

ISSUES IN THE CONVERGENCE OF CONTROL
WITH COMMUNICATION AND COMPUTATION

BY

SCOTT R. GRAHAM

B.S., Brigham Young University, 1993
M.S., Air Force Institute of Technology, 1999

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Any opinions, findings, conclusions, or recommendations
expressed in this publication are those of the author and
do not necessarily reflect the views of the United States
Air Force, Department of Defense, or the U.S. Government

Urbana, Illinois

© 2004 by Scott R. Graham. All rights reserved

ABSTRACT

We anticipate the next wave in the information technology revolution to be the convergence of control, i.e., sensing and actuation, with communication and computing. This dissertation addresses the broad set of issues that we believe to be important to the design, implementation, and proliferation of such systems. In particular, we expound on the topics of the architecture of such systems, methodologies for design, distributed time, services, and middleware. We describe our design and implementation of a testbed.

To my family

ACKNOWLEDGMENTS

Research in new areas requires great insights, often from different perspectives. I would like to thank the many wonderful people involved in my research who have provided such insights and perspectives. In particular, I would like to thank Professor P.R. Kumar, who served as my thesis advisor, committee chair, coach, and frequent cheerleader. I would also like to thank my thesis committee members – Professors Lui Sha, Peter Sauer, and Yi Ma – who have each contributed great thoughts and perspectives. I appreciate my research colleagues – Girish Baliga, Kun Huang, and others in the research group – who have provided friendship, generous support, and valuable advice. I am grateful for the support of the U.S. Air Force, through the AFIT/CI program. Lastly, and most important of all, I wish to thank my beautiful wife for her solid strength throughout this and many other experiences, and my extraordinary children who still seem to love me in spite of it all.

TABLE OF CONTENTS

LIST OF TABLES	xiii
LIST OF FIGURES	xiv
CHAPTER 1 INTRODUCTION	1
1.1 The Possible Next Phase of the IT Revolution	1
1.1.1 A preview of the end result	2
CHAPTER 2 CONSIDERATIONS INFORMING OUR APPROACH	6
2.1 Systems Design is the Establishment of Interfaces	6
2.2 Reliable Design in an Ever Changing Environment	7
2.2.1 Reliability and dependencies	7
2.2.2 Design for safety	9
2.2.3 Reliability in testbed design	10
2.2.3.1 Vision reliability	10
2.2.3.2 Controller reliability	11
2.2.3.3 Reliability through rapid restart	11
2.3 Proliferation of General Purpose Systems	12
2.4 Importance of Abstractions	13
2.4.1 The OSI abstractions in networking	13
2.4.2 The von Neumann serial computation abstraction	14
2.4.3 The Shannon separation of source/channel in digital communication	15
2.4.4 The plant, controller, and estimator abstractions in control systems	15
2.4.5 Importance of layering	15
2.5 The Shifting of Bottlenecks	16
2.5.1 The performance bottleneck	17

2.5.2	The design time bottleneck	17
2.6	The Need to Ensure Evolvability of Design	18
2.7	Convergence Towards a More Holistic Theory	19
2.8	The Necessity for Design Experiments on a Testbed	20
CHAPTER 3 THE CONVERGENCE LABORATORY		21
3.1	The Convergence Testbed	21
3.1.1	Testbed functionality	21
3.2	Description of the Testbed	23
3.2.1	Sensors and actuators	23
3.2.2	Controller	23
3.2.3	Supervisor	24
CHAPTER 4 THE CHALLENGES		25
4.1	Control Challenges and General Solutions	25
4.2	Communication Challenges and General Solutions	27
4.3	Computation Challenges and General Solutions	29
4.4	Distributed Challenges and General Solutions	30
4.5	General Purpose Challenges and Solutions	31
4.6	Concluding Remarks	33
CHAPTER 5 DESIGN PRINCIPLES FOR NETWORKED CONTROL		34
5.1	Local Temporal Autonomy	34
5.2	Stability	35
5.3	Dependence Reduction	35
5.3.1	Initialization sequence	35
5.3.2	Process dependence	36
5.3.3	Communication deadlock	36
5.3.4	Temporal dependence	36
5.4	Reliable Opportunism	37
5.5	Explicit Assumptions	37
5.6	Explicit Knowledge of Time and Delay	37
5.7	Virtual Collocation	38
5.8	Location Independence	38

5.9	Semantic Addressing	39
5.10	Migration for Self-Optimization and Reliability	39
5.11	Implementation of the Principles of Design	40
CHAPTER 6 A DESIGN PATTERN FOR INCREMENTAL EVOLUTION . .		41
6.1	The Need for the Incremental Evolution of Design	41
6.2	The Incremental Evolution Design Pattern	43
CHAPTER 7 DESIGN BASED ON LOCAL TEMPORAL AUTONOMY		47
7.1	Robustness to Blackout	47
7.1.1	Actuator autonomy	47
7.1.2	Controller autonomy	48
7.1.3	Vision system autonomy	49
7.2	Controller Blackouts	49
7.3	Capabilities Derived from Local Temporal Autonomy	50
CHAPTER 8 ARCHITECTURAL FRAMEWORK: MIDDLEWARE		51
8.1	Etherware	51
8.1.1	Active middleware	52
8.1.2	Event based communication	53
8.1.3	Invocation	53
8.1.4	Heterogeneous middleware	54
8.2	Core Services	55
8.2.1	Message passing	55
8.2.2	Semantic addressing	56
8.2.3	Time translation	56
CHAPTER 9 TIME: THE IMPORTANCE OF KNOWING IT		57
9.1	Introduction	57
9.2	Effect of Delay on System Performance	58
9.2.1	Effect of unknown delay on system performance: Experiment 1	58
9.2.2	State estimation	60
9.2.3	Effect of known delay on system performance: Experiment 2	61
9.3	Determining Overall Closed Loop System Delay	62

9.3.1	Using stability to determine delay: Experiment 3	63
9.3.2	Offline analysis of a step input: Experiment 4	65
9.4	Estimation of Plant Delay by Temporal Alignment	66
9.4.1	Online identification of plant delay: Experiment 5	67
9.5	Concluding Remarks	71
CHAPTER 10 THE CONTROL TIME PROTOCOL		72
10.1	Challenges of Time in Distributed Systems	72
10.1.1	Clock fundamentals	72
10.1.2	Simple clock model	73
10.1.3	Representation	73
10.2	Determining Offset with Communication Delay	74
10.2.1	Indeterminate offset in the presence of asymmetric delay	74
10.2.2	Least squares estimation of <i>skew</i> and <i>offset</i>	77
10.2.3	Numerical issues in recursive least squares estimation	78
10.2.4	Windowed least squares	80
10.2.5	Offset bounds	81
10.3	Time Translation Instead of Synchronization	81
10.3.1	Synchronization using the Network Time Protocol (NTP)	81
10.3.2	Control issues when using NTP	83
10.3.3	Control time protocol	84
10.3.4	Differences between NTP and CTP	85
10.3.5	Measuring the performance of NTP using the Control Time Protocol	86
10.4	On-Line Measurement of System Delay	86
CHAPTER 11 TESTBED DESIGN		88
11.1	Interfaces	88
11.1.1	Controller to actuator interface	88
11.1.2	Sensor to controller	90
11.1.2.1	StateEstimator	92
11.1.3	Actuator to sensor interface	93
11.2	Interface Layering	94
11.2.1	LocalPlanner	95

11.2.1.1	LocalPlannerStateEstimator	95
11.2.1.2	LocalPlanner to Tracker interface	96
11.2.2	Supervisor	96
11.2.2.1	SupervisorStateEstimator	97
11.3	Self Similar Hierarchy	97
11.4	Local Temporal Autonomy	98
11.4.1	Sensor autonomy	98
11.4.2	FeedbackServer autonomy	98
11.4.3	Controller autonomy	99
11.4.4	Actuator autonomy	99
11.5	Algorithms	99
11.5.1	Model predictive control	99
11.5.2	Actuator	100
11.5.3	FeedbackServer	100
11.5.4	VisionServer	100
11.5.5	LocalPlanner	101
11.5.6	Supervisor	101
11.6	Evolution View	102
11.6.1	Testbed version 1	102
11.6.2	Testbed version 2	103
11.6.3	Testbed version 3	106
11.6.4	Testbed version 4	106
11.6.5	Testbed version 5	107
11.6.6	Testbed version 6	108
11.6.7	Testbed version 7	109
11.6.8	Testbed version 8	110
11.6.9	Testbed version 9	110
11.7	Designing with Virtual Collocation	111
11.8	Methods for Reliability	112
11.9	Concluding Remarks	114

