A MIDDLEWARE FRAMEWORK FOR NETWORKED CONTROL SYSTEMS

BY

GIRISH BANTWAL BALIGA

B.E., Mangalore University, 2000
M.S., University of Illinois at Urbana-Champaign, 2002
M.S., University of Illinois at Urbana-Champaign, 2004

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

Networked control systems could possibly constitute the next logical step in the evolution of control, leading to the convergence of control with communication and computing. A central challenge is that traditional digital control methods cannot be directly applied to such systems. However, if appropriate system abstractions can be engineered, then such methods and theory can still be utilized. Our thesis is that a well designed middleware framework can indeed manufacture such an abstraction of virtual collocation, and thereby, propel the further proliferation of networked control systems.

In this thesis, we present such a middleware framework for networked control systems. Central to this framework is Etherware, a message oriented component middleware for such systems. We begin with a detailed description of the design and architecture of Etherware, and illustrate Etherware based development of networked control systems through a fairly complex traffic control testbed application. Building on the middleware, we address safety and security issues in networked control through a detailed scenario in this testbed, implementing safety preserving security overrides. We then address the issue of providing guarantees of overall system behavior in networked control systems. In particular, based on established properties of sub-systems, we prove system-wide guarantees for safety and liveness in the traffic control testbed.

To my parents and teachers

Acknowledgments

The research for this thesis has benefited by contributions from many people. My research adviser, Prof. P. R. Kumar, was instrumental in helping define research problems and formulate good solutions for this thesis, and his care and patience on my behalf have been vital. Prof. Lui Sha has also been a guiding force, and his insights and suggestions have been indispensable. My collaboration with Prof. Carl Gunter has been the third pillar of this thesis, and in particular, his ideas have molded the safety and security considerations in the middleware framework. Meetings and discussions with Profs. Jennifer Hou and Grigore Rosu, and their insightful suggestions and comments, have undoubtedly made this a much better thesis. I am extremely grateful to all of them for serving on my thesis committee, and making my research enjoyable and fruitful. Thanks also to Prof. Ralph Johnson for many meetings, and excellent feedback about Etherware.

A lot of this research was done in collaboration with my friends and colleagues: Arvind Giridhar, Scott Graham, Kun Huang, Sumant Kowshik, Craig Robinson, and Ajay Tirumala. I am very thankful for their constant support, feedback, criticism, and encouragement.

Francie Bridges and Rebecca Lonberger have been excellent secretaries, and have made many hard things seem easy. At the same time, the members of 139 CSL, both designated and honorary, have made some simple things seem much harder, and as a consequence, much more interesting. I am much obliged to all these people for making my research experience truly memorable.

My warmest gratitude is reserved for last. Thanks to my parents, sister, and the rest of my family for their unconditional love, support, and motivation.

Table of Contents

List of Figures . . . . **ix**

List of Tables . . . . **xi**

Chapter 1 INTRODUCTION **1**
- 1.1 Tools, standardization, architecture, and compatibility 3
- 1.2 Abstractions for networked control 5
- 1.3 Middleware frameworks 8
- 1.4 Organization of thesis 9

Chapter 2 TOWARDS NETWORKED CONTROL PROLIFERATION **12**
- 2.1 Networked control testbed 14
- 2.2 Exploratory implementation 14
- 2.3 Middleware framework requirements 16
  - 2.3.1 Operational requirements 16
  - 2.3.2 Non-functional requirements 18
  - 2.3.3 Management requirements 19
- 2.4 Research context 20
  - 2.4.1 Domainware 21
  - 2.4.2 Service continuity 23
  - 2.4.3 Operational semantics 24
  - 2.4.4 Other related work 25

Chapter 3 ETHERWARE **26**
- 3.1 Architectural considerations 26
  - 3.1.1 Application design 26
  - 3.1.2 Stability considerations 28
  - 3.1.3 Message-oriented communication 29
  - 3.1.4 Architecture 30
- 3.2 Etherware programming model 31
  - 3.2.1 Application model 32
  - 3.2.2 Messages 33
  - 3.2.3 Components 35
  - 3.2.4 MessageStreams 38
  - 3.2.5 MulticastStreams 39

3.2.6 Filters . . . 41
3.3 Etherware architecture . . . 42
3.3.1 Kernel . . . 42
3.3.2 Scheduler . . . 43
3.3.3 Shells . . . 44
3.3.4 Delivery addresses . . . 44
3.3.5 NetworkMessenger . . . 45
3.3.6 ProfileRegistry . . . 45
3.3.7 Network Time Service . . . 46
3.4 Etherware based formalization . . . 48
3.4.1 Formal application design . . . 49
3.4.2 Etherware support . . . 51
3.5 Etherware capabilities . . . 52
3.5.1 Operational capabilities . . . 52
3.5.2 Non-functional capabilities . . . 53
3.5.3 Management capabilities . . . 54

Chapter 4 ETHERWARE MECHANISMS . . . 56
4.1 Etherware initialization . . . 56
4.2 Message delivery . . . 58
4.2.1 Message headers . . . 59
4.2.2 Delivery mechanism . . . 60
4.3 System evolution . . . 61
4.3.1 Component restart . . . 63
4.3.2 Component upgrade . . . 65
4.3.3 Component migration . . . 66
4.4 Etherware performance . . . 67

Chapter 5 ETHERWARE BASED APPLICATION DESIGN . . . 74
5.1 Testbed redesign . . . 75
5.2 Robustness in control systems . . . 76
5.3 Design patterns for robustness . . . 78
5.3.1 State estimation . . . 79
5.3.2 Receding horizon control . . . 80
5.3.3 Testbed design enhancements . . . 82
5.4 Analysis of testbed enhancements . . . 82
5.4.1 Car Model . . . 83
5.4.2 Worst-case error bound . . . 85
5.4.3 Mean error bound . . . 86
5.5 Experimental validation . . . 87
5.5.1 Case study: Motorcade . . . 88

Chapter 6 SAFETY AND SECURITY CONSIDERATIONS 94
6.1 Control System Incident Response 95
6.1.1 Rationale 96
6.1.2 Strategies for Incident Response 97
6.1.3 Etherware mechanisms 99
6.2 Security in control 100
6.2.1 A dichotomy of control 100
6.2.2 Safety preserving security overrides 101
6.2.3 Etherware mechanisms 102
6.3 Safety in the testbed 103
6.3.1 Architecture 103
6.3.2 Algorithms 105
6.3.3 Analysis 107
6.4 Security case-study 108
6.4.1 Scenario 108
6.4.2 Architecture 109
6.4.3 Experiments 110

Chapter 7 PROOF OF SYSTEM-WIDE SAFETY AND LIVENESS . . 117
7.1 Scheduling renewal tasks 119
7.1.1 Renewal task model 120
7.1.2 Prior work 120
7.1.3 A non-resource-sharing policy 122
7.1.4 Resource-sharing policies 124
7.2 Controllability of a single car 134
7.2.1 Car Model 134
7.2.2 Controllability in a straight road 135
7.2.3 Controllability in a road network 143
7.3 System-wide safety and liveness guarantees 149

Chapter 8 CONCLUSIONS 153
8.1 Vision for the future 154

References 156

VITA 162

List of Figures

1.1 Component based software for Digital Control 6
1.2 A Networked Control System 7

2.1 Traffic Control Testbed 13
2.2 Software architecture of the preliminary testbed implementation 15

3.1 Enhanced design of the lower-level control loop in testbed 27
3.2 Application model in Etherware 32
3.3 Typical application components in Networked Control Systems 33
3.4 Programming model for a component in Etherware 35
3.5 MessageStream between two components 39
3.6 MulticastStream between multiple components 40
3.7 Filter for a MessageStream 41
3.8 Architecture of Etherware 42
3.9 Message exchanges in Control Time Protocol algorithm 47
3.10 Controller-Actuator protocol in testbed 49
3.11 Finite state machine for Actuator in testbed 50
3.12 Architecture of Maude-Etherware interface 52

4.1 Experimental setup 62
4.2 Error in car trajectory due to controller restarts 63
4.3 Error in car trajectory due to controller upgrade 65
4.4 Error in car trajectory due to controller migration 66
4.5 Distribution of round trip times for Ethernet 68
4.6 Distribution of round trip times for one-hop wireless network 69
4.7 Distribution of round trip times for two-hop wireless network (base-station mode) 70
4.8 Comparison of mean error for the three scenarios 72

5.1 Software architecture of the Etherware based testbed implementation 75
5.2 A simple control loop 78
5.3 State estimation design pattern 79
5.4 Receding horizon control design pattern 80
5.5 Design enhancements in the lower level control loop of the testbed 81
5.6 State evolution in the car model 83

5.7 Growth of error in state prediction 89
5.8 Distance between centers of cars in the motorcade 91
5.9 Deviation of leader from its trajectory 92

6.1 Federated software architecture for the traffic control testbed 100
6.2 Ordering of Filters in a MessageStream 102
6.3 Software architecture for collision avoidance in the testbed 104
6.4 Software architecture for the security case-study 110
6.5 Experiment 1: No collision avoidance or security override 111
6.6 Experiment 2: Security override without collision avoidance 112
6.7 Experiment 3: Collision avoidance without security override 113
6.8 Experiment 4: Security override with collision avoidance 114

7.1 Road network with single lane roads 118
7.2 Clockwise steering controls for a car 118
7.3 Real-time guarantee $t_{i+1} - t_i \leq T$ for car control 119
7.4 Renewal task set not schedulable under EDF 121
7.5 Renewal task set partial schedule under PSP 122
7.6 Partial schedule under PSP for task σ_i with renewal at t_m 123
7.7 Control Loop Model 134
7.8 Controllable subset characterization 136
7.9 Controllable subset and maximum value of α, β 137
7.10 Possible trajectory for Case I 138
7.11 Initial states and trajectories for Case I 139
7.12 Initial states and trajectories for Case II 141
7.13 Driving the car in a displaced lane 145
7.14 Control strategy for recovery from a bad initial condition 146
7.15 Controllable subset in a single lane 147
7.16 Modeling intersections in road network discretization 150

List of Tables

4.1 Round trip time statistics for wired link 71
4.2 Round trip time statistics for wireless link (one hop) 71
4.3 Round trip time statistics for Base station mode (two hop) 71

6.1 Safety measure for the four scenarios 115
6.2 Security measure for the four scenarios 115

Chapter 1

INTRODUCTION

Control systems have been designed and built since the earliest times. For instance, feedback control was used in float regulator mechanisms in Greece as early as 300 BC [1]. More recently, the use of automatic feedback control in industrial processes began with James Watt's fly-ball governor, which was developed in 1769 for controlling the speed of a steam engine [1]. However, all these systems were mechanically operated, and developed mostly by intuitive invention [2]. Although mathematical models were developed for some systems such as the governor [3], a general theory for control systems was developed only in the twentieth century.

Modern control systems are characterized by a more systematic design process. Indeed, based on the underlying technological and theoretical frameworks, control systems developed since the early twentieth century can be roughly classified into three generations.

The first generation of modern control systems were analog systems. These systems were mostly built using electronic feedback amplifiers as the underlying technology. Even today, many control systems are still built using similar principles. For instance, PID controllers are widely used as standard control components since they can be easily integrated into control systems through well-defined interfaces. Also, frequency-domain techniques pioneered by Bode, Nyquist, and Black, have served as the theoretical basis for designing such systems [4] [5].

The emergence of digital computers after World War II has led to a second generation of control systems based on digital control. Computers are the primary control components in such systems, and the underlying technology is the control software executing in these computers. The speed and accuracy of computers allow many system variables to be monitored and controlled simultaneously, and the flexibility and expressiveness of software languages support the implementation of fairly complex control algorithms. The design of these systems is governed by well understood techniques such as Kalman filters [6] and real-time scheduling theory [7]. Today, such systems are used in the vast majority of control applications ranging from airplane navigation to industrial process control.

We are now witnessing the emergence of a third generation of control systems, which are operated using computer networks. These networked control systems are characterized by distributed control loops operating over multiple computers in a network. They involve complex control software that needs to operate in a coordinated fashion over many interconnected computers. For instance, a typical mid-range automobile has about 45 micro-controllers connected by Controller Area Networks (CAN) [8]. Also, since the underlying infrastructure is usually quite expensive, these systems usually need to share the communication and computation resources among many different applications. As a result, these systems are much more complicated than previous generations of control systems.

Although networked control applications are increasing, such systems are still mostly custom-designed, and unlike earlier generations of control systems, there is still no comprehensive technological or theoretical framework for developing these systems. In particular, the control software, which is the major source of complexity in these systems, is still largely custom-made. For instance, over 60% of the software for the Boeing 777 airplane had to be developed from scratch [9].

In this thesis, we present a middleware framework for software development in networked control systems. We contend that such a framework is the key to the systematic design of such systems, and could consequently lead to the proliferation of networked control. To this end, our framework provides a comprehensive set of software primitives for application design, as well as a software infrastructure for application development in such systems. In particular, we promote the development of standardized and reusable networked control software, which will significantly reduce design cycle times and development costs in such systems. Finally, the application design standardization enabled through our framework, sets the stage for the development of a theory for networked control systems as well.

Our main contributions in this thesis are:

- A message-oriented component middleware for networked control systems called *Etherware* (Chs. 3 and 4).
- The design and implementation of a traffic control emulation testbed to illustrate Etherware based application development (Ch. 5).
- The Control System Incident Response (CSIR) framework for implementing safety and security sub-systems, and the principle of safety preserving security overrides in control systems (Ch. 6).
- A proof of overall system-wide safety and liveness for the traffic control testbed based on various sub-system level guarantees, which can possibly serve as a prototype for other such systems (Ch. 7).

1.1 Tools, standardization, architecture, and compatibility

We now discuss some of the main engineering concepts that we will draw upon for our middleware framework for networked control systems. In the process, we also trace the origins of these concepts, and illustrate their impact on the course of the history of modern technology.

The importance of *good tools* can be hardly overstated in the history of human technology. For instance, the invention of the steam engine is usually credited with starting the industrial revolution in England. However, when the first engines were built, there was a basic engineering problem. The large cylinders of engines could not be bored with sufficient precision, and a lot of power was lost to steam leaking around the pistons. This problem was eventually solved by the invention of a precision horizontal-boring machine by John Wilkinson in 1775. It was this breakthrough that made steam engines viable, and emphasized the importance of developing good tools in the subsequent revolution.

Meanwhile, the notion of *standardization* in manufacturing triggered the industrial revolution in North America. In 1798, the US government contracted Eli Whitney to produce 10,000 muskets in two years. The actual muskets were delivered in 1808, about eight years later than scheduled. However, Whitney delivered far more than just the muskets. He developed, and successfully applied, the “uniformity-system” to manufacture interchangeable parts for locks of the muskets [10]. In a remarkable demonstration to President Jefferson in 1801, he assembled complete locks for muskets from random parts. Today, Whitney’s system is considered to be the basis of standardization in manufacturing technology. In fact, standardization is a basic premise in most industries, where products are composed of interchangeable components developed according to well-defined interfaces.

A *well-designed architecture* has been the central reason for the proliferation of computers today. For instance, the work of Alonzo Church and Alan Turing in the early twentieth century, laid the theoretical foundations of sequential computing. They independently developed formal systems to define and model the notion of computable functions [11]. 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. In particular, the stored program concept [12] was a significant breakthrough. It has since led to the development of simple and uniform mechanisms to write, test, and maintain complex programs today.

The von Neumann architecture had a widespread influence on the first generation of digital computer engineers in the 1950s. However, even in the mid 1960s, commercial computers produced by IBM, Burroughs, UNIVAC, and others were still custom built machines. Individual computers had to be designed, built, and installed according to specific customer requirements. The IBM 360, first shipped in June 1966, changed all that. The IBM 360 was the first commercially successful general purpose computer. In fact, the name 360 (degrees) was intended to market its “all round” general purpose capabilities.

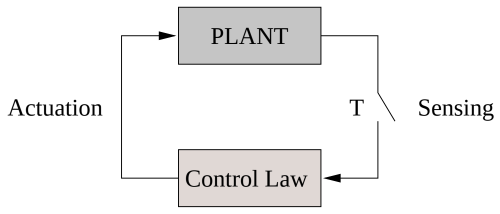
The 360/370 series introduced the notion of *compatibility* in computers [13]. All computer hardware was built to a common set of abstractions such as instruction sets, memory models, etc. Consequently, the same software could run on all the different machines in this series. This standardization tremendously reduced software development costs. More importantly, it solved problems due to changing customer requirements by supporting seamless upgrades of hardware and software. This architecture immediately propelled IBM to the top of the computer market, where it stayed for over two decades after the first 360 was produced.

These examples amply illustrate the importance of the associated engineering concepts. Standardization allows components to be independently and efficiently produced, and then assembled into customized applications. Good tools make crucial processes viable, improve the quality of systems, and increase productivity in manufacture. Well designed architectures and compatibility standards promote specialization of sub-systems, as well as their subsequent integration into working systems.

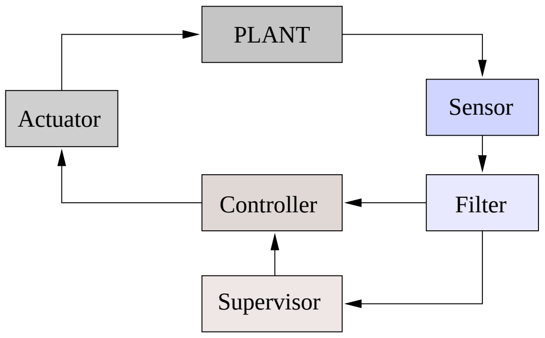
1.2 Abstractions for networked control

In this section, we examine similar considerations for the the design of networked control systems, and introduce some of the main abstractions underlying our middleware framework.

Let us consider the classic representation of a control system as shown in Figure 1.1(a). A simple implementation of this system would be a modular software program implementing the control laws and plant interfaces. However, established practice in software engineering advocates the use of *component architectures* for such systems [14] [15] [16] . This consists of decomposing the software into *components* as shown in Figure 1.1(b), where a *component* is an autonomous software module with well-defined functionality. For instance, Sensor,



(a) Periodic digital control



(b) Software components

Figure 1.1: Component based software for Digital Control

Actuator, Controller, Filter, and Supervisor are all possible components for such a system.

A component based software architecture has several benefits. It allows individual components to be developed separately and integrated later, which is very important for the development of large systems. Since components are well-defined, they can also be replaced without affecting the rest of the system. For instance, a zero-order hold filter in Figure 1.1(b) can be dynamically replaced by a Kalman filter without having to change the remaining software or restart an operational system. Further, such architecture promotes software reuse, since a well designed component such as a control algorithm, tested for one system, can be easily transplanted into another similar system.

Traditional digital control theory addresses the design of systems with tightly coupled plants and controllers as shown in Figure 1.1. Such controllers expect periodic feedback about plant state from sensors, and output periodic controls through actuators. This makes the system predictable, and simplifies the design of control laws. However, to obtain such pe-

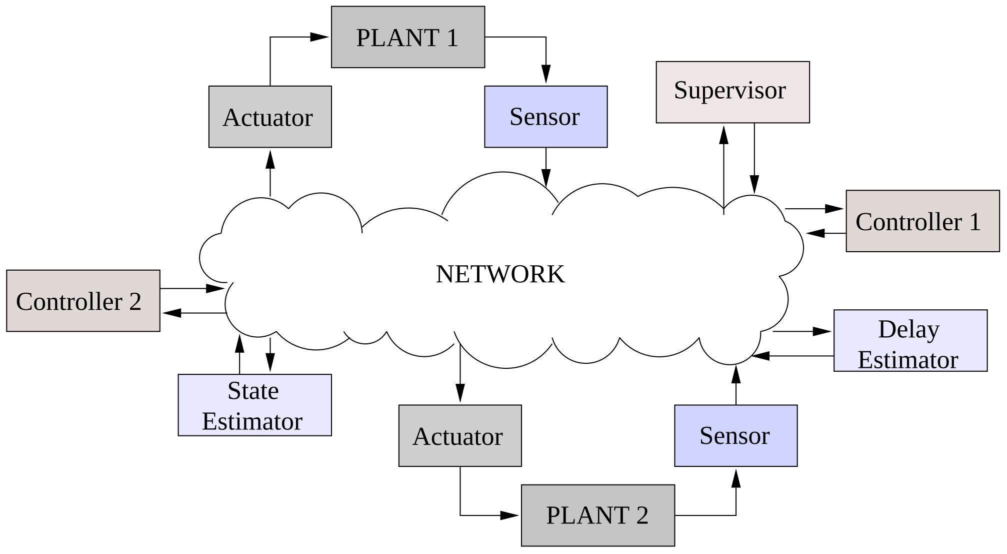


Figure 1.2: A Networked Control System

riodicity, the control software must be tightly integrated, and hard real-time deadlines must be strictly enforced. Hence, systems engineers have relied on effective scheduling algorithms based on conservative worst-case execution deadlines [7], and dedicated communication channels such as CAN [17] and FDDI [18] that provide hard real-time guarantees.

In networked control systems, on the other hand, software components execute on multiple computers connected over a general network. As illustrated in Figure 1.2, control loops may be distributed, and related software components may need to communicate over the network. In particular, it is difficult to guarantee periodic communication with hard real-time deadlines over best effort networks such as wireless and IP based networks. Hence, it is imperative to provide appropriate software abstractions, such as *virtual collocation* and *local temporal autonomy*, which allow digital control theory to be applicable in such systems [19].

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

Local temporal autonomy is a method for enhancing robustness and reliability. It consists of ensuring that a component can execute for a small amount of time even after another connected component has failed. For instance, using a state estimator can help a controller tolerate delayed or lost updates from a sensor. In addition, upon a software failure, if the sensor is restarted quickly enough, then the controller can continue executing without being aware of this. These and other similar mechanisms [19] allow network delays, transients, and failures to be tolerated, and help enhance the abstraction of virtual collocation.

1.3 Middleware frameworks

Middleware frameworks incorporate the engineering concepts presented in Section 1.1 quite well. They combine well-designed architectures with useful toolkits, and promote the development of applications with standardized and customizable interfaces. They also support the abstraction of virtual collocation in general purpose systems. Consequently, middleware has emerged as a preeminent framework for developing complex and distributed applications. Major software companies such as IBM [20], Microsoft [16], and SUN [21] increasingly promote their middleware based technologies as platforms for software engineering, and many successful applications have been developed using CORBA based architectures [22]. Today, middleware based technologies are considered the basic ingredient in the development of complex and distributed software systems.

Despite the considerable success of commercial middleware technologies, however, there is still a basic problem in their use for networked control. Traditional middleware technologies such as CORBA have been developed for transaction based applications in the banking and business sectors. They are based on Remote Procedure Call (RPC) semantics, which are ideal for ensuring logical correctness of programs. However, this causes components to block on remote function calls, which is undesirable for control applications. For instance, a critical component such as a real-time controller cannot block on updates from remote sensors, and would require a separate thread to block on each such sensor. This leads to complex multi-threaded component design, which is relatively hard to develop, manage, and verify. Further, component interfaces described in such systems cannot capture assumptions such as interaction protocols and failure semantics. This leads to numerous problems in systems integration, and may even cause unpredictable failures during system operation. A middleware for control applications must address these and related issues.

These considerations have motivated our design of a middleware framework specifically for networked controls systems, which we present in this thesis.

1.4 Organization of thesis

A middleware framework incorporates a set of architectural trade-offs for its domain of application, and in particular, the middleware itself is designed to address the infrastructural requirements of its applications. However, these trade-offs and requirements can be determined only after a detailed study of some of the applications in the domain. Consequently, we begin by studying an exploratory implementation of a prototype traffic control testbed in Chapter 2, and based on this, we develop the requirements for our middleware framework.

Central to a middleware framework is the underlying middleware itself. It incorporates the actual architectural trade-offs in the framework, and defines the set of design primitives for its applications. Its capabilities determine the capabilities of the framework itself. Hence, in Chapter 3, we present a detailed description of the design and architecture of *Etherware*, our middleware for networked control systems, and a discuss how the software infrastructure requirements for networked control systems are addressed therein. We complement this by describing the actual Etherware mechanisms for system operation and management in Chapter 4, and present a comparison of Etherware to other potential middleware for networked control systems. In particular, the descriptions in these two chapters constitute static and dynamic pictures of Etherware respectively.

A middleware framework is usually based on an application design philosophy. It promotes specific application designs and design patterns through appropriate abstractions and services. However, it is hard to make this explicit only through middleware description and cursory examples. Hence, we present the Etherware based design and implementation of our traffic control testbed in Chapter 5. In particular, we illustrate the design of distributed control loops, through a detailed analysis and experimental study of a representative control loop in the testbed.

Control systems interact with the real-world, and hence, *safety* and *security issues* are of vital concern in these systems. While sub-system level safety features are usually incorporated in control application design, there still are exceptional situations that may need to addressed at a system-wide level. For instance, a security breach that compromises safety usually requires the system to be operated in a safe-mode so that the corresponding threats can be mitigated. Also, the security overrides employed in such situations should not compromise low level safety features that provide important safety guarantees in the system. In Chapter 6, we present a systematic approach to addressing these issues in networked control systems using our middleware framework.

In Chapter 7, we address the issue of how one may envisage providing system-wide guarantees of safety and liveness for networked control systems. Specifically, we use subsystem level guarantees from the previous chapters, to prove system-wide safety and liveness guarantees in the testbed. For safety, we show that cars can be driven without collisions, and for liveness, we show that the system can be operated without gridlocks. During this development, we also introduce real-time scheduling policies for the renewal task model, which we then use to develop the control model for cars in the testbed.

Our thesis presents a middleware framework for networked control systems, but our vision is broader. We envision general purpose control, where control loops can be set up dynamically, and entire control systems can be assembled from toolkits of components, using well designed frameworks, and based on well understood system theory. This extends the notion of general purpose computing and communication, and anticipates the convergence of control with communication and computing. We elaborate on this vision and conclude in Chapter 8.

Chapter 2

TOWARDS NETWORKED CONTROL PROLIFERATION

In the past few decades, we have witnessed the development of several crucial technological enablers for networked control systems. Remarkable advances in VLSI technology, chip fabrication, and hardware architecture design have resulted in the availability of microprocessors with increasingly higher capabilities. This has significantly changed the nature of digital control, and enabled the use of higher performance control software for increasingly complex applications. For instance, ZiLOG's eZ80Acclaim [23] micro-controller used in industrial control applications has 256KB flash memory and 512KB fast SRAM. It is also supported by a real time kernel with a TCP/IP stack for Internet connectivity.

Wireless networking technology has also matured to the point where it can provide connectivity to embedded micro-controllers in control systems [24]. This in turn enables the deployment of increasingly complex control algorithms in such embedded computer networks. In addition, such software can even be updated dynamically using wireless connectivity. Further, the integration of sensing with computation and communication capabilities has enabled diverse new applications based on sensor networks. For instance, MICA2 [25] motes, a recent generation of experimental sensor nodes, can store programs of up to 128 Kbytes, and communicate at 38.4 kbps on a wireless link.

The above trends motivate significant extensions to traditional digital control. In particular, they enable distributed control loops, where control software executes in a coordinated fashion over networks of embedded computers in control systems. In addition, the powerful computation and communication capabilities in such computer networks can support standardized software architectures that can trade-off some performance for much better

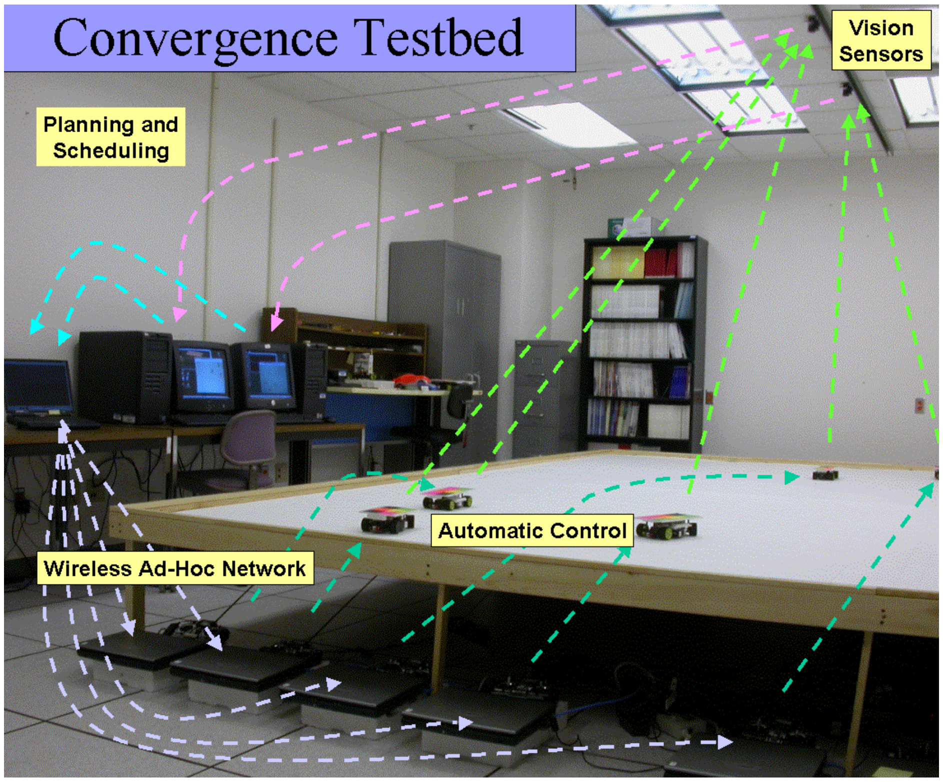


Figure 2.1: Traffic Control Testbed

software management and reuse. Consequently, such networked control systems are being increasingly deployed in applications ranging from automobiles to air traffic control.

The central challenge in the proliferation of networked control systems is the development and management of control software, and as we have argued in Section 1.3, a middleware framework is an important ingredient in addressing this challenge. In this chapter, we study this issue in further detail, using a prototype networked control testbed that involves traffic emulation through software controlled cars. In particular, we examine an exploratory software implementation of this system, and use this to develop the requirements for a middleware framework for networked control. We conclude with a presentation of the research context for this thesis.

2.1 Networked control testbed

The networked control testbed, which we use as a prototype for networked control systems, is shown in Figure 2.1. It consists of a set of remote controlled cars driven on a track. A car has no on-board computational capabilities, and is controlled by analog signals transmitted over a dedicated radio frequency channel. These signals control the speed and steering angle of the car. The corresponding radio transmitter is connected to the serial port of a dedicated laptop through a micro-controller, which converts discrete commands from the laptop into analog controls to operate the car. Hence, a car can be controlled by varying its speed and steering angle in discrete steps. Also, commands can be sent at a rate of up to 50 Hz, i.e., one command every 20ms.

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 10Hz, i.e., one update every 100ms. Also, all computers in the testbed are connected by wired Ethernet. In addition, the laptops are also connected by an ad hoc wireless network with IEEE 802.11 [26] PCMCIA cards.

2.2 Exploratory implementation

A preliminary implementation of traffic control was undertaken in the testbed to understand the various challenges involved in developing a middleware framework for such networked control. The application architecture of this implementation is illustrated in Figure 2.2. As shown in the figure, each car is controlled by a low-level model predictive Controller [27]. The desired operation of the car is specified to the Controller as a *trajectory*, which is a sequence of desired positions for the car at future time instants. The Controller then generates discrete controls to drive the car along the specified trajectory. Such controls are generated every

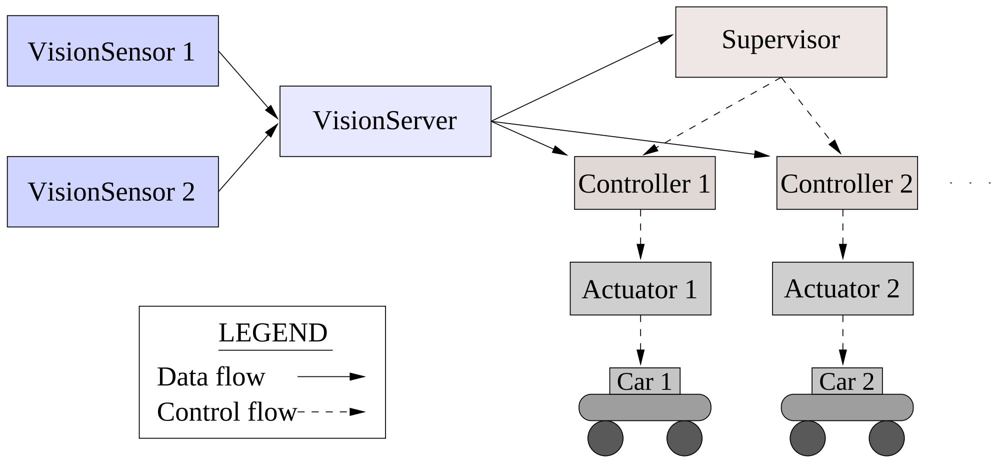


Figure 2.2: Software architecture of the preliminary testbed implementation

100 ms and sent to the Actuator module. The Actuator in turn sends the control to the car via the corresponding micro-controller and radio transmitter. Vision feedback from the image processing algorithms at the VisionSensors is collected at a VisionServer. The VisionServer then performs data fusion and provides suitably merged feedback information to the Controllers. This completes the low-level control loop in the system.

As noted above, the Controllers need to be given trajectories to operate the respective cars. In the testbed, such trajectories are generated by a central Supervisor, which also ensures global properties such as the avoidance of car collisions for safety, and the elimination of traffic gridlocks for liveness. For instance, in a traffic control scenario, the Supervisor generates trajectories to control the car along a pre-specified network of roads, with planning and scheduling algorithms that are based on the analysis described in [28]. Briefly, blocks of the road network are modeled as bins in a corresponding graph. The planning problem is then re-formulated as finding shortest paths in such 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. However, in practice, the Controllers are executed on the same laptop as the respective Actuators, and the Supervisor is executed on a separate dedicated laptop.

Traffic scenarios with up to eight cars operating simultaneously on the track have been tested using this implementation. The cars closely followed pre-specified trajectories in reasonably well behaved traffic scenarios. Another demonstrated scenario is pursuit-evasion in a leader-follower configuration, where a set of software controlled cars follow a manually controlled car. An automatic collision avoidance system has also been demonstrated.

2.3 Middleware framework requirements

The exploratory implementation described in Section 2.2 has helped determine some of the common functionalities in most networked control applications. Ideally, such functionalities should be reusable, and hence, part of the middleware framework. We now consider these requirements in detail.

2.3.1 Operational requirements

These are the basic functionalities required for correct operation in networked control.

- **Distributed operation:** Connecting diverse components executing on different computers in a network usually leads to numerous problems related to such distributed operation. These include locating components in the network, connecting related components, and supporting the exchange of messages between connected components. For instance, in the traffic control testbed, the Supervisor and the various Controllers execute on different computers. Initializing these components, connecting the Controllers to the Supervisor, developing protocols for their interaction, and synchronizing their operation to resolve conflicts represent some of the problems arising due to distributed operation in the testbed.

- **Location independence:** This is an important abstraction that provides an elegant solution to some of the problems related to distributed operation. It involves an addressing scheme for components that is independent of their actual location in the network. This 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 these details to be abstracted away from the components. Similarly, such an abstraction supports the use of other networking technologies such as Bluetooth [29], without having to update component code.

- **Service description:** Connecting related components in a network involves determining if appropriate components are executing in the system. Further, of the available components, the most suitable component has to be selected. For instance, consider a car controller that knows the geographic location of its car to be (10,25) in a given coordinate system. If it needs feedback, say from a vision sensor, then the controller should be able to connect directly to a sensor covering that location based on this information alone. In particular, the controller should not have to know about sensor locations in a network. This requires mechanisms to specify and discover services provided by components.

- **Interface compatibility:** Integrating independently developed software components almost always involves interface incompatibilities. For instance, the set of functions defined in the interface of a sensor module may not be compatible with the functions required by a controller component. Hence, integrating the two components would require reprogramming one or both of these components. This is eliminated if standard interfaces have been specified and supported for compatibility between such components.

- **Semantics:** A significant problem occurs due to incongruent assumptions in the implementation of components. The interaction between different software components is usually based on an implicit finite state machine. The exchanged messages are assumed to be in specific formats. However, current interface description languages such as the CORBA IDL [30] do not provide a mechanism to specify these assumptions properly. For example, suppose the Controller in Figure 2.2 is implemented so that it checks for an update at the VisionServer before computing controls. This assumes that the VisionServer responds immediately, returning an update if available, and none otherwise. However, if the VisionServer is implemented so that it checks with the VisionSensors for updates before responding, the additional delay may lead to failure as the Controller is also waiting for an update. Hence, the Controller and the VisionServer may have consistent interfaces, but the interaction semantics may still be incompatible. Consequently, there must be additional provision for specifying interaction semantics in the interface descriptions.

- **Distributed time:** Since components may execute on different computers, they may not share common clocks. Hence, there must be a mechanism to translate time between different clocks. For example, time-stamps on remote sensor updates must be translated to the local time for correct operation of controllers.

2.3.2 Non-functional requirements

The following features significantly improve system behavior and performance. However, they are not necessarily required for the correct operation of the system.

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

- **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 of them may be lost over the network.

- **Security:** Security is a key requirement in control systems as they interact with the real world. However, standard security mechanisms used in information management systems may not be directly applicable to control systems. Usually, additional constraints apply due to the direct interaction of such systems with the real world. For instance, when there is a security breach in the system, a standard approach is to suspend normal operation with a security override to address the intrusion. However, in control systems, low-level fail-safes that guarantee system safety must not be suspended as well. In fact, the implementation of such safety preserving security overrides is a key security principle for control systems.

- **Other requirements:** The protocols and algorithms implemented in the middleware framework must have good *performance* and *scalability*.

2.3.3 Management requirements

The following features considerably enhance system manageability.

- **Startup:** Programmable interfaces to *startup* procedures help ensure that all parts of the system are initialized up correctly. In particular, such an interface must allow the specification of dependencies between components at startup. For instance, to ensure proper initialization of the plant, a controller component may have to be started up before a related actuator component.

- **System evolution:** This requires the ability to update or migrate components at run-time. Component migration is required to optimize the configuration of software components to reduce or balance communication and computational loads. For example, a Controller in the testbed may be migrated to another computer, where it is both closer to the corresponding Actuator and the destination is less loaded computationally, thus reducing loop delay. This can and ought to be done dynamically, i.e., while the system is running, in order to self optimize the organization of what component is executing where. In particular, control engineers and system designers should not have to worry about such details. Run time updates also allow operations such as changing controllers, to address evolving plant goals, without having to restart the rest of the system.

2.4 Research context

The basis of a middleware framework is a sound architecture that governs the organization of software components and subsystems for the development of applications with desirable properties. Also, the application designs supported by an architecture are a direct consequence of the abstractions and programming models supported by the underlying software infrastructure. Accordingly, well designed middleware has emerged as the most widely used software infrastructure for distributed applications. As noted in Section 1.2, popular approaches for control are based on variants of CORBA [22] such as Real-Time CORBA [31], Minimum CORBA [32], and Fault tolerant CORBA [30].

This section presents a brief study of relevant software infrastructures that are being used in comparable systems. It also emphasizes the limitations of the associated frameworks vis-á-vis the requirements of networked control as these have motivated the main contributions of this thesis.

2.4.1 Domainware

A well designed software infrastructure effectively captures domain-specific trade-offs and provides an appropriate set of abstractions for application design. However, distinct domains typically have different operating conditions and constraints, and hence the appropriate trade-offs are usually different as well. For instance, the execution of correct transactions is necessary for business applications, but robustness and lower delays are more important in the control domain. This key insight guides the following considerations about software infrastructures.

