve the goal more efficiently since it can not only provide a simple programming model for real-time and fault-tolerant application development, but it can also reduce the unpredictable behavior of fault-tolerant operation by eliminating interactions over the communication network while preserving predictability in real-time scheduling.

2. REQUIREMENTS ANALYSIS

We begin our discussion by first introducing key domain requirements to show why both real-time and fault-tolerance issues are important to domainware for general purpose networked control systems.

2.1 Location Transparency

In distributed systems, entities comprising the system are not located in one physical location. They are usually connected to each other through a communication network. Therefore, an application developer wanting to

⋆ This material is based upon work partially supported by ORNL under Contract No. DOE-BATT-4000044522, NSF under Contract Nos. NSF ECCS-0701604, CNS-07-21992, CNS 05-19535, CCR-0325716, and USARO under Contract No. W-911-NF-0710287.

develop an application has to first deal with the existence of a network, in order to make application components interact each other. However, since network application programming is much harder to design, implement and debug than single computer applications, the time, cost and effort for application development are much higher in general. Therefore, if middleware can hide the existence of the underlying communication network from users, then it can mitigate a lot of burden of the application developer.

2.2 Hiding Time Discrepancy

All clocks are different in distributed systems. This can potentially cause some serious problems in control systems because the stability and performance of control systems is highly sensitive to delays and time synchronization accuracy, in general. For example, if a controller component receives a measurement from a sensor component, it needs to know when the measurement from the plant was taken to calculate an appropriate control command. There are two mechanisms to cope with time discrepancy problems. First, data exchanged between components can be time stamped. Second, the time at another computing node can be translated into the time at the local machine. Since these tasks should be done in every distributed control application, it is not an efficient way to make application developers handle these tasks by themselves whenever they develop an application.

2.3 Semantic Addressing

For interactions among components, each component should have a *name*. One way of naming each component is by using the IP address of the computing node where the component is executing. However, it is not desirable for portability or reusability to use IP address for naming. For example, if a controller component calls a sensor component using its IP address, the controller component has to be rewritten whenever either the sensor component is migrated to another computing node or the controller component needs to be used in another environment. Instead, if the controller component can use semantic names such as 'room temperature sensor', it might be not only easier to develop the controller component but it also increases portability of the sensor component and reusability of the controller component.

2.4 Supporting System Evolution

Deployed systems are always being changed. Not only the system itself, but also the decisions of system users, undergo change as time goes on. For example, the characteristics of a control system are continuously changing from its initial conception when the controller is designed, as its operation environment changes. System developers may want to deploy a newly designed control algorithm to compensate for system changes. In other situations, the system developer may want to move a software component from one computing node to another. This is called software component migration. In these situations, if the controller cannot be replaced or migrated at runtime, then there is no way to do it without stopping the whole system. However there are many control systems where the whole system cannot be stopped for runtime management. Therefore, it is desirable for a middleware to support *runtime management functionalities* to allow the system to continuously evolve.

2.5 Supporting Timeliness

The stability and performance of control systems are very dependent on the latencies involved in the interaction with physical entities through control behavior such as sensing and action. For example in a mobile robot system, the robot should be able to avoid a collision with obstacles while it is moving. However, this collision avoidance activity is possible only if the robot detects an obstacle in a timely fashion, the decision about how to avoid collision is made in a timely fashion, and finally the decision is delivered to the actuation module in a timely fashion. If any of these tasks does not execute in a timely fashion, for example the detection module suffers excessive delay in informing the decision module about the existence of an obstacle, then the robot can collide with the obstacle. Therefore, the ability to execute every critical task in time is an essential requirement for control systems. More generally, the middleware should provide mechanisms to build systems with guarantees on *temporally correct behavior*.

2.6 Enhancing Reliability

In many cases, a control system is a safety-critical system. A safety-critical system is one in which the cost of system failure is very expensive, such as severe damage or harm to people, equipment or environment. Avionics, road traffic control systems and robotic surgery systems are examples of safety-critical systems. However, it is very important for system developers to make sure that the control system will not fall into an uncontrollable state that threat safety. If this is not guaranteed or verifiable, then the system cannot be used in operation. Therefore, the reliability of systems is a crucial requirement for any type of safety-critical control system.