CHAPTER 12 A SKETCH OF A POSSIBLE LARGE SCALE APPLICATION FOR POWER DEMAND REGULATION	115
12.1 Background	115
12.2 Home and Industry Participation in Power Regulation	116
12.3 Peripheral Functionality	118
12.4 Incentive Pricing	121
12.5 Local Autonomy	121
12.6 Design of a Power Demand Response System	123
12.7 Concluding Remarks	126
CHAPTER 13 CONCLUDING REMARKS	127
APPENDIX A GLOSSARY OF TERMS	128
APPENDIX B ETHERWARE - CONTROL ORIENTED MIDDLEWARE ...	134
B.1 Etherware	135
B.2 Using Etherware	136
B.2.1 Collocation abstraction	136
B.2.2 Evolution	136
B.2.3 Migration	136
B.2.4 Containment of cascading failure	137
B.3 Application Design within Etherware	138
B.3.1 Design principles	138
B.3.1.1 Stability challenges	138
B.3.1.2 Error models	138
B.3.1.3 Component failures	138
B.3.1.4 Invoked components	139
B.3.1.5 Interactions	139
B.4 Design Abstractions and Philosophy	140
B.4.1 Event-based application	140
B.4.2 Local scheduling of events	140
B.4.3 Location independence	140
B.5 Core Etherware Services	141
B.5.1 Kernel	141

B.5.2	ProfileRegistry	142
B.5.3	GlobalEventBus	142
B.5.4	NetworkTimeService	143
B.6	Configuration and Startup	144
B.6.1	Initialization of a single Etherware node	144
B.6.1.1	Error handling through exceptions	145
B.6.2	Initialization of Etherware on multiple nodes	146
B.7	Steps of Design Using Etherware	147
B.8	Steps of Implementation Using Etherware	149
B.9	Steps of Execution Using Etherware	150
B.10	Conclusion	150
APPENDIX C VISION SYSTEM		151
C.1	Cameras	151
C.2	VisionServers	152
C.2.1	Performance	154
C.3	Other Positioning Possibilities	154
C.4	Reducing Functional Dependence on the Sensor	155
C.4.1	State estimation	156
C.4.2	Lighting dependence	156
C.4.3	Image format and transformations	157
C.4.4	Color space	158
C.4.5	Reducing patch dependence	160
C.4.6	From patches to positions: Reliable coding	162
C.5	Robust Vision through Dependency Reduction	164
REFERENCES		166
VITA		169

LIST OF TABLES

Table	Page
4.1 Control challenges and solutions.	26
4.2 Communication challenges and solutions.	27
4.3 Computation challenges and solutions.	29
4.4 Distributed challenges and solutions.	30
4.5 General purpose challenges and solutions.	32
9.1 System delay observed under stop/go control with time stamps.	66
10.1 Two different offset/delay combinations which produce identical time stamps.	76

LIST OF FIGURES

Figure	Page
2.1 A StateEstimator separating sensor and controller can reduce the execution and timing dependencies between them.	8
2.2 The Simplex switching architecture.	10
2.3 Deviation from desired trajectory under restart.	12
3.1 The Convergence Laboratory.	22
6.1 Incremental Evolution architecture.	43
6.2 Kalman filter inserted via Incremental Evolution.	44
6.3 Planning as Incremental Evolution.	46
9.1 Trajectory of car with no additional delay.	59
9.2 Trajectory of car with 300-ms additional delay.	59
9.3 Trajectory of car with 800-ms additional delay.	62
9.4 Determining delay by observing onset of instability in Experiment 1.	65
9.5 Estimated trajectory and the observed trajectory.	68
9.6 The summed difference measure attains a minimum at $r = 194$ ms.	69
9.7 Combined observed rotations at different time instants for both cameras. Left: the combined observed rotations at each camera. Right: Shifting the curve of the combined observed rotation of the real camera backwards by the detected time shift.	70
10.1 Message exchange for offset determination.	75
10.2 Ping message exchange for offset determination.	77
10.3 Jitter in offsets between two NTP enabled machines, studied through the Control Time Protocol.	83
10.4 Linear skew with infrequent step changes.	86

11.1	Initial components.	89
11.2	Self similar hierarchy of control.	98
11.3	Closing the loop.	103
11.4	Inserting a FeedbackServer.	103
11.5	Inserting a StateEstimator.	104
11.6	Inserting a LocalPlanner.	105
11.7	Inserting a StateEstimator for the LocalPlanner.	105
11.8	Upgrading the StateEstimator using the Incremental Evolution pattern.	107
11.9	Upgrading the Tracker using the Incremental Evolution pattern.	108
11.10	Adding another sensor.	109
11.11	Incorporating additional cars.	109
11.12	Inserting a Supervisor and its StateEstimator.	110
12.1	Basic configuration.	117
12.2	Communicating with a personal computer and/or the utility meter.	119
12.3	Incorporating control of other appliances.	120
C.1	RGB Color Space.	159
C.2	HSV and HLS color spaces.	159
C.3	Color patch layout.	161
C.4	Improved reliability when missed color patches are tolerated.	164

CHAPTER 1

INTRODUCTION

1.1 The Possible Next Phase of the IT Revolution

Over the past two decades we have seen the convergence and growth of communication and computation, which has given us the Internet with over 150 million hosts [1] that provide us the ability to passively exchange information in the form of email or to browse each other's web pages.

We anticipate that the next phase of the information technology revolution will be the ability to actively interact with the environment and alter it, by sensing and acting on it. It will be achieved by interconnecting sensors and actuators with computation elements, and providing all with communication capability. We believe it will lead to the convergence of control with computation and communication. By analogy to general purpose computing [2], we will refer to control systems created by hooking up computers with sensors and actuators over communication networks as general purpose control systems. Also by analogy with general purpose computing, we envision general purpose control to have widespread usage, and not necessarily by experts. Just as general purpose computing may not be applicable in very high performance problems such as scientific computing, so also general purpose control may not be applicable in environments where only a highly coupled solution that is highly customized in hardware and software for a niche application will meet the stringent requirements.

Two technological trends making general purpose control feasible are the growth in embedded computers and wireless networking. About 98% of all microprocessors sold are embedded, and their percentage is growing [3]. Currently these embedded devices function in an isolated

way and are not significantly interconnected. However, we may be on the cusp of a wireless revolution. Wi-Fi (IEEE 802.11x) has experienced double-digit growth since 2000 [4], and is now installed as a default on several makes of computers. Lower cost wireless connectivity is possible with Bluetooth available at a \$6 per chipset cost to manufactures [5]. Extrapolating these trends in wireless communication, we can envision a time, not far off, in which wireless connectivity is a commodity. With each embedded device functioning as a sensor or an actuator, and each wirelessly connected with others, the future could well see orchestras of sensors and actuators playing over the ether in vast interconnected control systems. Though our vision above is aimed at coarse granularity general purpose systems, even at the opposite and fine granularity end, the Berkeley Motes [6] already provide a combination of sensing, wireless communication, and computation, all in a package with a small spatial footprint and low energy usage. While there is much current interest in sensor networks, we believe that actuation based on sensing is inevitable and that these too will lead to sensor-actuator (i.e., control) networks at the low granularity form factor end.

In short, we anticipate the convergence of control with communication and computation [7]. This dissertation addresses the issues of what software, specifically middleware, will be the infrastructure for supporting these systems, how these systems will interoperate, what the obstacles and enablers of proliferation are, and what is the design process for such systems. We argue that the development framework, operational architecture, and middleware infrastructure support of these systems will play a critical role in their operation and proliferation, and propose such a software solution, framework, and architecture. We describe a testbed developed to identify the practically relevant research issues in this area, propose solutions to some of the identified problems, and exhibit the solutions in situ. The end result of our development efforts serves as an example of such a converged general purpose control system.

1.1.1 A preview of the end result

To illustrate our vision of the end systems that the convergence of control with communication and computation can lead to, and to motivate and make concrete some of the ensuing discussion, we provide a brief preview of our testbed and its capabilities.

The testbed features two cameras together observing several radio controlled (RC) cars which are each individually controlled by laptops and other computers that can be flexibly used for vision processing, planning or scheduling, all connected by an ad hoc wireless network interfaced with a wired ethernet.

The system is expandable at will. More cars can be added or some removed, more sensors of any kind can be added or some removed, more sensors of any kind can be added or removed, more computational resources can be added or removed, and communication infrastructure can be added or removed. Thus the entire hardware can be arbitrarily altered.