Real-Time CORBA (RT-CORBA) is representative of the class of middleware that have emerged as a control specific modification of pre-existing infrastructure developed for a different domain. In particular, RT-CORBA is a specialization of CORBA, which was originally developed for business applications requiring reliable communication. Hence, an implicit assumption in the specification of RT-CORBA is that communication overheads are tolerable by control applications. Consequently, most of the specification deals with models for scheduling threads and providing real-time guarantees on a single node. For instance, the notion of a distributable thread relies heavily on fast and reliable communication to avoid deadlocks. In wireless networks, however, communication delays are unpredictable and packet losses are high. Hence, a controller cannot depend on remote sensors through distributable threads. Also, the problem of sensor update losses can be addressed in the application itself by using good state estimation techniques. On the other hand, lower delays are much more important for state estimation and robust control as feedback must be available to controllers as soon as possible. Consequently, the ability to trade-off delay for reliability is a key feature that needs to be added to RT-CORBA. Reference [33] considers some of the other issues that need to be addressed in RT-CORBA. However, the main drawback with middleware such as RT-CORBA is that they inherit trade-offs made for unrelated domains.

Forcing functions capture the main constraints of a domain, and hence, constitute major determinants for domain-specific tradeoffs. We define a forcing function as a characteristic of an application domain that must be addressed by any application level solution. For instance, control applications operating on wireless networks are constrained by lossy links and unpredictable delays, due to which, real-time guarantees are impossible to support in middleware. Hence, the application itself is forced to address this issue by incorporating complex control laws and good state estimators that adapt to network conditions. Since applications are forced to directly address these constraints, this is a necessary cost to be paid.

Domainware is middleware that exploits application functionality already imposed by the forcing functions of the domain. In wireless networks, delay and bandwidth characteristics of communication channels have high variability, and 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 autopilot mode are controlled with a Global Positioning System (GPS) and a local Inertial Navigation System (INS). The INS is used for controlling the airplane during normal operation. But, as the reference point of the INS drifts with time, the GPS is used to correct it periodically. However, if communication with the GPS system is lost, then the default mode is to fly using just the INS. The key point is that airplanes, and in general most controllers over wireless networks, are designed to handle bad channel conditions. Domainware exploits this by relaxing its own requirements and having a much simpler architecture. As individual components are fail-safe, more important non-functional requirements such as fault tolerance and system evolution are addressed in a much simpler fashion.

Etherware, the middleware presented in this thesis, has been developed as Domainware for networked control. In particular, the forcing functions of wireless networks have been exploited to implement the system with soft real time mechanisms. Also, the ability of components to accept occasional delays have been exploited to support Etherware mechanisms for restart, upgrade, and migration as described in Chapter 4.

2.4.2 Service continuity

Control applications are subjected to numerous changes during prolonged operation. These include involuntary changes such as faults, and voluntary changes such as software upgrades. Since middleware based control applications are composed of interacting components, changes affecting one component must be isolated from other components that depend on it. For instance, restarting or updating the Controller in Figure 2.1 should not also require restarting the Actuator, as this might affect the operation of the car. Since components typically interact by providing services to each other, this requires service continuity to be maintained even in the presence of changes in a component.

Fault-tolerant CORBA [30] is a variant of CORBA that addresses service continuity during component failures. This is provided mainly by object replication and support for restarts. However, the mechanisms require at least one replicated server object to be operational for a procedure call from a client to be successful. If no server replica is available, then an exception is raised in the client due to Remote Procedure Calls (RPCs) based semantics. However, during failures due to logical errors in software, all replicas of a component will need to be restarted. Hence, such a failure will affect clients as well. Further, CORBA variants do not support service continuity during component upgrades, and do not support migration at all.

Etherware supports service continuity by providing efficient restart, upgrade, and migration mechanisms. As demonstrated by the experiments in Chapter 4, the involved delays are well within the bounds required by the testbed. Further, such communication channels are preserved across component restarts and upgrades. This allows other components to continue operating despite such changes. In particular, communication channels need not be reestablished after component restarts or upgrades.

2.4.3 Operational semantics

Interface description languages (IDLs) provided with CORBA, Jini, and other popular middleware mainly address the specification of functional interfaces, which are the declarations of function calls implemented by components. However, component behavior cannot be specified using these IDLs. Consequently, operational specifics such as communication protocols and failure modes of component implementations cannot be made explicit. For example, suppose an actuator component accepts control updates through the function *update*() in its interface. This may be implemented so that it accepts control updates most of the time, but raises an exception when *update*() is called at the same time as it is actually sending controls to its actuator device. However, a controller communicating with the actuator may assume such an exception to indicate failure and reset itself, thus leading to persistent faulty behavior. As most documentation is imprecise, such implicit assumptions make reusing components and integrating systems extremely difficult.

Many formal methods for system specification have been adapted for real-time systems [34]. For example, Statecharts [35] [36], a graphical language for specifying real-time systems, extends finite state machine descriptions by introducing composition operators such as AND and OR. In addition, some of these methods have also been used to extend the CORBA IDL to support the specification of operational semantics. For instance, TRIO/CORBA [37] [38] is based on TRIO [39], a first order temporal logic, while Cooperative Objects (CO) [40] [41] is an object oriented dialect of Petri Nets. Also, SpecTRM [42] [43] is a fairly comprehensive framework for safety critical systems specification. However, a major drawback with most of these tools is that they are not very well integrated with commercial middleware frameworks for control systems.

The middleware framework proposed in this thesis supports an IDL for the specification of operational semantics of components. In particular, aspects of a component's behavior that are relevant to other components can be specified. This includes the specification of interaction protocols and types of message exchanged between components. Such support should help eliminate a lot of the problems arising due to implicit assumptions about component behavior.

2.4.4 Other related work

As noted above, popular middleware based approaches for networked control build on variants of CORBA. For instance, Open Control Platform [44] is based on Real Time CORBA, and has been used to control unmanned aerial vehicles. OCP builds on top of a CORBA implementation by adding a Controls API, which provides the abstraction of components communicating by sending and receiving events through ports. This is similar to the Etherware programming model, but it also inherits the drawbacks of RT-CORBA noted in Section 2.4.1. CoSMIC [45] is another middleware for distributed real-time and embedded applications. It extends the Model Driven Architecture of OMG, and is based on an implementation of Real Time CORBA as well.

Other interesting approaches include the Giotto [46] system for embedded control applications, real-time framework [47] for robotics and automation, and OSACA [48] for automation. A fairly comprehensive overview of research and technology of software architectures for control systems is provided in [49].

Finally, several of the architectural constructs in Etherware have been influenced by J-Sim [50] [51], a component based, compositional simulation environment. In particular, the loosely coupled component programming model, and the autonomous component architecture in J-Sim have a similar conceptual basis to the corresponding aspects of Etherware.

Chapter 3

ETHERWARE

In this chapter, we present *Etherware*, the underlying infrastructure for our middleware framework for networked control systems. We present the main considerations influencing the design of Etherware, and describe the Etherware programming model and architecture. In particular, the programming model highlights the abstractions and design primitives supported by Etherware, and the architecture illustrates how middleware design considerations have been incorporated in its implementation. We conclude with a discussion of how the requirements of Section 2.3 are addressed in Etherware.

3.1 Architectural considerations

The design of Etherware has been considerably motivated by experiences with the exploratory testbed implementation described in Section 2.2. These and other considerations that have influenced the Etherware architecture are presented in this section.

3.1.1 Application design

The implementation of control loops over a network is a complex problem. This is further complicated in wireless networks due to high losses and unpredictable delays. However, these limitations cannot be addressed entirely by infrastructure based mechanisms, since such mechanisms would necessarily have to make trade-offs that affect application performance as well. For instance, reliable delivery of packets can be provided only with increased delays, which would then affect the performance of control loops in the system. Hence, applications

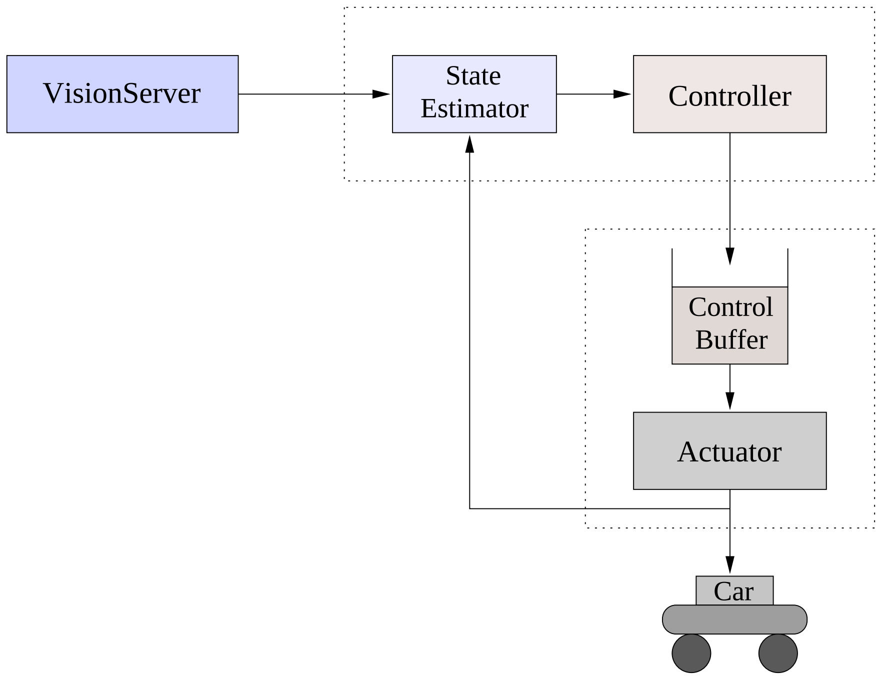


Figure 3.1: Enhanced design of the lower-level control loop in testbed

would either have to tolerate packet losses or increased delays.

Application design constraints are largely determined by the *forcing functions* of their domain, as described in Section 2.4.1. For networked control systems, a key forcing function is the lossy and delay-prone wireless channels used for communication. In the testbed, these constraints have led to various design enhancements to the software architecture of the testbed in Figure 2.2. For instance, enhancements to the lower-level control loop are shown in Figure 3.1. As shown in the figure, a Kalman Filter [6] is used in the Controller to reduce the impact of losses in sensory feedback. Similarly, a Control Buffer in the Actuator stores future commands to tolerate delayed or lost updates from the Controller. Chapter 5 considers these and other modifications to the testbed in detail.

The main idea behind the modifications to the testbed design is the reduction of network based dependencies between components by increasing their *Local temporal autonomy.* For example, the Kalman Filter allows the Controller to tolerate losses in sensory feedback, and the Control Buffer stores alternate controls for the Actuator to provision for delayed or lost Controller updates. This approach to application design has greatly influenced the architecture of Etherware. Since such local temporal autonomy allows critical components to endure some delays, Etherware has been developed assuming only soft real time control. In particular, this has shifted our focus from enforcing hard real-time guarantees, to supporting non-functional requirements such as fault-tolerance and software manageability.

3.1.2 Stability considerations

Robustness can be enhanced by, and even require, the ability to efficiently restart failed application components. For instance, suppose the Controller in Figure 3.1 fails due to a computation error. Restarting this component should not require reestablishing communication channels with the VisionServer, reinitializing the Actuator, or re-establishing Controller state, as these delays could themselves cause system instabilities, and thus result in car collisions.

The need for evolution of software in the system requires the ability to update components at run-time, i.e., while the system is executing. System optimization requires the ability to migrate components, also at run-time. Further, upgrade and migration must be done in an application aware fashion, so that the corresponding instabilities are minimized. For example, updating or migrating a Controller, without informing it that loop delays will consequently be reduced, can actually lead to instability or poor performance. All these requirements have motivated a simple and uniform design to externalize component state. This involves a mechanism to represent and capture the current state of a component so that it can be reinitialized with this state if it needs to be restarted at the same node, or even migrated to a different location and restarted there. This allows Etherware to capture component state, and use this to restart, update, or migrate components while maintaining system stability.

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. This is in contrast to a mechanism where such a requirement is optional, or even a special feature supported by additional middleware enhancements.

Components typically maintain several communication channels with other components. For smooth operation, restarting or upgrading of components should not require such channels to be re-established, and in particular, channels should be maintained across such changes. For example, if the Supervisor in Figure 2.2 is restarted, updated, or migrated, the connected Controllers should not have to reestablish communication channels with the new Supervisor. Consequently, maintenance of communication channels across such changes is also supported in Etherware, and in particular, identifiers for communication channels can be saved as part of check-pointed state. This allows restarted or upgraded components to continue using previously established channels. More importantly, it provides continuity of communication to other connected components during such changes.

3.1.3 Message-oriented communication

Control systems have fairly strict safety requirements. Components have to respond to changes as soon as possible. For example, upon detecting a safety violation, a Controller may not be able to wait for an acknowledgment from a Supervisor before it decides to take some safe action. Over a wireless channel in particular, delays can be fairly large due to deep fading and queued packets.

Conventional middleware for transaction based systems such as CORBA [30] are based on a synchronous mode of communication where components interact by making remote procedure calls. In this approach, the caller is blocked until it receives a reply from the callee. However, this may lead to complex multi-threaded component design in push-based communication channels commonly used in control applications. For instance, consider a controller operating multiple actuators over a wireless network. Since the network can have unpredictable delays, the controller should not be blocked while sending controls to any of the actuators. This would require a multi-threaded controller design with one thread to send controls to each actuator so that the main control loop does not block on the remote calls.

The above problem is addressed by providing an asynchronous mode of operation, wherein the caller is not blocked on a procedure call. For example, CORBA supports this by one-way functions. However, since this is the prevalent mode of operation in control systems, most communication would occur in the asynchronous mode. Hence, the additional overhead of supporting sparingly used synchronous communication as the default mode can be eliminated. Consequently, Etherware has been developed as a message-oriented middleware.

Messaging based communication requires a specification of message formats. Also, 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 implies ease of adding specifications for new functionality while still honoring the original specifications used by older components. Based on these requirements, Etherware uses XML [52] as the language for messages. All communication in Etherware is through messages, which are well-formed XML documents with appropriately defined formats. For platform independence, and due to availability of support for XML, Etherware has been implemented using the Java programming language [53]. While these choices incur some additional processing overhead, advances in computer hardware and the use of open standards more than compensate.

3.1.4 Architecture

The need to support system evolution has motivated a basic design choice in Etherware. All components on a given node are managed in a single process, and a component can have its own additional threads of control if necessary. This allows a fine grained control on the execution life cycle of components. It also provides communication savings by reducing expensive memory copy operations required to transfer data between different processes on the same node.

The services provided by middleware should also be restartable and upgradeable. This allows basic versions of services to be deployed initially for resource savings, which can then be dynamically updated based on changing application requirements. To accomplish this, invariant aspects of Etherware that cannot be changed dynamically have to be minimized.

The above considerations have motivated a micro-kernel [54] based design for Etherware. This concept proposes the use of a simple, efficient, and robust kernel that constitutes the system invariant that will not require changes during operation. For Etherware, this includes primitives for managing component life-cycles and delivery of messages between them. All other functionality is implemented just as other application components, and hence they can be restarted or upgraded without having to restart the system.

Pursuant to the above philosophy of flexibility, a bare minimum functional interface has been imposed for interactions between components and Etherware as well. This minimizes the dependence of component design on the Etherware architecture, due to which the latter can be modified with minimum corresponding changes required in the former. For uniformity, application components also interact with middleware services through messages.

3.2 Etherware programming model

An Etherware based application is composed of a set of components. The components collaborate by exchanging messages with each other. However, each component interacts directly with Etherware, and all messages are delivered by Etherware to respective components. However, for simplicity, the role of Etherware is abstracted out in the rest of this section, and components are shown to interact directly with each other.

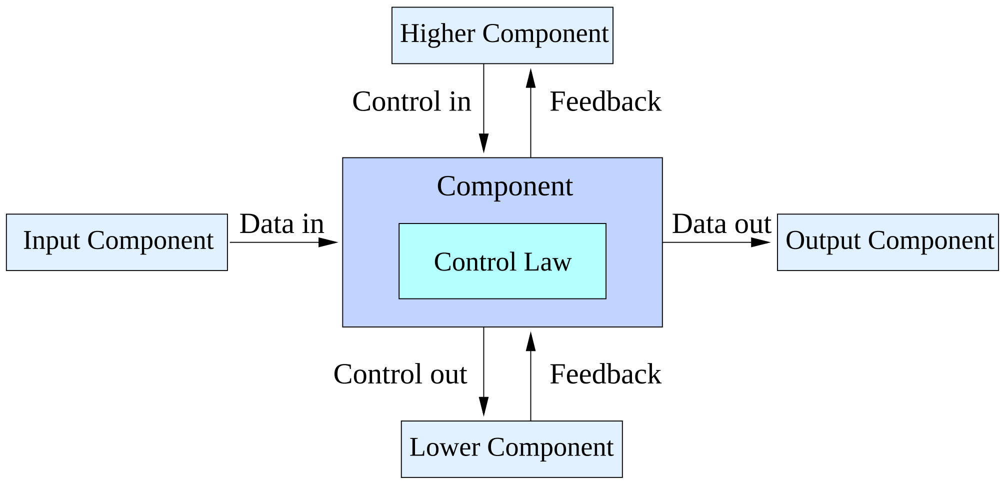


Figure 3.2: Application model in Etherware

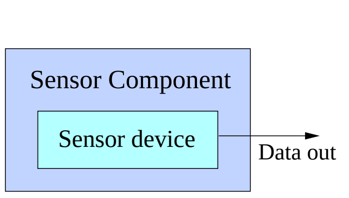
3.2.1 Application model

The possible interaction of a typical component in a networked control application is shown in Figure 3.2. Each such component encapsulates a control law, which represents the application logic associated with its functionality. As shown in Figure 3.3, this may include a set of equations for a controller, inputs from a sensor device, or output to an actuator device.

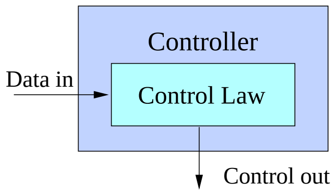
A generic component in a networked control application can interact with other components by participating in the following:

- **Control hierarchy:** A control hierarchy in Figure 3.2 is shown as a vertical stack of components, where a "higher" level component sends controls to a "lower" level component and obtains feedback from it. For example, the Controller in Figure 2.2 takes goals from the higher-level Supervisor and sends commands to the lower-level Actuator.

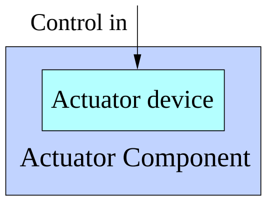
- **Data flow:** A data flow is shown as a horizontal chain of components in Figure 3.2, where an input component sends data to an output component. For example, the VisionServer of Figure 2.2 takes inputs from VisionSensors and provides sensor updates to the Controllers.



(a) Sensor Component



(b) Controller Component



(c) Actuator Component

Figure 3.3: Typical application components in Networked Control Systems

3.2.2 Messages

Based on the considerations of Section 3.1.3, Etherware has been designed as a message oriented middleware. Hence, in Etherware based applications, components communicate by exchanging messages that are well formed XML documents [52]. XML documents can be directly manipulated as large strings. However, Etherware also provides a hierarchy of Java classes to manipulate these documents. The classes provide various primitives to manipulate the underlying XML document across a Java based interface. Further, each class encodes the XML format of a specific message type in Etherware. This hierarchy can also be easily extended to define additional message types with user-defined XML formats.

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 addressed by associating service descriptions called *ServiceProfiles* to addressable components. A component can register multiple ServiceProfiles, and multiple components can register the same ServiceProfile. In the latter case, a given ServiceProfile is identified with one of the corresponding components, and hence ServiceProfiles do not uniquely identify components.

A ServiceProfile describes a specific service that a component provides. For example, a VisionSensor could register a Profile specifying that it operates a gray-scale camera covering the geographic region between coordinates (0,0) and (100,50), and tracks car positions based on this. Suppose a car Controller knows the location of its car to be within coordinates (25,37) and (45,52). It could use only this information to connect to a relevant vision sensor using an appropriate service query. Etherware would then match this query with the description of the VisionSensor, and forward the connection message to it. Note that the car controller need not specify the type of camera in its query, as this is irrelevant to the service it needs.

All messages in Etherware have the following three constituents as XML tags:

- **Profile:** This is used to determine the recipient of the message. A Profile can be one of the following:
  - **ServiceProfile:** Addressing messages using ServiceProfiles allows components to communicate without having to identify recipients. For example, for a one-time request response operation, a client need not discover a server's Binding before communicating with it. However, successive messages may be delivered to different components that have registered matching ServiceProfiles as these do not uniquely identify components.
  - **Binding:** This causes messages to be directly routed to the component with the given Binding. This is the preferred method for sending multiple messages as it ensures that all the messages are sent to the same component. In addition, this also reduces the overhead of matching ServiceProfiles for each message.
  - **Tap:** This is a special kind of Binding associated with an end-point of a *MessageStream*. For instance, when a component has opened several MessageStreams to another component, each such stream is identified by its associated Tap. MessageStreams are described in detail in Section 3.2.4.
- **Content:** This represents the contents of the message. All application specific information is contained in this tag. This can have application defined formats and is not

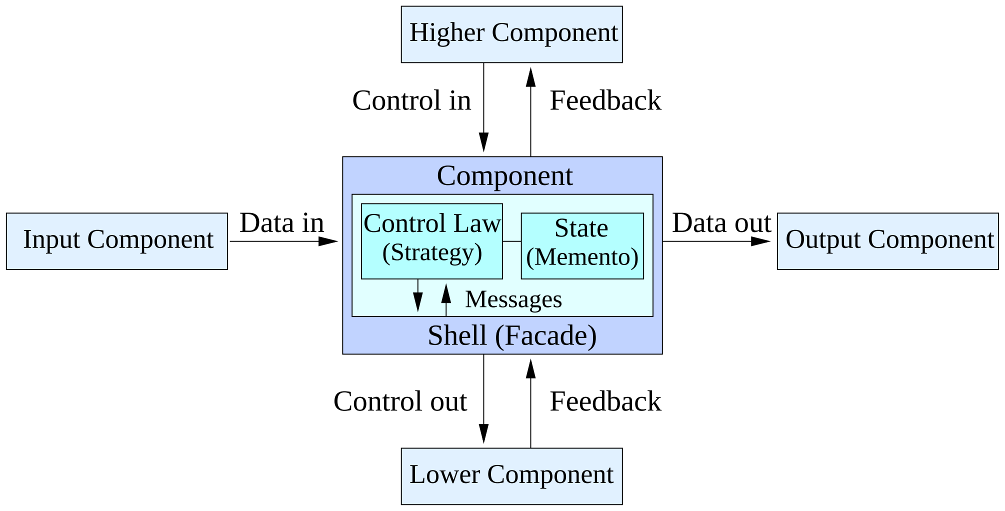


Figure 3.4: Programming model for a component in Etherware

processed by Etherware.

- **Time-stamp:** Each message has a time-stamp specifying when the message was generated. As a message moves from one node to another, the time-stamp is automatically translated to the local time of corresponding node. However, any time-stamps in the *content* tag will not be translated, and so any such time-stamp should be relative to the time-stamp of the message for proper interpretation.

3.2.3 Components

The design of a generic component in Etherware is shown in Figure 3.4. This design is based on several design patterns [55], where "a design pattern is a solution to a problem in a context" [55]. In software development, several problems may have a common recurring theme. Design patterns represent solutions to such problems that exploit the recurring theme. However, the solutions need to elaborated based on the context of the given problem. Accordingly, the design patterns shown in Figure 3.4 have the following rationale:

- **Memento:** Support for restarts and upgrades requires the ability to capture component state. This is addressed using the Memento pattern, based on which component state can be externalized as a Memento object, check-pointed, and then used for re-initialization.

- **Strategy:** System evolution requires the ability to replace components dynamically. In particular, this requires the functional (syntactic) interface of the old and new components to be compatible, so that a replacement does not require changing the rest of the system. This is an application of the Strategy pattern, whereby all components communicate with Etherware using the same functional interface. During component replacement, the Strategy pattern is used in conjunction with the Memento pattern for check-pointing component state.

- **Facade:** Application components interact with Etherware services by exchanging messages. However, if components have to directly interact with the corresponding subsystems, then the Etherware architecture needs to be exposed to component code. This introduces unnecessary dependencies and makes components and Etherware hard to evolve, as changing the Etherware architecture will require software of all the components to be updated as well. This dependence is eliminated by using the Facade pattern to provide a uniform functional interface and abstract away the architectural details of Etherware.

Based on the above design, components are required to interact with Etherware across a fixed functional interface. This specifies the set of functions that a component may call in Etherware and the set of call-backs that it must implement. However, the actual semantics of component interaction is determined by the messages that are exchanged during these function calls. Hence, the operational interface of a component is determined by the types of messages that it can generate or process, and not the actual functions used for sending or receiving them.

Components in Etherware based applications must belong to one of the following types:

- **Passive components:** These do not have active threads of control. They are only activated by call-backs triggered by incoming messages. Also, they can send messages only by returning them during a call-back from Etherware.

  Passive components are required to implement the following interface:

  ```
  public interface Component {
    /** This initializes the Component. */
    public List<Message> initialize(Memento memento, Binding binding);

    /** This processes an incoming Message. */
    public List<Message> process(Message message, Binding binding);

    /** This terminates the Component and captures its state. */
    public Memento terminate();
  }
  ```

  This interface requires passive components to implement the following call-backs:

  – *initialize():* This is called when the component is first created. The component is initialized with a Memento, which is a special kind of Message that contains component state. This can either be a pre-specified initial state for a newly created component, or a compatible check-pointed state returned in a *terminate()* method by a previous component. The Binding of this component is also passed as a parameter during initialization.

  – *process():* This is called when a component has an incoming message to process. The Binding of the sender of this message is also passed as a parameter. The implemented function can return a list of generated Messages.

  - *terminate():* This is used to terminate a component and capture its state so that it can be restarted, upgraded, or migrated. Accordingly, this method must return a Memento that can then be used to a re-initialize a compatible component.

- **Active components:** These have one or more independent threads of control. They can generate messages during call-backs as well as based on activities in their own threads of control. For instance, the VisionSensor of Figure 2.2 can be implemented as an active component with a separate thread that waits for camera updates and generates update messages when new sensor data is available.

  Active components are required to implement the following interface:

  ```
  public interface ActiveComponent extends Component {
    /** Activate the independent threads of control. */
    public List<Message> activate(Messenger messenger, Scheduler scheduler);
  }
  ```

  This interface requires active components to implement the call-backs listed above for passive components as well. In addition, they must implement the *activate()* call-back in which independent threads of control can be activated using the *Scheduler* interface. In addition, the *Messenger* interface can be used to send messages generated by the independent threads of control in the component.

3.2.4 MessageStreams

Messages addressed with ServiceProfiles and component Bindings are delivered reliably and in order by Etherware. However, as noted in Section 2.4.1, control applications need the ability to trade-off reliable delivery for low delays. This is supported by the notion of MessageStreams in Etherware. A *MessageStream* identifies a stream of messages from a source component to a sink component and supports various quality of service requirements

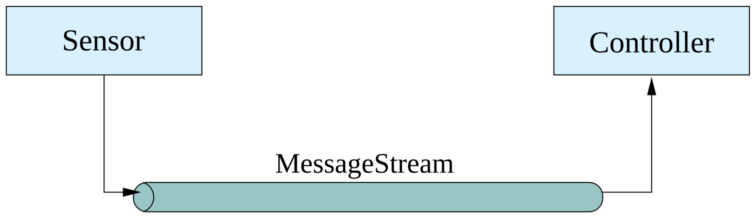


Figure 3.5: MessageStream between two components

for messages delivered through it. For instance, a feedback MessageStream from a Sensor to a Controller shown in Figure 3.5 can have unreliable and in-order delivery.

The source and sink components of a MessageStream are associated with corresponding *Taps*, which are special Bindings that identify access points into the MessageStream. In particular, the source component must address its messages to its source Tap for delivery over the MessageStream, and messages received by the sink component have its sink Tap as the sender of these messages.

A MessageStream is a first-class entity in Etherware. This means that messages can also be directly addressed to a MessageStream. However, such messages are processed by the Etherware entity associated with the source Tap and are not forwarded to the source component. In addition, a MessageStream can also be associated with a ServiceProfile, which is set as the profile for all messages delivered over the MessageStream. More specifically, the source component sets the profile of its messages to its source Tap, which is then reset to the given ServiceProfile when the messages are delivered over the MessageStream to the sink component. In particular, messages addressed to the ServiceProfile of the MessageStream are processed by the source Tap entity, and are not forwarded to the associated components.

3.2.5 MulticastStreams

MessageStreams are extremely useful for streams of messages from specified source components to sink components. However, a key limitation is that only one sender and one receiver can be associated with a MessageStream. Hence, Etherware supports Multicast-

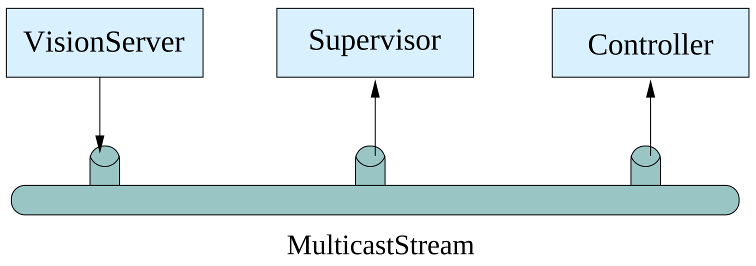


Figure 3.6: MulticastStream between multiple components

Streams to address situations where more than two components are associated with a stream of messages. A *MulticastStream* is a group communication primitive that allows a group of components to interact by exchanging related messages. For instance, a MulticastStream between three components is shown in Figure 3.6. In this example, there is one sender, a VisionSensor, sending feedback updates to multiple receivers, a Supervisor and a Controller. However, a MulticastStream can have multiple senders and receivers, and in particular, a component can both send and receive messages over a MulticastStream.

Similar to the MessageStreams described in Section 3.2.4, all components associated with a MulticastStream have corresponding Taps as well. However, these Taps can be used both to send and receive messages into the MulticastStream. More specifically, a sender addresses its messages to its corresponding Tap, and these messages are delivered to all other components associated with the MulticastStream. In particular, the sender does not get a copy of the message that it sent.

MulticastStreams are first class entities in Etherware, and hence a MulticastStream is associated with a ServiceProfile as well. Consequently, components join a MulticastStream by sending appropriate *join* messages addressed to the associated ServiceProfile. However, as noted before, senders must address multicast messages to the corresponding Taps. In particular, messages addressed to the ServiceProfile of the MulticastStream are processed in Etherware itself, and are not multicast to other components. Finally, the profiles of delivered messages are reset to the ServiceProfile of the MulticastStream as is done in MessageStreams.

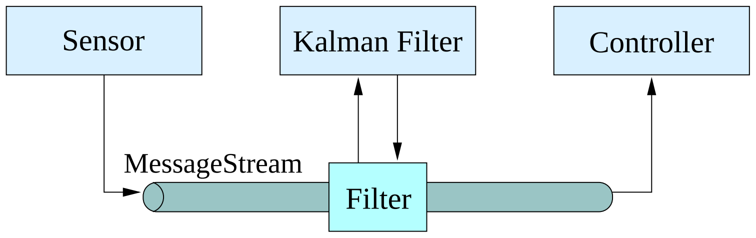


Figure 3.7: Filter for a MessageStream

3.2.6 Filters

Evolving operating conditions usually require appropriate changes in the application software as well, and the ability to upgrade component software in Etherware supports such changes. But in many cases, it may not be necessary to completely replace existing components, and adding an additional component that filters some of the exchanged messages is sufficient. For instance, updates from the VisionSensor could get noisy due to bad lighting conditions. This can be addressed by dynamically adding a Kalman Filter to the MessageStream from the VisionSensor to the Controller as shown in Figure 3.7. This approach is preferable to updating the Controller itself as the Kalman Filter is a simple component with well-defined functionality, while the Controller is much more complicated component and hence more difficult to change. More importantly, this change only affects the communication between the VisionSensor and Controller, and in particular, is transparent to both these components.

Filters are components that intercept messages sent or received by other components, and any component can register to be a Filter of another first class Etherware entity such as a component or a MessageStream. Also, to add itself as a Filter, a component just sends an appropriate *add-filter* message to Etherware. If successful, then an appropriate Tap is returned to the component, which is then used to receive and forward intercepted messages. In particular, a component Filter intercepts all messages sent and received by a component, while a MessageStream Filter only intercepts messages sent over the MessageStream. Finally, MulticastStreams do not support Filters as the set of receivers is not known due to broadcast semantics, and messages received by individual components can be intercepted by component

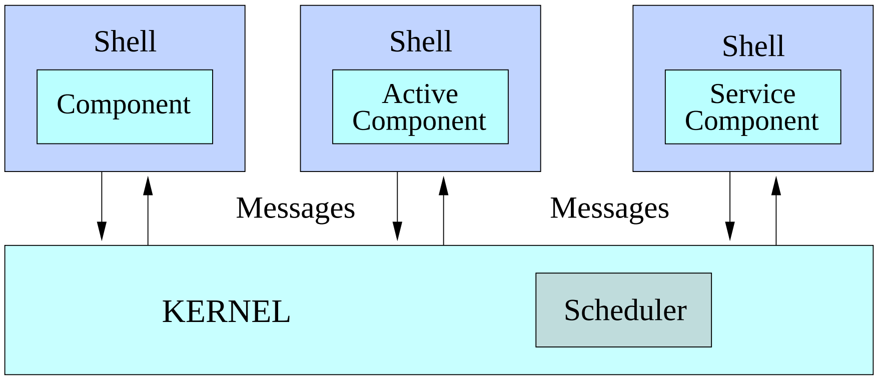


Figure 3.8: Architecture of Etherware

filters if necessary.

3.3 Etherware architecture

The architecture of Etherware is based on the micro-kernel concept as noted in Section 3.1. In particular, all the application components on a given node are managed in a single OS process whose configuration is shown in Figure 3.8. According to the micro-kernel concept, the process is managed by a simple Kernel, and all other functionality is implemented as application level components. This section describes the main Etherware entities and services in detail.

3.3.1 Kernel

As illustrated in Figure 3.8, at the heart of each Etherware process is its *Kernel*, which represents the minimum invariant entity in Etherware. This implies that all other aspects of Etherware can be changed dynamically with minimal impact to the system. It also allows Etherware to be highly customizable and reconfigurable, as only the required services can be used initially, and additional services can be added later as necessary.

The Kernel is simple, robust, and efficient, as it implements only two basic functionalities:

1. **Component management:** The Kernel manages all the components in the associ-

ated Etherware process. It can also accept messages to dynamically add new components as necessary.

2. **Message delivery:** The Kernel delivers messages to local components in the associated Etherware process. In particular, it only delivers messages addressed to Bindings of the components that are managed by it. All other messages are forwarded to special components called *ProfileRegistry* and *NetworkMessenger*, which are standard Etherware services described later in the section.

3.3.2 Scheduler

The Kernel has a separate *Scheduler* as shown in Figure 3.8. In Etherware, the Scheduler is responsible for scheduling messages delivered to local components. In addition, it also creates and manages threads in active components. Hence, the Scheduler determines the priorities and order of all activities in Etherware.

The Scheduler also provides a very useful *notification service* that simplifies a lot of application design. Many components in a control system need to sleep and wake up after a given delay. For instance, the car Controller of Figure 2.1 needs to be woken up every 100ms to generate a new set of controls. One solution is to have a separate thread for the Controller with a sleep function that is called periodically to sleep for the required duration. However, a simpler solution is to send periodic messages to the Controller to activate it as necessary. This also simplifies the Controller design as it eliminates the need to have a separate thread.

The Scheduler generates two kinds of notification messages:

- **Alarms:** These are one-time wake up messages generated after a given waiting interval. They are useful to components that need to wait for a given amount of time before performing an operation. For example, a Controller, which needs to wait for an Actuator to be initialized, can register to receive an Alarm after the required delay.

- **Ticks:** These are periodic messages generated in a tick stream, and can be used by passive components for periodic activations. In particular, this has been used to implement all soft real-time control in the testbed. For instance, the car Controller in Figure 2.2 operates at 10 Hz and has been implemented as a passive component. For periodic activation, the Controller registers with the notification service to receive periodic tick messages at 100ms intervals.

3.3.3 Shells

All components in Etherware are encapsulated by corresponding Shells as shown in Figure 3.8. A *Shell* presents a facade to the component, and performs the call-backs for component initialization, activation, termination, and message delivery. It also maintains component specific information such as associations with other components and participation in MessageStreams and MulticastStreams. Further, Shells implement all the functionalities associated with the Taps described in Section 3.2. Finally, most of the activities involved in component restart, upgrade, and migration are also performed by Shells.

As mentioned in Section 3.3.1, all other functionality in Etherware is implemented in service components. In particular, each Etherware process has a corresponding component for each of these services. The rest of the section describes the basic services used during the normal operation of Etherware.

3.3.4 Delivery addresses

As described in Section 3.2, components can address messages using various kinds of Profiles such as ServiceProfiles, Bindings, and Taps. However, these are only descriptive identifications of the corresponding components. For actual message delivery, a component is internally identified in Etherware by its *delivery address.* More specifically, the descriptive Profiles used by components are internally mapped to appropriate delivery addressed before they are actually delivered to the receivers.

The delivery address of a component consists of two parts:

- **Local part:** This is the globally unique Binding assigned to the component when it is created by the Kernel. As noted in Section 3.3.1, this is also the part used by the Kernel to deliver messages to local components that it manages.

- **Network part:** This is the network specific part of the delivery address. In particular, this is the network address to which messages from remote components must be sent over the network.

3.3.5 NetworkMessenger

The NetworkMessenger service is responsible for sending and receiving messages over the network. Consequently, it encapsulates all network specific information such as network addresses and protocols, and abstracts this away from the rest of Etherware. In particular, it specifies the network part of the delivery address of a component, which is basically the network address at which its local NetworkMessenger component receives messages using the network specific protocols.

Since all network specific information is encapsulated in the NetworkMessenger, this component needs to be used only if the system operates over a network. In addition, this abstraction also makes it quite simple to port Etherware to a different network technology as it only involves implementing a NetworkMessenger for the target network. Finally, the NetworkMessenger is an active component as it needs separate threads to receive messages from remote nodes.

3.3.6 ProfileRegistry

ProfileRegistry is the service that bridges the gap between the Etherware programming model and internal conventions. In particular, it maps ServiceProfiles and Bindings of components to their delivery addresses. So, when a new component is created by the Kernel, its Binding is also registered with the ProfileRegistry. Further, when messages are addressed with ServiceProfiles, they are forwarded by the Kernel to the ProfileRegistry by default. The ProfileRegistry then performs a lookup to map these Profiles to appropriate delivery addresses of matching components if any.

In the current implementation, each Etherware process has exactly one of the following kinds of ProfileRegistry components:

- **Global ProfileRegistry:** There is at least one such component in the network, and it maintains Profiles of components in all nodes of the network.
- **Local ProfileRegistry:** This registers and looks up Profiles of only the local components in the process. Hence, messages addressed to ServiceProfiles of non-local components are forwarded to a Global ProfileRegistry for further lookup. In addition, a Local ProfileRegistry also updates the Global ProfileRegistry with all registered local Profiles.

3.3.7 Network Time Service

In control systems, the time at which an observation or actuation occurred is as important as the action itself [56] [19]. Hence, multiple clocks in a distributed system pose a serious problem, as the same event is recorded with different time-stamps in different computers. For instance, the time-stamp of an observation at a remote sensor is not very useful at a controller if they use different clocks.