3. ETHERWARE

3.1 Component Model

Our earlier generation middleware (Baliga [2005]), *Etherware*, was developed to support *component-based application development*. In Etherware, each component interacts with other components by exchanging messages. *Message* is a well-defined XML document object. A new type of message can be defined by utilizing the Message class hierarchy. The type of message prescribes the semantics of interaction. In every message, *profile* tag and *content* tag are pre-defined. In content tag, arbitrary types of interaction semantics can be specified. The profile tag contains the message recipient's semantic address.

In designing the component model, several *software design patterns* (Gamma et al. [1995]) were used to support the domain requirement of system evolution. *Shell* performs a central role in the component model. First, it manages the life-cycle of a component which is encapsulated by it. Second, Shell provides a unique channel which allows a component to interact with the system. The basic design

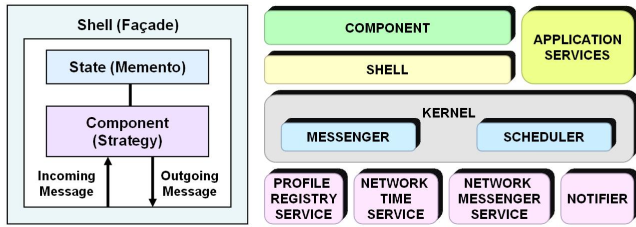

Fig. 1. Etherware architecture and component model

to implement Shell is based on the *Facade* design pattern. The *Strategy* design pattern is used to design a uniform interface between Shell and all components. Due to this Strategy design pattern, Shell can do *runtime component replacement*. For *component migration*, the execution state of a component should be continued smoothly after the migration to reduce the performance degradation. Therefore, the *Memento* design pattern was adopted to support this feature.

3.2 Etherware Architecture

The architecture of Etherware is based on the concept of microkernel in operating systems. Roughly, Etherware consists of a *Kernel*, Etherware *service components* and *application components*. In Kernel, several core functionalities for middleware operation, such as component lifecycle management, message delivery among local components, and scheduler for message delivery are implemented. In Etherware, a message can be delivered to a recipient component by scheduling a *job*, a scheduling entity of Etherware, which is a pair of message and Shell of recipient component, within Kernel. The scheduler's role in Kernel is to take the job from the messenger of Kernel and then enqueue it into the queue of the job execution module, called *TASK*.

3.3 Etherware Services

Etherware supports several functionalities that are commonly required for distributed control applications as Etherware services. *ProfileRegistry* is a naming service which is implemented in Etherware to support the semantic addressing requirement. All the details about the network are hidden from the user by the *NetworkMessenger* service which is responsible to deliver messages over the network. *NetworkTime* service performs automatic time-stamp translation. It translates the time stamp from the clock of the remote computing node to that of the local machine, for every message which is received by NetworkMessenger.

Basically, Etherware is an *event-driven system* such that a component gets executed only when it receives a message. However, in many cases, control actions need to be taken based on *time*. In such situations, the *Notification* service enables a component to execute at the time when it has to, by sending a notification message to that component.

4. ADDITIONAL FUNCTIONALITIES NEEDED BEYOND ETHERWARE

As shown in Table.1, Etherware satisfies several requirements of the distributed system. However, it still needs additional features to satisfy requirements which are essential for control systems. For example, Etherware does not support any timeliness guarantees at all. It also needs enhancements for reliability. Therefore, we propose several middleware mechanisms to support these requirements. In the following sections, we discuss these one by one.

Requirements	Etherware Implementation
Location Transparency	NetworkMessenger service
Hiding Time Discrepancy	NetworkTime service
Semantic Addressing	ProfileRegistry service
System Evolution	Component model
Timeliness	
Reliability	

Table 1. Domain requirements vs. Etherware

5. PROPOSED MECHANISMS FOR REAL-TIME GUARANTEES

5.1 Issues for Real-Time Properties

One of the most important issues for real-time behavior is *predictability* of the system (Buttazzo [2004]). However, predictability depends on everything, including H/W system, operating system, communication network, programming language, etc. Since we are looking at the problem at the level between application and platform (including communication network), the issues of platform predictability are out of the scope of our discussion. Therefore, from now on we assume that the underlying platform itself provides predictable characteristics. We focus on several issues at the middleware level which should be taken care of for real-time application.