We provide a middleware (software residing between operating system and application), called Etherware, which provides the system designer the luxurious abstraction of a collocated system where all information can be envisioned as centrally available. The system designer can thus proceed with her design as though it were a traditional control system, and focus, say, on the algorithmic or functional code for planning, scheduling, control, adaptation, estimation, or identification. The middleware hides all distracting details such as IP addresses, network protocols, and computational resources from the designer. All sensor and other information is automatically time-stamped, with clock alignment and translation issues automatically taken care of by the middleware. The designer can refer to services by content, and need not worry about details of replacing or upgrading a camera or a node with one IP address by another. She can ask for information from sensors on a regular basis, or as information push whenever updates are available, or as pull on demand by an information consumer. Hardware can be added or removed even while the system is running, and the designer need not deal with consequences of arcane, but overwhelming if neglected, issues like starting up computers in any order chosen.

The middleware also provides self-optimization capabilities. For example, whether to process all the pixels from the processor at a certain node, thus stressing its computational resource, or to transfer them to another node, thus stressing the communication network, is a mundane optimization issue whose solution depends on the power of the processor at a node and the current communication traffic load, and is a low level decision she need not deal with. The middleware can automatically migrate the computation from node to node. For example, if a planning algorithm requires intensive data which is available in one location, the algorithm can automatically migrate and run on that node. The designer need only specify rules or algo-

rithms for such migration, leaving the automatic implementation while a system is running to the middleware, thus avoiding static Y2K type solutions, as well as enabling designers to work at a high level in the space of algorithms that designers like best.

We also provide a design process for such systems motivated by the fact that large systems of this sort are always in a state of flux. New functionality is always being added, and the goal of the design process is to ensure that design can evolve in the future without sacrificing reliability. New functionality may be inserted reliably by following a design pattern we call Incremental Evolution.

We provide for reliability, which is emerging as the key performance measure in modern systems, at both the system and the software level. The individual subsystems, from the control designer's perspective, are loosely coupled in that, for example, upon failure of a vision system for a brief period, a Kalman filter continues to provide surrogate estimates of the positions of cars though with decreasing precision, but such that until the vision system comes back on line, the entire system continues to function. We also minimize dependencies within the software to enable automatic restart of failed components in a seamless way.

The design allows generality, as in general purpose computing, and the system can be used as a traffic coordination and transportation system for commuters, or as a zero-sum game situation where a manually driven evader is pursued by a pursuers in a formation, or to demonstrate repositioning and parking maneuvers of a simultaneous set of cars.¹ Functionality such as system identification or adaptive calibration can be added to an existent design relatively easily without a total redesign of interfaces and codes.

The abstractions supported in the design methodology and the architecture of the system are such that it is, to a large extent, context and application independent. Thus, by replacing only the algorithm code for a car control law with one for a thermostat, the system can function as a building-wide temperature control system rather than a vertically integrated automated traffic control system.

Thus, our vision is that of a general purpose control system with shortened design cycle-time, and whose designs can evolve, with the middleware allowing the virtual collocation abstraction

¹Movies are available at <http://decision.csl.uiuc.edu/~testbed>.

that control designers are immensely experienced at. The middleware supports enhanced performance recovery, through automatic services such as time-stamping of information that allows the control designers to specify how the information is to be optimally used as a function of its actual delay latency, and through supporting automatic migration to best exploit available communication and computation resources. At the same time, the design process allows for code reuse rather than rewrite, enhancing reliability. Through minimizing dependencies the system also supports enhanced reliability measures such as automatic restart of failed components, all in a seamless way. We believe that, in toto, the topics addressed in this dissertation will ensure the proliferation of general purpose convergence of control with communication and computation, the next frontier in the information technology revolution.

CHAPTER 2

CONSIDERATIONS INFORMING OUR APPROACH

One of our theses is that the following considerations are important for the convergence of control with computing. These considerations thus inform and drive our proposed solutions.

2.1 Systems Design is the Establishment of Interfaces

We argue that the essence of systems design is the establishment of interfaces. Within the boundaries of an interface, developers can employ theory and tools to create required performance. When the interfaces are complex or ill-defined, the complexity of interactions increases. We advocate simplicity in interfaces, relegating complexity to abide within well-defined boundaries. This approach is similar to the object oriented movement of the software engineering community [8].

As any domain matures, the knowledge acquired can and ought to be distilled into common solutions. A reduction in unsolved problems is clearly an improvement and allows focus on the new problems. Thus the art of design matures in stages towards the science of design. Throughout this dissertation, we present several “design patterns” [9], which we hope will facilitate the proliferation of general purpose control.

2.2 Reliable Design in an Ever Changing Environment

Mass production typically separates design from production. Design is completed before components are produced. While this model is very efficient for producing vast quantities of a product, it is not well-suited for general purpose control for several reasons.

First, new technological capabilities will become available during the development of the system. Users and designers must be able to incorporate suitable new technologies into the system rapidly.

Second, the requirements stipulated for general purpose control systems will themselves inevitably change as a system is built. With some of the components in place, designers and users will discover unforeseen capabilities and will want to take advantage of them to add value at relatively small additional cost.

Third, unlike building construction, general purpose control systems may be able to provide a minimal operational capability with few initial components. Early deployment of this capability can produce early revenue, or at least early justification for subsequent development, thus enabling proliferation.

Indeed we believe that system designs will always be in a continuous state of evolution. We therefore feel a spiral model of development [10] is much better suited to the domain of general purpose control. In this model, a system is designed, built, and tested in small increments, where each increment provides new capability.

A system undergoing continuous evolution will continuously struggle with the problem of system integration. As system integration is currently a substantial bottleneck, both in cost and schedule, we must search for and provide methods to facilitate system integration. Among the promising possibilities are efforts to reduce dependencies and improve reliability at all system levels.

2.2.1 Reliability and dependencies

We believe that two fundamental issues driving the choice of a design process are the need to reduce development time and the need to enhance reliability. Reliability is, in fact, a primary

performance measure. A primary advantage of development with a frozen design is that with only one design, careful attention can be given to safety and reliability. When the design is itself in flux, the task of analyzing reliability can quickly become intractable if the process of design itself is not properly designed.

To ensure reliability, we must begin addressing it from the lowest possible levels. This creates a hierarchy of reliability, with components carefully created to ensure reliable operation in spite of failures of other modules. Thus, reliability is provided by having robust components at every level, with some degree of autonomy to operate, in spite of the failure of other system components.

An example is illustrative. A controller operating in a closed loop uses sensor data as feedback in order to compute new controls. If the controller waits for a new data sample before computing new commands, then it is dependent on the communication channel and the sensor for its continued operation. As the controller cannot issue commands while waiting for sensor updates, it has effectively failed. We call this *execution dependence*. More generally, execution dependence exists when the failure of one module causes another module to fail.

In this example, the controller is also dependent on the sensor and communication channel for timing. Late data causes the controller to provide late controls. This represents a *timing dependence* of the controller on the sensor and communication channel.

To remove the execution and timing dependencies of the controller on the inherently unreliable communication channel, we interpose a module between them, shown in Figure 2.1.

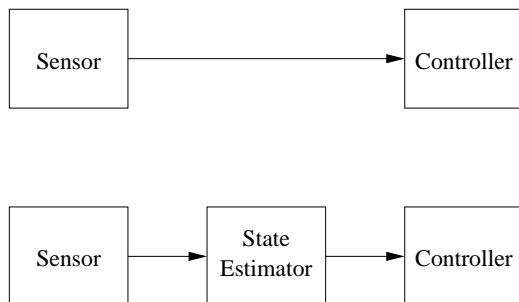


Figure 2.1 A StateEstimator separating sensor and controller can reduce the execution and timing dependencies between them.

This module, which we call a StateEstimator, accepts aperiodic sensor data as input, and provides continuous or periodic outputs to the controller. Given past controls in addition to sensor data, as in the Kalman filter [11], the StateEstimator can provide estimates of the state even in the complete absence of sensor information for some period of time. Thus, the controller update is made independent of the feedback delay and jitter. Because the controller can continue operation, even when no sensor feedback arrives, it can reliably function independent of the behavior of the sensor communication.

Architecturally, the StateEstimator represents a buffer between dissimilar components. That is, the controller prefers a periodic signal, but the communication channel can at best provide a perturbed periodic signal. With the StateEstimator serving as a mediating interface between incongruent models, the sensor, communication channel, and controller are free to operate and evolve as needed, both at run-time and over the system life cycle. This reduces dependencies and improves reliability, potentially reducing system integration problems.