This problem of distributed time can be addressed in two ways. The first approach is to synchronize all clocks in the system using a standard time synchronization protocol such as NTP [57]. However, this approach has the drawback that all computers in the system need to be in the same administrative domain. This can be particularly problematic if interacting components are in different administrative domains whose clocks cannot be

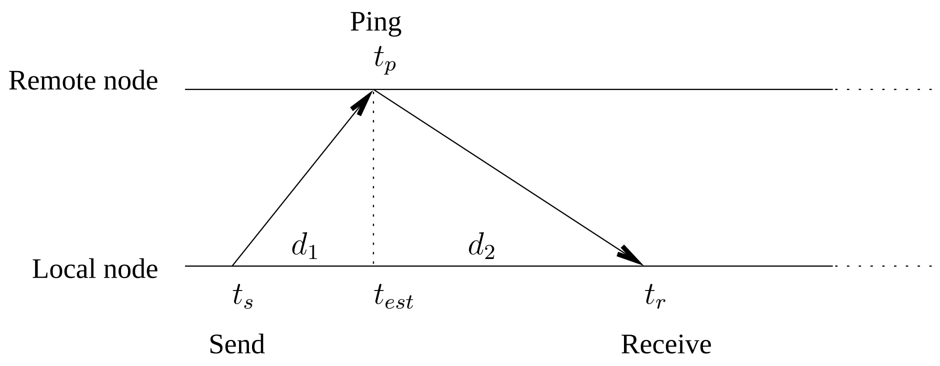


Figure 3.9: Message exchanges in Control Time Protocol algorithm

synchronized. The second approach is to translate time-stamps of messages as they move from one computer to another. Since this approach does not have the drawbacks of the time synchronization approach in multi-domain systems, it has been used in Etherware.

The NetworkTimeService (NTS) is the Etherware service that translates time-stamps of messages as they are transmitted from one node to another. In particular, a time-stamp t_{remote} generated on a remote node is translated to a local time-stamp t_{local} using the relation:

$$t_{remote} = \alpha\ t_{local} + \beta \tag{3.1}$$

In the above equation, the coefficient α represents the skew between clocks, and β represents the offset. These coefficients are computed using the *Control Time Protocol (CTP)* [56] [58] as follows. The local NTS component periodically sends a ping message, which is responded to by the corresponding remote component as shown in Figure 3.9. Assuming symmetric delays, 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. The coefficients α and β are then computed using a windowed least-squares approach [59], where a best-fit line representing the skew and offset between the nodes is computed. If the delays are indeed symmetric, the algorithm gives exact estimates. However, if the delays are asymmetric or noisy, the error is still less than half the round-trip delay time [56].

Based on the above mechanisms, each NTS component builds a table of skews and offsets for every other Etherware process on the network. In particular, these entries are tabulated according to the Bindings of the corresponding remote NTS components. Hence, given a time-stamp with the Binding of the associated remote NTS component, the local NTS component can translate it to the local clock using this table.

The mechanism by which actual message time-stamps are translated in Etherware is also quite interesting. The NTS component is initially added as a Filter to the local NetworkMessenger, and all messages that are sent to and received from remote components are intercepted. The local NTS component then filters outgoing messages to add its Binding to their time-stamps. Similarly, incoming messages are also filtered by the local NTS component, which translates the time-stamp based on the associated Binding and the table it has built up.

The main advantages of the time translation mechanism in Etherware are that the NTS component itself does not need to understand network addresses, and the NetworkMessenger does not need to know about time translation. Also, the natural use of component message filtering in Etherware may be noted.

3.4 Etherware based formalization

Most networked control application designs can be efficiently implemented using the various primitives of the Etherware programming model described in Section 3.2. However, these design descriptions are usually not formal specifications that can be used to prove properties such as safety and liveness in the system. In particular, the inter-component message interactions may have inconsistencies that cannot be easily detected. Hence, it is imperative to formally specify and verify component designs and interactions to ensure correct system operation.

Formal languages such as process algebras, Petri nets, and rewriting logic are widely used for system design, specification, and verification. However, such specifications usually have

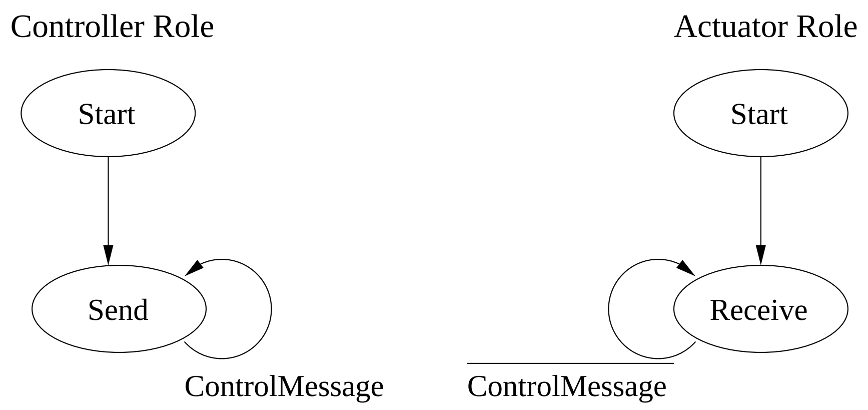


Figure 3.10: Controller-Actuator protocol in testbed

to be implemented in an appropriate programming language to be tested in an operational system. This approach has several disadvantages: the implementation may not faithful to the formal specification, the implementation itself cannot be formally verified, and the additional steps increase design cycle times. In this section, we describe the explicit support in the Etherware based framework to bridge this gap between formal specifications and rapid prototyping.

3.4.1 Formal application design

The interface of a component is the aspect of its behavior that affects other components that interact with it. For Etherware based components, this is the internal model of component behavior, and the set of messages that it sends and receives. Such interfaces are usually specified and analyzed using formal languages with corresponding primitives. In the following, we present a methodology for specifying Etherware based component interfaces using Maude [60], a high performance reflective language and system which supports equational and rewriting logic.

The complete interface of a given component is usually not required to interact with it. Typically, other components only need specific subsets of this interface for their interaction. For instance, while implementing the Controller component of the testbed in Figure 2.1, it

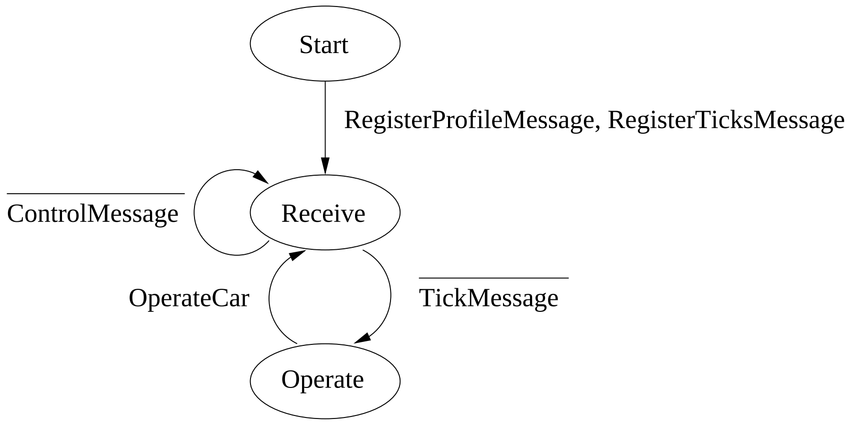

Start
RegisterProfileMessage, RegisterTicksMessage
$\overline{\text{ControlMessage}}$
Receive
OperateCar
$\overline{\text{TickMessage}}$
Operate

Figure 3.11: Finite state machine for Actuator in testbed

is sufficient to know the type of control messages that the Actuator consumes. Details about interactions between the Actuator and other components are not relevant. Consequently, the interface specification must clearly delineate the subset of the Actuator interface that is relevant to the Controller.

The following concepts capture the above notion of a component interface:

- **Protocols:** The interaction between components is captured in formally defined protocols. For example, the communication protocol used between the Controller and Actuator components of the testbed is shown in Figure 3.10. This specifies that both components begin in the *Start* state. The *Controller* then moves to the *Send* state and begins to send *ControlMessage* updates. Similarly, the *Actuator* goes to the *Receive* state, and can receive *ControlMessages* only after it is in this state. The overbar indicates the reception of a message. Note that this protocol specifies only the aspect of Actuator behavior that is relevant to the Controller, and vice versa.

- **Roles:** The protocol described in Figure 3.10 involves two components interacting in a prescribed manner. These constitute the roles in this protocol. Note that any two components can participate in this protocol as long as they honor their corresponding roles. In general, we can specify multi-party protocols with corresponding roles for each of the participants.

- **Interface:** A component participates in one or more protocols by assuming a specific role in each protocol. The component interface is then a combination of these roles. For example, Figure 3.11 specifies the actual interface of the Actuator component in the testbed. We see that the Actuator role in the protocol of Figure 3.10 is implemented by substituting the *Receive* state with two states, namely *Receive* and *Operate*. The interface also shows other messages received or sent by the Actuator, while interacting with Etherware services and the actuator device.

This approach to application design has several advantages. First, the interactions between components can be formally specified as protocols in Maude. Second, the formal specification helps in analyzing and minimizing dependencies between components. Third, the formal specification can be used to verify and prove properties about component interactions using the rich tool support in Maude [60]. Fourth, component behavior can be specified and verified as a correct composition of relevant protocols. Finally, the above component specifications can be composed into a formal specification of the entire system, which can then be used to study system level properties such as safety and liveness.

3.4.2 Etherware support

Maude specifications can be executed using the fairly powerful Maude execution environment. In particular, the environment supports a "loop mode" that allows other programs to exchange data with executing Maude specifications. This functionality has been exploited to support rapid prototyping of Maude component designs using Etherware. As shown in Figure 3.12, an Etherware based Maude proxy service manages a Maude execution environment in an Etherware component, and exchanges messages between the Maude prototype and the rest of Etherware. Interestingly, multiple instances of such components can also be executed at the same time.

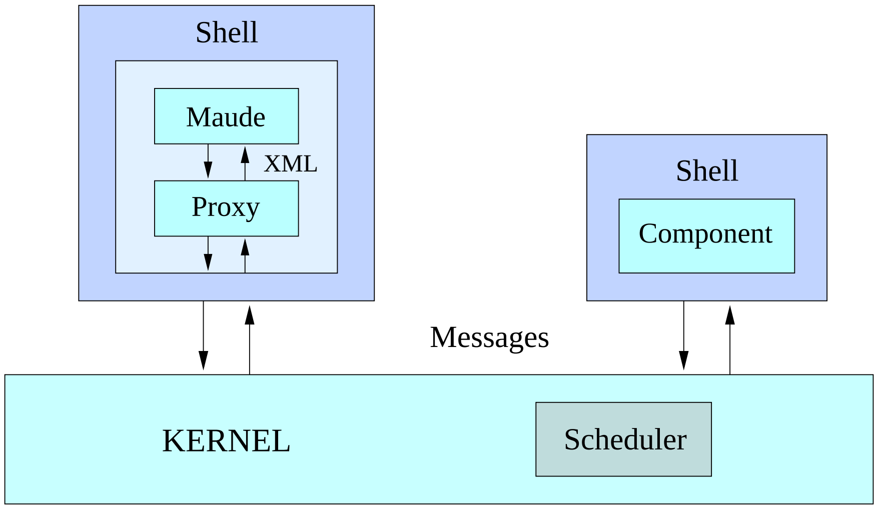


Figure 3.12: Architecture of Maude-Etherware interface

3.5 Etherware capabilities

This section describes how the requirements listed in Section 2.3 are addressed by the design principles and architecture of Etherware presented in this chapter.

3.5.1 Operational capabilities

The operational capabilities of Etherware are based on its programming model and services:

- **Distributed operation:** As described in Section 3.2, Etherware based components communicate by exchanging messages using the same primitives regardless of whether they execute on the same computer or on different computers. In particular, the problems of identifying and locating components are addressed by the ProfileRegistry and NetworkMessenger services described in Section 3.3.

- **Location independence:** This is achieved in Etherware by assigning each component a globally unique Binding, an addressing scheme that is independent of network conventions and topologies. As explained in Section 3.3.4, each Binding is mapped into a delivery address, which includes network specific details such as IP addresses. In particular, even though the network part of a component's delivery address changes when it is migrated from one computer to another, this is still transparent to other components as these details are abstracted away in the programming model.

- **Service description:** This is supported by addressable components being able to register ServiceProfiles. A component that wishes to access a given service can just address messages using an appropriate ServiceProfile. The Profile is then matched with registered Profiles by the ProfileRegistry. 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 the use of XML documents for communication between components, as described in Section 3.1. This helps solve many integration problems that arise due to incompatible interfaces. Besides, components use a simple and uniform functional interface for all interaction, as described in Section 3.2.3.

- **Semantics:** Application semantics are usually specified and analyzed using formal languages. As described in Section 3.4, the framework supports a protocol based specification of application design, and Etherware itself supports rapid prototyping by allowing executable design specifications in Maude to be deployed in an operational system.

- **Distributed time:** This issue is addressed by the automatic translation of message time-stamps provided by the NetworkTimeService described in Section 3.3.7.

3.5.2 Non-functional capabilities

The following non-functional capabilities of Etherware are primarily due to its component model and architecture:

- **Robustness:** This is facilitated primarily by the check-pointing of component state based on the Memento pattern. The effect of component failures are contained by efficient check-point and restart mechanisms in Etherware as demonstrated in Section 4.3.1.

- **Delay-reliability trade-off:** MessageStreams support trading off reliability for lower delays. They also provide a simple mechanism to incorporate support for other quality of service requirements as necessary.

- **Security:** Security overrides can be easily implemented by adding Filters to components and MessageStreams. Also, the order of Filters in a MessageStream essentially determines their priority. In particular, later Filters have higher priority as they intercept messages already filtered by earlier Filters. For instance, a security override Filter will be safety preserving if it has a lower priority than the safety Filter. These and other issues are considered in detail in Chapter 6.

- **Other requirements:** The current algorithms for various services scale well for the requirements in the testbed. In particular, they have been tested by operating up to eight cars at a time, which represents the scenario with the maximum load in the system. However, better algorithms can be easily incorporated if necessary.

3.5.3 Management capabilities

Provisions for configuration management and check-pointing in Etherware support the following management capabilities:

- **Startup:** Start-up configurations and dependencies are specified to Etherware using a simple configuration file for each computer. Etherware ensures that components are initialized in the correct order. In the current version, this consists of specifying the absolute order in which components need to be initialized on a given node. However, support for more complicated dependencies can be easily supported using a richer dependency evaluation system.

- **System evolution:** System evolution through component update and migration is supported by component state check-pointing. The application state is maintained across both operations using Mementos, and in particular, MessageStreams are maintained across component updates. As noted in Section 3.2, MessageStreams and Filters also provide an elegant mechanism for system evolution without having to change any existing connections between components.

Chapter 4

ETHERWARE MECHANISMS

In Chapter 3, the programming model and architecture of Etherware were described. However, this description only constitutes a static picture of the system. So, to further elaborate this picture, and illustrate the operation of Etherware, the dynamics and mechanisms of Etherware are presented in this chapter. In particular, the normal operation of Etherware is outlined, the mechanisms for component management are presented, and the performance of Etherware is compared to other middleware.

4.1 Etherware initialization

The constituents of Etherware have various inter-dependencies, which are highlighted during the initialization process. Specifically, the following sequence of activities occurs during the initialization of Etherware:

1. **Kernel startup:** When an Etherware process is started on a computer, the Kernel and Scheduler are initially started. The Kernel then instantiates and initializes all local components based on appropriate configuration specifications as described in the following steps.

2. **ProfileRegistry startup:** The first service initialized by the Kernel is the ProfileRegistry. The Kernel instantiates either a local or a global ProfileRegistry component as specified in the service configuration. Also, there must be at least one global ProfileRegistry in the network as noted in Section 3.3.6.

The ProfileRegistry maintains a mapping from ServiceProfiles and Bindings of components to corresponding delivery addresses as described in Section 3.3.6. However, since the NetworkMessenger has not yet been initialized, only the local part of delivery addresses is known at this point. Consequently, only local ServiceProfiles and Bindings can be resolved to corresponding components, and hence, only local message delivery is possible.

3. **NetworkMessenger startup:** This is the second service initialized by the Kernel. As noted in Section 3.3.5, the NetworkMessenger is responsible for sending and receiving messages on the network, and hence, it must have a network address, such as an IP address, to receive messages. As part of its initialization, the NetworkMessenger registers itself with the ProfileRegistry and specifies the local network address at which it can receive messages over the network.

   On receiving a registration message from the NetworkMessenger, the ProfileRegistry then performs the following actions:

   - It completes the partial delivery addresses in its repository with this network address.
   - It sends a message to the NetworkMessenger to be broadcast on the local network. This is received by other NetworkMessengers and delivered to corresponding ProfileRegistry components. In particular,
     - A Global ProfileRegistry broadcasts an *announcement* message to the local ProfileRegistries.
     - A Local ProfileRegistry broadcasts a *discovery* message to the global ProfileRegistry. This is responded by an announcement from a global ProfileRegistry, if one has been initialized.
   - When a Local ProfileRegistry receives an announcement, it notes the address of

the global ProfileRegistry, and sends back an update with all the Profiles that it has registered.

Since the Global ProfileRegistry has information about all components in the network, and the NetworkMessenger can send and receive messages over the network, all messages can be properly delivered at this point.

4. **Other services:** The remaining services are initialized by the Kernel according to the service configuration. In particular, the NetworkTimeService discovers Bindings of all its peers on the network by a specialized query to the ProfileRegistry. Of course, such a query can be used by an application component to discover a set of peers as well.

5. **Application components:** Finally, the Kernel initializes application components according to an application configuration file, which specifies the following for each component:

   - The Java class file corresponding to the component.
   - The XML document for the initialization state (Memento).
   - The total order for initializing components.

4.2 Message delivery

Components can address messages using Profiles such as ServiceProfiles, Bindings, and Taps. However, as noted in Section 3.3.4, these must be mapped to delivery addresses of the corresponding recipients for the messages to be delivered in Etherware. Hence, the two issues that need to be addressed for message delivery are mapping Profiles to delivery addresses and delivering messages based on these addresses. These issues are now considered in detail.

4.2.1 Message headers

Messages have headers that contain all the information related to message delivery. This includes:

- **Sender:** Delivery address of the sender of the message.
- **Receiver:** Delivery address of the receiver of the message.
- **Filters:** Delivery addresses of Filters that need to process the message, as well as those that have already done so.

A lot of this information is already available in the component Shell of the sender. In particular, the Binding of the component, which is the local part of the sender's delivery address, is provided to the Shell during component initialization. Also, if the message is part of a MessageStream, or even a response to a previous message, then the receiver's delivery address is cached in the Shell. Further, when Filters register with components or MessageStreams, their delivery addresses are also available as part of the header of the registration message. Consequently, all this information is automatically filled in the message header by the sender's Shell.

The remaining information in the message header, basically the receiver's delivery address and the network part of the sender's delivery address, has to be resolved before message delivery can be done. However, since this information is distributed among the various parts of Etherware, they are discovered by the following defaulting mechanisms:

- **ProfileRegistry:** This is the default recipient of all messages whose receiver's delivery address is not known.
- **NetworkMessenger:** This is the default recipient of all messages whose next receiver's delivery addresses have non-local network parts.

The following important points must also be noted:

- Messages are only delivered if the final receiver's address can be determined by the ProfileRegistry. In particular, if this cannot be determined, then an appropriate exception message is sent back to the message sender.
- Messages are delivered to actual message receiver only after they have been through all the Filters in the message header. Hence, the next receiver of a message is not necessarily its final receiver.

4.2.2 Delivery mechanism

The following steps constitute the actual message delivery mechanism in Etherware:

1. A component sends a message addressed with a Profile such as a ServiceProfile, Binding, or Tap.
2. The sender's Shell fills the message header with the local part of the sender's delivery address, as well as the delivery addresses of all the Filters for the message. It also fills in the receiver's delivery address if it is known.
3. The sender's Shell sends the message to the Kernel for delivery.
4. The Kernel encounters one of the following three cases while delivering a message:
    - If the receiver's address is known, and the next receiver is a component managed by the Kernel, then the message is delivered to this component's Shell.
    - If the receiver's address is known, but the network part of the next receiver's address is not local, then the message is dispatched to the NetworkMessenger for transmission over the network.
    - If the receiver's address is not known, then the message is sent to the ProfileRegistry for look-up.

5. The NetworkMessenger transmits a remote message over the network. This is then received by the corresponding NetworkMessenger on the remote node, which in turn forwards the message to its Kernel for delivery. Of course, since the NetworkTimeService (NTS) is added as a filter to each of the NetworkMessengers, the time-stamp of the message is translated correctly as described in Section 3.3.7.

6. The ProfileRegistry looks up the receiver's addresses based on the receiver's Profile in the message. If a Profile matching the message receiver's Profile has been registered, then the corresponding delivery address is filled into the message header, and the message is forwarded to the Kernel for delivery.

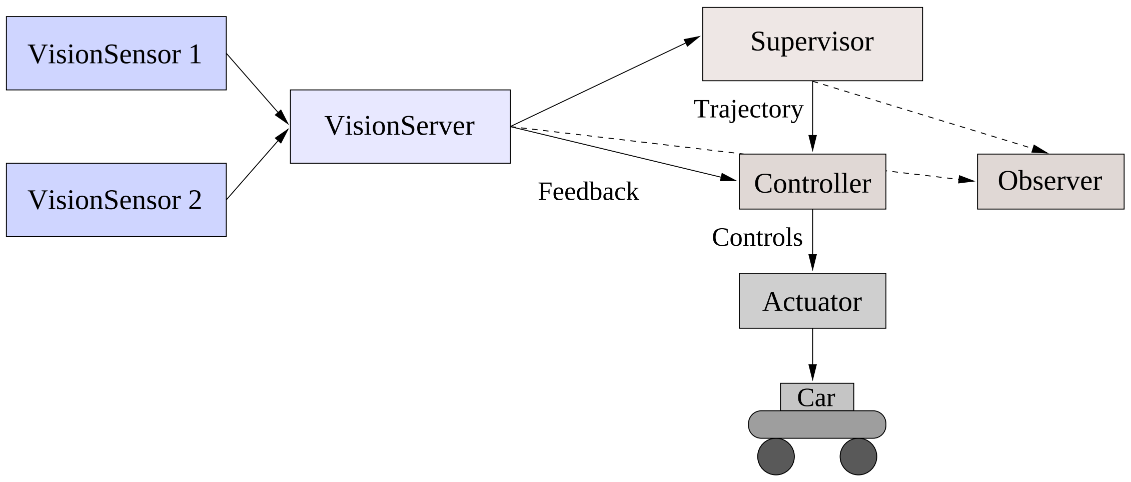
   However, if no matching Profile has been registered, then one of the following occurs:

   - A local ProfileRegistry forwards the message to the global ProfileRegistry for further look-up. However, if a global ProfileRegistry has not yet been initialized, then an exception message is sent back to the sender.
   - A global ProfileRegistry sends back an exception message to the sender.

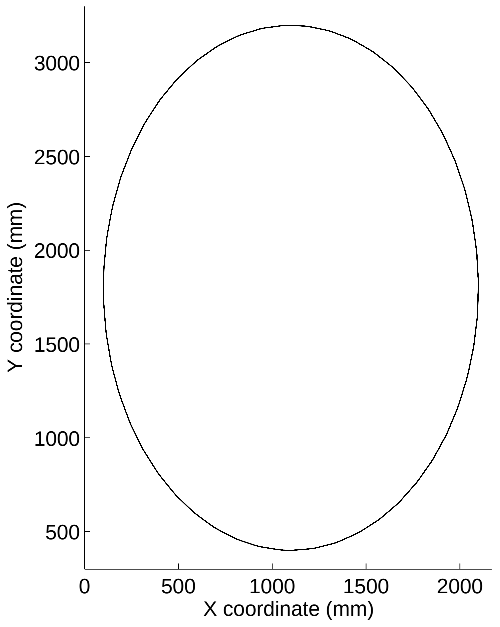
7. The message is received by the next receiver's Shell, which then removes the header and forwards it to the component. In particular, if the component is only a Filter and not the final receiver of the message, then the message header is stored and later appended when the message has been filtered.

4.3 System evolution

Etherware has fairly efficient mechanisms to support system robustness and evolution. In particular, the mechanisms for component restarts, upgrades, and migration have been analyzed through detailed experiments in the testbed. We now describe these mechanisms and their validating experiments in this section.



(a) Experimental testbed implementation



(b) Desired car trajectory

Figure 4.1: Experimental setup

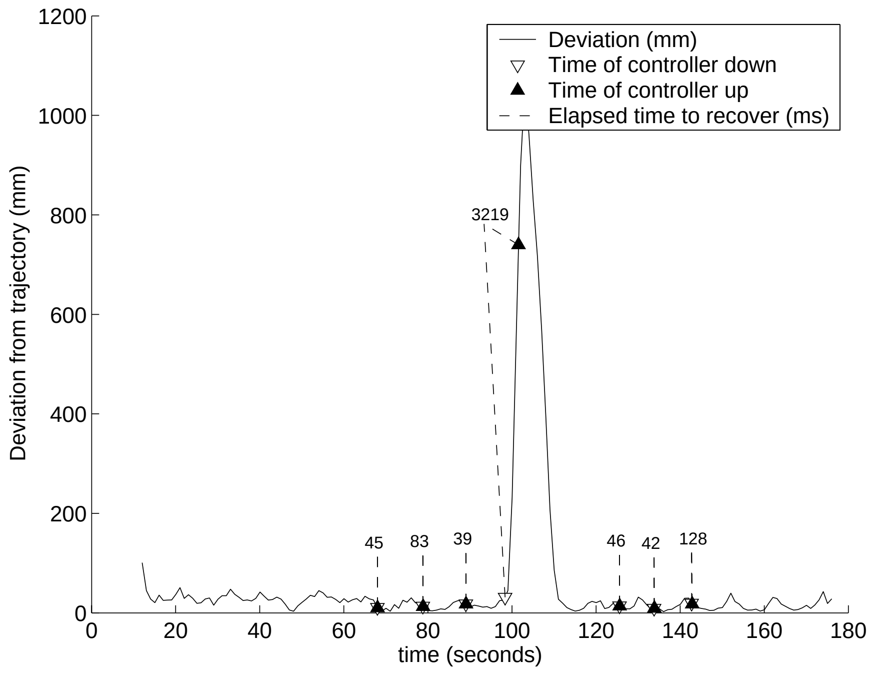


Figure 4.2: Error in car trajectory due to controller restarts

The modified architecture of the testbed for the experiments is shown in Figure 4.1(a). The components in this implementation have the same functionality as their in counterparts in the exploratory implementation of Figure 2.2. The only addition was an Observer component used to track the car positions as shown in Figure 4.1(a). As explained in Section 2.2, the Controller and Actuator were executed on the same computer, while all other components were executed on different computers. Also, in all experiments, the goal was to make the car traverse an oval trajectory as shown in Figure 4.1(b).

4.3.1 Component restart

The goal of this experiment was to analyze Etherware mechanisms for robustness. In particular, the ability to restart a faulty component was studied. To accomplish this, the Controller component was forced to be restarted several times as the car was being driven along the trajectory shown in Figure 4.1(b). Faults were injected at random by performing an illegal operation (divide by zero) in the Controller. Such a fault caused the Controller to raise an exception and be restarted by Etherware.

The deviation of the actual car trajectory from the desired trajectory, as a function of time, is shown in Figure 4.2. Restarts are indicated by pointers, and the accompanying numbers indicate, in milliseconds, the time for each restart. These are times-tamps at the Observer and include communication and synchronization times between the restarted Controller and the Observer.

For the first 60 seconds, the car operated without restarts and tracked the trajectory with an error of less than 50mm. 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 in Etherware. The plot in Figure 4.2 indicates that the error in the car position during these restarts was well within the system error bounds during normal operation.

Two Etherware mechanisms contributed to the quick recoveries. First, the Shell intercepted exceptions thrown due to Controller faults, and maintained the MessageStreams to the other components across the restarts. Second, before termination, the Controller state was check-pointed according to the Memento pattern, and this check-pointed state was then used for proper re-initialization.

To illustrate the effectiveness of these two mechanisms, the Etherware process managing the Controller was restarted at about 100 seconds after the start of the experiment. As shown in Figure 4.2, the subsequent restart of the Etherware process with the Controller took about three seconds. During this time, the actual car trajectory accumulated a large error of about 0.8 meters with respect to the desired trajectory. This clearly illustrates the necessity for efficient restarts. Furthermore, even though the Controller restarted after three seconds, additional error was accumulated before recovery. This was so because the Controller had to reconnect to the other components, rebuild the state of the car, and bring it back on track. This demonstrates the improvement that has been achieved by the check-pointing mechanism in Etherware.

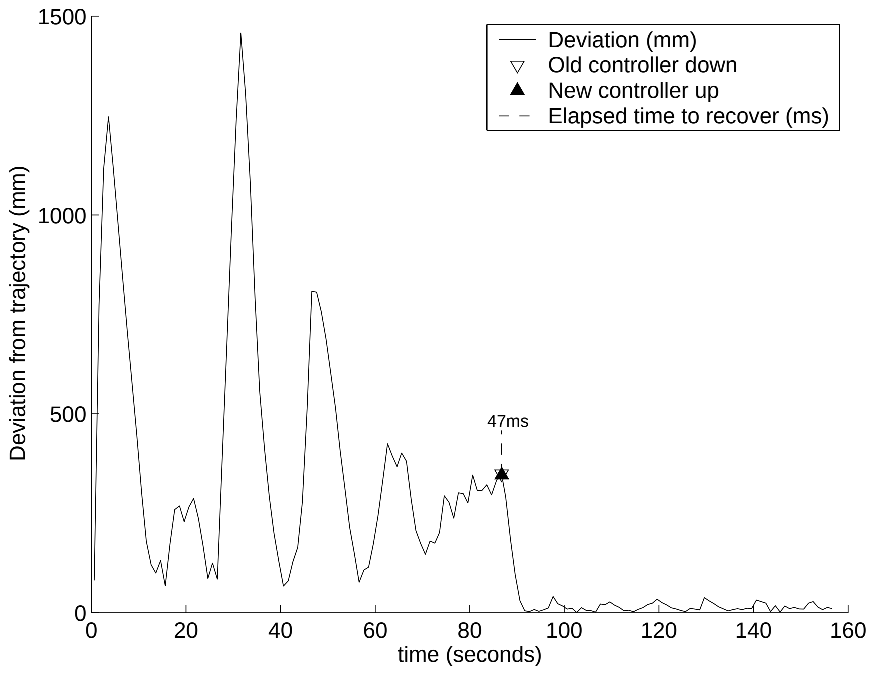


Figure 4.3: Error in car trajectory due to controller upgrade

4.3.2 Component upgrade

This experiment was used to study the mechanisms in Etherware for upgrading components dynamically. To test this, the car was initially controlled by a coarse Controller that operated myopically. Etherware was then commanded, at about 90 seconds after the start, to upgrade the coarse Controller to a better model predictive Controller. The plot of Figure 4.3 clearly shows the improvement in the car operation after update. As shown in the figure, the involved transients are well within the system error bounds.

This functionality is due to three key Etherware mechanisms. 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 MessageStreams to 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 as well as the operation of the car.

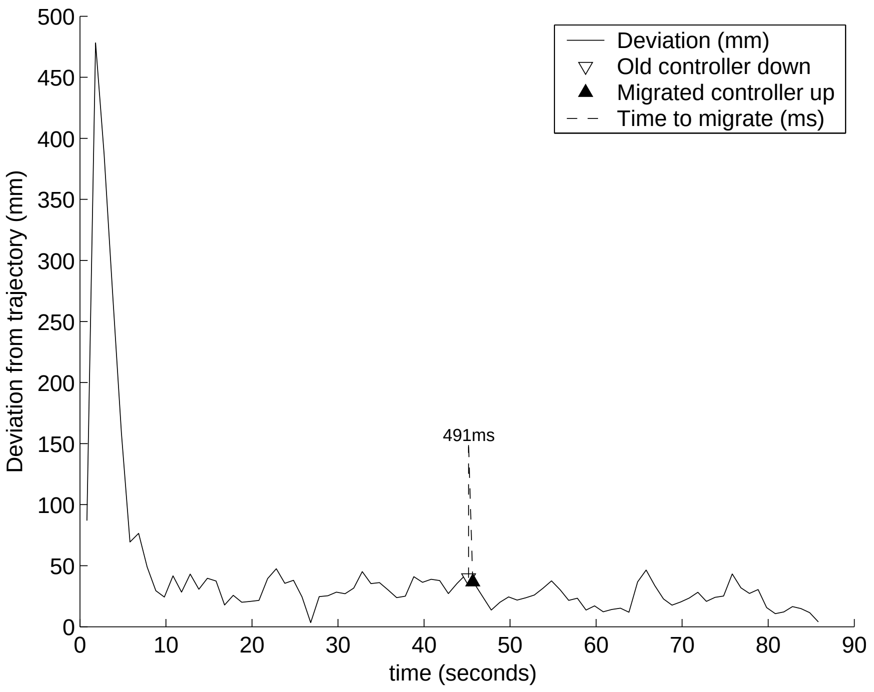


Figure 4.4: Error in car trajectory due to controller migration

4.3.3 Component migration

Etherware support for component migration was analyzed in this experiment. As before, the error in the car trajectory is shown in Figure 4.4. The large spike at the beginning of the graph was the transient error due to the car trying to catch up with its trajectory initially. This is achieved at about 10 seconds into the experiment, after which the car follows the trajectory within an error of 50mm. As shown in the Figure, the Controller was migrated from one computer to another at about 45 seconds into the experiment. Clearly 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 the 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 the old Controller.

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

4.4 Etherware performance

A comparison [58] of Etherware with other middleware for real-time and networked control systems is presented in this section. In particular, the performance of Etherware is compared with two widely available implementations of CORBA: ROFES 0.3b [61] and JacORB 2.0 [62]. ROFES supports the Minimum CORBA [32] and part of the Real-Time CORBA 1.1 [63] specifications, while JacORB supports the CORBA 3.0 specification.

For the comparison experiments, the Control Time Protocol (CTP) described in Section 3.3.7 was implemented in all three middleware, and a reference implementation in Java using sockets was used as the baseline. The configuration was basically a client pinging a remote server at a frequency of 10 Hz. The time line for this was as shown in Figure 3.9. 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, while the CTP estimation algorithm was run off-line on the accumulated log files. Hence, the delays in the experiments were mainly due to overhead of the middleware and their communication protocols.

Three scenarios were considered 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) - a fairly standard configuration for IEEE 802.11 wireless networks. The wired network was a 10 Mbps Ethernet, the IEEE 802.11 wireless networks used Cisco Aironet 350 series cards, and 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.