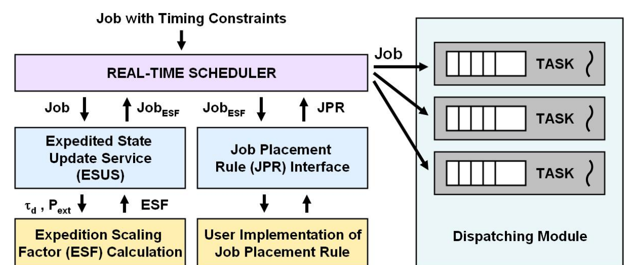

Fig. 2. Hierarchical real-time scheduling mechanism

5.2 Real-Time Scheduling

Real-time scheduling is the very first step for supporting predictability. Many different types of QoS, from static (e.g., period) to dynamic attributes (e.g., deadline), are used for real-time scheduling. The scheduling policy itself can be arbitrarily complex and have multiple layers of scheduling decisions using many types of QoS. In this paper, we adopt the concept of *hierarchical scheduling* (Gill et al. [2001]) to support an arbitrary scheduling policy. The main idea behind this design is that at the first stage the scheduler schedules jobs based on some *static QoS*, so that it can classify the static class, and then, at the second stage, the scheduler schedules the jobs within the same static class using some *dynamic QoS*. By combining these two static and dynamic scheduling hierarchies, it is possible to realize many scheduling policies from complete

static at one extreme to complete dynamic at the other. As shown in Figure 2, the *real-time scheduler* (*RT scheduler*) in middleware does not make any decision about the scheduling rule employed. Its primary role is to support any specified *job placement rule* (*JPR*), a decision about where a job will be placed in the queue of TASK. The module specifying the particular JPR should be implemented by the middleware user. Once the JPR is available, the RT scheduler picks TASK, and enqueues the job in the appropriate position within the queue.

5.3 CPU Resource Manager Service

Temporal protection is another important issue for predictability (Buttazzo [2004]). The schedulability analysis for real-time scheduling completely relies on the timing constraints specified in a job. Even though there are techniques, such as static analysis and sampled input based testing, to estimate or measure the WCET (Wegener et al. [2001]), it is very difficult to get the exact execution time parameters in practice. Therefore there is always some possibility that a job can overrun its WCET while it is executing. If this happens, it may lead to missing its deadline in many tightly scheduled systems. This is called a *temporal fault*. This can cause further temporal fault propagation and make yet other jobs to miss their deadlines too.

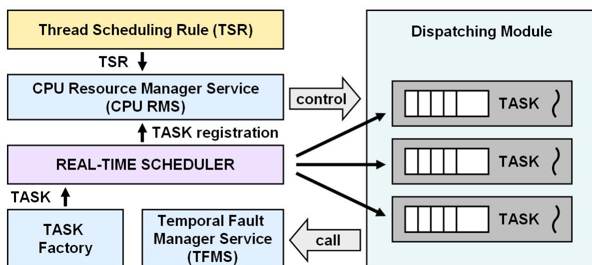

Fig. 3. CPU RMS and TFMS

CPU resource manager service (*CPU RMS*) is an Etherware service which is responsible for CPU resource sharing among TASKs. However, since a pure fixed priority based scheduling can be more appropriate in some situations, the decision about whether a temporal protection mechanism will be used or not should be made by the user. This decision should be described in a configuration file, called *Thread Scheduling Rule* (*TSR*). In TSR, information such as how many static classes will be running, which scheduling model will be used, and what are the attributes for the specified scheduling model, should be specified. The scheduling model can be either a fixed priority or a CPU resource sharing. If a fixed priority scheduling model is specified, then RT scheduler schedules jobs based only on the priorities inferred from JPR as explained in Section 5.2. If a CPU resource sharing scheduling model is specified, all TASKs share the CPU resource based on the specification. How to share the CPU resource among TASKs depends on the resource sharing model which has been used in implementing CPU RMS. For example, the constant bandwidth server (CBS, Buttazzo [2004]) can be used for implementing CPU RMS. For dynamic TASK life-cycle management, RT scheduler interacts with TASK Factory and CPU RMS. Whenever a new TASK needs to be created, RT scheduler first calls TASK Factory and then registers the created TASK into CPU RMS so that it can be controlled for CPU resource sharing. *Temporal fault manager service* (*TFMS*) is discussed in Section 6.4.