2.2.2 Design for safety

Similarly, we must design for safety beginning at the lowest level of control, the actuator, and continue up the control hierarchy. The actuator must have fail-safe properties built-in, such that failures of any of the higher level controllers can be tolerated, at least to some degree. As an example, in the testbed, the cars are designed to stop after not receiving commands for some predetermined amount of time.

At higher levels, the problem may be posed as one of using a complex controller whenever possible, reverting to a simpler, but more robust controller when the complex controller fails in some way. The ability to do this enables the use of complex, but unverified, components without depending on them. It must be designed into the architecture, particularly the communication architecture. This is similar to the Simplex architecture [12]; see Figure 2.2.

The significance of an architecture such as Simplex, which enables online guarded upgrade, is that it addresses the requirement of reliability without sacrificing evolvability. Through evolution, it enables performance improvement by providing a safe and reliable method of performance insertion. Moreover, the reliable backup can remain in place indefinitely, possibly

preventing future system integration errors from propagating. In addition, the ability to log the switchover, as well as system state at the time, can contribute to the debugging process throughout.

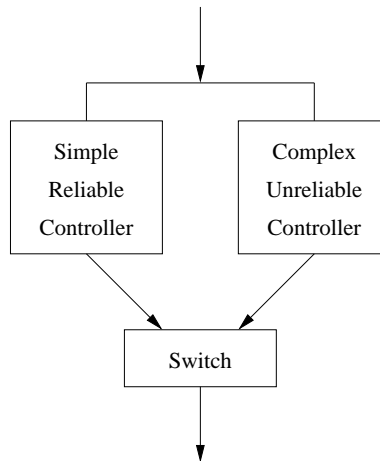


Figure 2.2 The Simplex switching architecture.

2.2.3 Reliability in testbed design

Beyond safety issues, reliability is also a performance measure. To improve performance, we have built reliability into the testbed in many places.

2.2.3.1 Vision reliability

Extracting position and orientation information for each car requires color segmentation. The vision system must group pixels in the track image into common colors in order to identify potential cars. This process is not perfect. The probability of the system properly identifying all six color patches on a car correctly is low. However, by incorporating redundancy into the car coding schemes, we can tolerate the loss of up to two color patches on each car and still properly identify it.

Another source of error is that brightness varies dramatically across the track. We therefore transform the color values from the red-green-blue RGB color space into another color coordinate system, the hue-luminance-saturation HLS color space, which captures brightness as a separate quantity, called luminance. By grouping the colors into a hue value and removing

brightness from that value, the HLS color space allows color to be captured as an independent measure, making the vision system much more immune to brightness fluctuations.

These two efforts result in improved sensor robustness, improving performance in turn for the whole system.

2.2.3.2 Controller reliability

The previously mentioned StateEstimator provides robustness to communication failures from the sensor to the controller. We have provided robustness against controller failures as well. This is enabled by three design elements. First, Model Predictive Control automatically provides a windowed horizon of future controls. Even though the quality of future controls degrades, they are still usable. Second, these future commands must be sent to, and stored by, the actuator for future use, even though most of these will be overwritten when the next sequence arrives. Third, by residing in a separate computer process, perhaps on a separate machine, the actuator will not fail in conjunction with a controller failure. In this manner, the failure of a controller is equivalent to a timeout failure.

2.2.3.3 Reliability through rapid restart

The original purpose of making the cars robust to controller failure was to prevent cars from driving blindly off the track when a controller failed. Thus, with the design noted above, after a period of open loop control, the cars are timed out for safety. However, having the ability to operate in the face of controller failure for a brief period prior to time-out has provided a new capability in the testbed, namely rapid restart. By checkpointing desired trajectories and start times, a failed controller can be restarted quickly, and resume operation.

Such a restart capability has been implemented [13]; Figure 2.3 shows the results of restarting multiple times. Note that the restarts have no noticeable effect. (The periodic spikes in deviation result from interpolation error by the smooth curve of a nominal polygonal trajectory.)

In this experiment, done in an earlier version of the testbed, controller failure was induced through the operating system. Another process monitored the liveness of the controller process

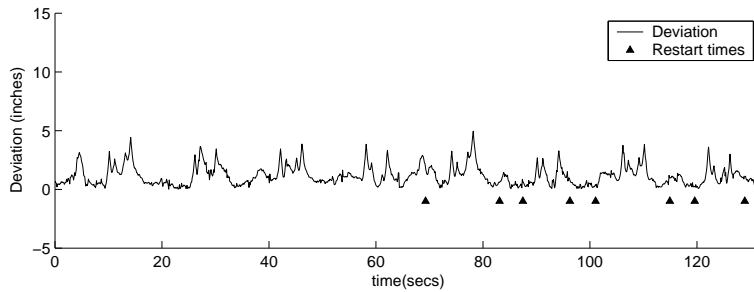


Figure 2.3 Deviation from desired trajectory under restart.

and restarted it immediately after it was terminated. Because the actuator was a separate process, it continued to function in the absence of the controller process. Upon restarting, the controller began to generate controls again before the sequence of commands provided to the actuator (just before termination or the controller) was exhausted. Thus, the car exhibited no visible signs of the outage.

Consider the implications of rapid restart. The mere fact that a controller can be restarted reduces the system dependence on correct operation of the controller under all conditions. If unusual conditions cause failure, but the controller can be rapidly restarted, the failure is masked. Furthermore, a system which tolerates such restart must have few dependencies. Indeed, the ability to tolerate brief controller outages is equivalent to tolerating communication outages from the controller. Minimal dependence also enables other capabilities such as online upgrades, as well as the ability to revert back to a safe controller in the event of failure in a complex controller. Thus, rapid restart provides temporal redundancy for improved controller reliability

2.3 Proliferation of General Purpose Systems

General purpose computation has proliferated extensively, arguably as a result of the clean interface of an instruction set architecture. Similarly, general purpose communication has proliferated, arguably as a result of the clean interface of the Internet Protocol. General purpose control is beginning to emerge with the proliferation of embedded devices. It is only a matter of time until the interconnection of the embedded devices proliferates into general purpose networked control systems. For this to happen, we believe that it is necessary to first identify what

the bottlenecks are in the development and proliferation of general purpose control systems, and subsequently to address them.

2.4 Importance of Abstractions

We believe that understanding what are the appropriate abstractions and what should be the proper architecture is fundamental to the proliferation of technology. As motivation, one may consider the IP stack in networking. It is present in all computers and has provided essential communication services by making interconnections transparent to the user, thus aiding proliferation. The question then arises as to what will the “IP stack” equivalent be for general purpose control with its distributed and interconnected embedded systems? A suitable architectural construct will need to provide the appropriate services for sensors, actuators, computation, and communication to work together. Such infrastructure code should self-organize, taking care of details such as which computation is running on which host, what time it is on various machines, and relieve the designer from mundane details such as IP addresses and the problem of start-ups, etc. The software should provide the right abstractions and interfaces to application programmers, and a rich set at that, so that they can concentrate on developing applications. Some examples of successful technologies taken from the domains of communication, computation, and control are revealing.

2.4.1 The OSI abstractions in networking

We contend that the reason for the success of networked communication, e.g., the Internet, is fundamentally architectural, and only secondarily algorithmic, though that too is very important.

Consider the layered Open Systems Interconnection (OSI) architecture [14], which consists of a hierarchy of abstractions. The bottom layer, the physical link layer, handles the many details of point-to-point communication, including timing issues, signal processing, encoding and decoding schemes, etc. The second layer, the data link layer, handles error detection and correction for point-to-point communication. Next, the network layer is responsible for routing over the set of links provided by the data link layer, and provides the service of unreliable

delivery of packets to a particular destination. The layer above is the transport, or end-to-end, layer, which builds upon the service provided by the underlying layer network layers and adds reliability through end-to-end handshaking protocols in TCP (but not UDP).

Each layer can be oblivious to the details of lower layers and in turn provides a service to the layer above it. Thus the data link layer hides the details of the physical link below such as fiber optic or satellite links, and allows technology below it to evolve independently. Another architectural feature is the notion of peer-to-peer protocols, whereby TCP mediates between two end hosts at the transport layer. Over the course of the years, different versions of TCP have been proposed and could be deployed, without necessitating a change in other layers. These capabilities make the network robust and evolvable, give longevity to the basic design, and allow heterogeneous systems to be composed in a plug-and-play fashion, making it amendable to massive proliferation.