The distribution of round trip times for pings in the three cases are shown in Figures 4.5, 4.6, and 4.7, and the corresponding statistics are tabulated in Tables 4.1, 4.2, and 4.3

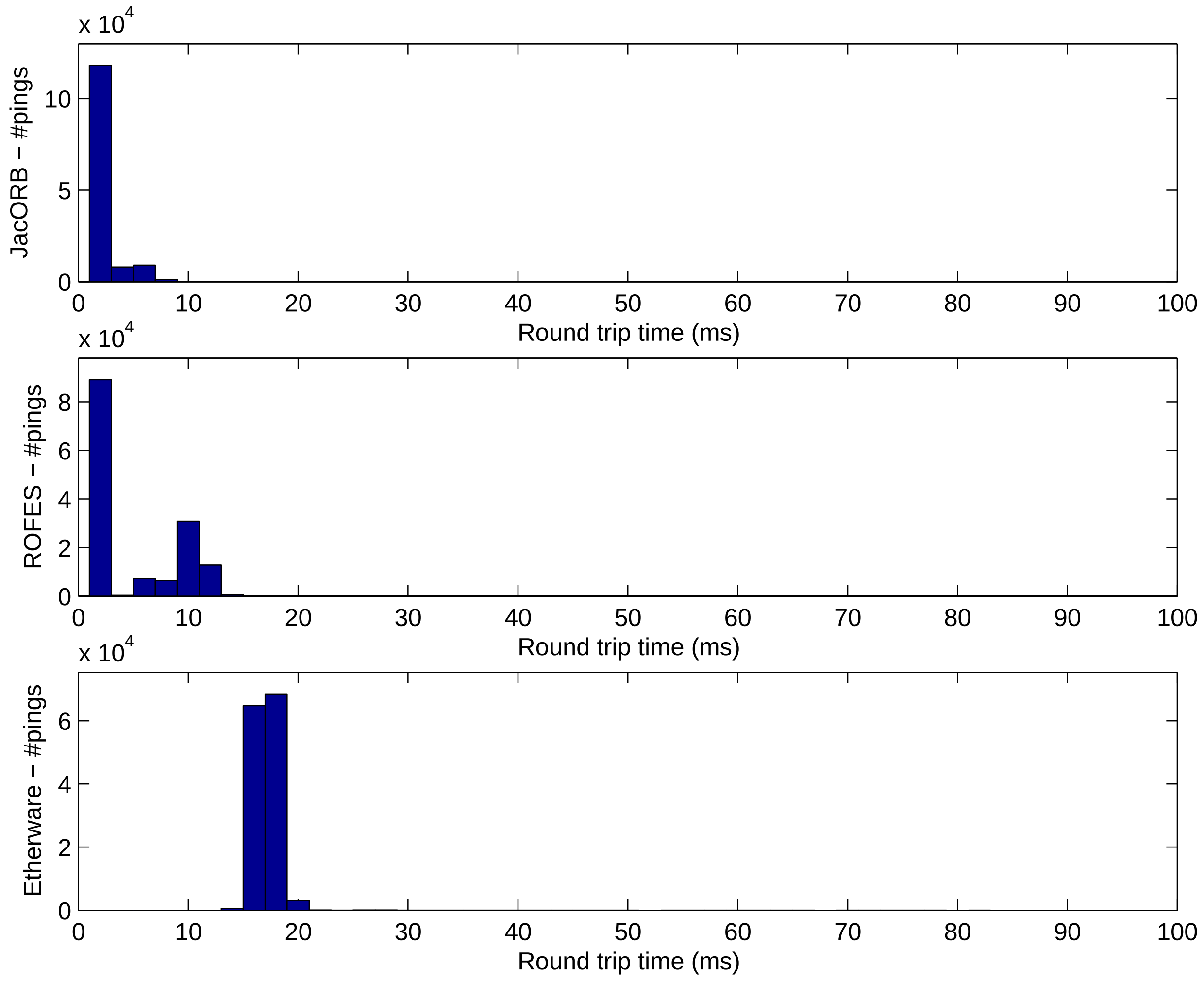


Figure 4.5: Distribution of round trip times for Ethernet

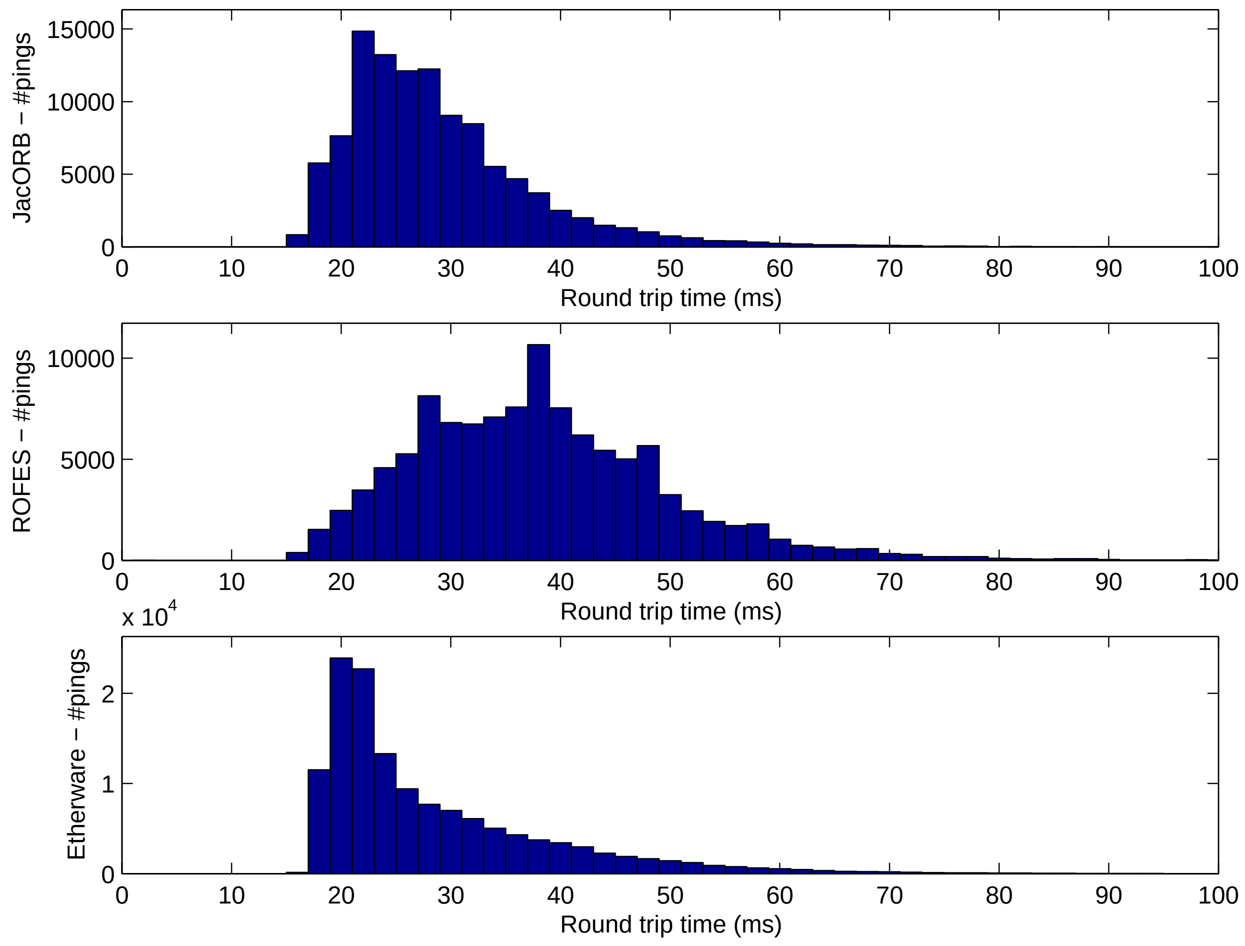


Figure 4.6: Distribution of round trip times for one-hop wireless network

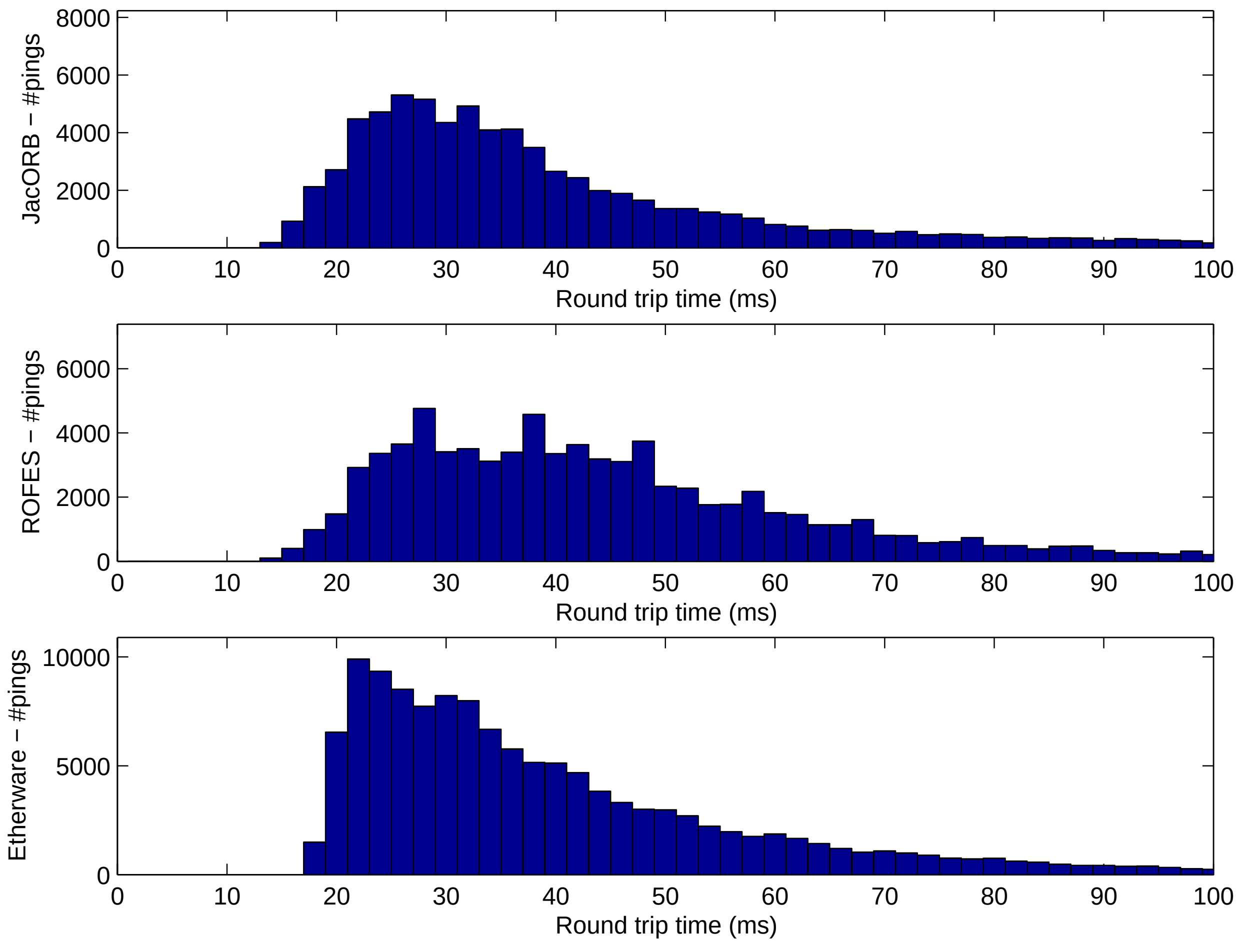


Figure 4.7: Distribution of round trip times for two-hop wireless network (base-station mode)

	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 4.1: 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 4.2: Round trip time statistics for wireless link (one hop)

respectively. Since the delays are fairly small in the Ethernet case, Figure 4.5 and Table 4.1 essentially illustrate the overhead in the implementations of the various middleware. In particular, the relatively high mean delay for Etherware is mainly due to the overhead of the XML implementation in Java. Consequently, the remaining comparisons clearly favor the CORBA implementations. Interestingly, however, in the one-hop wireless link case shown in Figure 4.6 and Table 4.2, 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 4.3.

The impact of these statistics on the estimation algorithms of CTP is also interesting. In particular, the long term behavior of the estimate error $err = t_p - t_{est}$ can be studied by considering its exponentially weighted moving average e of err, which is computed iteratively as follows:

	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 4.3: Round trip time statistics for Base station mode (two hop)

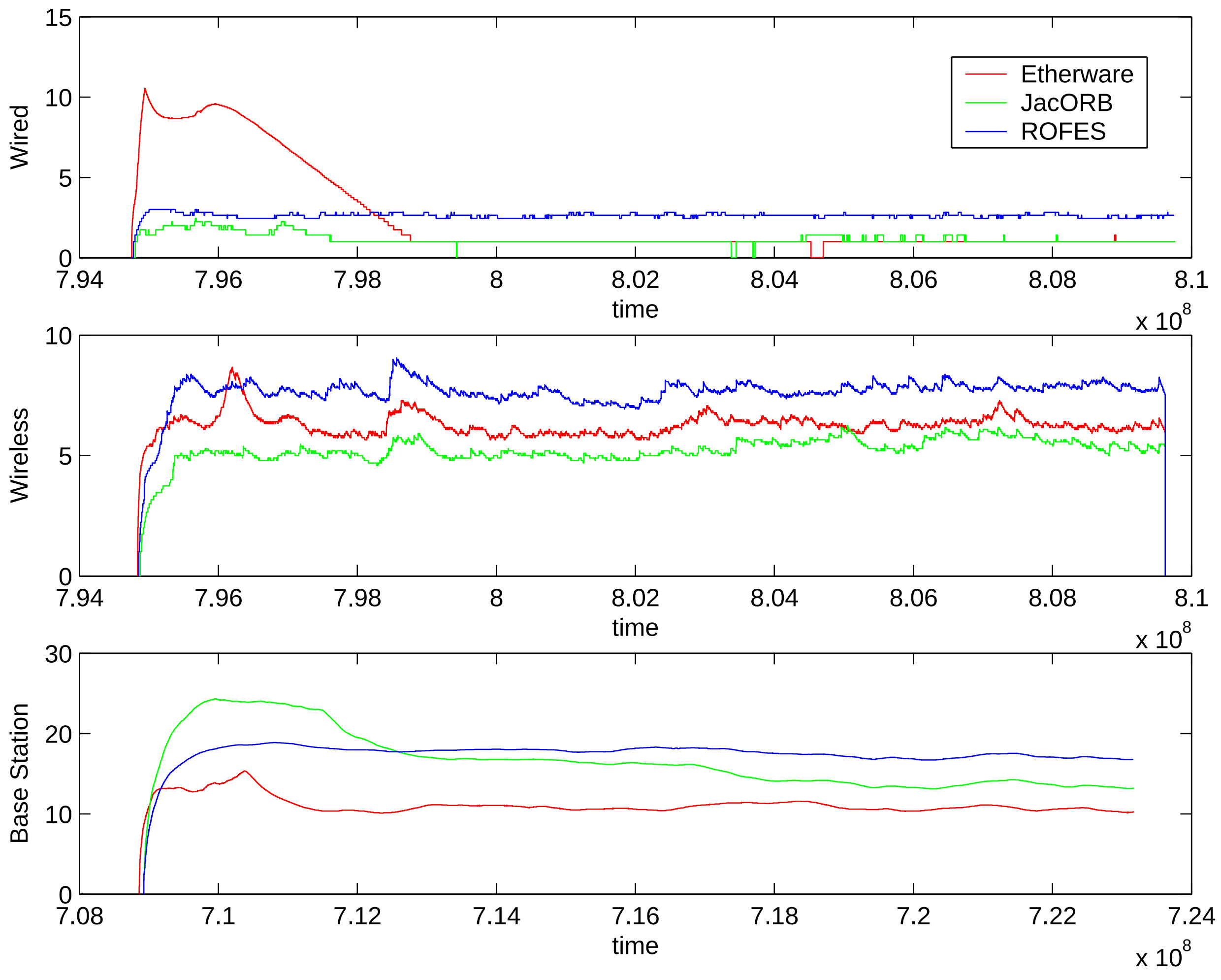


Figure 4.8: Comparison of mean error for the three scenarios

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

Since the error $err = t_p - t_{est}$ could be positive or negative, the squared error err_t^2 at time t is used in the above equation. Also, e_t is the average up to time t. In addition, it must be noted that the CTP implementation assumed symmetric delays, which implies $d_1 = d_2 = d$ in Figure 3.9.

The plots with $\lambda = 0.9995$ are shown in Figure 4.8. 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 despite 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 performance comparisons for the one-hop and two hop wireless cases, respectively. We see that performance degrades in all cases, but the degradation of Etherware is lesser than that of the CORBA implementations.

The principal reason for the relatively better performance of Etherware in these experiments is the additional option of using UDP based communication to trade-off reliability for lower delays. In particular, this option is not available in the other middleware as the CORBA specification specifically mandates the use of TCP due to RPC based communication semantics. As discussed in Section 3.1, and demonstrated in these experiments, the ability to trade-off reliability for lower delays is very important for networked control applications. It must be noted, however, that option does not significantly affect single processor systems and wired networks as seen in Figure 4.5 and Table 4.1, and more importantly, as established by the success of Real-Time CORBA for various control applications [22].

The main point is that networked control applications need the ability to trade-off reliability for lower delay in communication over networks, and the Etherware support for this trade-off is crucial for such applications.

Chapter 5

ETHERWARE BASED APPLICATION DESIGN

Etherware is a message oriented component middleware for networked control systems. The primitives in the Etherware programming model have been designed specifically for control applications, and its architectural trade-offs have been engineered for such systems. Consequently, systems such as the traffic control testbed described in Section 2.1 can be architected and implemented using Etherware in a straightforward fashion. To emphasize this, and illustrate application development using the associated middleware framework, an Etherware based design and re-implementation of the traffic control testbed is presented in this chapter.

Etherware has been designed as domainware for networked control. Several forcing functions of this domain have been exploited in its design as described in Section 3.1. However, application design is still constrained by these forcing functions, and in particular, Etherware based applications must have properties such as fault tolerance and local temporal autonomy for effective operation. In practice, these constraints can usually be addressed by using appropriate design patterns for control applications. In this chapter, we describe two such design patterns - state estimation and receding horizon control - that have been used in the Etherware based implementation of the traffic control testbed [64]. We also present a detailed analysis of the real-time performance of their implementation, and validate this with an experimental case study in the testbed [65].

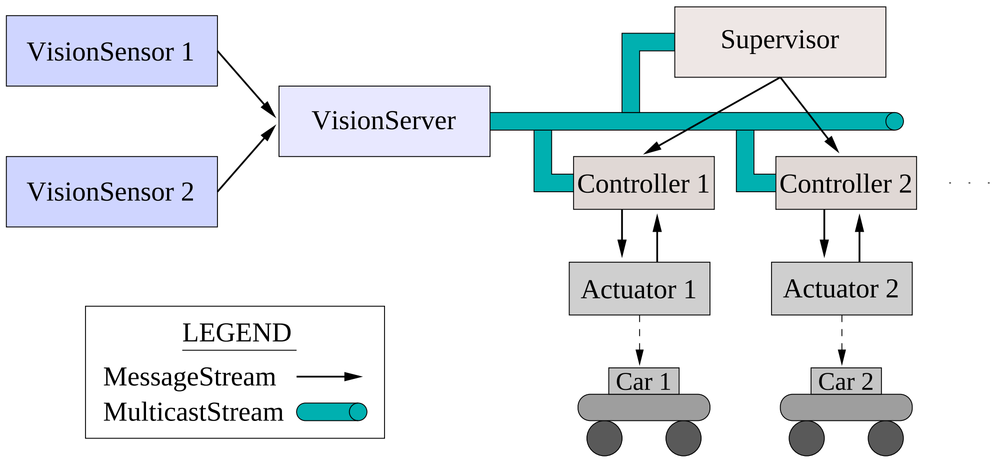


Figure 5.1: Software architecture of the Etherware based testbed implementation

5.1 Testbed redesign

The first implementation of the traffic control testbed, described in Section 2.2, was based on operating system level primitives such as processes, threads, and sockets [54]. This endeavor served as a testing ground for trying out various programming models, and choosing primitives that were most suitable for such systems. In particular, the various components and sub-systems had to be designed from scratch, and their interactions had to be carefully engineered for the desired performance. Hence, this served as the basis for the development of both Etherware itself, as well as an Etherware based implementation of the testbed.

The software architecture of the redesigned testbed is shown in Figure 5.1. Since the various entities in the exploratory implementation of Figure 2.2 had well-defined functionalities and interfaces, they were directly implemented as Etherware components. Although the image processing algorithm and the actuation interface were still implemented in C++ for library support, they were also encapsulated in Etherware based components for system integration. In addition, all the components were implemented passively, and the notification service described in Section 3.3.2 was used for periodic activation.

The data streams from the VisionSensors to the VisionServer were implemented as MessageStreams as shown in Figure 5.1. In particular, this allowed the ability to trade-off reliability for low-delay. Similarly, the control streams from the Supervisor to the Controllers were also implemented as MessageStreams. Further, the interactions between the Controller and the Actuator were implemented as two different MessageStreams: a control stream from the Controller to the Actuator, and a feedback stream conversely. Finally, the feedback stream from the VisionSensor to the Supervisor and the Controllers was implemented as a MulticastStream.

The above description illustrates how the design of the testbed was directly implemented using the primitives of the Etherware programming model presented in Section 3.2. In addition, this design can also be evolved quite naturally using MessageFilters and component Profiles as demonstrated in Chapter 6.

5.2 Robustness in control systems

The ability to tolerate faults is the basis for safe and reliable operation of control systems, and robustness is the basic system property that ensures this. In general, system faults are application specific, and the corresponding design considerations for robustness have to be suitably tailored. However, many of these faults are common to most applications, and the corresponding solutions can be reused effectively. In this chapter, we address two such common kinds of faults: communication faults and software failures, and begin with a brief overview of related work in this section.

Communication faults primarily manifest as delays and losses of messages between software components. Consequently, the problem of tolerating delays and errors, particularly in sensory feedback, has been addressed quite extensively. For instance, Kalman Filters [6] are widely used as state estimators to overcome noisy feedback. In addition, the effect of delays on the operation and stability of a controller has been studied [66], and the use of a state estimator to stabilize controllers in the presence of random delays has also been analyzed [67].

Software failures, on the other hand, can have a variety of symptoms and effects - from erroneous software computations to component failures. Simplex [68] is a popular architecture that uses analytical redundancy to provide robustness against such failures in controllers. In Simplex, two controllers are employed: a simple robust controller, and a complex and possibly defective controller. The complex controller usually has better performance and operates the system most of the time. However, a supervisor constantly monitors the state of the system and can quickly switch to the simple controller if the system approaches instability. This is a simple and uniform approach to address software failures in the complex controller. However, Simplex may not always be applicable as some control systems may not have such a back-up controller. Also, Simplex does not address communication faults, and the problem of replica determinism in real-time systems is quite challenging [69].

Traditional fault-tolerance mechanisms in distributed control systems have relied primarily on redundancy and replication of components [70] [71]. While replication is an effective technique against hardware failures and transient errors, it is usually quite resource intensive, and in particular, does not address software errors that would cause all replicas to fail on identical inputs [72]. Proactive recovery [73] is another technique used to pro-actively prevent errors such as failures due to resource exhaustion. However, it is difficult to predict such errors related to transients or software bugs. Consequently, restart based recovery may perhaps be the only viable solution in such instances. Indeed, popular techniques such as software rejuvenation [74] and recovery-oriented computing [75] [76] are based on restarts of individual components for system recovery.

The importance of low-cost fault-tolerance techniques without the use of redundant controllers has been considered before [77] [78]. However, fault tolerance in distributed control systems needs to be addressed at a system-wide level instead of just at the level of an individual controller [79] . In the following, we adopt this approach to fault-tolerance, where we guarantee correct system operation, even in the presence of communication delays and restarts of individual components.

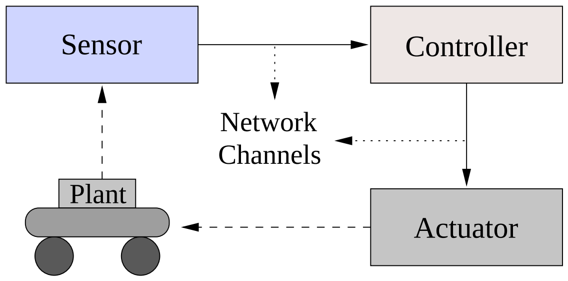


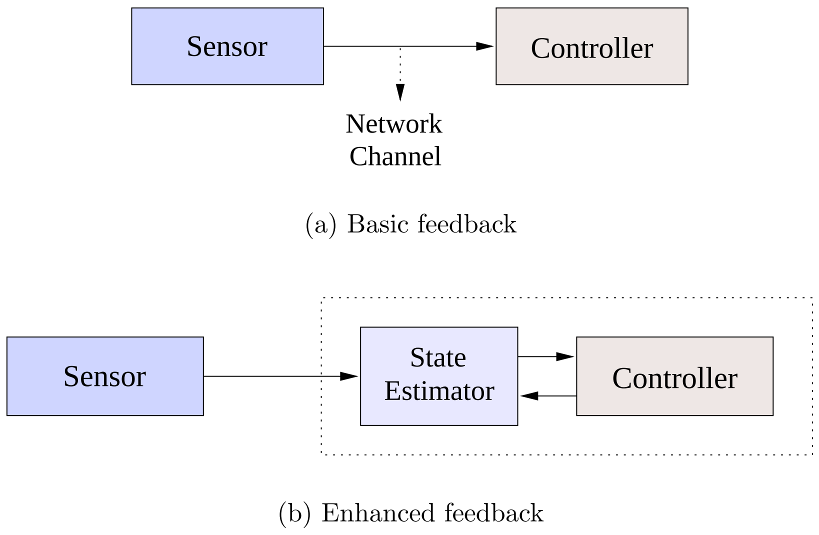
Figure 5.2: A simple control loop

5.3 Design patterns for robustness

Control systems are characterized by components operating in control loops. In networked control systems, such loops may involve communication over network links as well. For instance, a simple control loop is shown in Figure 5.2, where feedback from the sensor to the controller and controls from the controller to the actuator are transmitted over network channels.

Networked control loops involve many failures that need to be tolerated for effective operation. For instance, communication channels in best effort networks, and particularly in wireless networks, are prone to delays and packet losses. Also, the failure of a remote component, such as the controller in the above example, could potentially disrupt the operation of the entire system. Even in the presence of efficient fault tolerance and restart mechanisms, the involved transients could still lead to system instabilities.

Local temporal autonomy is a key property that can be used to address most of the above problems. It is the ability of components to tolerate disruptions in other components for a certain amount of time, and with graceful degradation. This not only shields components from the faults listed above, but also provides precious additional time to recover from such faults as well. In this section, we present two design patterns that use this property to address faults in each of the communication links in Figure 5.2.



(a) Basic feedback

(b) Enhanced feedback

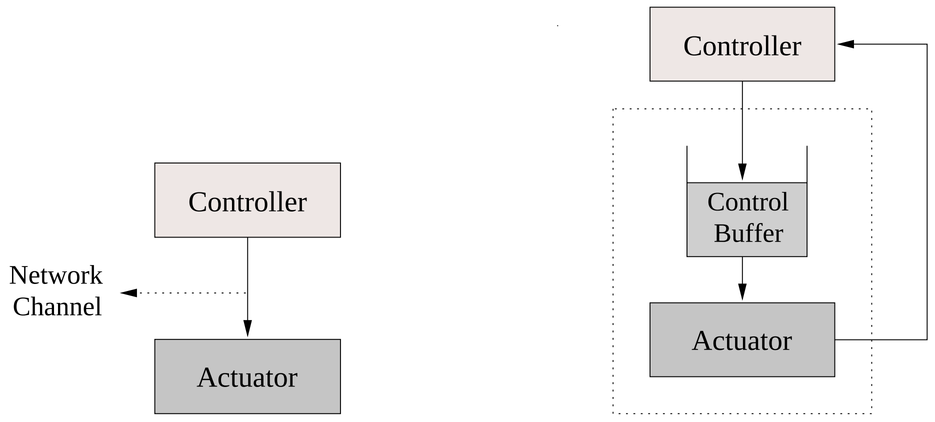
Figure 5.3: State estimation design pattern

5.3.1 State estimation

Sensory feedback from the sensor to the controller is the basis of feedback control. In digital control design, such feedback is expected to be periodic with hard real-time guarantees. However, such guarantees cannot be provided over best effort and wireless network channels. Hence, to apply digital control theory in the design of networked control systems, this problem has to be addressed effectively.

State estimation [59] is a widely used technique in control systems to overcome noise in sensory feedback. For this, a state estimator maintains a model of the plant, which it then uses to predict plant behavior based on received feedback and applied controls. Due to modeling errors, the error in state estimates grow with time. However, regular feedback improves these estimates and keeps the error bounded. In particular, the state estimator can take in feedback with delays and jitter, and still provide reasonably accurate and periodic state estimates.

The essence of the *state estimation* design pattern is to use a state estimator at the controller to tolerate faults in sensory feedback as shown in Figure 5.3. Since the state estimator can provide reasonably accurate estimates without feedback for a limited duration,



(a) Basic control (b) Enhanced control

Figure 5.4: Receding horizon control design pattern

this pattern increases the local temporal autonomy of the controller. In addition, the state estimator also provides periodic estimates to the controller enabling the application of digital control theory in networked control systems.

5.3.2 Receding horizon control

Traditional digital controllers operate periodically, computing one set of controls for every period. These controls are then sent to the actuator, which effects them in the plant. This simple design works well in practice when the controller and the actuator execute on the same computer. However, when they communicate over a network link, then the above design is vulnerable to the attendant faults of networked operation. In particular, when periodic controls do not arrive from the controller due to software or communication failures, then the actuator has no controls to effect. In such cases, actuators may have default fail-safe controls that maintain system safety. While this approach may work with occasional failures, it is certainly quite sub-optimal in most networked control systems.

Under the *receding horizon control* design pattern, the controller computes a sequence of future controls during every period. These controls are then stored in a *control buffer*

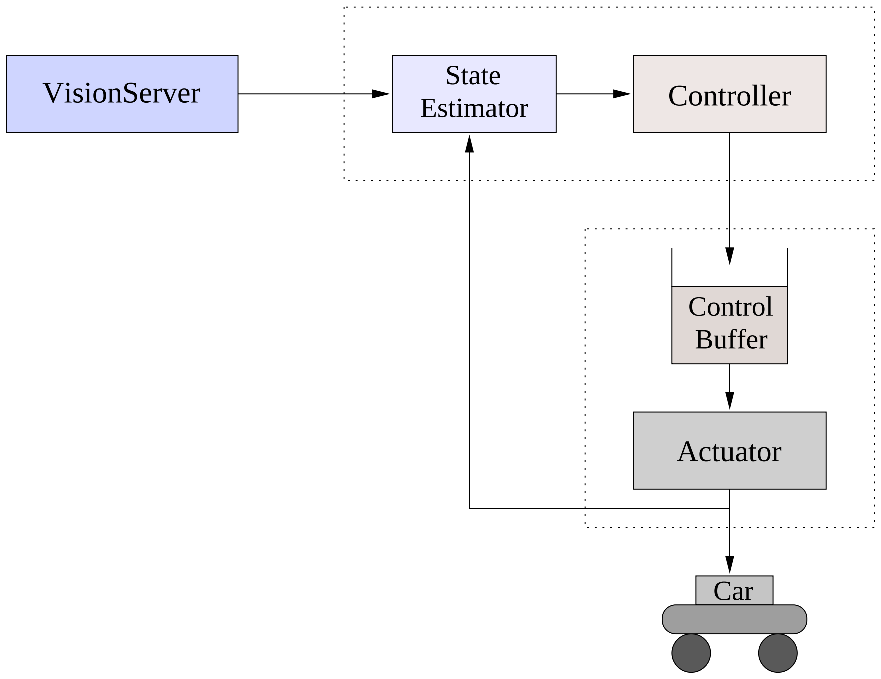


Figure 5.5: Design enhancements in the lower level control loop of the testbed

at the actuator as shown in Figure 5.4(b). Hence, if a subsequent control update from the controller is delayed or lost, then the actuator can use the pre-computed controls from the control buffer. Consequently, this improves the local temporal autonomy of the actuator, and promotes graceful degradation in the presence of controller failures.

Two important points regarding receding horizon control must be noted. First, the controller needs a model to estimate future plant behavior so that it can compute a sequence of controls. One interesting option is to use the state estimator itself as a state predictor since the estimator already has a model of the plant. Alternatively, control laws such as model predictive control [27], which automatically compute sequences of future controls, may also be employed. Second, since the actual controls effected by the actuator are not known in advance, this information must be fed back to the controller and its state estimator as shown in Figure 5.4(b).

5.3.3 Testbed design enhancements

The state estimation and receding horizon control design patterns have been applied in both the control loops of the testbed shown in Figure 5.1. In particular, the design enhancements to the lower level control loop are shown in Figure 5.5. The enhanced Controller uses a state estimator to filter the sensory feedback from the VisionServer. Also, the Controller computes and sends a sequence of future controls to the Actuator, which then stores them in a control buffer according to the receding horizon control design pattern. The Controller in the testbed uses model predictive control [27] as noted in Section 2.2. Hence, it automatically computes a sequence of future controls that are then sent to the Actuator. In particular, the state estimator and the Controller use the same plant model.

In the rest of the chapter, we analyze and experimentally validate the above design enhancements in the lower level control loop of the testbed.

5.4 Analysis of testbed enhancements

The design enhancements to the lower level control loop in the testbed, shown in Figure 5.5, improve system robustness by increasing the local temporal autonomy of components. In effect, the deadlines for sensory feedback and control updates are extended, and the overall system has graceful degradation in the presence of communication and software failures. We now analyze these deadline extensions, and in particular, characterize the worst-case and average-case degradation in system performance during failures.

The following factors determine the extension of deadlines for the controller in the enhanced control loop:

1. System tolerance and operational error bounds for the plant.

2. Growth of error in plant state predictions.

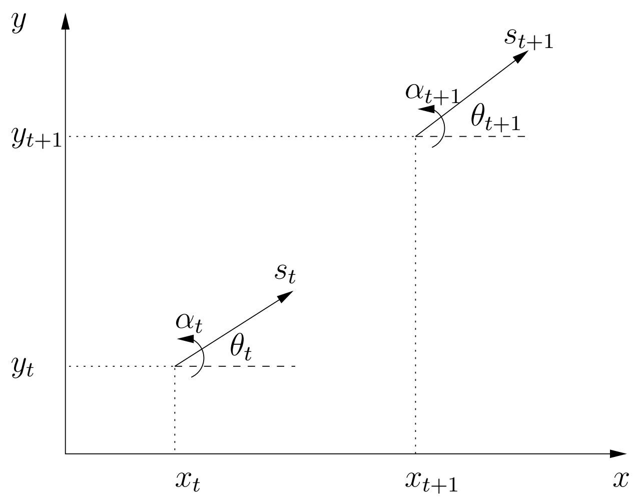


Figure 5.6: State evolution in the car model

3. Computational resources available at the controller to compute sequences of future controls during each period.

4. Size of the control buffer at the actuator to store the future controls.

5. Communication bandwidth between the controller and the actuator to send a sequence of future controls during each period.

The last three factors depend on system deployment constraints and can be engineered as necessary. However, the growth of prediction error is determined by the plant model, and the future control horizon is essentially the interval up to which this error is within system tolerance. Consequently, in the rest of the chapter, we focus primarily on the second factor, and analyze how deadlines have been extended in the testbed due to this. Finally, since both state estimation and receding horizon control use the same plant model in the testbed, we note that the following analysis applies to deadline extensions for sensory feedback as well.

5.4.1 Car Model

In the traffic control testbed, we model a car by considering its position and orientation on the track. More specifically, the state of a car $\mathbf{x}_t$ at time t is given by $\mathbf{x}_t = [x_t \; y_t \; \theta_t]^T$, where

x_t and y_t are the coordinates of the center of the car, and θ_t is its orientation. Further, the car has a speed s_t and steering α_t at time t.

The evolution of car state $\mathbf{x}_t$ in one time step is shown in Figure 5.6. The orientation of the car after one time step is given by

$$\theta_{t+1} = \theta_t + h * \alpha_t + w_{\theta t}$$

where h is the length of a time-step, and $w_{\theta t}$ is the update error due to noise. During this interval, the car is moving at speed s_t. Further, we assume that h is small enough so that the car moves at an average orientation of $\vartheta_t = \theta_t + (\alpha_t/2)$ during the interval. Consequently, the car has a displacement of $h * s_t$ at an orientation of ϑ_t. These relations are summarized in the following equation.

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & hs_t \cos \vartheta_t \\ 0 & 1 & 0 & hs_t \sin \vartheta_t \\ 0 & 0 & 1 & h\alpha_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \\ \theta_t \\ 1 \end{bmatrix} + \begin{bmatrix} w_{xt} \\ w_{yt} \\ w_{\theta t} \\ 0 \end{bmatrix} \tag{5.1}$$

Equation (5.1) can be written more compactly using vectors as follows:

$$\mathbf{x}_{t+1} = \mathbf{M}_t \mathbf{x}_t + \mathbf{w}_t \tag{5.2}$$

where $\mathbf{x}_t$ is the *state* of the car, $\mathbf{M}_t$ is the *car model*, and $\mathbf{w}_t$ is the *update error*, all at time t. Also, the state has been extended to $\mathbf{x}_t = [x_t \; y_t \; \theta_t \; 1]^T$ to account for state evolution.

5.4.2 Worst-case error bound

During normal operation, the state estimator maintains $\hat{\mathbf{x}}_{\mathbf{t}}$, an estimate of the state $\mathbf{x}_t$, which evolves according to the equations

$$\begin{aligned} \hat{\mathbf{x}}_{t+1}^{(-)} &= \mathbf{M}_t \hat{\mathbf{x}}_t & (5.3) \\ \hat{\mathbf{x}}_{t+1} &= f(\hat{\mathbf{x}}_{t+1}^{(-)}, \mathbf{y}_{t+1}) & (5.4) \end{aligned}$$

where $\mathbf{y}_{t+1}$ is the observation (sensor feedback) at time $t+1$ and is used to correct the estimate using the correction f in (5.4). This is a common approach to account for the error term $\mathbf{w}_t$ in (5.2) [80].

During state prediction, however, there are no observations, and $\hat{\mathbf{x}}_{\mathbf{t}}$ evolves according to

$$\hat{\mathbf{x}}_{t+1} = \mathbf{M}_t \hat{\mathbf{x}}_t \tag{5.5}$$

Since the error term $\mathbf{w}_t$ in (5.2) cannot be accounted for in the prediction update of (5.5), the *prediction error* $\tilde{\mathbf{x}}_t = \mathbf{x}_t - \hat{\mathbf{x}}_t$ grows with time. In particular, if $\tilde{\mathbf{x}}_0 = 0$, and the error term can be bounded as $\mathbf{w}_t \leq \mathbf{w}_{max}$, then the prediction error grows as

$$\tilde{\mathbf{x}}_{t+1} \leq \mathbf{w}_{max} t \tag{5.6}$$

due to the special structures of M_t (upper triangular form) and w_t (last component is zero) shown in (5.1). This is equivalent to specifying uncertainty in the prediction of the car position by a bounding box whose area grows linearly with time. Consequently, if the acceptable uncertainty is $\tilde{\mathbf{x}}_{max}$, then the prediction is acceptable for all time t with

$$\tilde{\mathbf{x}}_{t+1} \leq \mathbf{w}_{max} t \leq \tilde{\mathbf{x}}_{max} \tag{5.7}$$

Hence, the deterministic (worst case) upper bound on the prediction error grows linearly with time. Further, since prediction is equivalent to estimation without observations, the same analysis applies for computing deadlines for the sensor as well. In particular, the maximum value of t satisfying (5.7) is the *hard real-time deadline* for sensory feedback.

5.4.3 Mean error bound

The deterministic analysis of Section 5.4.2 can be used to compute hard-real time deadlines using (5.7). In particular, this bound uses the maximum error $\tilde{\mathbf{w}}_{max}$, and hence the prediction error bound grows linearly with time. However, if the error $\tilde{\mathbf{w}}_t$ is "well behaved", and we can tolerate occasional failures, then we can obtain significantly higher deadline extensions.

In equation (5.2), the exact values of $\mathbf{w}_t$ cannot usually be determined in practice, and only the probability distributions are known. Hence, $\mathbf{x}_t$ and $\mathbf{w}_t$ are modeled as random variables in the car model as well as in the updates of (5.3), (5.4), and (5.5). Further, we may reasonably assume that the error terms $\mathbf{w}_t$ are independent and identically distributed (i.i.d) with mean zero. We can then use (5.2) and (5.5) to compute the mean squared error as follows:

$$\begin{aligned} E[\tilde{\mathbf{x}}_{t+1}^T\tilde{\mathbf{x}}_{t+1}] &= E[(\tilde{\mathbf{x}}_t^T\mathbf{M}_t^T + \mathbf{w}_t^T)(\mathbf{M}_t\tilde{\mathbf{x}}_t + \mathbf{w}_t)] \\ &= E[\tilde{\mathbf{x}}_t^T\mathbf{M}_t^T\mathbf{M}_t\tilde{\mathbf{x}}_t] + E[\mathbf{w}_t^T\mathbf{w}_t] + E[\tilde{\mathbf{x}}_t^T\mathbf{M}_t^T]E[\mathbf{w}_t] + E[\mathbf{w}_t^T]E[\mathbf{M}_t\tilde{\mathbf{x}}_t] \\ &= E[\tilde{\mathbf{x}}_t^T\mathbf{M}_t^T\mathbf{M}_t\tilde{\mathbf{x}}_t] + E[\mathbf{w}_t^T\mathbf{w}_t] \qquad (5.8) \end{aligned}$$

The expectations factor out in the first step since $\tilde{\mathbf{x}}_t$ is independent of $\mathbf{w}_t$, and the last two terms disappear in the second step as $\mathbf{w}_t$ has mean zero.

Equation (5.8) is a recursion in $\tilde{\mathbf{x}}_t$. Assuming $\tilde{\mathbf{x}}_0 = 0$, we can solve the above recursion to get $\tilde{\mathbf{x}}_{t+1}$ in terms of the error terms $\mathbf{w}_j$ as

$$E[\tilde{\mathbf{x}}_{t+1}^T\tilde{\mathbf{x}}_{t+1}] = E[\mathbf{w}_t^T\mathbf{w}_t] + \sum_{j=1}^{t-1} E[\tilde{\mathbf{w}}_j^T(\prod_{i=j+1}^{t}\mathbf{M}_i^T)(\prod_{i=0}^{t-(j+1)}\mathbf{M}_{t-i})\tilde{\mathbf{w}}_j] \tag{5.9}$$

Due to the special structures of M_t (upper triangular form) and w_t (last component is zero) shown in (5.1), each term in the above summation reduces to $E[\mathbf{w}_j^T\mathbf{w}_j]$. Further, since the $\mathbf{w}_j$ were assumed to be i.i.d, we have $E[\mathbf{w}^T\mathbf{w}] = E[\mathbf{w}_j^T\mathbf{w}_j]$ for all j. Consequently, (5.9) simplifies to give

$$E[\tilde{\mathbf{x}}_{t+1}^T\tilde{\mathbf{x}}_{t+1}] = E[\mathbf{w}^T\mathbf{w}] * t \tag{5.10}$$

We can now conclude that the *mean prediction error* ϵ grows as the square root of time t, i.e.,

$$\epsilon = \sqrt{E[\tilde{\mathbf{x}}_{t+1}^T\tilde{\mathbf{x}}_{t+1}]} = k\sqrt{t} \tag{5.11}$$

for constant k. In practice, this gives much higher deadline extensions than (5.6) as we demonstrate in Section 5.5.

The key result in this section is summarized in the theorem below.

Theorem 5.4.1 *For the car model given by (5.2), the worst-care error grows linearly with time. Further, if error terms $\mathbf{w}_t$ are independent and identically distributed (i.i.d) with mean zero, then the mean error grows as the square root of time.*

Hence, to operate the system with the hard guarantee that a car will stay within the system tolerance, the deadline is the largest t for which (5.7) is satisfied. However, if occasional failures can be tolerated, then using (5.11) gives much larger deadline extensions in practice.

5.5 Experimental validation

This section presents a case-study to validate the conclusions of Section 5.4.

5.5.1 Case study: Motorcade

Through our experiments we intend to answer the following questions:

1. Does the empirical growth in state prediction error correspond to (5.11) in Section 5.4?

2. What is the deadline extension achieved?

3. Is our new design indeed tolerant to delayed restarts of sensors and controllers?

To answer these questions, we consider a motorcade scenario with two cars: a leader and a follower. The cars move around in an elliptical trajectory with a major axis of length 2.8m and a minor axis of length 2m. The cars themselves are about 225mm long, travel at an average speed of 371mm/s, and take about 21.7 seconds for one iteration of the trajectory. An elliptical trajectory was chosen since it exercises different steering angles at different points.