5.4 Expedited State Update Scheduling

The *state estimator design pattern* was proposed in Etherware for *Local Temporal Autonomy* (LTA) (Graham et al. [2004], Robinson et al. [2005]). However even though an estimated state can be used as a surrogate when measurements are delayed, performance will be degraded as the update delay time increases (Baliga [2005]). Therefore, the new data for a component which has been experiencing update delay longer than other components should be delivered in an expedited fashion to prevent performance degradation. *Expedited state update scheduling* (*ESUS*) is an end-to-end type of scheduling to support this. The main idea of ESUS is that whenever a real-time job is scheduled, it advances the deadline of the job based on the update delay time. This advanced deadline, called expedited deadline, will be used by the real-time scheduler in scheduling. For this purpose, a scaling factor, called *expedited scaling factor* (*ESF*), is computed by calling the *ESF calculation module* which should be implemented by the user. More generally, this feature allows modification of priority based on runtime information, i.e., adaptation.

6. PROPOSED MECHANISMS FOR RELIABILITY

6.1 Issues for Reliability Enhancement

Etherware's approach to reliability is to increase local temporal autonomy by reducing operational dependencies among components. However, the system cannot be made reliable enough without having appropriate *fault detection and management mechanisms*, since faults can occur in unexpected situations. Therefore, in this section, we discuss our proposed systematic fault detection and management mechanisms to improve reliability.

6.2 Types of Faults

One type of important fault is an *execution fault*. This fault can occur when a software component or the computer system crash while it is executing. In a distributed environment, the operational correctness of a component might depend on the operational correctness of other components or the communication network. Therefore, an operational fault can occur at runtime even though the component itself operates correctly. We call this an *interaction fault*. A third type of fault, called *semantic fault*, can occur even though everything in the system is operating functionally correctly, i.e., even though there are no execution or interaction faults. For example, the performance of a control system depends on the performance of the control algorithm. For the same target system, some controllers can control the system with very small error, while others can even make the system unstable. However, there is nothing wrong with the latter controller vis-a-vis its functional correctness. We call this a semantic fault because the semantics of a component's operation can cause a failure.

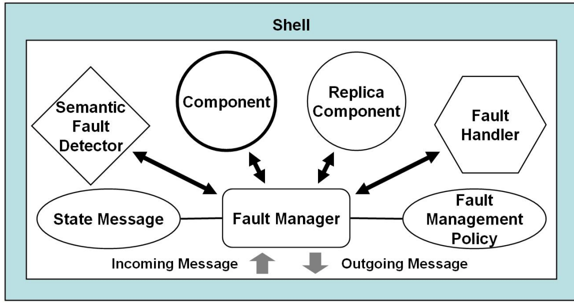

Fig. 4. Component model for fault-tolerance

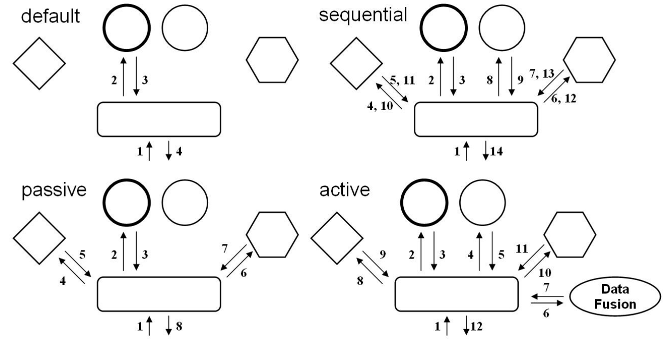

Fig. 5. Fault management policies

6.3 Component Model for Fault-Tolerance

To improve reliability, we propose a *fault-tolerant component model* (*FT component model*). The fundamental philosophy behind this is that the best place to detect and handle faults is the very place where the fault occurs. The FT component model is designed on top of Etherware's component model. So, most of the fault management activities are performed within Shell. The FT component model provides an application developer with a mechanism to detect and handle semantic faults, execution faults and interaction faults. In addition to this, the FT component model also supports redundancy based fault-tolerance by allowing multiple components which are functionally equivalent to be managed within Shell for redundant operation.