2.4.2 The von Neumann serial computation abstraction

In serial computation, we note the importance of an instruction set architecture, called the “von Neumann bridge” in [15]. Here the software meets the hardware. The instruction set architecture is an interface and represents an abstraction of the hardware for the software, and an abstraction of the software for the hardware. It allows Intel and Microsoft (say) to proceed separately, ensuring only that each conforms to the abstraction of the others. Software programmers need not know about the number of gates used in the adder, etc. CPU designers no longer need be concerned with program correctness. The two activities can be largely separated. Valiant [15] argues that this has been the reason for the proliferation of serial computation. He also claims that it is precisely the lack of a von Neumann bridge which has prevented the widespread success of parallel systems. For parallel algorithms, the programmer must write code according to the details of the underlying hardware structure. Thus, parallel software is not portable.

2.4.3 The Shannon separation of source/channel in digital communication

One of the rare architectural results that is the outcome of mathematical theory is the source-channel separation theorem of Shannon in information theory. It is at the heart of the digital communication revolution. This theory proves that it is ϵ -optimal to separate the functions of source and channel coding, thereby allowing source compression algorithms such as JPEG and MPEG to work independent of channel coding techniques such as QPSK, etc. Nowadays source coding is often done in software (e.g., JPEG), while channel coding is done in hardware by a network interface card that is tailored to a specific channel [16].

2.4.4 The plant, controller, and estimator abstractions in control systems

In control, it is standard to separately view the plant from its controller. This has proven itself as a very useful abstraction. However, it is obvious only in retrospect and not routinely exploited in other fields, e.g., simulation software where plant and controller may be inter-mixed. Another control theoretic abstraction is the separation of estimation and control, which considerably simplifies the design of control systems [17, 18].

This leads to what we believe is a fundamental question: What are the appropriate abstractions and what is the appropriate architecture for the convergence of control with communication and computing? As far as possible, we wish to design an application-independent and context-independent architecture. If one replaces car-specific code with aircraft-specific code, the same architecture should support air traffic control. Or if we replace aircraft-specific code with code for a thermostat, it should support building temperature control.

2.4.5 Importance of layering

The network layer is responsible for routing. As all nodes cannot possibly be directly connected to one another, there must exist the capability to route a packet, or group of bits, from node to node. This is accomplished using a globally unique identifier, such as the IP address, and algorithms which create routing tables used by routers to forward packets. In the Internet, the IP address is intended to be globally unique, but is hierarchically assigned. This facilitates simple routing tables to make routing fast. We note that this hierarchy is also

a hindrance to mobility. That is, a node which moves can physically leave the hierarchical domain in which it belongs, thwarting efforts to send it a packet.

According to the OSI model, the layer above the network layer is the transport, or end-to-end, layer. This layer builds upon the service provided by underlying layers, namely, unreliable delivery of packets to and from a particular destination. In the transport layer we may add reliability through end-to-end handshaking protocols. The two ends can communicate and ask for retransmissions, etc. At this layer, the protocols need not be concerned about how packets are delivered. The abstraction at this layer is that a packet will be delivered to the destination, but without guarantees. In this layer, we can add reliability through protocols such as the widely used Transmission Control Protocol (TCP). Several versions of TCP have been proposed. It is important to note that the specific version of TCP being used does not actually matter to layers above and below the transport layer, with the possible exception of performance issues. This fact demonstrates the importance of proper abstractions and layering of functionality.

We now consider what advantages and capabilities this layering and abstraction provide. The data link layer hides the details of physical links below, such as fibre optic, satellite, or twisted pair links. This provides the ability to evolve. If a particular link is too slow, for example, and is replaced by another technology, the layers above function exactly as before, but can now take advantage of the faster speed. Moreover, there is no need to upgrade all of the physical links in the system simultaneously. Instead of macro evolution, the system can progress through micro evolution.

These capabilities make the network robust and evolvable. They also allow for mass production by heterogeneous vendors in the marketplace. Companies can build custom routers and links, but because of the standardization existing in the interface of the layers, the heterogeneous systems can be composed in a plug-and-play fashion.

2.5 The Shifting of Bottlenecks

As technology advances, one often witnesses the shift of bottleneck problems. Given that great advances have been made over the last two decades in computing, communication, and

control capabilities, we contend that the first task is to understand how to compose systems efficiently, more so than with individual performance issues.

The envelope of performance itself has widened to include such attributes as reliability, robustness, security, evolvability, etc. These so-called nonfunctional requirements represent the current bottleneck in system design and must be addressed.

2.5.1 The performance bottleneck

The design paradigm of “performance first” emphasizes performance over architecture. By optimizing for performance early in the design cycle, decisions which initially support performance impede clean architectural interface design, reducing system flexibility. Performance oriented systems are initially better, but may soon fall behind in performance as agile and properly designed systems adapt and evolve toward better performance. E.g., the IP stack which sacrifices performance through redundancy in managing layers, ultimately allows massive proliferation which has improved its performance.

We argue that in light of the substantial performance capabilities available today, the current bottleneck has shifted away from traditional performance, toward nonfunctional requirements. To this end, our approach begins with a fundamental paradigm which places traditional performance issues (speed, power, etc.) behind nonfunctional requirements deemed to be ultimately more important than performance. Indeed, we believe that performance can be enabled precisely by designing for flexibility and evolution. While early performance may be low, later performance rapidly improves, surpassing the level of performance originally attainable by an initially performance-oriented design.

2.5.2 The design time bottleneck

We contend that a chief bottleneck to the proliferation of general purpose networked control systems is the designer’s time. This is true of both large scale systems in which the complexity is overwhelming, as well as of small scale applications in which the weekend designer desires to compose a small network of devices into a particular task. In all cases, we believe that the central issue is how to improve the process of design with an eye toward reducing the time

required of the designers. Because downstream work cannot proceed until after the designer, or system architect, has specified the structure, or interfaces, of a system, the time pressure on a designer is high. Implementers justifiably want rigid requirements so that they can properly encapsulate their work without concern for whether or not it works with other pieces of the system. Implementers do not like moving targets. However, the designer may not have a complete idea of the present and future needs of the system before having to commence the design. Ideally, the implementation of a design not only produces the capabilities currently specified by the user, but affords the flexibility in meeting changes to these requirements over time. In fact, the process of design itself must already anticipate that the design will evolve since designs are ever evolving. (Witness the ever newer versions of Microsoft Word, for example.)

Much of the specification at the design level today includes details which can and should be properly handled by the implementation level. For instance, it is temptingly easy to hard wire in an IP address when building a system, but if the address changes, and references to the address are scattered in various pieces of code, then we risk a Y2K problem. If mundane details such as IP addresses are instead acquired at run time, either through configuration files or some sort of “discovery,” the system is flexible to IP address changes, albeit with a penalty in the increase of system startup time and slightly more complex component code. Designing for the long term dictates that long-term flexibility should take precedence over short-term simplicity.

2.6 The Need to Ensure Evolvability of Design

There are two primary drivers of change in systems. First is the desire to incorporate additional capability provided by new technology. The second is that users of the system change the requirements of the system, possibly as a result of using the system and discovering additional possibilities or unforeseen limitations. With ever increasing capability required of large systems, complexity grows, making the process of defining requirements very difficult. It is expected that requirements mistakes will be made. By designing a system to be evolvable, those mistakes can be corrected much more easily, reducing cost and adding capability. For smaller systems, evolution is the preferable mode for change in order to allow for customization and applicability over a wider range of uses. The weekend designer may compose a system to do

one thing this weekend, but after using it may conceive of many more things. Evolvable design requires proper layering and abstractions to enable change by restricting the cascade of change which follows from changing one portion of the system. Proper interfaces are also required to reduce the required changes as functionality is added.

2.7 Convergence Towards a More Holistic Theory

The aforementioned technological developments are leading to accompanying changes in research directions which are aimed at a more integrated view of systems theory. Though it may not be completely accurate to put too clear a historical marker, it can be said that the last half of the twentieth century was the age of development of the individual areas of control, communication, and computation. Von Neumann's idea of a stored program (1944) and the ENIAC (1946) are about a half century old, and roughly mark the beginning of the age of computers. Wiener's World War II work, embodied in his "Yellow Peril" book [19] (so known for the color of its cover and its perceived incomprehensibility) dates to 1949. Shannon's [20] foundational information theory was published in 1948. Kalman's [21] work on providing a foundation for state-space control theory dates to around 1960. In signal processing the seminal work of Cooley and Tukey [22] is slightly more recent, around 1965.

In contrast, we anticipate that the next few decades will witness the development of a more integrated system theory combining all these areas. For example, signal/image processing methods with information theoretic performance assessment and connections are already emerging [23, 24]. Networking is seeing the confluence of computer science with more traditional communications research conducted in electrical engineering departments. (INFOCOM, for example, is jointly organized by the IEEE Computer and IEEE Communications Societies.) Communication and control have a long history of involvement, dating back to the work of Wiener and Nyquist.