The main goal is to maintain a separation of about 400mm between the centers of the cars, i.e., about 175mm between their bumpers. Hence, the deviation of the leader car from its trajectory is constrained to be less than half the separation between the cars in order to avoid collision. To ensure this, we set the maximum allowable deviation for the leader car to be 50mm.

Error in state prediction

In the first experiment, we seek to answer the first two questions posed in Section 5.5.1 by measuring the growth of error in state prediction as a function of time. For this, the leading car is driven along the elliptical trajectory, and a separate state estimator for the car is executed in parallel. At a designated point, the feedback to the state estimator is turned off, after which the estimator essentially operates as a state predictor. However, the actual car is still operated by a controller with complete feedback. Hence, the distance between the actual position of the car and the predictions of the state estimator is the growth of error

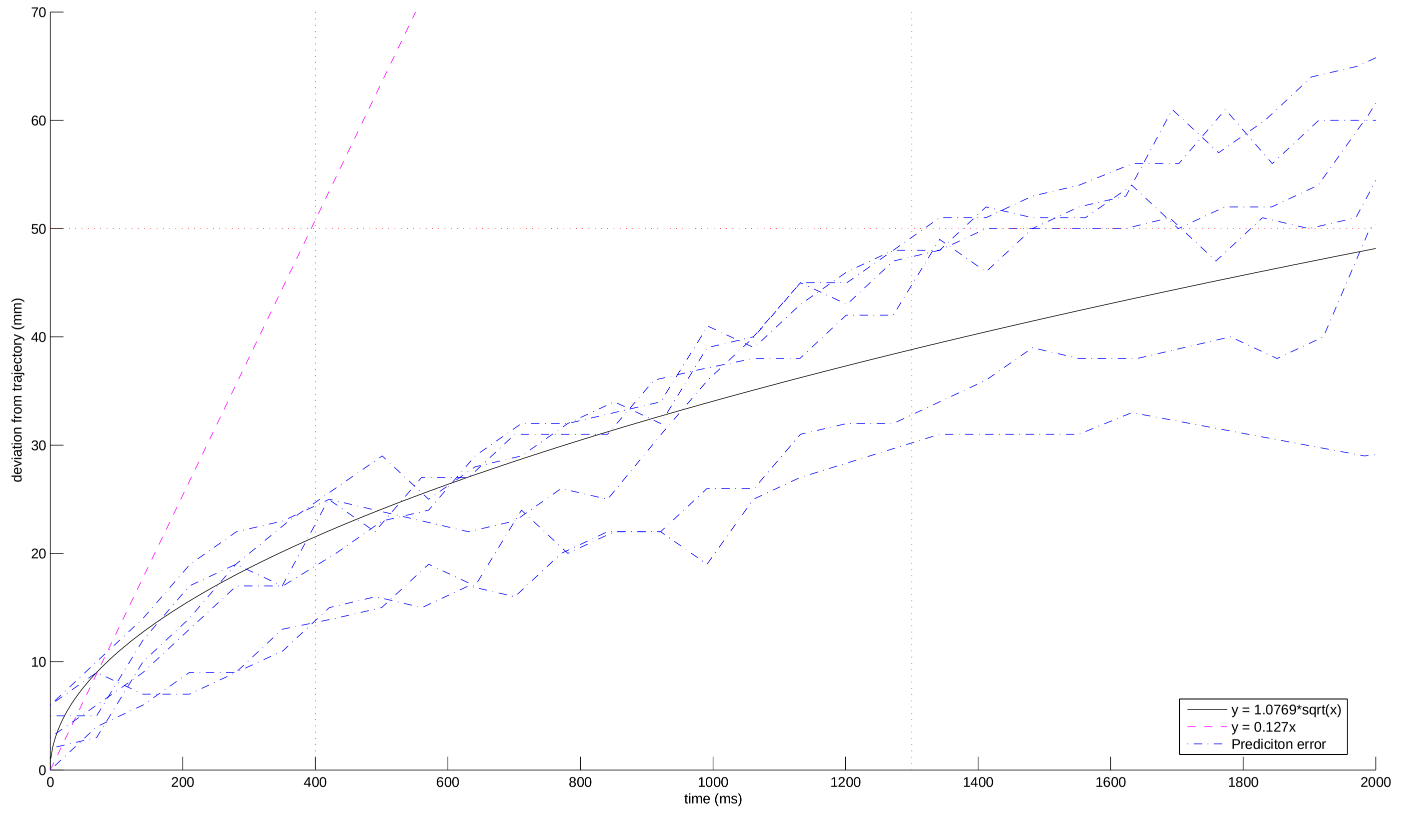


Figure 5.7: Growth of error in state prediction

in state prediction. Figure 5.7 plots the prediction error for six different points along the elliptical trajectory of the motorcade.

As shown in Figure 5.7, the prediction error grows as the square root of time. In fact, the figure also shows the solid curve $y = k\sqrt{x}$, where $k = 1.0769$ is the value for which this function has the least mean squared error from the prediction error curves. This is in accordance with the average case analysis of Section 5.4.3 and validates (5.11). Also, the maximum difference between calibrated speeds is 127mm/s. Assuming this to be the worst case error in (5.7) for the duration of two seconds, Figure 5.7 also shows the growth of the worse case error bound with time.

To determine the deadline extensions, we observe when the various error curves hit the tolerable error bound of 50mm in Figure 5.7. The original deadline for the sensor and the controller was 100ms - the operating period of the lower level control loop. However, the design enhancements presented in this chapter extend even the worst case deadline to 400ms. If occasional failures can be tolerated, then the average error curves show that this deadline can be further extended to 1300ms. In particular, these extended deadlines are more than sufficient to tolerate most delays in the network.

Effect of delayed restarts

The second experiment addresses the third question raised in Section 5.5.1. In this experiment, both cars in the motorcade have trajectories that make them go along an ellipse with a separation of about 400mm between their centers. Faults are then injected into the controller (cf. Figure 5.5) of the leader at random points in the trajectory. The subsequent restart of the controller is also delayed for random intervals to observe the behavior of the car based on future controls stored in the control buffer at the actuator. Note that restarting the vision sensor would have a similar effect on the behavior of the car, since in either case the controls are being computed by using the same plant model.

The distance between the leader and the follower is shown in Figure 5.8, and the cor-

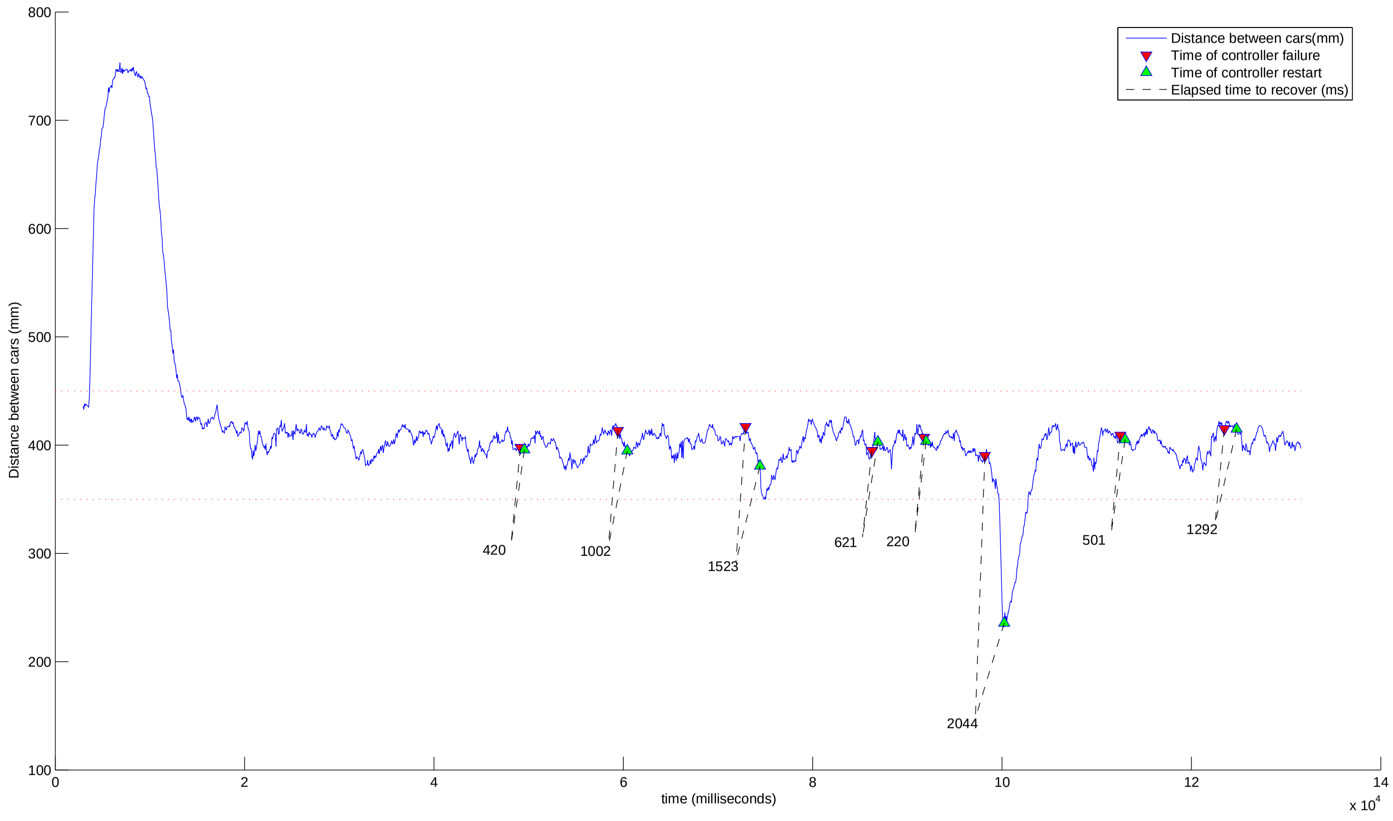


Figure 5.8: Distance between centers of cars in the motorcade

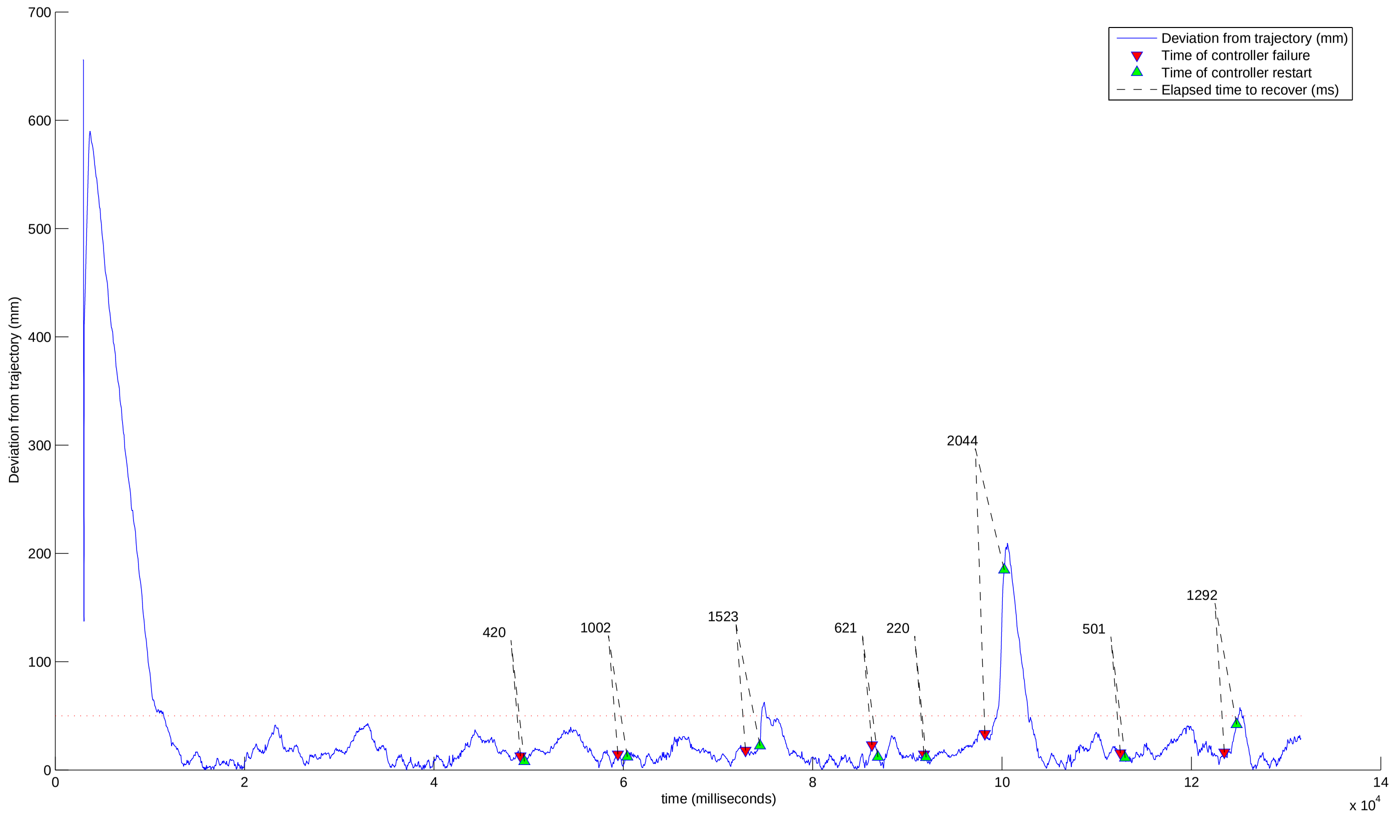


Figure 5.9: Deviation of leader from its trajectory

responding deviation of the leader from its trajectory is plotted in Figure 5.9. The initial increase in the distance between the two cars is due to the cars catching up with their trajectories during start-up. This transient is resolved in about 10s, and the cars stay quite close to their trajectories for the rest of the experiment.

The effect of restarting the controller of the leading car is clearly reflected in both figures. In particular, for restart delays of less than 1.3 seconds, the distance between the cars does not vary by more than 50mm. This is consistent with the behavior observed in Figure 5.7. However, when the restarts are delayed for longer intervals, we see that the distance reduces as the follower comes closer to the leader. In particular, for the two second restart at about 100s into the experiment, the future controls in the control buffer of the actuator are exhausted and the car stops. Consequently, we see that the cars collide as expected.

We conclude that the experimental results are in conformity with the conclusions of Section 5.4, and more importantly, the design patterns presented in this chapter effectively address the forcing functions of the networked control domain.

Chapter 6

SAFETY AND SECURITY CONSIDERATIONS

Control systems interact with the real world, and hence, safety is a central concern in their design and operation. Indeed, safety considerations influence many of the engineering decisions and trade-offs in systems design. For instance, multimedia applications are typically executed as soft real-time tasks, while mission critical systems generally require hard-real time guarantees and over-provisioned or dedicated resources. Also, appropriate safety measures can usually be incorporated at different levels in application design as well. For example, in the traffic control testbed of Figure 5.1, a default fail-safe control such as a stop command is an effective safety measure in the Actuator, while gridlock free scheduling of cars by the Supervisor addresses system-wide safety concerns. In general, such safety features can and ought to be incorporated into components and sub-systems as an integral part of control system design.

While mechanisms such as collision avoidance can be employed during normal operation, it may not be useful, or even practical, to enforce some safety measures all the time. For instance, if there are not too many cars in the traffic control testbed, then the probability of gridlocks occurring is quite small. In this case, low overhead gridlock detection can be used during normal operation, so that higher overhead resolution algorithms are employed only when a gridlock is detected. Such an approach is quite valuable as necessary safety can be enforced with low operating overhead and much better scalability. In this chapter, we present an Etherware based Control System Incident Response (CSIR) framework that allows such mechanisms to be implemented as Strategies for Incident Response (SIRs) in control systems.

Many security problems can also be addressed in a similar fashion; low overhead detection mechanisms can be used during normal operation, and more expensive responses can be triggered only when a security failure is detected. However, care must be taken while enforcing security responses in control systems as the associated override mechanisms must respect underlying plant dynamics. In particular, some of the underlying safety measures may still need to be preserved during security overrides.

We capture the above insight by introducing the principle of *safety preserving security overrides* in the following, and establish the importance of this principle through a detailed case-study in the testbed. In the process, we also describe safety and security mechanisms in the testbed, and illustrate the ease with which these can be integrated into our Etherware based testbed implementation.

6.1 Control System Incident Response

Complex control systems usually have imprecise models of their plants. Consequently, during normal operation, such systems may occasionally get into states that violate the operational constraints of control algorithms in the system. For instance, in a distributed traffic control system, a small segment of the road network may become congested or blocked due to an accident. Such situations are usually exceptional in that they are expected to occur quite infrequently, but additional strategies would be required to bring the system back to normal operation. Such an exceptional situation is called an *incident*, and an associated response strategy is called a *Strategy for Incident Response (SIR).*

Control System Incident Response (CSIR) is a framework that allows strategies addressing incidents in control systems to be correctly and systematically incorporated into the system architecture. In particular, the CSIR framework operates in conjunction with the regular control system, and enables the addition of SIRs with minimal impact to the rest of the system. For example, using a CSIR framework, an SIR to appropriately manage congested traffic could be added to respond to accidents in traffic control systems.

6.1.1 Rationale

A central concern in the CSIR based approach is the following: if incidents in control systems can indeed be modeled, then why should SIRs not be part of the normal algorithms in control systems. For instance, in a traffic control system, why wouldn't managing traffic around accidents be an active function of a high-level supervisor? To address this concern, we provide the following justifications for CSIR:

- *System tractability:* Plants are usually modeled at different levels of detail in different layers of a control hierarchy. This allows tractable control algorithms to be developed in the higher control layers, rather than using detailed models in these layers, which would make their design unnecessarily complex, or even intractable. However, most incidents are usually detected in lower level detailed models, but require higher level response. Hence, an SIR would have to be used to recover from such incidents.

  We illustrate the above argument using the traffic control testbed in Figure 5.1. The high-level traffic Supervisor models the traffic network at the granularity of roads, while the low-level Controllers monitor actual states of their associated cars. This allows tractable control algorithms to be developed to manage city traffic as well as to control individual cars. However, the Supervisor cannot completely avoid congestion due to accidents since it does not directly track or control individual cars. Also, a congested region cannot be controlled using "normal" traffic rules, as controlling all the cars through a high-level Supervisor would result in a very centralized system. In particular, such a design is not desirable due to poor scalability and very high communication overheads. Hence, an SIR would be necessary to address such a situation.

- *Performance trade-off:* Although control algorithms based on detailed models may be tractable in some cases, the amount of information to be processed, and the detail of controls to be specified, could still require the system to be operated in a fairly sub-optimal regime. This is usually not desirable as the cost of such performance degradation is usually much greater than that associated with a CSIR framework.

For instance, in the traffic control testbed, it is possible for the Supervisor to track and control individual cars if they moved very slowly. However, such poor performance is not acceptable during normal operation. On the other hand, this could be the only recourse in the case of an accident. Hence, it would be sensible to employ an SIR involving detailed supervisory control only during accidents.

- *Modeling limitations:* Physical models of the actual plant under control, are imprecise. Consequently, there is always the potential for incidents due to modeling inaccuracies. Hence, a CSIR framework would be necessary even if the above considerations are inapplicable to a system. Since it is impossible to predict or avoid all accidents in traffic control systems, SIRs are therefore necessary to respond to such incidents.

- *Goal changes:* Some incidents can potentially change system objectives so that important though temporary goals may emerge as a result. In such situations, while it may be impractical to change the entire system to respond to such a transient goal, using a CSIR framework would still allow an appropriate SIR to be employed. For example, giving higher priority to an ambulance would be the most important goal after a traffic accident. However, once the accident has been appropriately responded to, the system can return to normal operation where all cars are scheduled with equal priority. A CSIR framework would allow such an incident to be addressed using an appropriate SIR.

6.1.2 Strategies for Incident Response

A *Strategy for Incident Response (SIR)* is the sequence of activities undertaken in response to an incident in a control system. While the particular set of undertaken activities depends on specific incidents, the activities can still be categorized based on the underlying strategy. For instance, in a Detect-Investigate-Respond (DIR) strategy, the following types of activities are involved:

- *Detection activities:* These are used to monitor the plant state for symptoms of incidents. Since these are continuously performed in the system, they usually have very low impact and overhead. Typical detection activities include monitoring data flows to detect outliers, and operating special sensors to detect incidents. For instance, activities such as the detection of road obstructions and accidents directly through special sensors, or indirectly through traffic congestion, belong in this category.

- *Investigation activities:* On detection of incidents, further investigative activities are initiated. These typically involve an initial response, followed by further analysis of the symptoms to suitably classify the incident and determine appropriate recovery procedures. These are higher impact activities that might place the affected part of the system into a safety response mode. For instance, when a road gets congested, the blocked cars have to be moved out of the road, and the road has to be temporarily removed from the traffic grid. Concurrently, the cause of the congestion, which could be a road block, an obstruction, or an accident, should also be investigated so that appropriate recovery activities can be initiated.

- *Recovery activities:* Once an incident has been suitably classified, appropriate recovery activities can then be employed to bring the system back to normal operation. These are potentially very high impact activities that could affect a larger part of the system. For instance, once the cause of traffic congestion in a road has been identified, recovery involves routing appropriate emergency response vehicles with high priority in the system. Also, when the situation has been addressed suitably, normal scheduling of traffic is resumed, and the repaired road is eventually added back to the traffic grid.

In some cases, classification of incidents may not require further investigation, and a Detect-Respond (DR) strategy may be sufficient. Although many other strategies could also be deployed in a CSIR framework, in this chapter, we focus mainly on DR and DIR strategies.

6.1.3 Etherware mechanisms

The Etherware programming model described in Section 3.2 enables the design of fairly complex safety mechanisms. In particular, the following primitives provide specific support for the CSIR framework:

- *MulticastStream:* MulticastStreams are efficient data distribution mechanisms. For instance, the vision based feedback in the traffic control testbed is communicated through a MulticastStream as shown in Figure 5.1. Detection mechanisms in a DIR strategy can also be supported quite efficiently through such a feedback MulticastStream. In particular, detection components can be easily added or removed at runtime without the involvement of, or impact to, other operational components in the system.

- *Filter:* Filters intercept messages in MessageStreams. Hence, detection mechanisms can be implemented as message Filters as well. In particular, they can be added or removed dynamically in an operational system. Also, since Filters actually intercept all messages before they reach the destination, recovery activities that involve turning off, modulating, or even injecting messages into MessageStreams can be implemented as Filter components. In fact, this functionality is the basis of the CSIR framework support in Etherware as it allows flexible and extensible management of SIRs.

As we describe later in the chapter, these primitives have been extensively used in the safety and security mechanisms in the traffic control testbed.

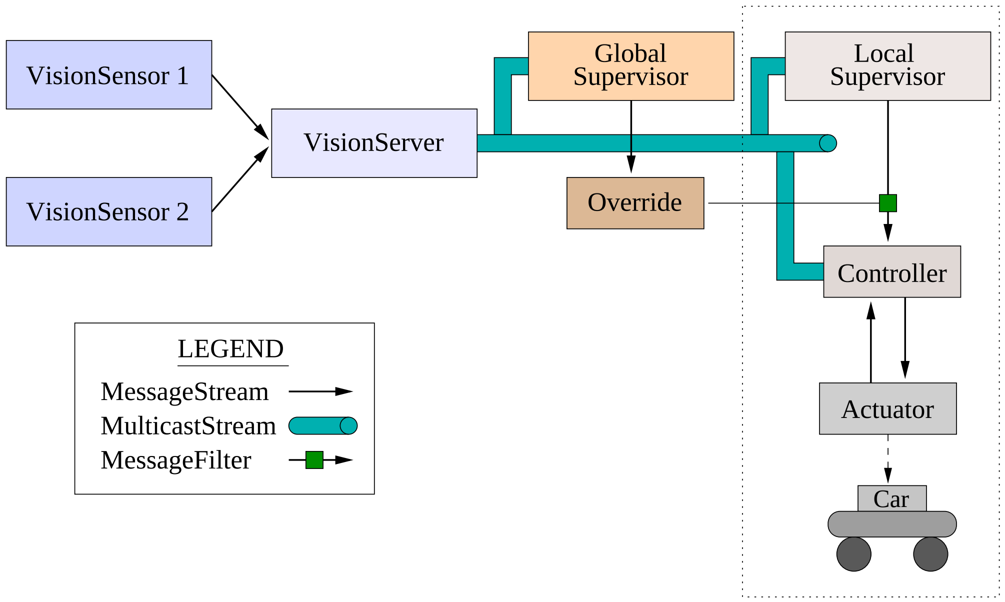


Figure 6.1: Federated software architecture for the traffic control testbed

6.2 Security in control

A centralized control system is relatively easy to secure as all software is executed on a single computer. The security considerations regarding access control can usually be addressed using standard techniques [81], and the desired security mechanisms can be directly enforced on the corresponding computer. On the other hand, networked control systems can have much more complex interactions, particularly between distributed control loops, and hence, standard security techniques may not be directly applicable. In this section, we present the principle of *safety preserving security trade-offs* for networked control systems.

6.2.1 A dichotomy of control

Networked control systems are usually composed of many interacting sub-systems, and the overall system goals can be decomposed into corresponding sub-goals. In particular, such a hierarchy of goals usually imposes different considerations at different layers of the hierarchy. For instance, consider the federated architecture for the traffic control testbed shown in Figure 6.1. In this configuration, each car has a Local Supervisor that supervises the

Controller based on individual goals of the car. However, such distributed operation does not address the global goal of gridlock free operation. Instead, this is addressed by a Global Supervisor that preempts the Local Supervisor when necessary. This modified architecture clearly brings out the different considerations at different layers in a control hierarchy.

The part-whole relationship between a networked control system and its sub-systems illustrated in Figure 6.1 usually leads to the following dichotomy of control:

- *Discretionary control:* Lower level controllers exert discretionary control over their corresponding sub-systems. In Figure 6.1, the Local Supervisor operates the car based on individual goals.

- *Mandatory control:* Higher level controllers exert mandatory control over the sub-systems that they supervise. In Figure 6.1, the Global Supervisor overrides Local Supervisors to avoid gridlocks in the testbed.

This architecture is much more scalable than that in Figure 5.1 since Local Supervisors can efficiently operate in a distributed fashion, while the Global Supervisor can still address gridlocks in the system when necessary.

6.2.2 Safety preserving security overrides

The dichotomy of control illustrated in Figure 6.1 is particularly significant for security considerations. During normal operation, most of the sub-systems are operated under discretionary control for efficiency, and the local components have regular permissions. However, when a security breach occurs, the local permissions need to be reduced, and the system needs to be operated under mandatory control so that the failure can be addressed much more effectively.

This characterization is well represented in regular city traffic. During normal operation, drivers operate their respective cars to accomplish individual goals, and all drivers have

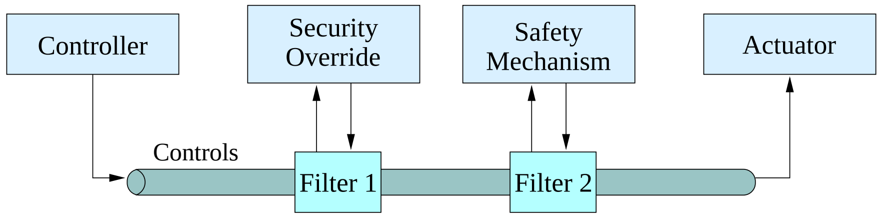


Figure 6.2: Ordering of Filters in a MessageStream

equal permissions at roads and intersections. However, when a security breach occurs due to a rogue car driver, mandatory control is exerted by police cars so that they can effectively bring the situation under control. In particular, during such override, regular cars have reduced permissions and must yield to police cars.

Control systems typically have many low-level fail-safes to provide safety guarantees in the system. Some of these guarantees could be critical to system safety, and hence, the corresponding safety mechanisms must not be preempted by security overrides. For instance, in the scenario discussed above, drivers must still ensure that they yield safely and do not cause accidents. Similarly, even though police cars have higher permissions for a road, they must still wait for other cars to clear out so that they can drive without accidents as well. This important observation is stated in the following principle.

Principle 6.2.1 *The* principle of safety preserving security overrides *states that higher level security overrides must preserve lower level safety features as far as possible.*

In the remainder, we establish the importance of this principle through a detailed case-study in the testbed.

6.2.3 Etherware mechanisms

The goals of security are usually addressed by prevention, detection, and recovery activities [81]. In networked control systems, detection and recovery mechanisms are typically expressible as SIRs. Hence, the Etherware mechanisms described in Section 6.1.3 are applicable to these mechanisms as well. On the other hand, prevention measures are typically part of standard system design and must be considered on an application specific basis.

There is, however, an interesting Etherware feature that can be directly used in applying the principle of safety preserving security overrides. There can be multiple message Filters for an entity in Etherware, and such Filters always have a well defined order. For instance, of the two Filters shown in Figure 6.2, Filter 2 has a higher priority than Filter 1 because all messages forwarded by Filter 1 are in turn intercepted by Filter 2. As the figure demonstrates, this ordering between Filters can be used to implement safety preserving security overrides in a natural fashion.

In general, security overrides are usually safety preserving if they do not preempt the control exerted by safety mechanisms over system actuators.

6.3 Safety in the testbed

Collision avoidance is the main safety issue that we address in this section. The objective is to ensure that cars do not collide while accomplishing individual goals. Although the city traffic Supervisor in Figure 5.1 generates collision free trajectories for cars, some collisions still occur as Controllers cannot always accurately follow their given trajectories. Also, obstacles such as stationary cars are not taken into account by the Supervisor and may cause collisions as well. Further, in other scenarios such as trajectory tracking, Supervisors may not even provide collision free trajectories. Consequently, there is a need for a simple and uniform collision avoidance sub-system to ensure system safety in the testbed.

6.3.1 Architecture

The collision avoidance sub-system was developed in collaboration with Hans-Joerg Schuetz [82]. The software architecture for this sub-system is shown in Figure 6.3. The main components of this sub-system are the CollisionAvoidanceFilter (CA Filter) and the CollisionAvoid-

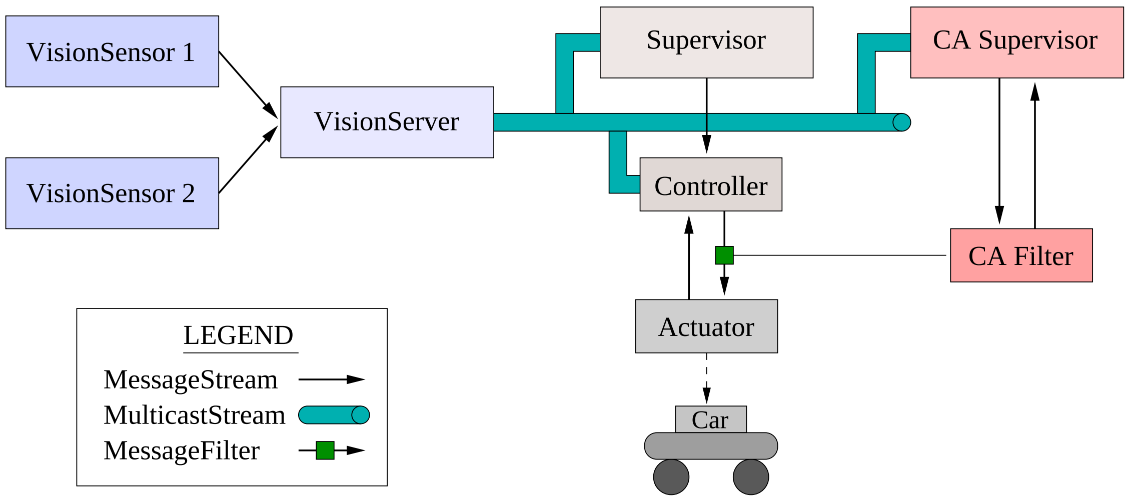


Figure 6.3: Software architecture for collision avoidance in the testbed

anceSupervisor (CA Supervisor). The CA Supervisor is a centralized entity that monitors all cars in the testbed, while there is a CA Filter for each operational car.

The collision avoidance sub-system is activated by adding a CA Filter as a Filter to the control MessageStream from the Controller to the Actuator of each car. Subsequently, the CA Filter intercepts all controls sent by the Controller, and forwards only those controls to the Actuator that ensure collision-free operation of the car. In particular, if a sequence of controls would cause the car to collide with another car, then the CA Filter sends fail-safe stop controls instead. Also, the pre-clearance computations in the CA Filter are based on future car position estimates generated by the Controller as part of the control MessageStream.

The main idea in the above approach to collision avoidance is to ensure that cars move only in pre-cleared areas. Since these areas are guaranteed to be non-intersecting, the cars do not collide as long as they are within their respective pre-cleared areas. The pre-clearance of cars is regulated by the CA Supervisor, and individual CA Filters must request pre-clearance from it. Since the CA Supervisor maintains the set of all assigned pre-cleared areas, it can ensure that subsequent pre-clearance assignments cause no collisions. Also, the CA Supervisor tracks obstacles such as stationary cars that can be monitored by the vision subsystem, and ensures that pre-clearance assignments do not intersect these obstacles either.

Finally, CA Filters request *sequences* of pre-cleared areas so that they do not have to contact the CA Supervisor on every control update. This improves overall system performance and scalability.

6.3.2 Algorithms

We now describe the main algorithms implemented in the collision avoidance sub-system in Figure 6.3. This description will also inform our subsequent analysis to establish safety guarantees provided by this sub-system.

CA Supervisor algorithm:

1. Connect to the vision feedback MulticastStream to get feedback about the testbed.
2. Wait for updates from the VisionServer and connection requests from CA Filters.
   (a) When an update from the VisionServer arrives, update the set of areas occupied by obstacles in the testbed.
   (b) When a CA Filter connects, open a MessageStream to send responses to pre-clearance requests.
   (c) When a request for a sequence of pre-clearance arrives from a CA Filter:
      i. Remove the pre-cleared areas previously assigned to the associated car from the set of pre-cleared areas.
      ii. Find the maximum sub-sequence that does not intersect with assigned pre-cleared areas or perceived obstacles.
      iii. Add the maximum sub-sequence to the set of assigned pre-cleared areas.
      iv. Send a pre-clearance response to the CA Filter approving the computed sub-sequence of pre-cleared areas.

CA Filter algorithm:

1. Connect to the CA Supervisor and create a MessageStream for sending pre-clearance requests. If a CA Supervisor is not found, then disable collision avoidance for the car.
2. Create a Filter to intercept control messages from the Controller to the Actuator.
3. Wait for intercepted control messages and responses from the CA Supervisor.
   (a) When an intercepted control message arrives:
      i. If there is pre-clearance for a sufficiently long sub-sequence of controls, then send the cleared controls to the Actuator.
      ii. If there isn't sufficient pre-clearance for these controls, then request for additional pre-clearance from the CA Supervisor, and send a stop control to the Actuator.
   (b) When a pre-clearance response arrives from the CA Supervisor, send any pending cleared controls to the Actuator.

There are a two important protocol considerations in the operation of a CA Filter. First, the CA Filter sends a pre-clearance request to the CA Supervisor only after it has received a pre-clearance response for a previous request. This ensures that both the CA Filter and the CA Supervisor have a consistent view of the pre-clearance for the associated car.

Second, due to possible failure of the CA Supervisor, or packet losses over the network, a pre-clearance request may not always result in a response. In such a situation, a CA Filter gets into a deadlock as it awaits a response from the CA Supervisor, which in turn is waiting for a subsequent pre-clearance request from the CA Filter. This is addressed by having a time-out at the CA Filter, so that if there is no response for a pre-clearance request for a given interval of time, then the CA Filter assumes that the request was lost, and sends another request if necessary. Also, if a sufficient number of consecutive requests do not elicit response, then the CA Filter assumes that the CA Supervisor is down, and disables collision avoidance for the car.

6.3.3 Analysis

We now prove that there are no collisions under the collision avoidance sub-system of Figure 6.3. In the following analysis, we assume that the car position estimates provided by the Controller in the control MessageStream have bounded error according to Theorem 5.4.1, and that this is accounted for in the pre-clearance computations of the CA Filter.

Lemma 6.3.1 *If the CA Supervisor is operational, then the pre-cleared areas assigned to different cars do not intersect.*

Proof The CA Supervisor is the only component in the system that can assign pre-cleared areas. Also, the CA Supervisor assigns a pre-cleared area to a car only if it does not intersect with any pre-cleared area previously assigned to other cars in the testbed. Hence, the pre-cleared areas assigned to different cars do not intersect. □

Lemma 6.3.2 *If the CA Filter of a car is operational and not preempted, and the CA Supervisor is operational, then the corresponding car moves only within its set of pre-cleared areas.*

Proof Firstly, the set of pre-cleared areas stored in a CA Filter is always a subset of the pre-cleared areas assigned to its car by the CA Supervisor. This follows from the fact that the CA Filter has at most one pre-clearance request awaiting response from the CA Supervisor, and the fact that the CA Filter sets its pre-cleared areas only after it receives a response from the CA Supervisor.

Secondly, since the CA Filter is not preempted by another Filter in the control MessageStream from the Controller to the Actuator, the controls that are finally executed by the Actuator are exactly the controls approved by the CA Filter.

Finally, the CA Filter only approves controls that operate the car within the set of pre-cleared areas for the car. Since the collision avoidance sub-system is active when the CA

Supervisor is operational, it follows that the corresponding car moves only within its set of pre-cleared areas assigned by the CA Supervisor. □

Theorem 6.3.3 *If the CA Filter of all cars are operational and not preempted, the central CA Supervisor is operational, and all obstacles in the track are stationary and can be monitored by the vision system, then there are no collisions between cars or obstacles in the testbed under the collision avoidance sub-system.*

Proof The pre-cleared areas assigned to different cars do not intersect with each other by Lemma 6.3.1. Also, since the obstacles are observable, the CA Supervisor ensures that the pre-cleared areas assigned to a car do not intersect with these as well. Further, since the obstacles are stationary and accounted for during pre-clearance assignment, there cannot be any obstacle in any pre-cleared area for any car in the testbed. Finally, since cars move only within their respective pre-cleared areas by Lemma 6.3.2, they do not collide with other cars or obstacles in the testbed. Consequently, it follows that there are no collisions in the testbed under the collision avoidance sub-system. □

6.4 Security case-study

In this section, we present a detailed case-study to validate the principle of safety preserving security overrides in the traffic control testbed. In particular, the collision avoidance sub-system described in Section 6.3 serves as the low-level safety mechanism for the case-study.

6.4.1 Scenario

For the case-study, we consider three kinds of cars: regular cars, rogue cars, and police cars. All these cars operate in a city traffic scenario with well-defined traffic rules. The regular cars and police cars are operated by Controllers and Actuators, while rogue cars are operated manually. The regular cars drive along the traffic grid based on traffic rules, and visit various points on the grid based on individual goals. The rogue cars are driven arbitrarily, while the police cars pursue the rogue cars following traffic rules as well.

The security scenario studied is as follows. Initially, there are no rogue cars in the traffic grid. The regular cars are driven according to individual goals, while the police cars are parked in strategic locations. A single rogue car is then introduced into the traffic constituting a *security breach*. As a *security response*, the police cars must then pursue the rogue car as closely as possible while still following traffic rules. Meanwhile, the regular cars continue driving based on their individual goals.

The two main objectives for the scenario are:

- *Safety objective:* The safety objective is to ensure collision-free operation of cars.
- *Security objective:* The security objective is to ensure that police cars pursue the rogue car as closely as possible.

6.4.2 Architecture

The software architecture for the security case-study is shown in Figure 6.4. This incorporates the collision avoidance safety sub-system of Figure 6.3, along with the federated supervisory control of Figure 6.1. In particular, the security override of local supervision preserves the safety mechanisms of the collision avoidance sub-system.