Since a semantic fault is related to a component's operational semantics, only the component developer can determine whether the result of component operation is a fault or not. Therefore, the user of the FT component model should implement a semantic fault detector within the component model. There are two different types of execution faults, *exception faults* and *temporal execution faults*. Exception faults can be caused by logical programming errors such as divide-by-zero, and this type of fault can be treated within Shell by making use of the exception handling mechanism of programming languages such as Java. Temporal execution faults can occur when a component overruns its WCET due to some reasons such as an infinite loop or blocking. Since this type of fault cannot be detected within Shell, middleware support is necessary. The source of interaction faults, such as network failure or another component's execution fault, resides outside of a component. Therefore, their detection also should be done outside of a component, using middleware support. In any case, it is important to handle these faults, no matter where they are detected, at one place, where an application developer can appropriately manage them. The fault handler in FT component model is the place where a user can implement such fault handling logic.

The primary role of fault manager is to coordinate all the interactions among components within the FT component model. The *fault management policy* (*FM policy*) provides decisions about how to coordinate them. As replication strategies in FT-CORBA, four types of FM policies are pre-defined and implemented in current FT component model. These FM policies are shown in Figure 5.

A conventional Etherware component model can be emulated by using the *default* FM policy. In the *passive* FM policy, the primary component processes all incoming messages first, and then the result, called state message, is processed by a semantic fault detector to check for any semantic fault. If a semantic fault is detected, the fault manager calls fault handler to make it handle the fault. The *sequential* FM policy repeats the passive policy until the processed result is non-faulty. If all redundant components generate faulty results, then fault manager calls fault handler. In the *active* FM policy, fault manager calls all functional components one by one and collects results from all of them before it calls the semantic fault detector. When the active FM policy is used, a user needs to implement an appropriate data fusion method, which is an abstract method of fault manager.

6.4 Relevant Middleware Services

As mentioned above, temporal execution faults and interaction faults need to be detected outside of the FT component model. For this purpose, we propose two middleware services. Basically, *temporal fault manager service* (*TFMS*) supports temporal protection at the granularity of a job. The detection of a job's temporal fault is possible if the platform supports overrun (or deadline if necessary) handler registration and callback mechanism (real-time specification for Java (Bollella et al. [2000])). TFMS is registered as a default callback handler for all TASKs in the system by the real-time scheduler when it creates a new TASK. Then, as shown in Figure 3, whenever a temporal fault is detected while executing a job, the platform calls TFMS. Once TFMS is called, it first determines which job is the source of the fault. Then it sends a temporal fault message notifying the temporal execution fault to the fault manager of the corresponding component. The *interaction fault detector service* (*IFDS*) determines whether an interaction fault has occurred or not, using the maximum data update delay time specified. If a component wants to receive interaction fault detection service from IFDS, it first needs to send a request message with the maximum data update delay time in it. Whenever the component receives new data from other component, the component sends to IFDS a reset message to reset the watchdog timer in IFDS. If the watchdog timer expires, IFDS sends an interaction fault message to the component. This fault message is then to be processed by the fault handler of the component.