In the future, at the theoretical and architectural levels, issues such as addressing messages, and combining sensory inputs, while computing based on locally available data, will all be seen simply as tradeoffs in the context of design of a larger system.

2.8 The Necessity for Design Experiments on a Testbed

Last, we believe that research in the area of convergence is well served by actual implementations and testing on a flexible experimental testbed at all stages from conception to implementation. Rather than merely serving as a “demo,” it should be a fully experimental platform where one can learn from experiments. Such a testbed has been developed as part of this dissertation and serves as the basis for the propositions contained therein.

CHAPTER 3

THE CONVERGENCE LABORATORY

3.1 The Convergence Testbed

We now describe the testbed used in our study. The description of the testbed will enable clearer and simpler explanation of issues, since we can refer to how they arise in an actual context. The description of the testbed will be enhanced in subsequent sections as we describe the design and functionality of the software infrastructure and solutions.

The Convergence Laboratory features a concrete example of a general purpose control system with an indoor track upon which remotely controlled cars are driven. The tops of these cars have color codings visible by two cameras mounted in the ceiling. Each camera covers half of the track with a slight overlap. The images from these cameras are processed to determine the identities, positions, and orientations of cars on the track. This information is then distributed over an ad hoc wireless network to laptops which compute commands to send to the cars. Figure 3.1 shows the physical components of the testbed.

3.1.1 Testbed functionality

The primary function of the testbed is to have a Controller direct a car along a trajectory. Using this primitive, we can demonstrate several modes of operation. Running many cars along a predetermined roadway, we may wish to schedule their routes to avoid collisions. This can be done using a centralized scheduler [25] that produces a deadlock-free and collision-free schedule which, if adhered to, enables many cars to operate simultaneously in their various routes.

In another mode, we may designate a particular car as a leader, and command other cars to follow it with some offset. This may be done in a formation with a single leader, or with multiple leaders and followers, i.e., with cars following each other in succession.

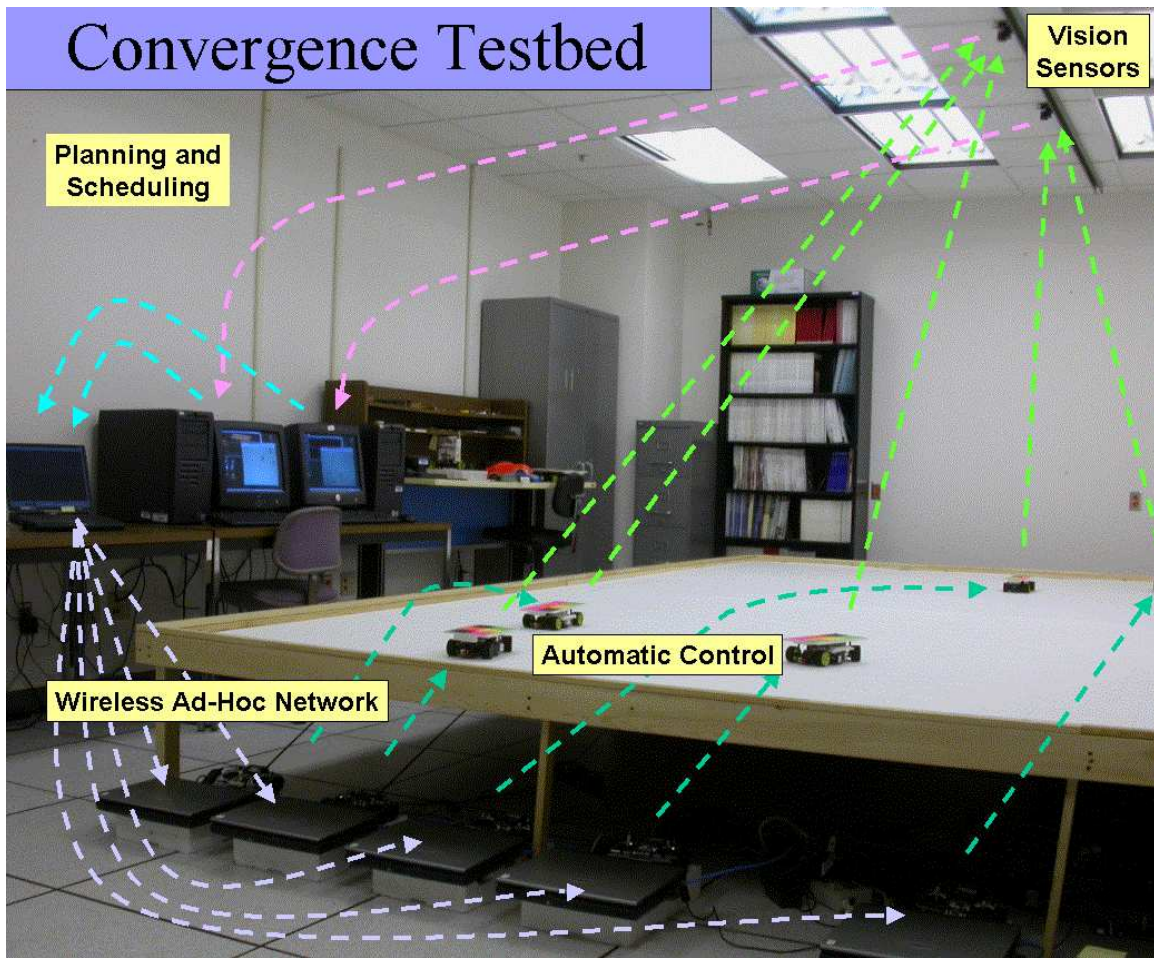


Figure 3.1 The Convergence Laboratory.

Additionally, we have created local collision detection capabilities as well as collision avoidance maneuvers. These operate by having each car receive a global picture of the track, monitor all other cars, and determine when a collision is imminent. A car may then choose to stop, pass on the left or right, or wait for the traffic to clear.¹

The testbed is also capable of adaptive calibration. It further represents a hybrid system, involving the interaction of discrete-event and continuous dynamics.

¹Movies are available at <http://decision.csl.uiuc.edu/~testbed>.

3.2 Description of the Testbed

3.2.1 Sensors and actuators

Currently, two vision systems serve as the only sensors in the testbed system. The ceiling mounted cameras capture images which are then processed to provide position and orientation information for each car. A data fusion component, called the FeedbackServer, combines the sensor data from both vision systems and distributes the information to all interested controllers via an ad hoc wireless network.

The actuators are simple radio-controlled cars. A software module, called the Actuator, feeds commands to the radio-control system. For fail safe, the Actuator is designed to buffer a small horizon of future commands, tolerating brief Controller outages, while stopping the car if a Controller is down for too long.

3.2.2 Controller

The Controller is separated into two components, a planning component, called the LocalPlanner, and a component responsible for computing the best controls to track a set of waypoints. This latter component is thus called a Tracker. The Tracker's control law is Model Predictive Control (MPC) wherein the Tracker searches a large space of potential control sequences at each iteration, choosing the control which minimizes the distance between the desired set points and the predicted set points over a receding horizon.

Incoming sensor data is passed to a state estimation module which utilizes a Kalman filter to provide reliable periodic state estimation to the Tracker, thus buffering the Tracker against the delays and jitter associated with communication systems.

Residing in the layer above the Tracker, the LocalPlanner provides a desired trajectory aperiodically to the Tracker. At this higher layer, the LocalPlanner monitors the positions of other cars for possible collisions, or if in a leader-follower mode, replans the desired trajectory for the follower. In the case of scheduled traffic, each LocalPlanner provides a desired route to the centralized scheduler, which then returns a scheduled route, or trajectory to each LocalPlanner.

As the LocalPlanners are collocated with the real-time Tracker, they are able to provide local autonomous control for fail-safe purposes.

To provide the LocalPlanner with an accurate estimation of the positions of other cars on the track, another state estimator is utilized as part of the Controller. This state estimator receives feedback information from the FeedbackServer, but for other cars rather than its own position. This state estimator could also communicate directly with nearby cars in order to improve the estimates for collision avoidance purposes. Thus, a complete Controller includes two state estimators, a LocalPlanner, and a Tracker.

3.2.3 Supervisor

Directing the actions of multiple cars is the responsibility of the Supervisor. Again, a state estimator is used for the Supervisor, illustrating the separation of estimation and control at each level in the control hierarchy. The Supervisor includes components for planning and scheduling the actions of the various cars. Such planning may be abstract, such as in the planning of routes within a city street scenario. Here a small section of street is referred to as a bin, and routes then consist of a sequence of bins. The Scheduler then must create a timed sequence of bins for each car to traverse such that collisions are avoided. The Supervisor's state estimator must be able to determine which bin a car is in, possibly using the information from previous plans to improve the estimates.