The Local Supervisors exert discretionary control, while the Global Supervisor enforces mandatory control with security overrides. The Local Supervisors for the regular cars plan routes for their respective cars, based on the discrete bin traffic grid model described in Section 2.2. Similarly, the Local Supervisors for the police cars plan routes for pursuing the rogue car, based on the same model. On the other hand, the Global Supervisor implements both these functionalities, i.e., routing regular cars to individual destinations, and supervis-

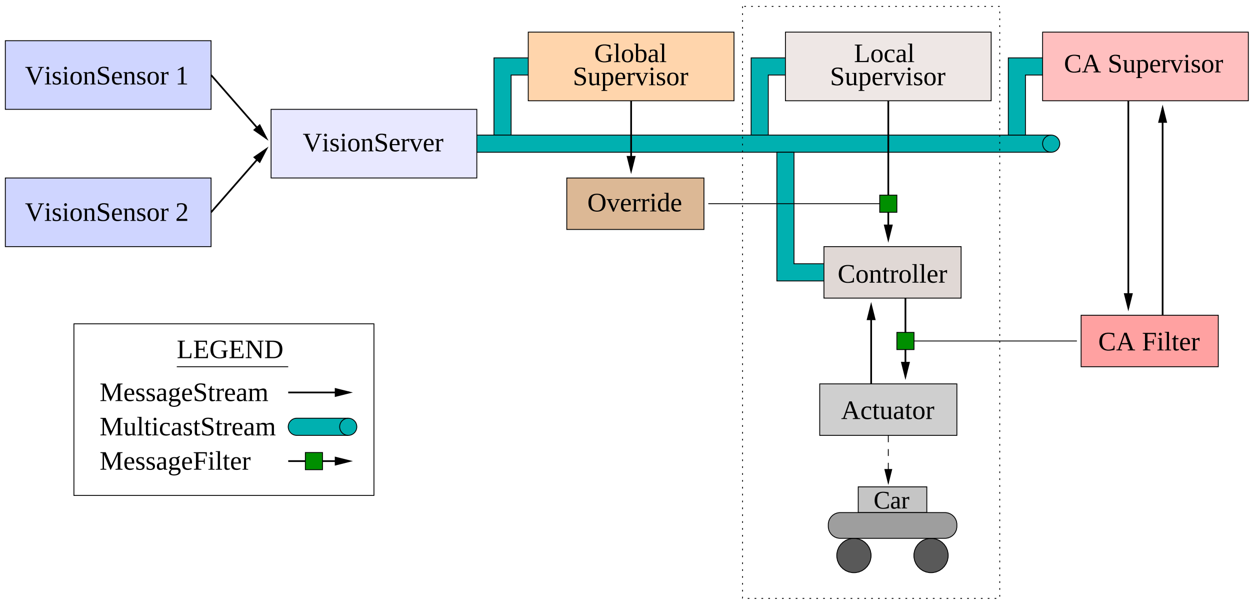


Figure 6.4: Software architecture for the security case-study

ing police cars to pursue the rogue car. In addition, the Global Supervisor also schedules collision-free trajectories based on the scheduling algorithms described in Section 2.2. Finally, the Global Supervisor overrides Local Supervisors using the security override Filter shown in Figure 6.4.

6.4.3 Experiments

We consider four different experiments in the traffic control testbed, based on the scenario presented in Section 6.4.1. These experiments correspond to the following configurations:

1. *No collision avoidance or security override:* This corresponds to local discretionary control of cars without low-level safety.

2. *Security override without collision avoidance:* This corresponds to global mandatory control of cars, also without low-level safety.

3. *Collision avoidance without security override:* This corresponds to local discretionary control with low-level safety.

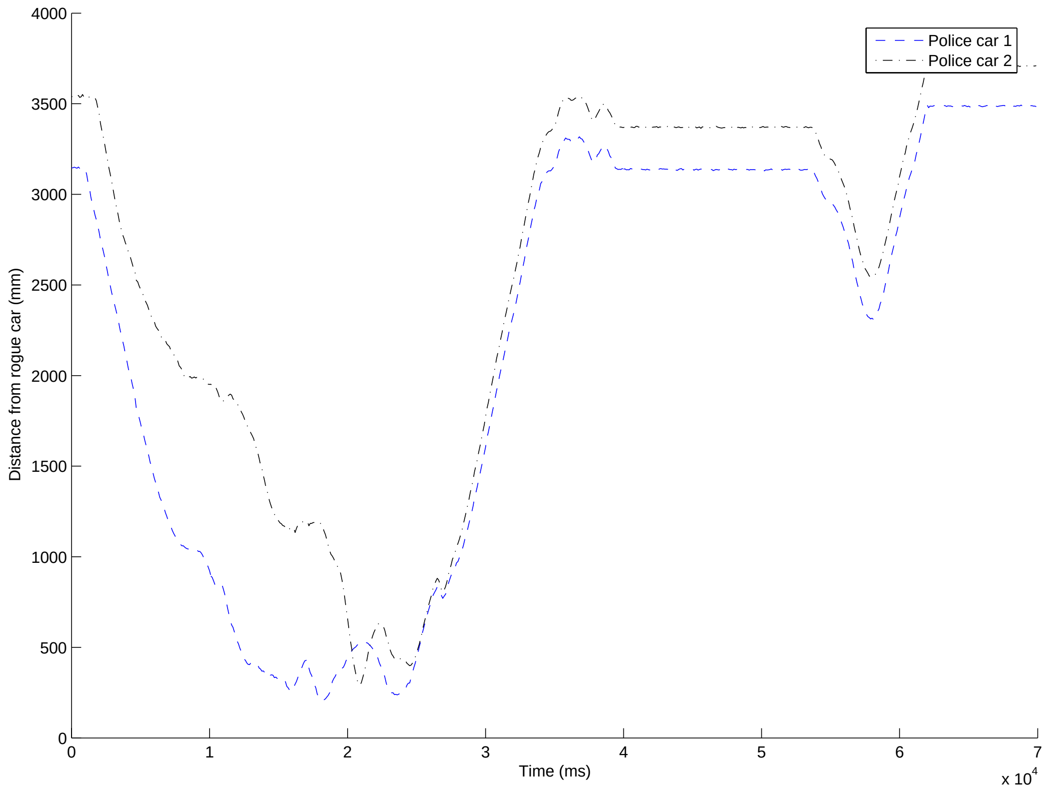


Figure 6.5: Experiment 1: No collision avoidance or security override

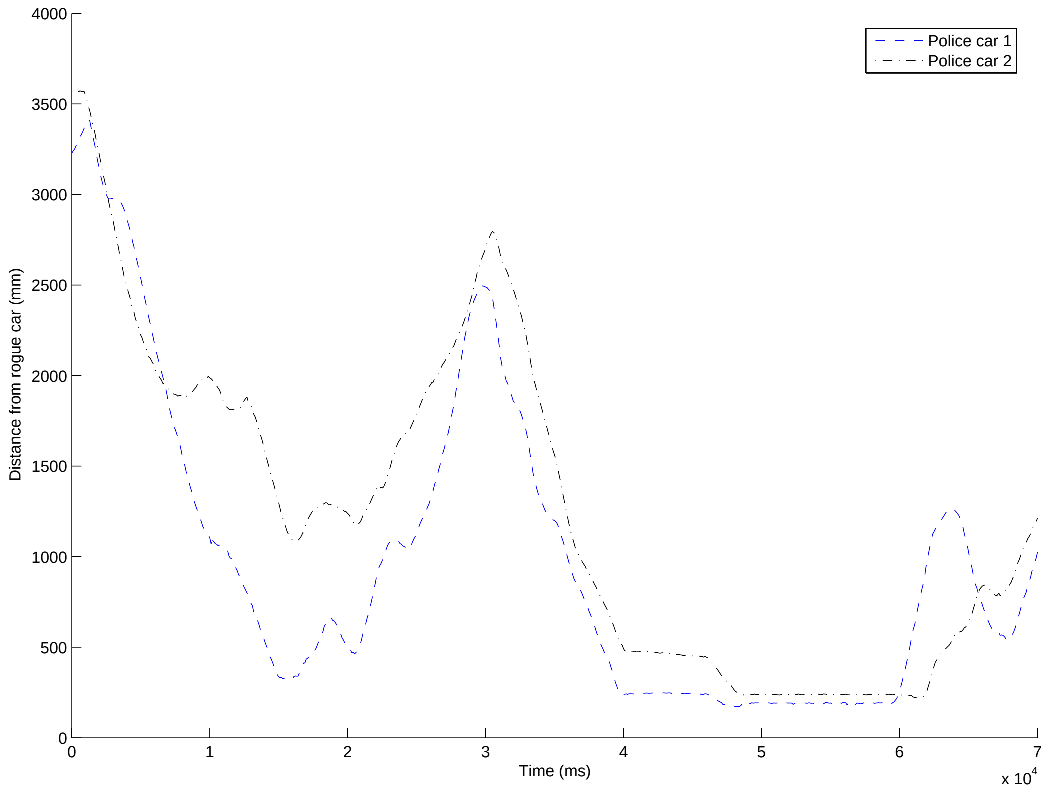


Figure 6.6: Experiment 2: Security override without collision avoidance

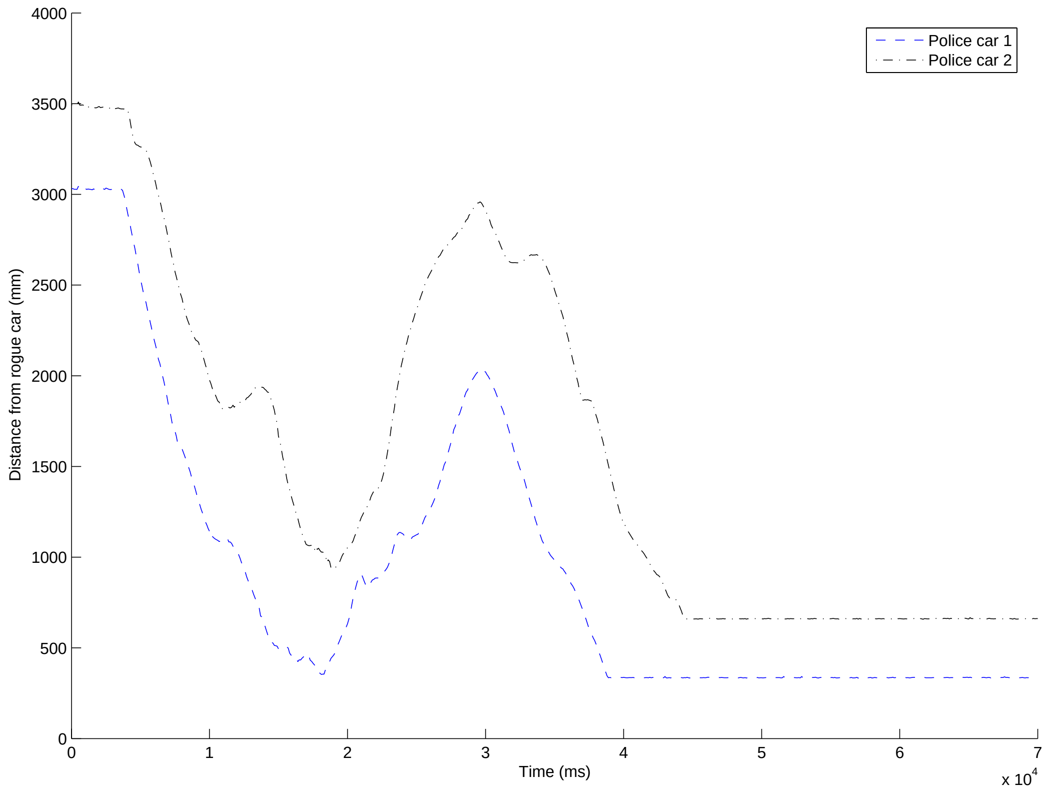


Figure 6.7: Experiment 3: Collision avoidance without security override

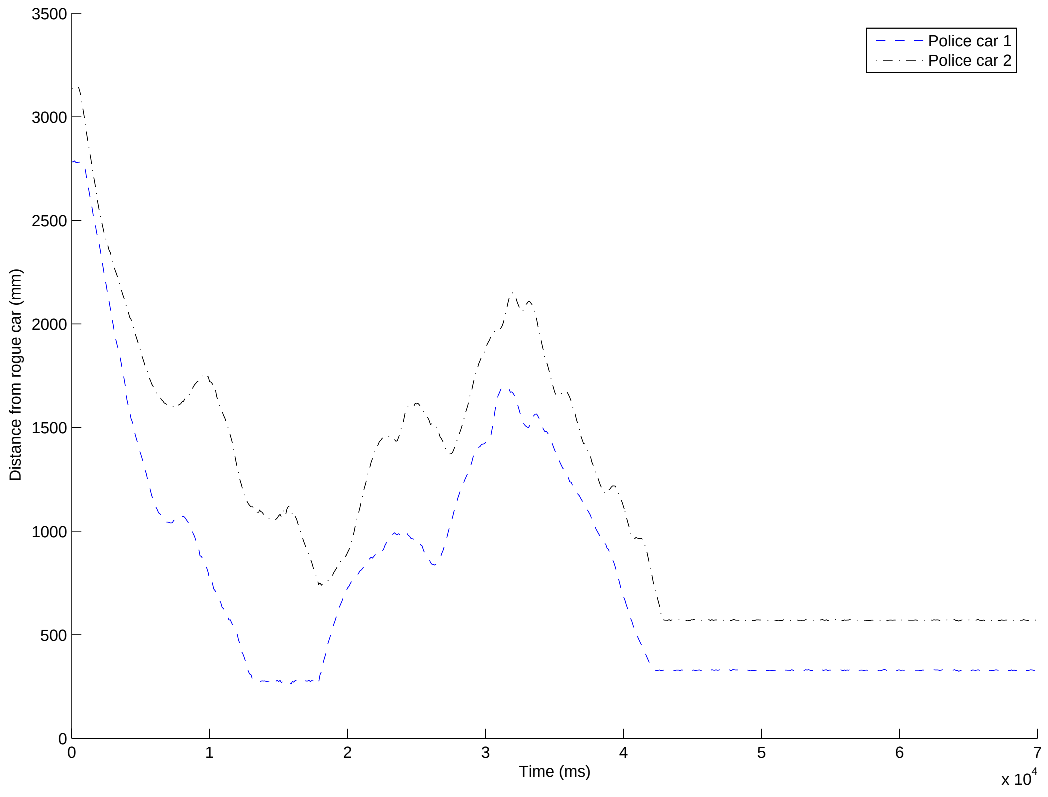


Figure 6.8: Experiment 4: Security override with collision avoidance

4. *Security override with collision avoidance:* This corresponds to global mandatory control with low-level safety as well.

Among the above experiments, we are particularly interested in the second and fourth experiments, since the former corresponds to security override that overrides safety as well, while in the latter, the security override is safety preserving. Also, in all the four experiments, we have two police cars that pursue a single rogue car, and two regular cars that follow pre-specified goals. The videos for all four experiments can be viewed at our testbed website [83].

Table 6.1: Safety measure for the four scenarios

Collision Avoidance	Security Override	Time to first collision (seconds)
No	No	2
No	Yes	10
Yes	No	No collisions
Yes	Yes	No collisions

Table 6.2: Security measure for the four scenarios

Collision Avoidance	Security Override	Minimum of police cars' distance to rogue car	
		Mean distance (mm)	Std. deviation (mm)
No	No	2182.7	1195.5
No	Yes	925.6	833.5
Yes	No	913.0	785.0
Yes	Yes	766.1	578.2

In Table 6.1, we note the safety performance of the system for all the experiments. In particular, our safety measure is the time to the first collision in each experiment. Similarly, we also tabulate the security performance in Table 6.2, where the security measure is the minimum distance between the rogue car and any of the police cars. In addition, we plot the graphs for the distance between the rogue and each police car in corresponding figures.

For the first experiment, the distance between the police cars and the rogue car is shown in Figure 6.5. As the plots illustrate, the police cars are not very successful in following the rogue car. This is primarily due to the frequent collisions that occur as the cars operate without coordination or collision avoidance. In particular, the first collision occurs only 2s into the experiment as we note in Table 6.1.

The situation is drastically improved in the second experiment when the security override enforces global supervision. The police cars now have higher priority in the system, and their improved performance is shown in Figure 6.6. However, while global supervision enforces collision free car schedules, there still are some collisions due to the slower response times of the supervisor. In fact, the first collision occurs about 8s into the experiment as Table 6.1 shows.

The safety in the system is further improved by the collision avoidance mechanism in the third experiment, where there are no collisions as shown in Table 6.1. In fact, as illustrated in Figure 6.6 and Table 6.2, the overall security performance is even better than in the second experiment even though there is no security override. However, police cars can be preempted by regular cars as demonstrated by the degraded performance of the second police car. More importantly, the lack of global supervision can lead to gridlocks[1] in the system and severely affect security performance.

However, the best performance, in terms of both safety and security, was during the fourth experiment, where the security override was enforced while preserving low-level safety. There were no collisions as shown in Table 6.1, and the cars were able to pursue the rogue car more effectively than in the other three experiments, as shown in Figure 6.8 and Tables 6.2. The global supervisory control ensured that the police cars had a higher priority than the regular cars. The overall safety was improved due to the collision avoidance mechanism, and in particular, the absence of collisions also contributed to the better performance of the police cars.

These experiments demonstrate the importance of preserving low-level safety mechanisms while enforcing high-level security overrides in control systems.

[1]Interestingly, at the end of the third experiment, there was a gridlock involving the two regular cars at a major intersection. This can be seen in the corresponding video at the testbed website [83].

Chapter 7

PROOF OF SYSTEM-WIDE SAFETY AND LIVENESS

The main result we prove in this chapter is that, given a road network with single-lane straight roads of sufficient length and width, angles of lane intersections, a set of cars with specified steering radii and bounded speed, real-time guarantees for renewal feedback about cars, and initial positions of the cars, the cars can be driven to their destinations without collisions (safety guarantee) or gridlocks (liveness guarantee), while staying within the confines of corresponding lanes (tracking guarantee). This result establishes properties of an overall system involving the convergence of control with communication and computation, and can serve as a possible prototype of such proofs for other networked control systems.

We begin with an overview of the main result. For the analysis, we consider road networks with straight line road segments and angled intersections as shown in Figure 7.1. Let L be the minimum length of the straight line segments, γ the maximum angle of intersection between any two lane segments, and W the minimum width of lanes. Suppose also that the cars are equipped with four steering controls: two causing anti-clockwise trajectories of radii $\underline{R}$ and $\overline{R}$, and the other two causing clockwise trajectories of radii $\underline{R}$ and $\overline{R}$. In particular, let $\underline{R} < \overline{R}$ as shown in Figure 7.2. Denote by s the maximum speed of a car. Finally, suppose we have a real-time guarantee[1] which ensures that the maximum time interval between successive observations and controls for any car is at most T time units as shown in Figure 7.3. So, if we define

$$\alpha := \frac{sT}{\underline{R}}$$

[1]We will later show real-time scheduling algorithms that guarantee such schedulability

Constraints: $L_i \geq L, \overline{W}_i \geq \overline{W}, \gamma_i \leq \gamma$

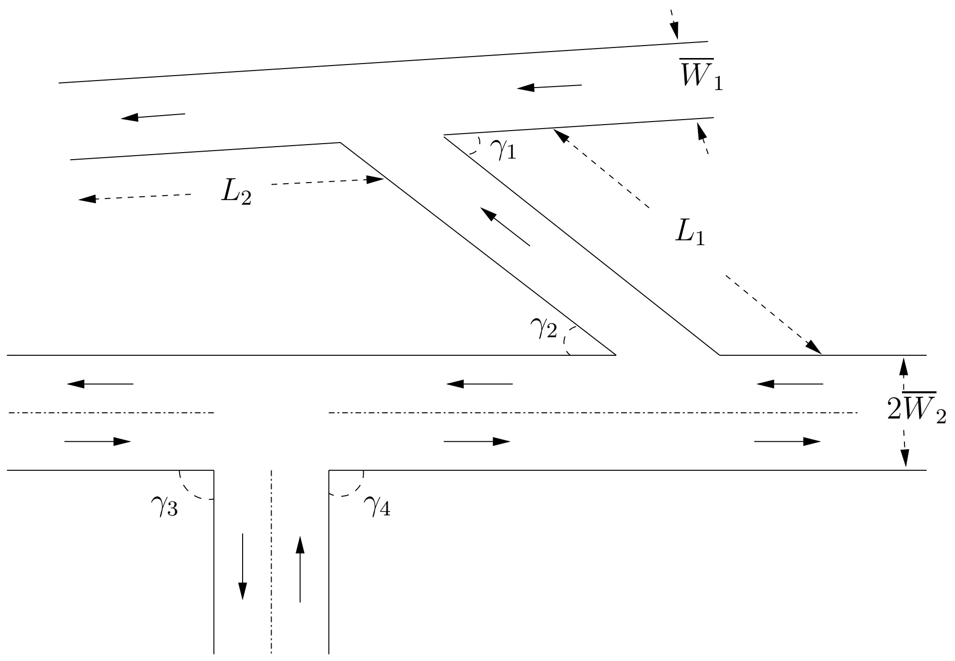


Figure 7.1: Road network with single lane roads

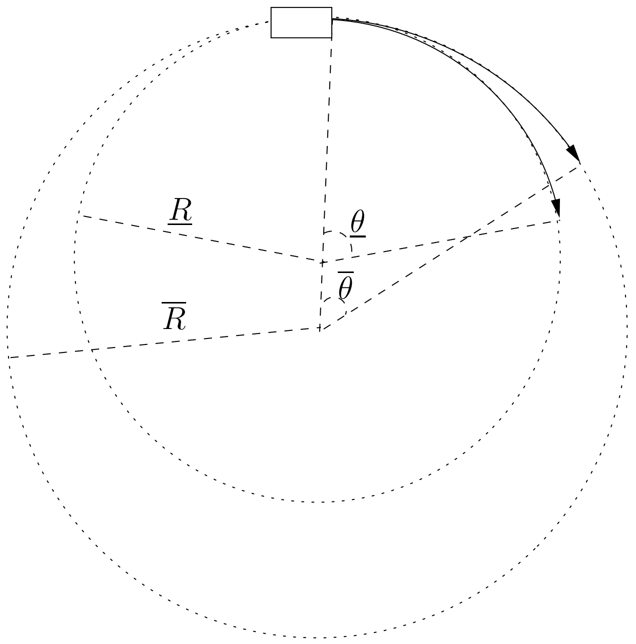


Figure 7.2: Clockwise steering controls for a car

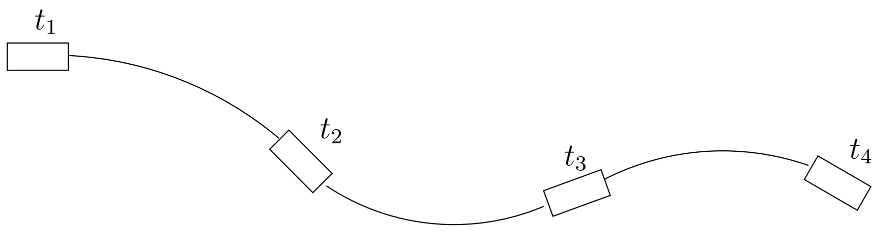


Figure 7.3: Real-time guarantee $t_{i+1} - t_i \leq T$ for car control

and $\beta = \cos^{-1}(2\cos\alpha - 1)$, then the real-time guarantee implies that a car moves a distance of at most $D = sT = \underline{R}\alpha$, before it is next observed and controlled.

Given the above characteristics of individual sub-systems, we show that a set of cars can be driven to their destinations in the road network without collisions or gridlocks, while staying within the confines of corresponding lanes, if

$$\begin{aligned} L &\geq \frac{2\gamma\overline{R}\underline{R}}{\overline{R} - \underline{R}} \\ \overline{W} &\geq \underline{R}(2 - \cos\beta(2\cos\alpha - \cos\gamma)) \\ \alpha &\leq \pi/3 \end{aligned}$$

7.1 Scheduling renewal tasks

The evolution of state prediction error characterized in Theorem 5.4.1 shows that, with accurate feedback, the error in car trajectory can be bounded if the maximum time interval between any two consecutive feedback updates can be also bounded. In other words, an accurate feedback update essentially resets the error in the car position estimate to zero, and acts as a "renewal point" for the state prediction process. Hence, the real-time guarantee of interest is to ensure that the maximum time interval between any two consecutive feedback updates is bounded. In this section, we formalize this requirement as the renewal task model, and present scheduling policies for renewal task sets.

7.1.1 Renewal task model

A task is a sequence of jobs that execute on, or are computed by, a specified resource. In this chapter, we only consider renewal tasks, which are defined as follows:

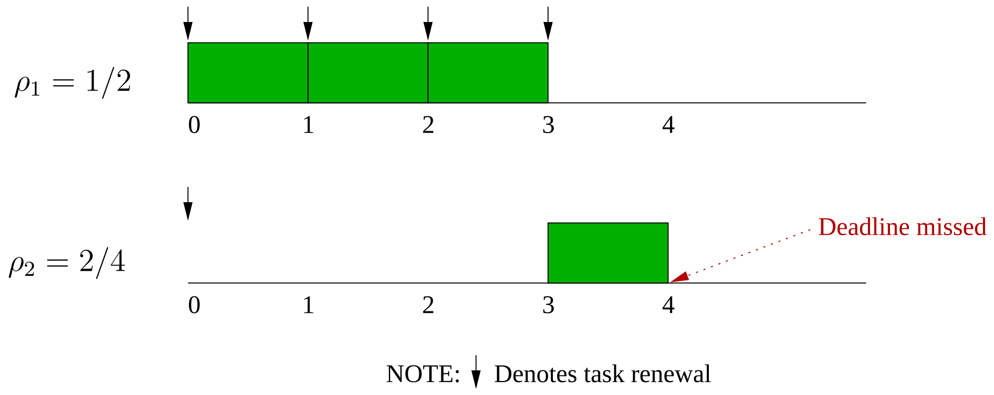
Definition 7.1.1 *In a* renewal task*, each job has a fixed computation time C, a relative deadline D, and a new job is released immediately after a previous job is completed. A renewal task is characterized by its task* density *$\rho = C/D$.*

In this section, we consider the problem of scheduling a set of n renewal tasks $S = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, on a common resource, starting at time $t = 0$. Each task $\sigma_i \in S$ is characterized by its task density $\rho_i = C_i/D_i$, where $C_i \in \mathbb{Z}^+$ is the fixed computation time and $D_i \in \mathbb{Z}^+$ is the relative deadline of each job of σ_i. In the remainder, we assume that $\rho_i > 0$ for all tasks $\sigma_i \in S$.

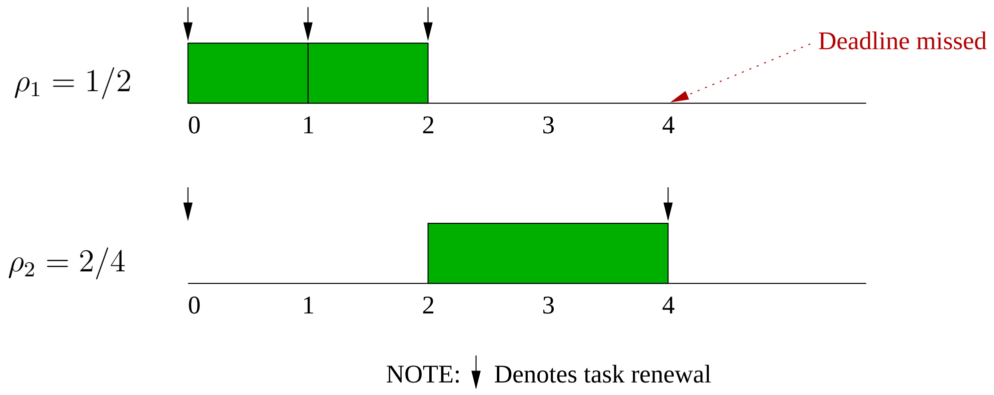
7.1.2 Prior work

Renewal tasks are interesting from a real-time scheduling point of view because they cannot be properly scheduled by standard non-idling policies such as Earliest Deadline First (EDF) [84] [7]. We illustrate this with a simple example. Consider a renewal task set $S = \{\sigma_1, \sigma_2\}$ with $\rho_1 = 1/2$ and $\rho_2 = 2/4$, by which we mean $C_2 = 2$ and $D_2 = 4$. This task set satisfies the EDF schedulability criterion $\rho_1 + \rho_2 \leq 1$. The schedule generated by EDF for the task set S is shown in Figure 7.4.

Since EDF basically schedules the job with the earliest deadline at any given time t, the first job of task σ_1 is scheduled at time $t = 0$ as it has the earliest deadline. Similarly, the second job of σ_1 is scheduled at time $t = 1$. At time $t = 2$, there are two choices since the corresponding jobs for both tasks σ_1 and σ_2 have the same deadline. However, choosing σ_1 at $t = 2$ leads to a deadline miss for σ_2 at time $t = 4$ as shown in Figure 7.4(a), while choosing σ_2 at $t = 2$ leads to a deadline miss for σ_1 at time $t = 4$ as shown in Figure 7.4(b). Consequently, renewal task sets cannot be scheduled by EDF in general.



(a) Task σ_1 chosen at time $t = 2$



(b) Task σ_2 chosen at time $t = 2$

Figure 7.4: Renewal task set not schedulable under EDF

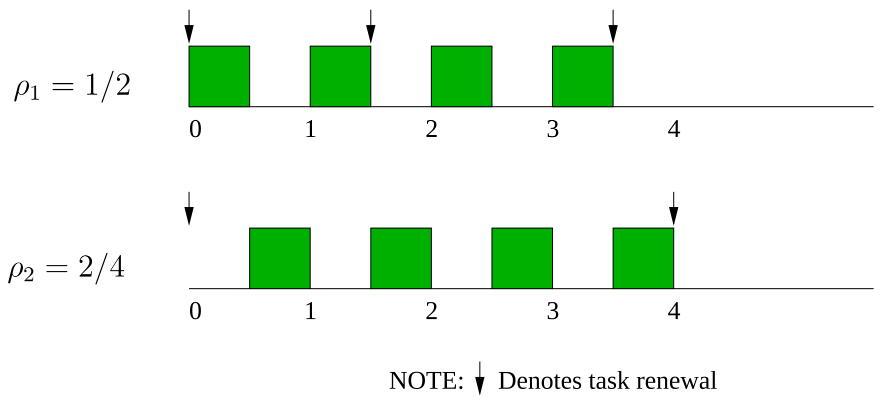


Figure 7.5: Renewal task set partial schedule under PSP

An elegant policy for scheduling renewal task sets has been presented by Han, Lin, and Hou in [85]. The main idea is to map this scheduling problem into a pin-wheel scheduling problem and apply well known pin-wheel scheduling algorithms such as **Sa**, **Sx**, **Sbc**, **Sby**, and **Sxy** [86], [87]. However, the schedulability condition is $\sum_i \sigma_i \leq 1$ only if the D_i's are all multiples of some D_j, and $\sum_i \sigma_i \leq n(2^{1/n} - 1)$ in the general case. In contrast, the policies presented in this section can all schedule any renewal task set that satisfies $\sum_i \sigma_i \leq 1$.

7.1.3 A non-resource-sharing policy

All the policies presented in Section 7.1.2 are non-resource-sharing policies since only one task consumes the resource at a given time. We now consider the Proportional Scheduling Policy (PSP), which is a non-resource-sharing policy defined as follows.

Definition 7.1.2 Proportional Scheduling Policy (PSP) *schedules tasks periodically so that in each successive interval of unit length, each task gets a time-share that is at least as large as its task density.*

We illustrate PSP using the same task set $S = \{\sigma_1, \sigma_2\}$ considered in Section 7.1.2, with $\rho_1 = 1/2$ and $\rho_2 = 2/4$. A partial schedule for this task set under PSP is shown in Figure 7.5. In this schedule, each successive unit time interval is divided into two equal sub-intervals, so that σ_1 is scheduled during the first sub-interval, and σ_2 is scheduled during the

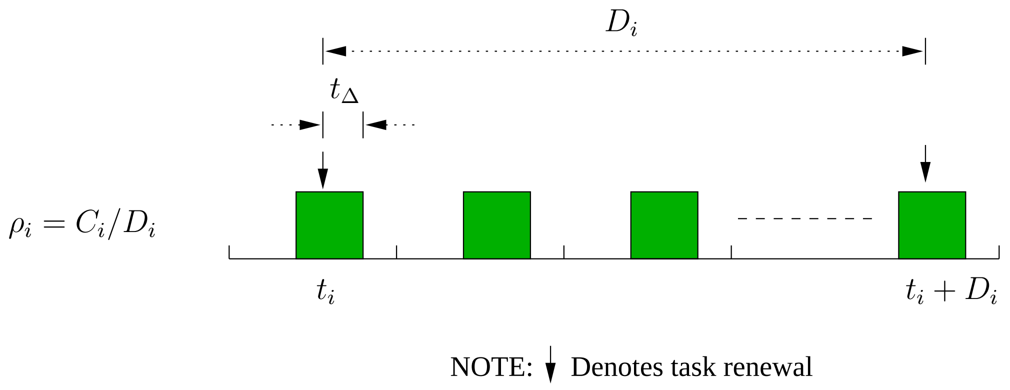


Figure 7.6: Partial schedule under PSP for task σ_i with renewal at t_m

second sub-interval. We can easily extend this partial schedule to verify that all deadlines are satisfied.

We now characterize the schedulability condition under which renewal task sets are schedulable[2] under PSP in the following theorem.

Theorem 7.1.3 *A renewal task set $S = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ is schedulable under PSP if and only if the task densities satisfy*

$$\sum_{i=1}^{n} \rho_i \leq 1.$$

where $C_i, D_i \in \mathbb{Z}^+$ for all $\sigma_i \in S$.

Proof If $\sum_i \rho_i > 1$, then the task set is clearly not schedulable as the demands exceed capacity. So, this condition is clearly necessary.

Suppose a task set $S = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ satisfies $\sum_i \rho_i \leq 1$. Consider a task $\sigma_i \in S$, and suppose that it has a renewal at time t_i. The partial schedule for the next job of σ_i is shown in Figure 7.6. In particular, the deadline of this job is $(t_i + D_i)$, and it needs at least C_i time units of the resource for completion.

Now, in each successive unit time interval within $T_i = (t_i, t_i + D_i)$, the task σ_i is scheduled for at least ρ_i time units under PSP. Also, as shown in Figure 7.6, σ_i is scheduled

[2]A task set is said to be *schedulable* by a policy if it can be scheduled by the policy so that no deadlines are missed.

for t_Δ time units immediately after its renewal at time t_i, and for $(\rho_i - t_\Delta)$ time units immediately before $(t_i + D_i)$. Hence, in the interval T_i, the task σ_i is scheduled for at least $t_\Delta + (D_i - 1)\rho_i + (\rho_i - t_\Delta) = D_i * \rho_i = C_i$ time units. Consequently, the next job of σ_i completes by its deadline. Finally, since we arbitrarily chose a job and a task, the same argument applies to all jobs in the system. $\square$

7.1.4 Resource-sharing policies

We now consider resource sharing policies for scheduling renewal task sets where more than one task can share the resource at a given time. We begin with some additional definitions for renewal tasks.

Definitions

Consider a set of renewal tasks $S = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. For a renewal task $\sigma_i \in S$, since a new job is released immediately after a previous job is completed, a job completion is also called a *task renewal.* For a given time instant $t \geq 0$, $\zeta_i(t)$ represents the time of the next task renewal of σ_i after time t. In particular, we always have $\zeta_i(t) > t$ for all $\sigma_i \in S$ and time instants $t \geq 0$. Also, subsequent task renewals are represented as $\zeta_i^2(t) = \zeta_i(\zeta_i(t))$, $\zeta_i^3(t) = \zeta_i(\zeta_i(\zeta_i(t)))$, etc. However, we define $\zeta_i^0(t) = t$ for convenience.

Further, $\zeta(t)$ is the *next renewal* of any task in S after time $t \geq 0$, i.e.,

$$\zeta(t) = \min_{\sigma_i \in S} \zeta_i(t).$$

Similarly, $\hat{\zeta}(t)$ is the *last renewal* of any task in S before time[3] $t > 0$, i.e.,

$$\hat{\zeta}(t) = \max\{\zeta^i(0) \leq t \mid i \in \mathbb{Z}^+\}.$$

[3]$\hat{\zeta}^i(0) = 0$ for all $i \in \mathbb{Z}^+$.

Finally, a *system renewal* occurs when all tasks have a renewal at the same time. For a given time instant $t \geq 0$, $Z(t)$ represents the next system renewal of the task set S after t, and $\hat{Z}(t)$ represents the previous system renewal before t. As before, subsequent system renewals are represented as $Z^2(t) = Z(Z(t))$, $Z^3(t) = Z(Z(Z(t)))$, etc.

Admissible scheduling policies

A scheduling policy allocates the common resource among currently executing jobs of the task set $S = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. More formally, given a time interval T, a scheduling policy allocates $\tau_i(T) \in \mathbb{R}^+$ time units of the resource to each task $\sigma_i \in S$ during the interval T. Also, $\tau_0(T) \in \mathbb{R}^+$ represents the idle time of the resource during T which is not allocated to any task in S. For convenience, define $\tau_i(t) = \tau_i([0,t])$ for $i = 0, \ldots, n$.

Since not all possible scheduling policies are reasonable, an admissible scheduling policy is defined as follows.

Definition 7.1.4 *An* admissible scheduling policy *is an allocation* $(\tau_0(t), \tau_1(t), \ldots, \tau_n(t))$ *for* $t \geq 0$ *such that* $\tau_i(t)$ *are continuous and non-decreasing functions of time t for* $i = 0, \ldots, n$, *and*

$$\sum_{i=0}^{n} \tau_i(t) = t.$$

Proportional Share Scheduling Policy

We now define a resource sharing version of the Proportional Scheduling Policy (PSP) presented in Section 7.1.3.

Definition 7.1.5 *The* Proportional Share Scheduling Policy (PSSP) *allocates the resource proportionally to all tasks in a task set* $S = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, *i.e., for any task* $\sigma_i \in S$, *we have*

$$\tau_i(T) = |T| * \frac{\rho_i}{\sum_{j=1}^{n} \rho_j}.$$

PSSP is clearly an admissible scheduling policy. The schedulability criterion for renewal task sets under PSSP is also easily obtained as follows.

Theorem 7.1.6 *A renewal task set $S = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ is schedulable under PSSP if and only if the task densities satisfy*

$$\sum_{i=1}^{n} \rho_i \leq 1.$$

Proof If $\sum_i \rho_i > 1$, then the task set is clearly not schedulable as the demands exceed capacity. So, this condition is clearly necessary.

Suppose a task set $S = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ with $\sum_i \rho_i \leq 1$ is not schedulable under PSP. Since task renewals are countable, we can consider the first time t_m at which a deadline is missed in a PSP schedule for this task set. Let $\sigma_m \in S$ be the corresponding task that missed its deadline at t_m. Now, the last task renewal of σ_m before t_m occurred at time $(t_m - D_m)$. However, in the time interval $T_m = (t_m - D_m, t_m)$, we have

$$\tau_m(T_m) = |T_m| * \frac{\rho_m}{\sum_{j=1}^{n} \rho_j} \geq |T_m| * \rho_m = C_m$$

under PSSP since $\sum_i \rho_i \leq 1$, $\rho_m = C_m/D_m$, and $|T_m| = D_m$. Hence, σ_m cannot miss a deadline at t_m, which is a contradiction. $\square$

Highest Scaled critical-ratio First policy

We now exhibit schedulability of renewal tasks through another interesting class of resource sharing admissible scheduling policies.