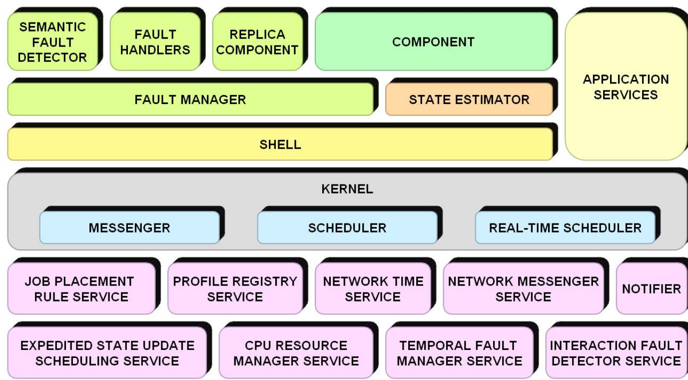

Fig. 6. Real-time and fault-tolerant Etherware architecture

7. CONCLUSION

In this paper, we first investigate what are the requirements that a domainware for (wireless) networked control systems should satisfy, in order to support rapid, reliable and evolvable application development. Based on this, we determine that two essential requirements that need to be provided, beyond what is provided by Etherware, are real-time support and support for mechanisms to enhance fault-tolerance. Based on this, we propose several middleware mechanism designs to support these domain requirements. For real-time support, we design a hierarchical real-time scheduling mechanism which can flexibly adapt to many different types of scheduling policies. We also provide a mechanism for temporal protection in TASK level via CPU RMS, and job level via TFMS. ESUS is an end-to-end scheduling model which considers the interaction delay time in real-time scheduling, to expedite data delivery if the delay is longer than specified. To enhance reliability, we categorize faults into three types. An FT component model is proposed to provide a simple but effective fault detection and management mechanism for the application developer. Two middleware services for fault detection are also designed to complete the fault-tolerant middleware mechanism.

Figure 6 is the proposed Etherware architecture which represents all the important functional modules which comprise the overall proposed middleware infrastructure. Since the proposed architecture and mechanisms are based on the assumption of the predictability of underlying platform, they can be used in any types of networked control systems regardless of communication medium.

8. FUTURE WORK

The proposed design for a real-time and fault-tolerant Etherware is currently being implemented. Many parts of the proposed FT component model have been implemented and have undergone functionality testing through several examples. However, several mechanisms such as TFMS and IFDS still need to be implemented. For real-time support, we use a real-time java virtual machine which implements RTSJ as an implementation platform. To implement the hierarchical RT scheduler, the existing Etherware has already been modified to support multiple concurrent message processing. Currently, the hierarchical RT scheduler itself is under implementation. The CPU RMS and ESUS will also be implemented after implementing scheduler.

9. ACKNOWLEDGEMENT

We thank Professors M. Caccamo, C.D. Gill, L. Sha, K.G. Shin, P. Tabuada for valuable discussions.

REFERENCES

K. Arzen, A. Bicchi, G. Dini, S. Hailes, K.H. Johansson, J. Lygeros and A. Tzes. A component-based approach to the design of networked control systems. *European Journal of Control*, vol. 13, no. 2–3, June 2007.

G. Baliga. A middleware framework for networked control systems. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

J. Balasubramanian, S. Tambe, A. Gokhale, D.C. Schmidt, C. Lu, and C. Gill. FLARe:a Fault-tolerant lightweight adaptive real-time middleware for distributed real-time and embedded systems. Unpublished, http://www.cse.wustl.edu/sinschmidt/PDF/rtss2007.pdf

G. Bollella, B. Brosgol, S. Furr, D. Hardin, P. Dibble, J. Gosling and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

U. Brinkschulte, A. Bechina, F. Picioroaga, E. Schneider, T. Ungerer, J. Kreunzinger and M. Pfeffer. A microkernel middleware architecture for distributed embedded real-time systems. *Proceedings of 20th IEEE Symposium on Reliable Distributed Systems*, pp. 218–226, New Orleans, USA, 2001.

G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd edition. Springer, 2004.

E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Resuable Object-Oriented Software*. Addison-Wesley, 1995.

C.D. Gill, D.L. Levine and D.C. Schmidt. The design and performance of a real-time CORBA scheduling service. *Real-time Systems*, vol. 20, n. 2, March 2001.

A.S. Gokhale, B. Natarajan, D.C. Schmidt and J.K. Cross. Towards real-time fault-tolerant CORBA middleware. *Cluster Computing*, vol. 7, issue 4, pp. 331–346, October 2004.

S. Graham, G. Baliga and P.R. Kumar. Issues in the convergence of control with communication and computing: proliferation, architecture, design, services, and middleware. *Proceedings of the 43rd IEEE Conference on Decision and Control*, vol. 2, pp. 1466–1471, Bahamas, December 14-17, 2004.

Object Management Group (OMG). The Common Object Request Broker Architecture: Core Specification, version 3.0.3. March 2004.

Object Management Group (OMG). Real-Time CORBA Specification, version 2.0. November 2003.

C.L. Robinson, G. Baliga and P.R. Kumar. Design patterns for robust and evolvable networked control *Proceedings of the 3rd Annual Conference on Systems Engineering Research*, New Jersey, USA, March 2005.

J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, vol. 21, issue 3, pp. 241–268, November 2001.