CHAPTER 4

THE CHALLENGES

Several challenges arise from the need to match the stringent requirements of control systems with the behavior of general purpose communication and computation systems. In addition, the ever changing nature of general purpose technologies demands and dictates that systems be capable of evolving in a cost effective manner. Moreover, in order for general purpose control to proliferate, the cycle time for design must be drastically reduced. This puts a great premium on the designer's time, and in turn necessitates the development of suitable abstractions to simplify design. These abstractions also need to be matched to an appropriate architecture for the overall system. We now elaborate on these challenges and in doing so outline our vision for general purpose control.

We begin by outlining the challenges of control, communication, and computation separately, as well as the challenges of making their convergence useful in a general purpose environment. Here we aim to show the overlap in challenges from each of these issues. Ultimately, we wish to present crosscutting solutions for these challenges.

4.1 Control Challenges and General Solutions

To understand the specific challenges that arise in the current context of general purpose control and its convergence with communication and computing, it is helpful to first step back and review the challenges faced in traditional dedicated control systems, and how they are met. Table 4.1 summarizes these challenges and solutions.

Table 4.1 Control challenges and solutions.

Challenge	Solution
Plant uncertainty	Feedback
Sensor noise	State estimation
Unknown plant model	System identification Adaptive control
Stability	Incremental Evolution
Reliability of computational system controller logic	Incremental Evolution

First, of course, one needs to decide on control authority. What and how much control can be exerted over the physical system of interest? Ideally, the choice of control authority takes into account not only the plant constraints, but also the constraints on what is achievable by control. As this challenge depends greatly on the specific application, we do not address it here.

The next challenge, plant uncertainty, applies in virtually all control systems. Uncertainty is an inherent feature of physical systems. Actuators never perform precisely as expected. The revolutionary design solution is feedback, which solves one problem but introduces the next, the challenge of sensing.

Sensors report an imperfect and incomplete view of the state of a system. As computing good actuator inputs, or controls, depends heavily on good knowledge of the current state of the system, we must make the best possible estimate of the state. So the solution to sensor noise is state estimation, e.g., the Kalman filter.

StateEstimators can provide better estimates when they are provided with a model of the physical system, or plant, and past controls sent to the plant actuators. Controllers themselves often depend on the model of the plant as well. Moreover, the plant itself may be slowly drifting and changing over time. This leads to the next challenge of control, plant modeling or parameter estimation, commonly called system identification.

The last challenge of isolated control which we address here emerges from the influence of the previous solutions. Another challenge also arises from trying to control a plant that is itself changing, adaptive control.

A major challenge that arises in the context of convergence of control with computation is the very reliability of the computation system used in the controller. It is common experience that modern computation systems are still orders of magnitude more prone to failure than traditional engineering systems (e.g., automobiles). The challenge therefore is to make the controller itself reliable, or to be more explicit, to make the computational system in the controller more reliable to meet the reliability demands of control systems. We propose to address the reliability of the computational system or the controller logic by additional control monitoring, using backup controllers when primary controllers exhibit instability. We shall discuss this in greater detail later in Section 6, and introduce the Incremental Evolution design pattern.

4.2 Communication Challenges and General Solutions

Likewise, communication systems solutions have also been developed to meet certain challenges as well. We summarize the communication challenges for networked control systems in Table 4.2, and elaborate on the solutions later in the section.

Table 4.2 Communication challenges and solutions.

Challenge	Solution
Addressing	Globally unique identifiers Semantic addressing
Routing	ProfileRegistry GlobalEventBus
Delay	State estimation Control Time Protocol
Loss	State estimation TCP

First, of course, is the very physical connection which is met through wires, radio, microwave, and optics. As this challenge is largely solved, we do not address it here. The next challenge is more conceptual. For communication systems capable of reaching multiple destinations, how will the destinations be distinguished or addressed? The typical solution for addressing is to create globally unique identifiers, e.g., IP addresses, and to require senders to know these

identifiers. Coupled with the addressing problem is the challenge of routing (assuming multihop systems), i.e., how will intermediate nodes forward information? The solution to routing is often coupled with the addressing solution. For instance, IP addresses are arranged hierarchically such that routing decisions are based upon address prefixes. Note that while this simplifies routing in the static case, it prevents logical mobility. Thus, for general purpose communication with mobility, addressing and routing may need to be solved separately. In the case of Mobile IPv6, addressing remains fixed and hierarchical, but a home agent is appointed for the mobile node [26]. This home agent forwards packets to a temporary “care-of” address being used by the mobile node. For efficiency, the routing layer may have to cache limited connection state.

Another challenge for communication is that of standards and protocols such that heterogeneous systems can cooperate in the communication. This challenge is largely solved for general purpose networks through the use of the OSI networking model, discussed in Section 2.4.1, and will not be addressed further here.

In addition to the challenges, there are some realities of communication that depend upon the particular technologies used. While they are not a problem for communication itself, they affect the applications using the communication service. The first reality of communication is that of delay. Communication requires time to complete. For packet switched networks, the delays may appear random, depending on traffic conditions. Unfortunately, for control, delay can destabilize otherwise stable systems. The second is constrained bandwidth. The communication system has physical limitations in its ability to transmit information [20]. As this constraint is application specific, we do not address it here. The last important reality is that of packet loss. Whether by cutting a wire, moving out of range of an antenna, losing power, buffer overflowing, or electromagnetic interference, communication systems are subject to losses, although with cost and effort they can be made reasonably reliable.

In the context of control, several of these challenges and realities can be overcome. We shall go into greater detail for each solution later. For now, we briefly introduce the solutions.

To handle addressing for control applications, we simply provide a globally unique identifier to each component. The second issue in addressing is how senders/recipients can know the identifiers of potential recipients/senders. It is not actually necessary that components know

the address of a server. Instead, we wish to address by function. Thus the solution is semantic addressing, which will be discussed later in Section 8.2.2.

Routing is accomplished on two levels. First, the system can use the routing protocols of IP as a basic service. We will overlay on this service a semantic addressing and routing service, accomplished via the ProfileRegistry and GlobalEventBus, each of which will be described later.

The effect of delay on a control system can be mitigated, if it is known; thus a protocol that makes per-message delays known can help to solve delay problems. Our protocol for this is called the Control Time Protocol and will be discussed in Chapter 10.

The loss of some packets is usually tolerable in control, provided that the controller can function in the absence of the updates. This ability can be enhanced by employing a StateEstimator. To compensate for critical update losses, reliable delivery can be used, as in the Transmission Control Protocol (TCP).

4.3 Computation Challenges and General Solutions

One primary performance measure in computation is execution time. While this remains a problem for many applications, we feel that at least for some general purpose control systems, current computation speeds are ample. Moreover, with Moore’s Law still in effect [27], the class of such systems can only expand over time. We do not address this challenge here. Instead, we focus on issues affecting general purpose control systems, summarized in Table 4.3.

Table 4.3 Computation challenges and solutions.

Challenge	Solution
Execution time	Not addressed here
Numerics Stability Liveness Correctness	Incremental Evolution
Dependence Heterogeneity	Etherware

Another computation challenge is how to compute on different hardware and software platforms. One major solution to this challenge is the use of platform-independent languages such as Java. Another solution is to employ middleware capable of hiding differences among platforms.

A third major challenge of computing is that of dependence. How can a component continue to operate in spite of the failure of other components? There are many aspects of dependence, some of which can be addressed through design techniques provided in the middleware. We expand on this in Section 5.3.

Computation also raises such issues as liveness, correctness, numerics, and computational stability. While each of these is application specific, we can protect an application against many of these problems through monitors and the Incremental Evolution design pattern.

4.4 Distributed Challenges and General Solutions

Distributed components, in many physical and logical locations, are pervasive in the context of general purpose networked control systems that we envision. However, they introduce several design challenges. Table 4.4 summarizes the distributed challenges and solutions.

Table 4.4 Distributed challenges and solutions.

Challenge	Solution
Communication	Addressed in communication section
Initialization Configuration	Etherware
Communication failure	Local Temporal Autonomy
Coordination	Global state estimation Hierarchical planning and scheduling Timescale decomposition

Some of these challenges relate to the same challenges of communication, such as addressing and routing, loss, and delay. Others arise from the difficulties of initializing or configuring an application in many locations, some of which can be handled in middleware. Because of the high potential for communication failure, as well as the need for safe control of physical systems, local components must be endowed with the ability to continue to function, albeit for a short

period of time, on their own. We refer to this as local temporal autonomy and use it as a fundamental abstraction and design goal.