We begin with some additional definitions for renewal task sets. Consider a set of renewal tasks $S = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. For a renewal task $\sigma_i \in S$, and a given time instant $t \geq 0$, $\eta_i(t)$ is the number of jobs completed for task σ_i until time t, i.e.,

$$\eta_i(t) := \max\{n \in \mathbb{Z}^+ \mid \zeta_i^n(0) \leq t\}.$$

A renewal task always has exactly one unfinished job that needs to be scheduled. Hence, define $d_i(t)$ as the *deadline* of the current job of task σ_i at time $t \geq 0$. The *remaining computation time* $c_i(t)$ of the current job of task σ_i is then given by

$$c_i(t) := (\eta_i(t) + 1)C_i - \tau_i(t), \tag{7.1}$$

which leads to the following definition of a *critical ratio*.

Definition 7.1.7 Critical ratio $\kappa_i(t)$ *of a task* $\sigma_i \in S$ *at time* $t \geq 0$ *is defined as*

$$\kappa_i(t) := \frac{c_i(t)}{d_i(t)}$$

Instead of using critical ratios directly, we break jobs virtually into "sub-jobs" so that the corresponding sub-jobs of all tasks have the same deadlines. Consequently, we render the corresponding critical ratios all have the same denominator at any given time $t > 0$. Define $\bar{d}(t)$ to be the *next deadline* among the jobs in the system at time $t > 0$, i.e.,

$$\bar{d}(t) := \min_{\sigma_i \in S}\{d_i(t)\}, \tag{7.2}$$

and the *remaining computation time* $\bar{c}_i(t)$ of the "sub-job" of task $\sigma_i \in S$ as

$$\bar{c}_i(t) = \kappa_i(\hat{\zeta}(t))\bar{d}(\hat{\zeta}(t)) - \tau_i([\hat{\zeta}(t), t]), \tag{7.3}$$

where $\tau_i([\hat{\zeta}(t), t])$ is the *amount of work done on the current job* of task σ_i in the interval $[\hat{\zeta}(t), t]$. Now we define what we mean by *scaled critical ratios*.

Definition 7.1.8 *The* scaled critical ratio $\bar{\kappa}_i(t)$ *of a task* $\sigma_i \in S$ *at time* $t \geq 0$ *is defined as*

$$\bar{\kappa}_i(t) := \frac{\bar{c}_i(t)}{\bar{d}(t)}.$$

Definition 7.1.9 *The* scaled system critical ratio $\bar{\kappa}(t)$ *is the maximum of the scaled critical ratios of all tasks in* S *at time* $t \geq 0$*, i.e.,*

$$\bar{\kappa}(t) := \max\{\bar{\kappa}_i(t) \mid \sigma_i \in S\}.$$

Definition 7.1.10 Scaled critical task set $S_{\bar{\kappa}}(t)$ *is the set of tasks whose critical ratio is equal to the scaled system critical ratio at time* $t \geq 0$*, i.e.,*

$$S_{\bar{\kappa}}(t) = \{\sigma_i \in S \mid \bar{\kappa}_i(t) = \bar{\kappa}(t)\}.$$

Also, for a time interval T*, we have*

$$S_{\bar{\kappa}}(T) = \bigcup_{t \in T} S_{\bar{\kappa}}(t).$$

Finally, the class of *Highest Scaled critical-ratio First* policies is defined as follows.

Definition 7.1.11 Highest Scaled critical-ratio First (HSF) *is the class of admissible scheduling policies that enforce the condition*

$$\int_0^\infty \mathbb{I}(\bar{\kappa}_i(t) < \bar{\kappa}(t)) d\tau_i(t) = 0, \tag{7.4}$$

for all tasks $\sigma_i \in S$*, where* $\mathbb{I}(\cdot)$ *is the indicator function.*

Equation 7.4 basically ensures that a task $\sigma_i \in S$ is actively processed only when $\bar{\kappa}_i(t) = \bar{\kappa}(t)$. In the following development, we refer to the class of HSF policies collectively as HSF for convenience.

Clearly, HSF is a non-idling policy as it always schedules tasks with the highest critical ratio. In addition, we also have the following properties for critical functions under HSF.

Proposition 7.1.12 *The following properties hold for a renewal task set* $S = \{\sigma_1, \ldots, \sigma_n\}$ *scheduled by HSF:*

1. *$\bar{c}_i(t)$, $\bar{d}(t)$, $\bar{\kappa}_i(t)$, $\bar{\kappa}(t)$, $c_i(t)$, $d_i(t)$, and $\kappa_i(t)$ are right continuous for $i = 1, \dots, n$ and $t \geq 0$.*

2. *$\bar{c}_i(t)$, $\bar{d}(t)$, $\bar{\kappa}_i(t)$, $\bar{\kappa}(t)$, $c_i(t)$, $d_i(t)$, and $\kappa_i(t)$ are continuous in the interval $T_\zeta(t) = (\hat{\zeta}(t), \zeta(t))$ for $i = 1, \dots, n$ and $t \geq 0$.*

Proof

1. Since HSF is an admissible scheduling policy, $\tau_i(t)$ are continuous for $i = 0, \dots, n$ and $t \geq 0$. Consequently, $\bar{c}_i(t)$ are right continuous by eq (7.3). Also, $d_i(t)$ are right continuous as they increase linearly with time t except for jumps at renewals of corresponding σ_i. Hence, $\bar{d}(t)$ is right continuous as it is equal to the smallest of the $d_i(t)$, and jumps only when the corresponding task σ_i has a renewal. By definition, it follows that $\bar{\kappa}_i(t)$ are also right continuous. Also, since $\bar{\kappa}(t)$ is the maximum of finitely many right continuous functions, it is also right continuous. Similarly, since $\tau_i(t)$ are continuous for $i = 0, \dots, n$ and $t \geq 0$, $c_i(t)$ are right continuous by eq (7.1). Hence, it follows that $\kappa_i(t)$ are right continuous functions as well.

2. As noted above, $\tau_i(t)$ are continuous for $i = 0, \dots, n$ and $t \geq 0$ under HSF. Also, there are no job completions during the interval $T_\zeta(t) = (\hat{\zeta}(t), \zeta(t))$ for any task in S. Hence, $\bar{c}_i(t)$ are continuous in $T_\zeta(t)$ by eq (7.3). Similarly, $d_i(t)$ and $\bar{d}(t)$ are also continuous in $T_\zeta(t)$. Consequently, $\bar{\kappa}_i(t)$ are continuous in the interval $T_\zeta(t)$ as well. Also, since $\bar{\kappa}(t)$ is the maximum of finitely many continuous functions, it is also continuous in $T_\zeta(t)$. The functions $c_i(t)$ and $\kappa_i(t)$ are continuous in the interval T_ζ due to the same reason.

□

Since the functions $\bar{\kappa}_i(t)$ and $\bar{\kappa}(t)$ are continuous in the interval $T_\zeta(t) = (\hat{\zeta}(t), \zeta(t))$ for a given time instant $t > 0$, it follows that there is a sub-interval $T \subseteq (t, \zeta(t))$ during which the scaled critical task set is constant, i.e., $S_{\bar{\kappa}}(T) = S_{\bar{\kappa}}(t)$. Hence, the following interval is well-defined.

Definition 7.1.13 *For a given time instant* $t > 0$*, the* job-constancy interval $T_{\bar{\kappa}}(t)$ *is defined to be the maximal interval starting at* t *during which the scaled critical task set is constant, i.e.,* $S_{\bar{\kappa}}(T_{\bar{\kappa}}(t)) = S_{\bar{\kappa}}(t)$*.*

We now characterize the evolution of $\bar{\kappa}(t)$ in the interval $(t, \zeta(t))$.

Lemma 7.1.14 *For a renewal task set* $S = \{\sigma_1, \ldots, \sigma_n\}$ *scheduled by HSF, if for some* $t_0 \geq 0$ *we have*

$$\sum_{\sigma_i \in S_{\bar{\kappa}}(t_0)} \bar{\kappa}_i(t_0) \leq 1 - \epsilon,$$

then for any interval $T = (t_0, t) \subseteq T_{\bar{\kappa}}(t_0)$*, we have*

$$\bar{\kappa}(t) \leq \bar{\kappa}(t_0) - \frac{\epsilon |T|}{m(\bar{d}(t_0) - |T|))},$$

where $m = |S_{\bar{\kappa}}(t_0)|$*.*

Proof Suppose there is a time interval $T_f = (t_0, t_f) \subseteq T_{\bar{\kappa}}(t_0)$ such that

$$\bar{\kappa}(t_f) > \bar{\kappa}(t_0) - \frac{\epsilon |T_f|}{m(\bar{d}(t_0) - |T_f|))}.$$

Since t_f is in the job-constancy interval of t, we have

$$\bar{\kappa}_i(t) > \bar{\kappa}_i(t_0) - \frac{\epsilon |T_f|}{m(\bar{d}(t_0) - |T_f|))}$$

for all $\sigma_i \in S_{\bar{\kappa}}(t_0)$. By definition of $\bar{\kappa}_i(t)$, this is equivalent to

$$\frac{\bar{c}_i(t_0) - \tau_i(T_f)}{\bar{d}(t_0) - |T_f|} > \frac{\bar{c}_i(t_0)}{\bar{d}(t_0)} - \frac{\epsilon |T_f|}{m(\bar{d}(t_0) - |T_f|))},$$

which holds if and only if

$$\frac{\bar{c}_i(t_0)}{\bar{d}(t_0)} > \frac{\tau_i(T_f)}{|T_f|} - \frac{\epsilon}{m}.$$

Summing the above inequality for all $\sigma_i \in S_{\bar{\kappa}}(t_0)$, we have

$$\sum_{\sigma_i \in S_{\bar{\kappa}}(t_0)} \frac{\bar{c}_i(t_0)}{\bar{d}_i(t_0)} > \sum_{\sigma_i \in S_{\bar{\kappa}}(t_0)} \frac{\tau_i(T_f)}{|T_f|} - \sum_{\sigma_i \in S_{\bar{\kappa}}(t_0)} \frac{\epsilon}{m}.$$

Since HSF is an admissible policy, and $T_f \subseteq T_{\bar{\kappa}}(t_0)$, we have $\sum_{\sigma_i \in S_{\bar{\kappa}}(t_0)} \tau_i(T_f) = T_f$. Hence, the above inequality simplifies to

$$\sum_{\sigma_i \in S_{\bar{\kappa}}(t_0)} \bar{\kappa}_i(t_0) > 1 - \epsilon,$$

which is a contradiction. □

Theorem 7.1.15 *For a renewal task set $S = \{\sigma_1, \ldots, \sigma_n\}$ scheduled by HSF, if for some $t_0 \geq 0$ we have*

$$\sum_{\sigma_i \in S} \bar{\kappa}_i(t_0) \leq 1$$

then $\zeta(t_0) = \sum_{\sigma_i \in S} \bar{c}_i(t_0)$.

Proof Suppose $S_{\bar{\kappa}}(t_0) \subsetneq S$ and let $\sigma_j \in S \setminus S_{\bar{\kappa}}(t_0)$ be a non-critical task at t_0. By Lemma 7.1.14, the scaled critical ratio $\bar{\kappa}(t)$ decreases in the job-constancy interval $T_{\bar{\kappa}}(t_0)$, and $\bar{\kappa}_j(t)$ increases in the same interval. Hence, there is a first time instant t_j at which $\sigma_j \in S_{\bar{\kappa}}(t)$ joins the scaled critical task set and is scheduled by HSF. Clearly $(t_j - t_0) < \bar{d}(t_0)$, otherwise the sub-jobs of the tasks in $S_{\bar{\kappa}}(t_0)$ would complete, and we would have $\bar{\kappa}(t') = 0$ for some $t' < \bar{d}(t_0)$, which is impossible.

We will now show that

$$\sum_{\sigma_i \in S} \bar{\kappa}_i(t_j) \leq 1. \tag{7.5}$$

By Lemma 7.1.14, the above inequality holds if we have

$$\sum_{\sigma_i \in S_{\bar{\kappa}}(t_0)} \left(\bar{\kappa}_i(t_0) - \frac{\epsilon |T_j|}{m(\bar{d}(t_0) - |T_j|)} \right) + \sum_{\sigma_i \notin S_{\bar{\kappa}}(t_0)} \bar{\kappa}_i(t_j) \leq 1$$

where $m = |S_{\bar{\kappa}}(t_0)|$, $T_j = (t_0, t_j)$, and $\epsilon = 1 - \sum_{\sigma_i \in S_{\bar{\kappa}}(t_0)} \bar{\kappa}(t_0)$. This can be simplified to get

$$m\bar{\kappa}(t_0) - \frac{\epsilon |T_j|}{\bar{d}(t_0) - |T_j|} + \frac{\sum_{\sigma_i \notin S_{\bar{\kappa}}(t_0)} \bar{c}_j(t_0)}{\bar{d}(t_0) - |T_j|} \leq 1.$$

By rearranging terms, and noting that $\bar{\kappa}_i(t_0) = \bar{c}_i(t_0)/\bar{d}(t_0)$, we can rewrite the above inequality as

$$|T_j|(1 - \epsilon + m\bar{\kappa}(t_0)) \leq \bar{d}(t_0)(1 - \sum_{\sigma_i \notin S_{\bar{\kappa}}(t_0)} \bar{\kappa}_j(t_0) - m\bar{\kappa}(t_0))$$

But this inequality holds since $0 < |T_j| < \bar{d}(t_0)$ and $\sum_{\sigma_i \notin S_{\bar{\kappa}}(t_0)} \bar{\kappa}_i(t_0) \leq \epsilon$. Hence, the inequality (7.5) holds as well.

Since the number of tasks in S is finite, we can repeat the same argument to find a time t with $(t - t_0) < \bar{d}(t_0)$ such that $S_{\bar{\kappa}}(t) = S$. Hence, all the tasks share the resource in the job-constancy interval $T_{\bar{\kappa}}(t)$, at the end of which the corresponding sub-jobs are completed. Since HSF is non-idling, this occurs at time $\zeta(t_0) = \sum_{\sigma_i \in S} \bar{c}_i(t_0)$. □

We now characterize the schedulability condition for renewal task sets under HSF.

Lemma 7.1.16 *For a renewal task set $S = \{\sigma_1, \ldots, \sigma_n\}$ scheduled by HSF, if $\sum_{i=1}^{n} \rho_i \leq 1$ and $\bar{\kappa}_i(0) \leq \rho_i$ for all $\sigma_i \in S$, then $\bar{\kappa}_i(\zeta^k(0)) \leq \rho_i$ for all tasks $\sigma_i \in S$ and $k = 0, 1, 2, \ldots$.*

Proof We prove this by induction on $k = 0, 1, 2, \ldots$. The induction basis for $\zeta^0(0) = 0$ is trivial. For the induction step, suppose the result holds for some k, and for convenience define $t_k = \zeta^k(0)$ and $t_{k+1} = \zeta^{k+1}(0)$. Again, a task $\sigma_i \in S$ with $d_i(t_k) = \bar{d}(t_k)$ has a renewal at t_{k+1} by Theorem 7.1.15, and hence $\bar{\kappa}_i(t_{k+1}) = \kappa_i(t_{k+1}) = \rho_i$ by (7.3). However, if $d_i(t_k) > \bar{d}(t_k)$ then $\tau_i([t_k, t_{k+1}]) = \bar{c}_i(t_k)$, and since $t_{k+1} \leq \bar{d}(t_k)$ from Theorem 7.1.15, by

(7.3) we have

$$\bar{\kappa}_i(t_{k+1}) = \kappa_i(t_{k+1}) = \frac{c_i(t_k) - \bar{c}_i(t_k)}{d_i(t_k) - t_{k+1}} \le \frac{c_i(t_k) - \bar{c}_i(t_k)}{d_i(t_k) - \bar{d}(0)},$$

which can be simplified as

$$\bar{\kappa}_i(t_{k+1}) \le \frac{c_i(t_k) - (c_i(t_k)/d_i(t_k))\bar{d}(0)}{d_i(t_k) - \bar{d}(0)} = \frac{c_i(t_k)}{d_i(t_k)} = \kappa_i(t_k) \le \rho_i.$$

□

Theorem 7.1.17 *For a renewal task set $S = \{\sigma_1, \ldots, \sigma_n\}$, if $\sum_{i=1}^{n} \rho_i \le 1$ and $\bar{\kappa}_i(0) \le \rho_i$ for all $\sigma_i \in S$, then the task set is schedulable under HSF.*

Proof We prove this by induction on task renewals $\zeta^k(0)$ for $k = 0, 1, 2, \ldots$. For the induction basis, by Theorem 7.1.15, the jobs of tasks $\sigma_i \in S$ with $d_i(0) = \bar{d}(0)$ are completed at $\zeta(0)$ by their deadlines. All other tasks have $d_i(0) > \bar{d}(0)$, and hence no deadlines are missed until $\zeta(0)$.

For the induction step, suppose no deadlines are missed until $\zeta^k(0)$, and for convenience define $t_k = \zeta^k(0)$ and $t_{k+1} = \zeta^{k+1}(0)$. By Proposition 7.1.16, we have $\sum_{i=1}^{n} \bar{\kappa}_i(0) \le 1$. Hence, by Theorem 7.1.15 the jobs of tasks $\sigma_i \in S$ with $d_i(t_k) = \bar{d}(t_i)$ are completed at t_{k+1} by their deadlines. As before, all other tasks have $d_i(t_k) > \bar{d}(t_k)$, and hence no deadlines are missed until t_{k+1}. □

As a corollary of this theorem, it follows that any admissible policy that provides a computation time of at least

$$\tau_i(T_\zeta^k(0)) = \frac{\rho_i |T_\zeta^k(0)|}{\sum_j \rho_j}$$

for each task $\sigma_i \in S$, and each interval $T_\zeta^k(0) = (\zeta^k(0), \zeta^{k+1}(0))$ with $k \in \mathbb{Z}^+$, will meet all deadlines.

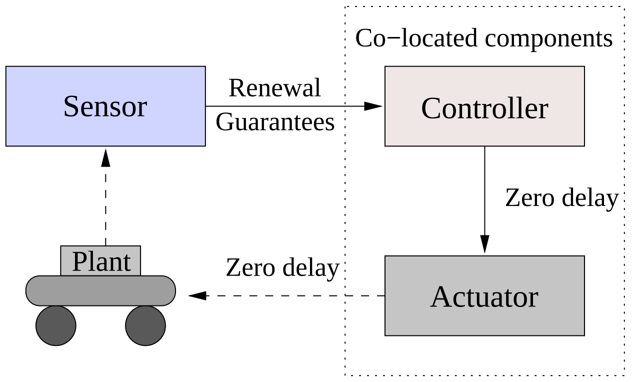


Figure 7.7: Control Loop Model

7.2 Controllability of a single car

The renewal task model bounds the maximum separation between any two consecutive completions of jobs in a task, and the scheduling policies presented in Section 7.1 demonstrate that this model can be implemented in practice. In this section, we use this guarantee to formulate and analyze the lower level control loop in the traffic control testbed design of Figure 5.1. In particular, we prove that a car can be controlled so that it stays within a road lane with a given width.

7.2.1 Car Model

The control loop model that we use for our analysis is shown in Figure 7.7. The feedback channel from the Sensor to the Controller is scheduled according to the renewal task model. On the other hand, the Controller and the Actuator are collocated, and there is negligible delay between a Controller issuing a control, and the Actuator effecting it in the car. But for the renewal guarantee, this model is a slightly simplified version of the current configuration in the testbed described in Section 2.2.

We model a car itself as an oriented point in a lane, and define the state of a car as the pair (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. In practice, this point would represent the center of the car. Also, 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 the clockwise direction. The two *anti-clockwise* controls move the car in anti-clockwise circles of radii $\underline{R}$ and $\overline{R}$ with $\underline{R} < \overline{R}$. Similarly, the two *clockwise* controls move the car in clockwise circles of radii $\underline{R}$ and $\overline{R}$. Based on the renewal task model for sensory feedback, we also assume that the car can move a distance of at most $\underline{R}\alpha$ 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; equivalently, the car can move a distance of at most $\underline{R}\alpha$ before the next control is sent.

7.2.2 Controllability in a straight road

We begin by analyzing the controllability of a car along a straight road. Specifically, we will try to formulate a control strategy so that a car can be indefinitely maintained within a straight lane of a given width using the above control loop model. Our strategy will be to find a subset S_C of the car state space, so that if the car is in S_C when a control is sent, then it can be indefinitely maintained within S_C. For brevity, we will call this set S_C, which can be held invariant through control, a *controllable subset.*

In the following analysis, we only allow steering controls that move the car in circles of radius $\underline{R}$. Suppose the car is initially in state $(0, 0)$, i.e. it is positioned at the center of the lane, and oriented along the direction of the road, which is assumed to be from left to right as noted before. If the car is given the *clockwise* control, then it moves in a clockwise circle of radius $\underline{R}$ and tangential to the center of the lane as shown in Figure 7.8. When we get to send the next control, the car may have moved a distance of up to $\underline{R}\alpha$ along its circle. Suppose the car does indeed move a distance of $\underline{R}\alpha$ before we send the next control. Since we next see the car to be below the lane, we issue an *anti-clockwise* control instead. As

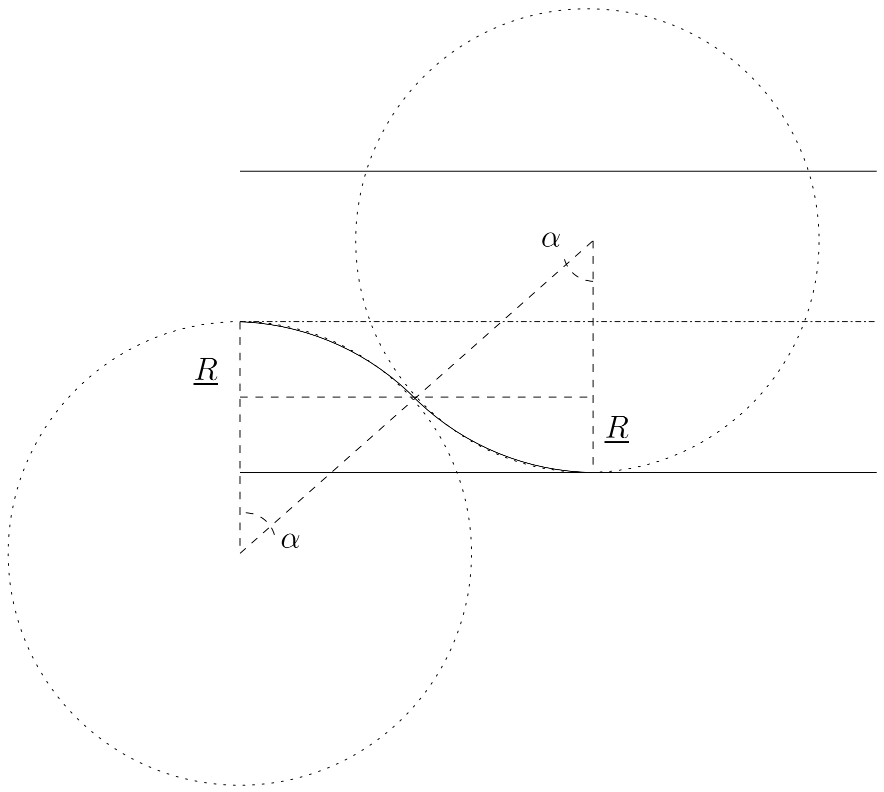


Figure 7.8: Controllable subset characterization

shown in Figure 7.8, the car will now move in an anticlockwise circle of radius $\underline{R}$ tangential to the previous circle. However, the car will still move away from the center of the lane for sometime, before it recovers. Consequently, the lane must be wide enough to account for this maneuver.

From Figure 7.8, we can also see that if the car is below the center of the lane initially, then it can be made to stay within this lane using the *anti-clockwise* control as long as its distance and orientation is bounded by the second circle in the figure. This subset is further illustrated in Figure 7.9.

One important constraint we want to ensure is that the car is never moving backwards, i.e., $\theta \leq \pi/2$ always. This implies that the maximum permissible angle $\beta \leq \pi/2$ in Figure 7.9. But, from Figure 7.9, we also have $\underline{R}(1-\cos\beta) = 2\underline{R}(1-\cos\alpha)$, which implies $\cos\beta =$

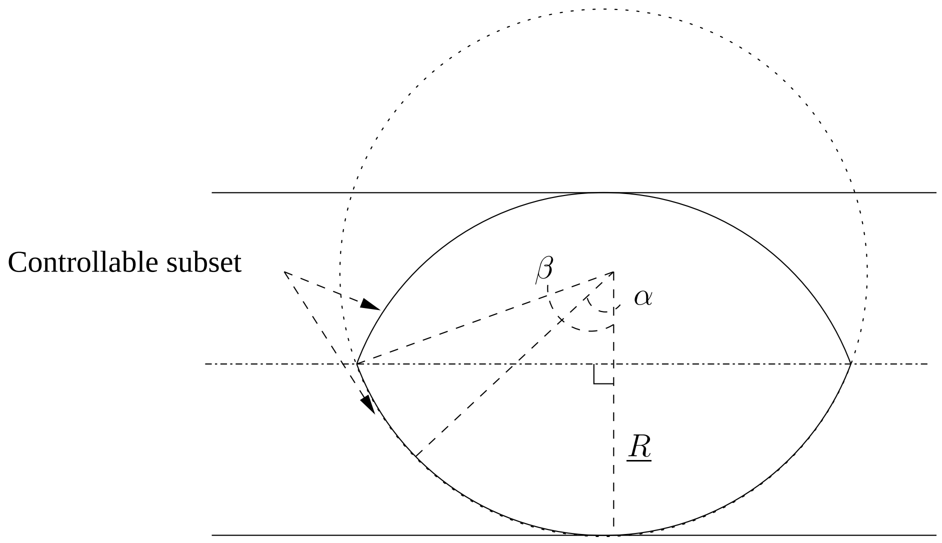


Figure 7.9: Controllable subset and maximum value of α, β

$2\cos\alpha - 1$, and $\beta \leq \pi/2$ in turn implies $\cos\alpha \geq 1/2$, i.e.,

$$\alpha \leq \frac{\pi}{3}. \tag{7.6}$$

In the following theorem, we prove that the subset illustrated in Figure 7.9 is actually a controllable subset for the car.

Theorem 7.2.1 *For an infinitely long road of width* $2W$*, where* $W = \underline{R}(1 - \cos\beta(2\cos\alpha - 1))$*,* $\alpha \leq \pi/3$*, and* $\beta = \cos^{-1}(2\cos\alpha - 1)$*, the set*

$$S_C = \{(d, \theta) \mid d + \underline{R}(1 - \cos\theta) \leq W\} \tag{7.7}$$

is a controllable subset of the car. That is, if the state of the car is initially within the subset S_C*, then the car can be controlled to stay indefinitely within the lane of width* $2W$*.*

Proof We prove this by showing that if the initial car state is in the subset $S_C = \{(d, \theta) \mid d + \underline{R}(1 - \cos\theta) \leq W\}$, then there exists a control which ensures that the car trajectory will remain in S_C until the next control can be sent.

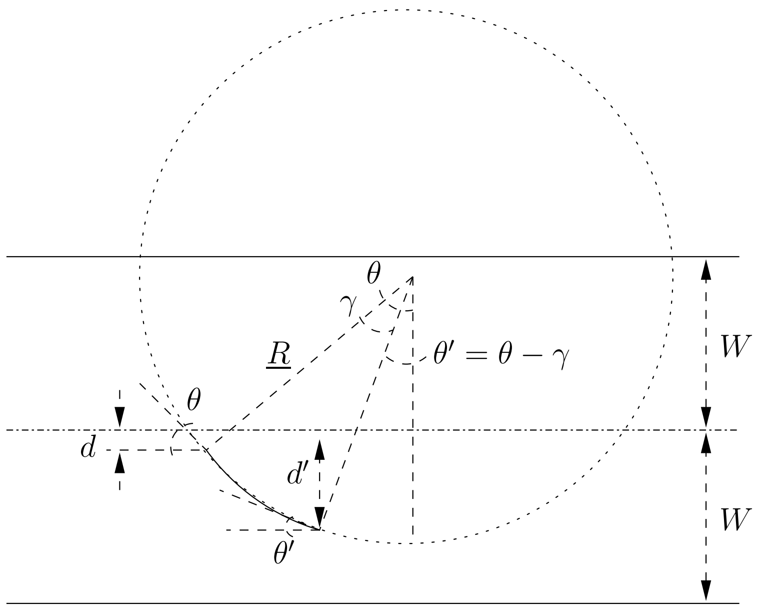


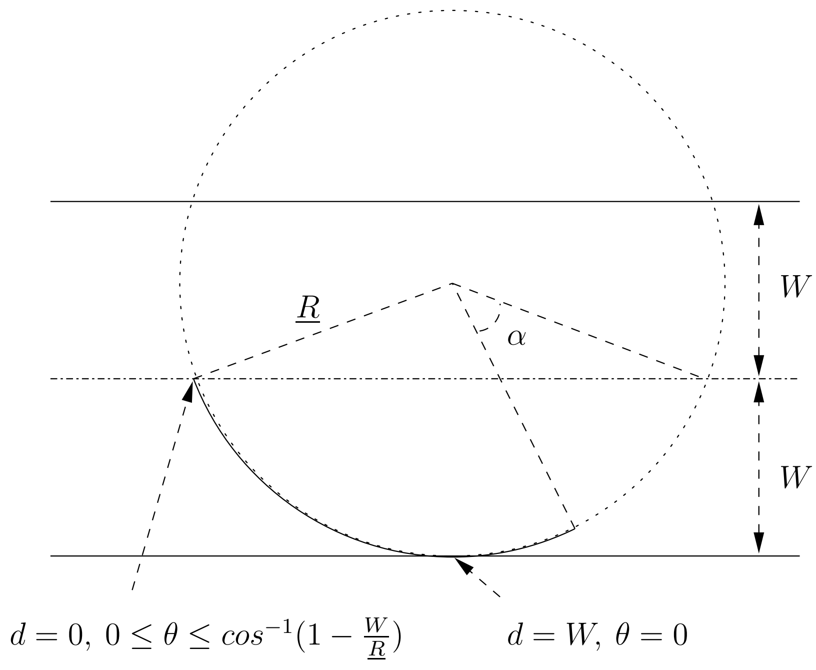
Figure 7.10: Possible trajectory for Case I

We need only consider initial car positions in the lower half of the lane, since a symmetrical argument applies to the upper half as well. We have two cases for possible initial car distance d in the lower half of the lane: on the solid curve segment in Figure 7.11(a), and on the solid curve segment in Figure 7.12(a). In either case, for a given distance d from the center of the lane, the initial car orientation θ satisfies $d + \underline{R}(1 - \cos\theta) \leq W$.

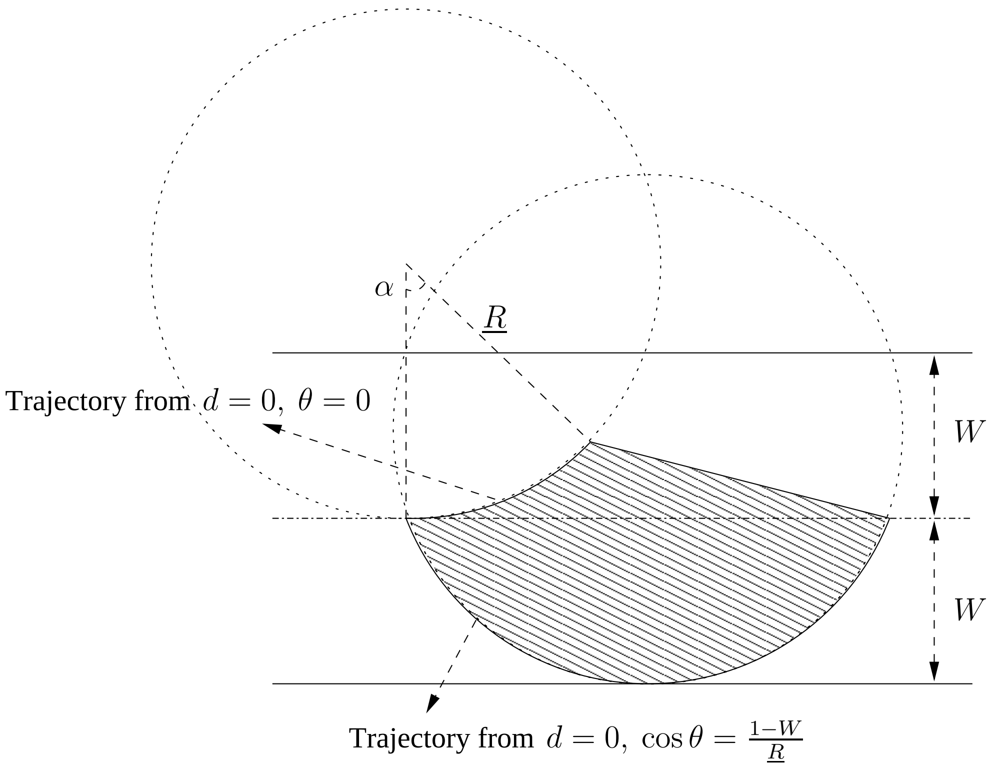
We consider each of these cases in detail.

CASE I

Suppose the initial car position is on the solid curve segment in Figure 7.11(a) so that distance d and angle θ satisfy $d + \underline{R}(1 - \cos\theta) \leq W$. It is easy to see that, under the *anti-clockwise* control, the subsequent trajectory of the car will be contained in the shaded region in Figure 7.11(b). Clearly, at all points on all these trajectories, the car state will satisfy $d + \underline{R}(1 - \cos\theta) \leq W$. To see this, suppose the initial position (d, θ) satisfies $d + \underline{R}(1 - \cos\theta) \leq W$. Then, after moving through an angle $0 \leq \gamma \leq \alpha$, the new coordinates


$\underline{R}$
α
W
W
$d = 0,\ 0 \leq \theta \leq cos^{-1}(1 - \frac{W}{\underline{R}})$
$d = W,\ \theta = 0$

(a) Initial states for Case I


α
$\underline{R}$
Trajectory from $d = 0,\ \theta = 0$
W
W
Trajectory from $d = 0,\ \cos\theta = \frac{1-W}{\underline{R}}$

(b) Trajectory subset enclosure for Case I

Figure 7.11: Initial states and trajectories for Case I

are $d' = d + \underline{R}(\cos(\theta - \gamma) - \cos\theta)$ and $\theta' = \theta - \gamma$ as shown in Figure 7.10. Now,

$$\begin{aligned} d' + \underline{R}(1 - \cos\theta') &= d + \underline{R}(\cos(\theta - \gamma) - \cos\theta) + \underline{R}(1 - \cos(\theta - \gamma)) \\ &= d + \underline{R}(1 - \cos\theta) \\ &\leq W \end{aligned}$$

as required.

CASE II

Suppose the initial car position is on the solid curve segment in Figure 7.12(a) so that the distance d and the orientation θ satisfy $d + \underline{R}(1 - \cos\theta) \leq W$. In this case, the *anti-clockwise* control may not always ensure that car-states in the subsequent trajectory of the car will satisfy $d + \underline{R}(1 - \cos\theta) \leq W$.

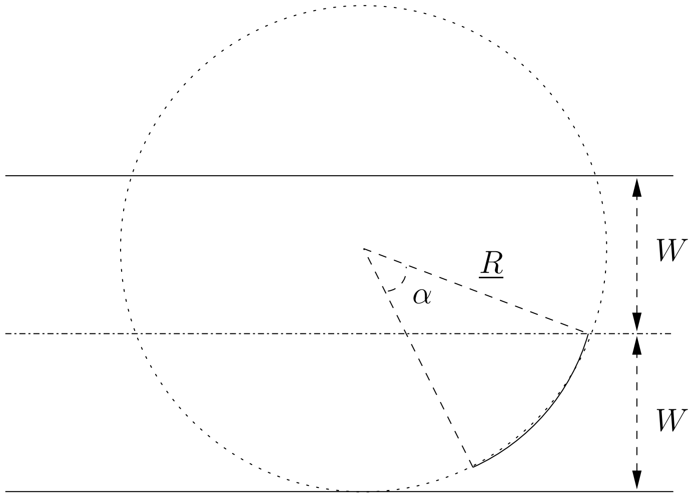
For a given distance d in Figure 7.12(a), we seek an angle θ^* such that car states will satisfy $d + \underline{R}(1 - \cos\theta) \leq W$ in the subsequent car trajectory up to the next renewal for both *anti-clockwise* and *clockwise* controls. That is, such an initial (d, θ^*) is an indifference state, so that both anti-clockwise and clockwise controls will maintain feasibility up until the next renewal. If we can find such an indifference angle θ^*, then for initial angles $\theta < \theta^*$, the *anti-clockwise* control ensures a valid subsequent trajectory, while for initial angles $\theta > \theta^*$, the *clockwise* control ensures the same.

Suppose the initial car position is (d, θ) as shown in Figure 7.12(a). If the *anti-clockwise* control is used, then, in the worst case, the car can at most reach the state (d_1, θ_1) shown in Figure 7.12(b) with[4] $d_1 = \underline{R}(\cos\theta - \cos\theta_1) - d$ and $\theta_1 = \alpha + \theta$. Since we require $(d_1, \theta_1) \in S_C$, we need

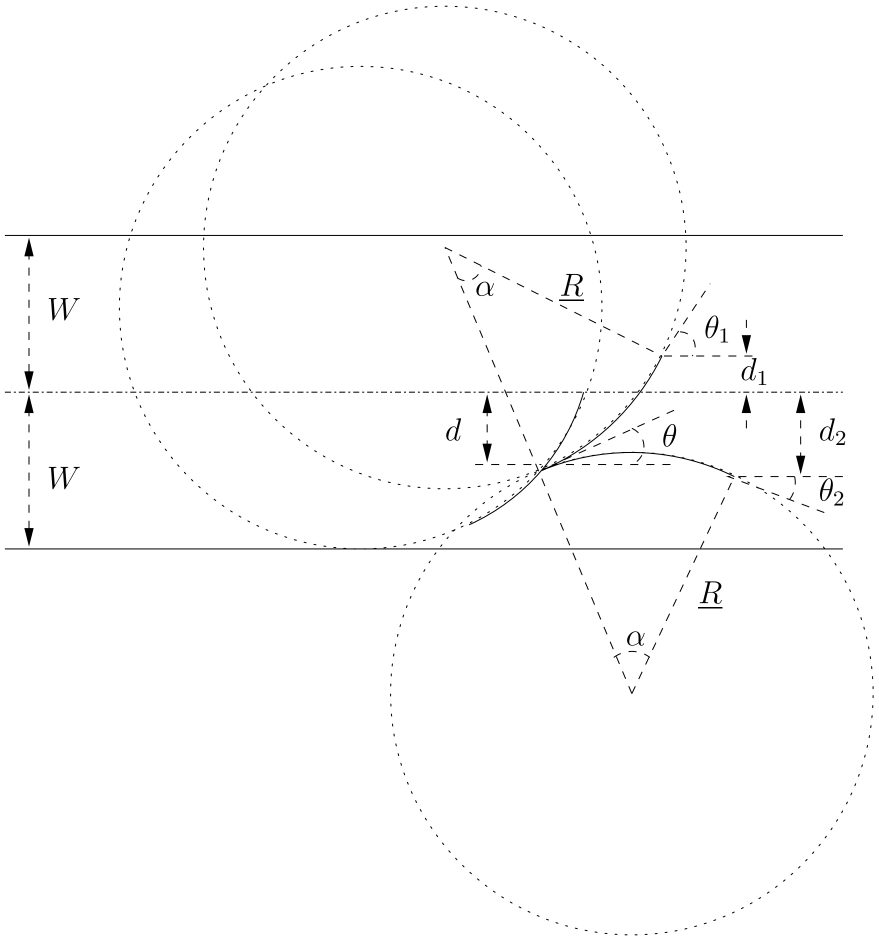
$$\underline{R}(1 + \cos\theta - 2\cos(\alpha + \theta)) - d \leq W, \tag{7.8}$$

to ensure that feasibility is maintained until the next renewal.

[4]Note that $d_1 \geq 0$ by convention since it is the absolute distance from the median.



(a) Initial states for Case II



(b) Trajectory subset enclosure for Case II

Figure 7.12: Initial states and trajectories for Case II

Similarly, if the *clockwise* control is used, then, in the worst case, the car can at most reach the state (d_2, θ_2) with $d_2 = \underline{R}(\cos\theta - \cos\theta_2) + d$ and $\theta_2 = \alpha - \theta$, as shown in Figure 7.12(b). Since we also require $(d_2, \theta_2) \in S_C$, we need

$$\underline{R}(1 + \cos\theta - 2\cos(\alpha - \theta)) + d \leq W, \tag{7.9}$$

to ensure that feasibility is maintained until the next renewal.

Suppose the left-hand side (LHS) expressions of both (7.8) and (7.9) are equal for some θ^*. We can then equate the two expressions to get

$$\sin\theta^* = \frac{d}{2R\sin\alpha}. \tag{7.10}$$

Since the θ^* in (7.10) is such that the LHS expressions of (7.8) and (7.9) are equal, the initial state (d, θ^*) will satisfy both these inequalities if and only if the sum of these LHS expressions is bounded by $2W$, i.e.,

$$(\underline{R}(1 + \cos\theta^* - 2\cos(\alpha + \theta^*)) - d) + (\underline{R}(1 + \cos\theta^* - 2\cos(\alpha - \theta^*)) + d) \leq 2W$$

This inequality can be simplified to get

$$(1 + \cos\theta^* - 2\cos\theta^*\cos\alpha) \leq \frac{W}{\underline{R}}.$$

Since $W = \underline{R}(1 - \cos\beta(2\cos\alpha - 1))$, the above inequality is equivalent to

$$\cos\theta^*(2\cos\alpha - 1) \geq \cos\beta(2\cos\alpha - 1). \tag{7.11}$$

Since $\alpha \leq \pi/3$, the above inequality will hold if $\theta^* \leq \beta$.

Recall that θ^* is such that the LHS expressions of (7.8) and (7.9) are equal. Now, for $\theta = 0$, the LHS of (7.8) becomes $\underline{R}(2 - 2\cos\alpha) - d$, which is less than $\underline{R}(2 - 2\cos\alpha) + d$, the corresponding value of the LHS of (7.9). Hence, for $\theta = \beta$, if the LHS of (7.8) is greater than that of (7.9), then the two expressions will be equal by the intermediate value theorem for some $0 \leq \theta^* \leq \beta$. So, for $\theta = \beta$, we need to show that

$$\underline{R}(1 + \cos\beta - 2\cos(\alpha + \beta)) - d \geq \underline{R}(1 + \cos\beta - 2\cos(\alpha - \beta)) + d,$$

which is equivalent to

$$\cos(\beta - \alpha) - \cos(\beta + \alpha) \geq \frac{d}{\underline{R}}.$$

However, from Figure 7.12(a), we see that the maximum value of $d = \underline{R}(\cos(\beta - \alpha) - \cos\beta)$. Hence, the above inequality will hold if

$$\cos\beta \geq \cos(\beta + \alpha),$$

which is true since $(\beta + \alpha) \leq \pi$, and $\cos\theta$ is a decreasing function in the interval $[0, \pi]$.

Consequently, for a given initial distance d in the subset specified in Figure 7.12(a), there indeed exists an orientation θ^* given by (7.10), such that for the initial position (d, θ^*), both *anti-clockwise* and *clockwise* controls will maintain feasibility until the next renewal. Hence, if the initial angle $\theta < \theta^*$, then the *anti-clockwise* control will ensure a valid trajectory, and if $\theta \geq \theta^*$, then the *clockwise* control will do the same. □

7.2.3 Controllability in a road network

In Section 7.2.2, we have proved that a car can be driven within a straight lane indefinitely, using feedback with renewal task guarantees. In a road network, however, roads have finite lengths, and turns need to be made at intersections. In this section, we show that, given sufficiently wide roads, a car can also be driven along such a road network. In the following, we use the same car model presented in Section 7.2.1, while taking advantage of the availability of the second set of steering controls with the larger turning radius $\overline{R} > \underline{R}$.

The main problem in navigating turns is that a car has to move from one road to another at their intersection. In particular, the car state (d, θ) satisfying $d - \underline{R}(1 - \cos\theta) \leq W$ on the first road, may not necessarily satisfy this condition in the second road. Hence, the car must be controlled so that its state is brought back, within a bounded distance, into the controllable subset corresponding to the second road. If this can be accomplished, then the car can be controlled to stay withing the roads in a road network as well. The following theorem shows that, under certain constraints on initial position, a car can indeed be controlled to get it back into the controllable subset within a bounded distance.

Theorem 7.2.2 *If the initial state (d, θ) of a car satisfies $d \leq W$ and $\theta \leq \pi/2$, then the car can be controlled to a state (d_1, θ_1) that satisfies $d_1 + \underline{R}(1 - \cos\theta_1) \leq W$ with $W = \underline{R}(1 - \cos\beta(2\cos\alpha - 1))$, $\alpha \leq \pi/3$, and $\beta = \cos^{-1}(2\cos\alpha - 1)$, within a distance of at most*

$$L = \frac{2\gamma\overline{R}\underline{R}}{\overline{R} - \underline{R}}, \tag{7.12}$$

where γ satisfies $d + \underline{R}(1 - \cos(\theta - \gamma)) \leq W$.

Proof In the following, we only consider initial car positions in the lower half of the lane, since a symmetrical argument applies to the upper half as well.

Suppose the car is initially in a state (d, θ) with $d \leq W$, but with $d + \underline{R}(1 - \cos\theta) > W$. Since $d \leq W$, there is an angle γ such that $d + \underline{R}(1 - \cos(\theta - \gamma)) \leq W$. For instance, the situation when $d = W$ is illustrated in Figure 7.13. The key observation is that the car can still be driven within an appropriately displaced lane of width $2W$ by the same control strategy used in the proof of Theorem 7.2.1. So, the problem now is to find a sequence of controls that can "shift" this lane so that the "excess" angle offset γ is "negated".

A control strategy that can be used to negate γ is shown in Figure 7.14. At each point, the car is controlled as if it is being driven in an appropriately displaced lane, illustrated in Figure 7.13, using the same control strategy as in the proof of Theorem 7.2.1. When the

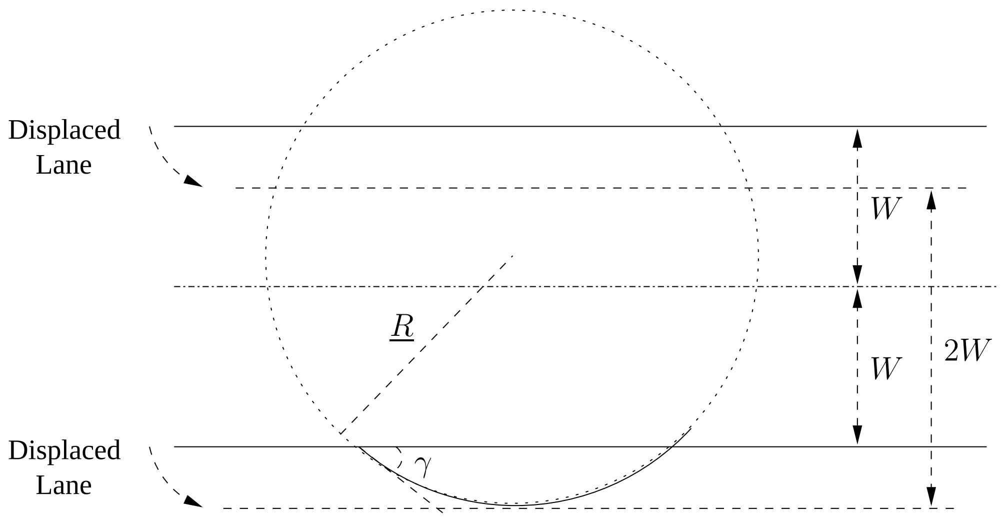


Figure 7.13: Driving the car in a displaced lane

anti-clockwise control is to be applied, we use the control with steering radius $\underline{R}$ as before. However, when the *clockwise* control is to be enforced, we use the control with steering radius $\overline{R}$ instead.

As shown in Figure 7.14, the control with the larger steering radius $\overline{R}$ ensures that the angle $\overline{\theta}$ at the next control point will be lesser than the $\underline{\theta}$ that it would have been, had we used the control with steering radius $\underline{R}$ instead. This is effectively an "improvement" of $(\underline{\theta} - \overline{\theta})$ towards negating γ. From Figure 7.14, we also see that $(\underline{\theta} - \overline{\theta}) = (\underline{\theta'} - \overline{\theta'})$ and

$$\underline{\theta'} = \frac{\overline{R} * \overline{\theta'}}{\underline{R}},$$

and hence the improvement achieved is

$$\underline{\theta} - \overline{\theta} = \overline{\theta'}\left(\frac{\overline{R} - \underline{R}}{\underline{R}}\right). \tag{7.13}$$

Such an improvement is achieved each time we use a *clockwise* control. Hence, the car state will be brought back into the controllable subset when $\underline{\theta} - \overline{\theta} = \gamma$ for *clockwise* controls. During this time, the car travels a distance of $\overline{R} * \overline{\theta'}$. Also, we use the *clockwise* control half the time, since the original control strategy maintains the car within a displaced lane

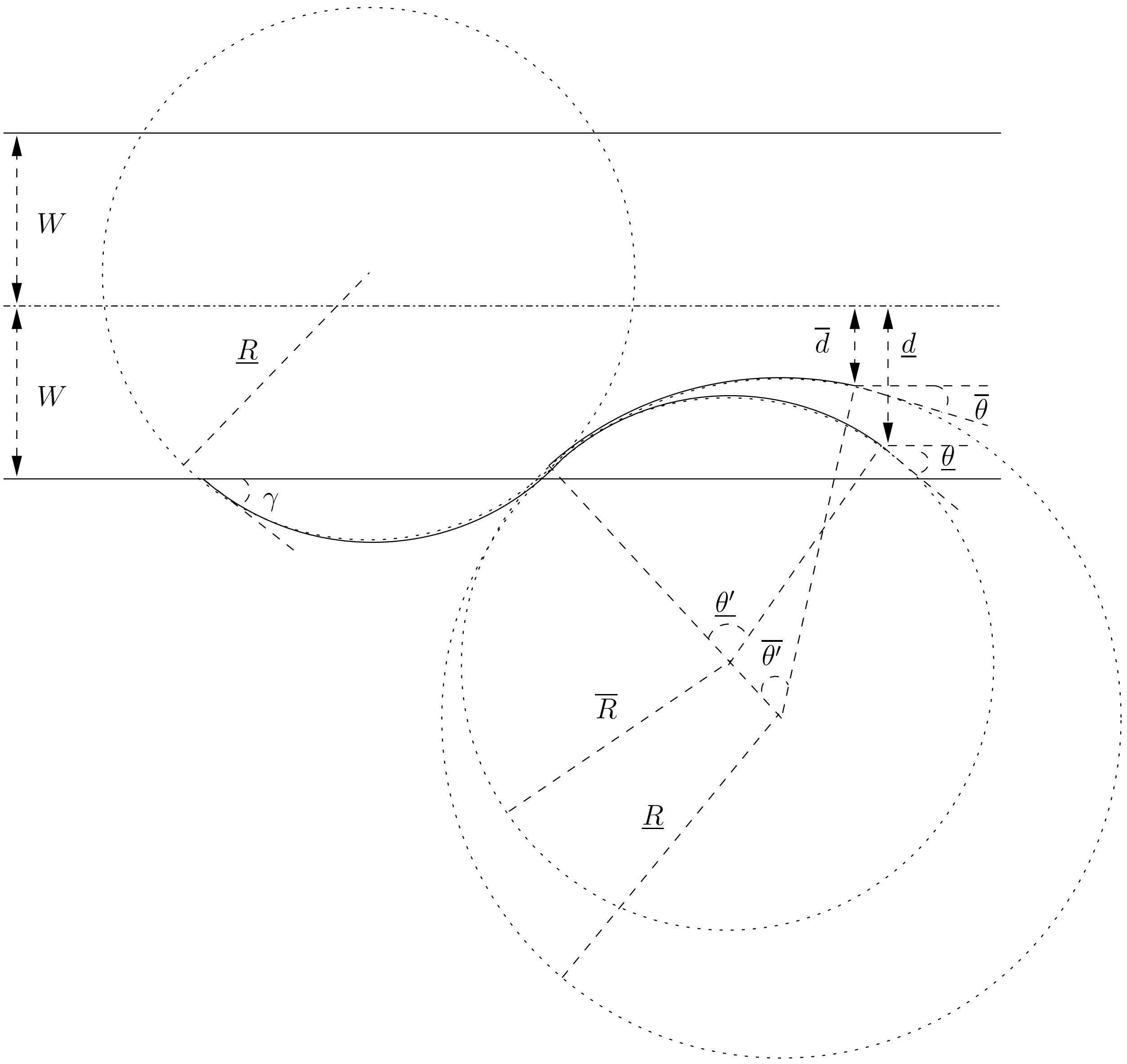


Figure 7.14: Control strategy for recovery from a bad initial condition

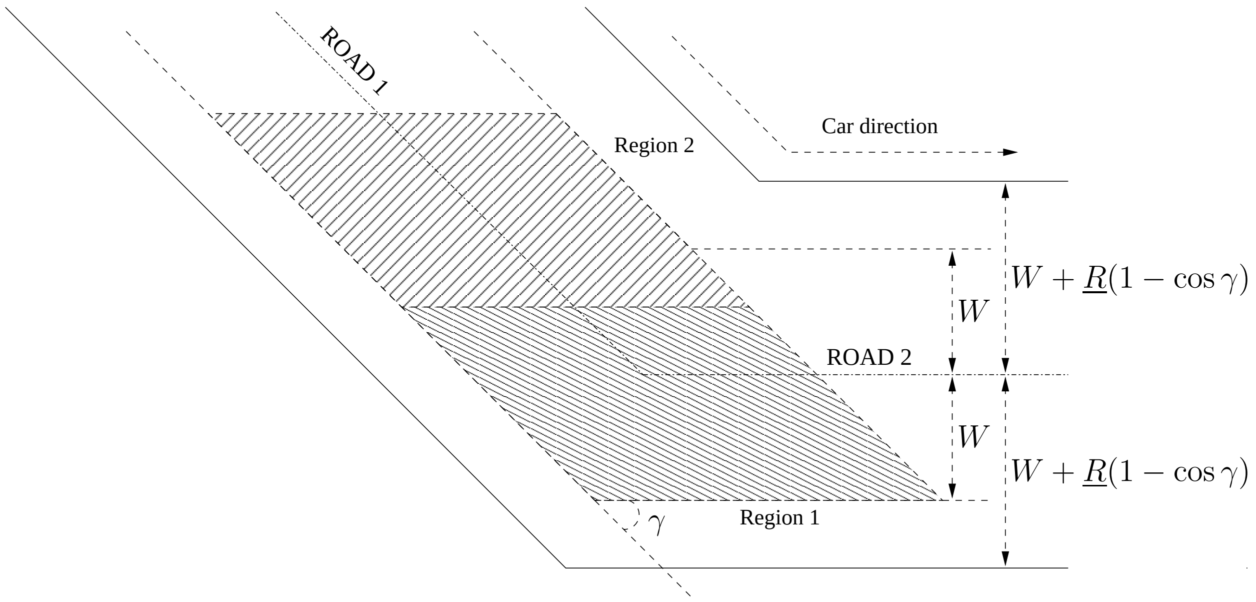

ROAD 1
Region 2
Car direction
$W + \underline{R}(1-\cos\gamma)$
W
ROAD 2
W
$W + \underline{R}(1-\cos\gamma)$
γ
Region 1

Figure 7.15: Controllable subset in a single lane

of fixed width by Theorem 7.2.1. Hence, the maximum distance that the car travels by the time we have $\underline{\theta} - \overline{\theta} = \gamma$ is $L = 2\overline{R} * \overline{\theta}'$, and by (7.13), this is given by

$$L = \frac{2\gamma\overline{R}\underline{R}}{\overline{R} - \underline{R}}.$$

It may be noted, however, that the car may recover quicker than the worst case described above, since it also moves the closer to the center of the lane as $\overline{d} < \underline{d}$ by Figure 7.14. $\Box$

We now consider how a car can be controlled at the intersection of two roads. This situation is illustrated in Figure 7.15. In this figure, Region 1 is the area such that, if the car is seen there at a renewal time, and the appropriate control corresponding to Road 1 is applied, then the car may move beyond the lower edge of Road 2 before it is seen next. Region 2 is the area in which the car is seen one iteration before it is seen in Region 1. In the following lemma, we show that a car can indeed be controlled at such an intersection as well.

Lemma 7.2.3 *If two roads intersect at an angle $\gamma \in [0, \pi/2]$ (cf. Figure 7.15), and each* *road is at least* $2(W+\underline{R}(1-\cos\gamma))$ *units wide with* $W=\underline{R}(1-\cos\beta(2\cos\alpha-1))$, $\alpha\leq\pi/3$, *and* $\beta=\cos^{-1}(2\cos\alpha-1)$, *then a car starting at a position* (d,θ) *with* $d+\underline{R}(1-\cos\theta)\leq W$ *on the first road, can be controlled so that it will satisfy this condition on the second road, within a distance of*

$$L=\frac{2\gamma\overline{R}\underline{R}}{\overline{R}-\underline{R}}.$$

Proof The basic idea in the control strategy is that when the car crosses the boundary between Regions 1 and 2 in Figure 7.15, we assume that the car is in Road 2 and send appropriate controls. When the car enters Region 1, we can have one of the following cases:

1. The car can be in the controllable subset $d+\underline{R}(1-\cos\theta)\leq W$ of Road 2 as well. In this case, we are done since feasibility with respect to the Road 2 can thereafter be maintained by Theorem 7.2.1.

2. The car can be outside the controllable subset, but still satisfy $d\leq W$ and $d+\underline{R}(1-\cos\theta)\leq d+\underline{R}(1-\cos(\theta+\gamma))$. In this case, the car can be brought back to the controllable subset in Road 2 by Theorem 7.2.2. In particular, since the maximum angle is γ when the car is a distance $d=W$, the car can be maintained within a road of width $2(W+\underline{R}(1-\cos\gamma))$ by the control strategy in Theorem 7.2.2.

3. The car can be outside the controllable subset with $d>W$ or $d+\underline{R}(1-\cos\theta)>d+\underline{R}(1-\cos(\theta+\gamma))$.

We now address the third case. There are two possible ways the car could end up in state 3: below the center of the second lane, or above it. Since the car is seen before it goes off the lower edge of the second lane, if the car is below the center of the second lane, then we will have $d<W$. In addition, we can use the *anti-clockwise* steering in Region 2 to ensure that $d+\underline{R}(1-\cos\theta)\leq d+\underline{R}(1-\cos(\theta+\gamma))$ as well. On the other hand, if the car is above the center of the second lane, then we can again use the *anti-clockwise* steering in Region 2

to ensure that the car ends up in cases 1 or 2. □

We can now prove that a car can be driven in a city road network.

Theorem 7.2.4 *In a road network with single-lane straight roads of length at least*

$$L = \frac{2\gamma \overline{R}\underline{R}}{\overline{R} - \underline{R}}.$$

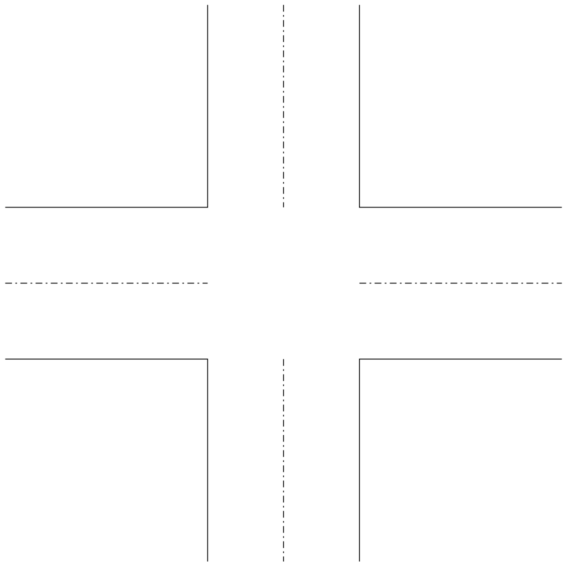
and lane-width at least $2(W + \underline{R}(1 - \cos\gamma))$ *with* $W = \underline{R}(1 - \cos\beta(2\cos\alpha - 1))$*,* $\alpha \leq \pi/3$*, and* $\beta = \cos^{-}1(2\cos\alpha - 1)$*, a car can be driven so that it stays within the corresponding lanes.*

Proof Since the roads in the network are straight, by Theorem 7.2.1, a car can be driven within a lane on such a road. Also, if we make the car normally drive in a sufficiently narrow sub-lane within each lane, then at intersections, a car can be properly controlled from a lane on an incoming road to a lane on the outgoing road by Lemma 7.2.3. Hence, a car can be driven so that it stays within specified lanes in the road network. □

7.3 System-wide safety and liveness guarantees

In this section, we use results from previous sections to prove system-wide safety and liveness guarantees in the traffic control testbed.

The Supervisor is the higher level control component that regulates the behavior of all the cars in the testbed as shown in Figure 5.1. In particular, the Supervisor for city traffic control also ensures that cars are scheduled so that there are no conflicts or gridlocks [28]. As described in Section 2.2, the basic approach is to map the road network into a discrete graph. A road is modeled as a path of bins, and the edges between bins show how cars can be moved between these bins. On the other hand, an intersection is modeled using the


(a) Intersection in a road network

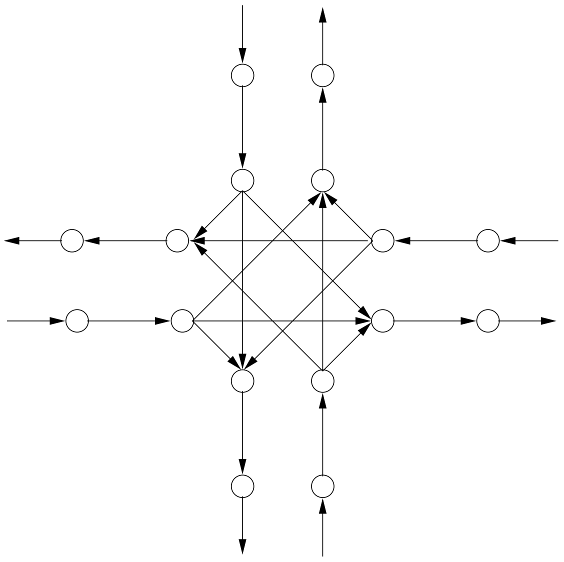

(b) Sub-graph modeling an intersection

Figure 7.16: Modeling intersections in road network discretization

pattern shown in Figure 7.16. In Figure 7.16(b), we see that there exist pairs of edges along which cars cannot be simultaneously without collisions. This is modeled by the concept of *edge conflicts*, whereby, cars cannot be simultaneously moved along conflicting edges.

Cars are initially assigned to respective bins in the road network graph. The traffic scheduling problem is then modeled as computing schedules for moving cars along edges so that there are no conflicts or gridlocks. As noted above, conflicts are eliminated by defining appropriate edge conflicts. On the other hand, gridlocks basically consist of an occupied cycle in graph, i.e., a cycle of bins in the graph occupied by cars such that each car needs to move to the next bin in the cycle.

Now we invoke the guaranteed property of the traffic scheduling algorithm from [28]:

Theorem 7.3.1 *If the directed graph $\mathcal{G}$ modeling a road network is such that each bin has either in-degree 1 or out-degree 1, the system has no occupied cycle initially, and cars exit the system after reaching their destinations, then the supervisory algorithm schedules the system so that all cars reach their destinations in finite time.*

Proof The main idea of the supervisory algorithm is to move cars along edges so that there are no conflicts, and no occupied cycles are formed. It can also be shown that, if there are no occupied cycles, and each bin in the road network graph has either in-degree 1 or out-degree 1, then it is always possible to move a car so that no occupied cycles are formed consequently. Since cars have finite routes to their destinations, it then follows that all cars reach their destinations in finite time. The detailed proof is presented in [28]. □

The next theorem shows that the discrete model requirement of at most one car in a bin at any given time, can indeed be enforced in continuous time.

Theorem 7.3.2 *In a road network satisfying the model of Theorem 7.3.1, cars can indeed be scheduled so that there is at most one car in a bin at any given time.*

Proof The scheduling algorithm in Theorem 7.3.1 ensures that there are at most two cars in a bin at a given time. This occurs when the next car enters a bin while the previous car has not completely left it. However, we can ensure that at most one car is in a bin at a given time by introducing a "virtual" car in front of each real car in the system. Specifically, the virtual car has the same route as the corresponding real car, and ensures that the bin in front of each real car is empty. □

We now prove main result in this chapter assuming the car model of Section 7.2.1.

Theorem 7.3.3 *In a road network with single-lane straight roads of length at least*

$$L = \frac{2\gamma \overline{R}\underline{R}}{\overline{R} - \underline{R}}.$$

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* $\beta = \cos^{-1}(2\cos\alpha - 1)$, *a set of cars can be driven along pre-specified routes without collisions (safety guarantee) or gridlocks (liveness guarantee).*

Proof Since the roads have single lanes and are sufficiently long and wide, by Theorem 7.2.4, a single car can be driven so that it stays within the corresponding lane. Also, since the network only has single lane roads, the corresponding road network graph satisfies the constraints of Theorem 7.3.1 as shown in Figure 7.16. Further, by Theorem 7.3.2, the cars can be scheduled so that there is at most one car in a bin at any given time, so that there are no collisions. Consequently, multiple cars can be operated so that there are no conflicts or gridlocks. $\square$

Chapter 8

CONCLUSIONS

We now summarize our main contributions in this thesis, and conclude with a vision of general purpose control for the future.

In this thesis, we have made the case that a key to the proliferation of networked control systems lies in solving the problem of software management in these systems. We have argued that, unlike earlier analog and digital control systems, the technological and theoretical aspects of design of these systems are not very well understood, and in particular, the complexities in these systems are mainly in the development and management of control software, besides the fundamental theoretical issues in decentralized control. Our thesis is that a well-designed middleware framework is a central ingredient in addressing these challenges.

Thus motivated, we have presented a middleware framework for networked control systems. Our efforts have been focused on a traffic control testbed, which we have used as a prototype to study networked control systems. Specifically, we have used an exploratory implementation of the testbed to understand the requirements of such systems in general, and motivate the design of Etherware, our middleware for networked control.

We have presented Etherware both from a middleware designer's perspective as well as an application developer's perspective. We have described the programming model, architecture, and mechanisms of Etherware, and described how to develop applications using them. In particular, we have described the Etherware based design and implementation of our traffic control testbed, and illustrated the design of distributed control loops through a detailed case study. This is one of the key contributions of this thesis.

Next, we have built on this to study the importance of safety and security issues in networked control systems. In particular, we have considered a system-wide perspective to safety, whereby exceptional incidents such as security breaches can be addressed by appropriate Strategies for Incident Response (SIR). In addition, we have also presented an Etherware based Control System Incident Response (CSIR) framework for incorporating SIRs into operational systems. Further, we have advocated and illustrated the principle of safety preserving security trade-offs, which states that high-level security overrides such as mandatory supervision, must not disable low-level safety features such as collision avoidance in traffic systems.

We have addressed the issue of how to establish system-wide guarantees. We have done this by proving system-wide safety and liveness guarantees in the testbed, based on various sub-system level guarantees. Specifically, we have proved that under a renewal task model based control model for cars, the traffic control testbed can be operated without car collisions or gridlocks. We have also presented accompanying scheduling policies for renewal task models as part of this analysis. This type of analysis can potentially serve as an example for other networked control systems of the future.

8.1 Vision for the future

We have presented a middleware framework that supports the development of manageable and reusable software for networked control systems. While our primary motivation has been the proliferation of networked control, our contributions, it is hoped, enable more than the systematic design of distributed control loops that characterize these systems. Our framework provides a basis for general purpose control.

By *general purpose control*, we mean the notion of using generic components as building blocks for control applications. These components could be standardized sensors, actuators, and controllers that incorporate both hardware and software capabilities, so that they can be seamlessly assembled or integrated into control loops dynamically. For instance, a building-wide distributed temperature control system could automatically detect and integrate new sensors and air-conditioning units, while still maintaining proper control of temperatures inside the building. The controllers of an automated car could add a new sensor device, such as a GPS unit, and exploit the additional sensor feedback available. Such cars could dynamically integrate into an autonomous traffic system and use advisories for traffic conditions. The possibilities are immense.

Our middleware framework provides a basis for this vision by supporting the necessary interoperability for independently developed control software. Our framework can be used to develop libraries of generic control software, which can then be used as high-level design primitives for implementing new control systems. In particular, such software can be bundled together with corresponding control components to form the building blocks for operational control systems. This lays the foundations for general purpose control, where users can essentially synthesize new working applications from such components.

References

[1] O. Mayer, *The Origins of Feedback Control.* Cambridge, Mass: M.I.T. Press, 1970.

[2] R. C. Dorf, *Modern control systems*, 5th ed. Addison-Wesley, 1989.

[3] J. C. Maxwell, "On governors," *Proc. of the Royal Society of London*, 1868.

[4] H. W. Bode, "Feedback - the history of an idea," *Selected Papers on Mathematical Trends in Control Theory*, pp. 106–123, 1964.

[5] M. D. Fagen, *A History of Engineering and Science on the Bell Systems.* Bell Telephone Laboratories, 1978.

[6] P. Varaiya and P. R. Kumar, *Stochastic Systems: Estimation, Identification and Adaptive Control.* Englewood Cliffs, N. J.: Prentice Hall, 1986.

[7] L. Sha and et. al., "Real time scheduling theory: A historical perspective," *Real-Time Systems*, vol. 28, pp. 101–155, 2004.

[8] R. Bannatyne, "Microcontrollers for the automobile," *Micro Control Journal*, 2003.

[9] R. J. Pehrson, "Software development for the boeing 777," tech. rep., The Boeing Company, 1996.

[10] C. M. Green, *Eli Whitney and the Birth of American Technology.* Addison Wesley Longman, Inc, 1956.

[11] H. P. Barendregt and E. Barendsen, "Introduction to lambda calculus," in *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*, 1988.

[12] J. von Neumann, "First draft of a report on the edvac," tech. rep., June 1945.

[13] C. H. Ferguson and C. R. Morris, *Computer Wars - The Fall of IBM and the Future of Global Technology.* Three Rivers Press, 1994.

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

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

[16] Microsoft Inc, *Microsoft .NET.*

[17] K. Etschberger, *Controller Area Network.* IXXAT Automation GmbH, Aug 2001.

[18] B. Albert and A. P. Jayasumana, *FDDI and FDDI-II: Architecture, Protocols, and Performance.* Artech House Publishers, Jan 1994.

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

[20] IBM, *Middleware is Everywhere.*

[21] Sun Microsystems, *Java 2 Platform, Enterprise Edition (J2EE).*

[22] OMG Inc, *CORBA Success Stories.*

[23] ZiLOG, *eZ80F91 Module product specification.*

[24] N. J. Ploplys, P. A. Kawka, and A. G. Alleyne, "Closed-loop control over wireless networks," *IEEE Control Systems Magazine*, vol. 24, pp. 58–71, June 2004.

[25] Crossbow Technology Inc, *MICA2 wireless measurement system datasheet.*

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

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

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

[29] "The Official Bluetooth Website,",". http://www.bluetooth.com/.

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

[31] Object Management Group, Inc, *Real-Time CORBA Specification Version 2.0*, Nov 2003.

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

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

[34] C. Heitmeyer and D. Mandrioli, Eds., *Formal Methods for Real-Time Computing*, vol. 5 of *Trends in Software.* John Wiley & Sons, 1996.

[35] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer Programming*, vol. 8, no. 1, pp. 231–274, 1987.

[36] D. Harel and E. Gery, "Executable object modeling with statecharts," *IEEE Computer*, vol. 30, no. 7, pp. 31–42, 1997.

[37] A. Coen-Porisini, M. Pradella, M. Rossi, and D. Mandrioli, "A formal approach for designing corba based applications," *ACM Trans. on Software Engineering and Methodology*, vol. 12, April 2003.

[38] F. Marotta, A. Morzenti, and D. Mandrioli, "Modeling and analyzing real-time corba and supervision & control framework and applications," in *Proceedings of the 21st Intl. Conf. on Distributed Computing Systems*, April 2001.

[39] C. Ghezzi, D. Mandrioli, and A. Morzenti, "Trio - a logic language for executable specifications of real-time systems," *Journal of Systems and Software*, vol. 12, May 1990.

[40] R. Bastide and P. Palanque, "Cooperative objects: A concurrent, petri net based object-oriented language," in *IEEE-SMC*, 1993.

[41] R. Bastide, O. Sy, D. Navarre, and P. Palanque, "A formal specification of the corba event service," in *Formal Methods for Open Object-Based Distributed Systems*, vol. 4, 2000, pp. 371–396.

[42] N. G. Leveson, J. D. Reese, and M. Heimdahl, "Spectrm: A cad system for digital automation," in *Proceedings of the 17th Digital Avionics Systems Conf.*, Nov 1998.

[43] G. Lee, J. Howard, and P. Anderson, "Safety-critical requirements specification and analysis using spectrm," in *Proceedings of the 2nd Meeting of the US Software System Safety Working Group*, Feb 2002.

[44] L. Wills, S. Kannan, S. Sander, M. Guler, B. Heck, J. V. R. Prasad, D. Schrage, and G. Vachtsevanos, "An open platform for reconfigurable control," *IEEE Control Systems Magazine*, June 2001.

[45] A. Gokhale, D. C. Schmidt, B. Natarajan, and N. Wang, "Applying model-integrated computing to component middleware and enterprise applications," *Communications of the ACM*, vol. 45, no. 10, pp. 65–70, 2002.

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

[47] A. Traub and R. Schraft, "An object-oriented realtime framework for distributed control systems," in *Proceedings of the IEEE Conf. on Robotics and Automation*, Detroit, Mi, May 1999.

[48] P. Lutz, W. Sperling, D. Fichtner, and R. Mackay, "Osaca - the vendor neutral control architecture," in *Proceedings of the European Conf. on Integration in Manufacturing*, Dresden, Germany, Sept 1997, pp. 247–256.

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

[50] "J-Sim Home Page,",". http://www.j-sim.org.

[51] A. Sobeih, W.-P. Chen, J. C. Hou, L.-C. Kung, N. Li, H. Lim, and H. Zhang, "J-Sim: a simulation and emulation environment for wireless sensor networks," in *Proceedings of the 38th Annual Simulation Symposium*, San Diego, CA, April 2005.

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

[53] "Java 2 Platform, Standard Edition (J2SE),",". http://java.sun.com/j2se/.

[54] A. Silberschatz, P. Galvin, and G. Gagne, *Applied Operating System Concepts*, first ed. John Wiley and Sons Inc, 2000.

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

[56] S. Graham and P. R. Kumar, "Time in general-purpose control systems: The control time protocol and an experimental evaluation," in *Proceedings of the 43rd IEEE Conf. on Decision and Control*, Dec 2004.

[57] "NTP: The Network Time Protocol,",". http://www.ntp.org/.

[58] G. Baliga, S. Graham, L. Sha, and P. R. Kumar, "Etherware: Domainware for wireless control networks," in *Proceedings of the 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, May 2004, pp. 155–162.

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

[60] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott, *Maude Manual (Version 2.1)*, Mar 2004.

[61] "ROFES: Real-Time CORBA for embedded systems,",". http://www.lfbs.rwth-aachen.de/users/stefan/rofes/.

[62] "JacORB,",". http://www.jacorb.org.

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

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

[65] S. Kowshik, G. Baliga, S. Graham, and L. Sha, "Co-design based approach to improve robustness in networked control systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, Yokohama, Japan, June 2005.

[66] W. Zhang, "Stability analysis of networked control systems," Ph.D. dissertation, Case Western Reserve University, 2001.

[67] J. Nilsson, "Real-time control systems with delays," Ph.D. dissertation, Lund Institute of Technology, 1998.

[68] L. Sha, "Using simplicity to control complexity," *IEEE Software*, July/August 2001.

[69] S. Poledna, *Fault Tolerant Real-time Systems.* Kluwer Academic Publishers, Nov 1995.

[70] R. J. Patton, "Fault tolerant control systems: The 1997 situation," in *IFAC Symposium on Fault Detection Supervision and Safety for Technical Processes*, vol. 3, August 1997, pp. 1033–1054.

[71] F. Cristian, B. Dancey, and J. Dehn, "Fault-tolerance in air traffic control systems," *ACM Transactions on Computer Systems*, vol. 14, pp. 265–286, Aug 1996.

[72] L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving dependable real-time systems," in *Proceedings of the Aerospace Applications Conference*, vol. 1, 1996, pp. 335 – 346.

[73] S. Pertet and P. Narasimhan, "Proactive recovery in distributed corba applications," in *IEEE Conference on Dependable Systems and Networks*, Florence, Italy, June 2004.

[74] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: analysis, module and applications," in *Proc. of the 25th Int. Symposium on Fault-Tolerant Computing*, Pasadena, CA, June 1995.

[75] G. Candea and A. Fox, "Crash-only software," in *Proc. 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, HI, May 2003.

[76] G. Candea, J. Cutler, and A. Fox, "Improving availability with recursive microreboots: A soft-state system case study," vol. 56, March 2004.

[77] J. C. Cunha, R. Maia, M. Rela, and J. Silva, "A study on the failure models in feedback control systems," Goteborg, Sweden, July 2001.

[78] J. C. Cunha and M. Rela, "On the use of disaster prediction for failure-tolerance in feedback control systems," Washington D.C., USA, June 2002.

[79] J. Cunha, "Low-cost fault tolerance for continuous real-time control systems," Ph.D. dissertation, Faculdade de Ciencias e Tecnologia, Universidade de Coimbra, July 2003.

[80] C.-T. Chen, *Linear System Theory and Design*, 3rd ed. Oxford University Press, 1999.

[81] M. Bishop, *Computer Security: Art and Science.* Addison Wesley Professional, 2003.

[82] H.-J. Schuetz, "Collision avoidance for sensorless cars in a networked control system," tech. rep., Coordinated Science Lab, Univ. of Illinois at Urbana-Champaign, and Darmstadt University of Technology, 2005.

[83] "IT Convergence Lab, CSL, UIUC,",". http://decision.csl.uiuc.edu/∼testbed/.

[84] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," in *Journal of the ACM*, vol. 20, New York, NY, USA, ACM Press, 1973, pp. 46–61.

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

[86] M. Y. Chan and F. Y. L. Chin, "Schedulers for larger classes of pinwheel instances," *Algorithmica*, vol. 9, pp. 425–462, 1993.

[87] M. Y. Chan and F. Y. L. Chin, "General schedulers for the pinwheel problem based on double-integer reduction," *IEEE Transactions on Computers*, vol. 41, no. 6, pp. 755–768, 1992.

VITA

Girish Baliga was born in 1979 in Bangalore, India. He grew up in Bangalore, where he also did his basic schooling. He then pursued his undergraduate education at the National Institute of Technology, Surathkal, India, and graduated with a B.E. in Computer Engineering in 2000. He received an M.S. in Computer Science in 2002, and M.S. in Mathematics in 2004, both from the University of Illinois at Urbana-Champaign. After the completion of his PhD, Girish plans to move to the Silicon Valley to join Google as a software engineer.