Yet another challenge with distributed operations is that of collecting and analyzing information from multiple sources as well as coordinating the actions of several entities in the system. These challenges are application specific, but share common features which can be addressed by global state estimation, hierarchical control, and timescale decomposition. Fortunately, the timescales required for global action are typically more forgiving than those of local control and stability. Thus, at a higher level, centralized or perhaps decentralized components can provide planning and scheduling services in a reliable and timely manner.

4.5 General Purpose Challenges and Solutions

To move control into the realm of general purpose use, there are further important challenges to face. The first arises from the importance and necessity for proliferation. We can today build one-of-a-kind systems tailored for just about any single specific use, although such systems could be, and often are, enormously expensive. This is not the future we envision. We envision widespread usage of general purpose control systems, which are easily configured for specific applications. When these systems are mass-produced, they become inexpensive, and the demand for their use increases. This in turn leads to improvements, which further increases demand, driving down cost, and so on. Many of the eventual uses may be of limited value, and hence would never support large-scale costs of development on their own, but when the costs come down and are amortized over a huge number of applications, these lower value needs will begin to drive the market. In this spiral, performance also can increase across the board. Our goal, thus, is to move from an era of custom hand-crafted control systems to mass production of interconnectable devices, with easy to configure interfaces, such that systems which feature the convergence of control with communication and computation are routinely deployed with short design and development time, while incorporating flexibility to meet changing needs.

Second is the need for reliability. As systems grow in number and complexity of components, the interaction of system components becomes more difficult to analyze and predict. Moreover, with more components available to fail, the likelihood of all components functioning correctly

decreases. For large interconnected systems, reliability is becoming the key measure of performance. For systems which interact directly with the physical world, reliability is also a safety issue. The challenge of reliability is not only to reduce individual component errors, but to tolerate the inevitable errors arising from unforeseen interactions. We believe that systems must be capable of graceful degradation, safely containing errors. Such design places emphasis on protecting components from the actions of other components, and giving components limited autonomous capability to function in the face of erroneous behavior from other components.

Last is the need for evolvability. The design of a large system is always in flux. It is never complete. As a system is built, new features are always added. It would be erroneous to design today's large and complex systems under the assumption that software is easy to change, and therefore adaptable. Only if the system is well-designed, with flexible architecture, can one hope that the resulting system will be adaptable. Incremental development can be used to manage this process. One starts with a modest goal, and an eye toward future changes, and completes it reliably. Then one inserts additional functionality to make the system more useful. Incremental development may also be necessary from an economic point of view in the proliferation and mass adoption of a technology, since it can facilitate revenue even at the outset, making proliferation financially viable. Incremental development also provides useful feedback in the design and application of the system. As increments are tested, identified problems can be resolved before future increments suffer from the need for redesign.

Each of these is addressed in a crosscutting fashion through local temporal autonomy, middleware, incremental evolution, proper abstractions, and layering; see Table 4.5.

Table 4.5 General purpose challenges and solutions.

Challenges	Solutions
Reliability	Local Temporal Autonomy
Proliferation	Incremental Evolution
Evolution	Etherware
	Layering and abstractions

4.6 Concluding Remarks

The aforementioned challenges represent what we believe to be the fundamental challenges to general purpose networked control systems. We now turn attention to design principles to meet these challenges.

CHAPTER 5

DESIGN PRINCIPLES FOR NETWORKED CONTROL

Following from the challenges identified for general purpose networked control systems, and what we believe to be appropriate considerations, we now identify several design principles discovered and distilled throughout the development of the testbed. These principles are cross-cutting, and therefore overlap at times with each other. They represent our understanding of design principles which are needed for general purpose control systems to be designed quickly, reliably, and with the ability to evolve.

5.1 Local Temporal Autonomy

As communication failures represent a large class of probable system failures, we contend that distributed systems ought to be designed with local decision making capability, which we will call local temporal autonomy, at every node in the system. While such autonomy may be very limited, we believe that every remote component must be capable of self preservation in the absence of communication. That is, communication loss must not result in component shutdown, at least for some reasonable amount of time. Moreover, components which direct physical action must have sufficient autonomy to perform in a fail safe manner. For example, instead of providing just the current control inputs, at each update a controller may provide a horizon, or time sequence, of control inputs, which this actuator can sequentially consume over time should new updates fail to arrive. This gives it local temporal autonomy, since it is not critically dependent on updates for a short horizon. This autonomy should exist in all hierarchical layers and between all controllers and their actuators. Such autonomous behavior

is not only useful for operational robustness, but is also useful in design and test. It simplifies design and testing.

5.2 Stability

Stability is closely related to safety, and one desires to ensure it for as large a class of input conditions of the system as possible, including a large class of failures. For our purposes, we will divide stability into two main areas. First is the stability of the physical system itself under control, e.g., keeping the aircraft flying as opposed to tumbling. This is largely addressed by the rich area of control theory. Second is the stability of the underlying controller itself which we need to remind ourselves is implemented in software. For example, rebooting the controller online may not be acceptable, hence the computing system must itself remain stable throughout system operation.

5.3 Dependence Reduction

To understand the issue of computational stability, we need to examine a common cause of cascading failures, which is dependence. If a component cannot provide its intended service functionality without the services of another component, it is functionally dependent on the other component. Such dependence is often natural and unavoidable. Unfortunately, design and implementation choices often create additional unnecessary dependencies for various reasons. These unintentional dependencies reduce flexibility, affect reliability, prevent graceful evolution, and give rise to a large class of system integration problems. We refer to such unnecessary dependencies as implementation dependencies.

Implementation dependencies arise from system design and implementation choices, rather than from natural functional dependence. Some examples are given in Sections 5.3.1-5.3.4.

5.3.1 Initialization sequence

Components may require an initialization sequence as a result of functional or implementation dependencies. Consider a client which connects to a server as part of its initialization,

but which has not been designed to attempt a reconnect in the event of server failure. In this scenario, server failure and subsequent restart cascades into a restart of the client. This cascade may continue into a much larger execution failure scenario. While individual dependencies may be tolerable in isolation, the cascading effect possible in larger systems is not.

5.3.2 Process dependence

Two unrelated functions may execute in a common software operating system process. Even though the two functions may be completely unrelated, if one function misbehaves, such as executing a divide by zero, and the operating system reacts by shutting down the whole process, then the innocent function is terminated as well.

5.3.3 Communication deadlock

Suppose two intercommunicating components follow a “stop and wait” communication protocol without timeouts. Then, failure of a remote component leaves the local component waiting indefinitely for a response that is not coming. Although the local component has not technically failed, it has effectively failed as it no longer functions.

5.3.4 Temporal dependence

While timing problems can be even more general, as we will consider in Chapter 9, we focus here on the specific case where the execution of one component is dependent on the timing properties of another. As an example, we will consider a periodic update. Assume that new information needs to be outputted by one component every sampling interval, and that another component waits to process each update, without timeout. If the first component fails to produce an update or the update is lost, the second component will not execute, resulting in failure of the second to produce any output. Although this may be logically correct behavior, it is not temporally correct behavior; for control of physical systems it is usually unacceptable. The requirements of system safety and stability require that the system maintain control in the presence of errors, although graceful degradation is allowed and expected.

5.4 Reliable Opportunism

Evolvable systems ought to be opportunistic. They ought to be capable of using capability, without expressly depending on it. In this fashion, the system is agile, switching from that which does not work to that which does in a continuous process of optimization. Such dynamic adaptation is only possible with systems designed for such flexibility.

For control systems, safety requires that the system always exert “safe” control over the physical system. As an example, if the temperature feedback from a blast furnace is suspect, the controller ought to be capable of reverting to a known safe open loop control, perhaps turning the furnace off. A “stuck” sensor may indeed cause major damage at a blast furnace facility. In this case, feedback may have been desirable, but fail-safe was required.

5.5 Explicit Assumptions

Interfaces attempt to reduce the interaction between entities to a manageable level. While current interface specifications specify explicit interactions well, they often fail to capture implicit interactions and assumptions. For example, the Ariane 5 disaster occurred when a component was reused from the Ariane 4. The faster speed of the Ariane 5 caused a numerical overflow in a module fully tested on the Ariane 4, and thus assumed to be safe for the Ariane 5 [28]. We contend that interfaces must be capable of specifying more than simple object types. In our testbed middleware, we employ enhanced messages, such as those provided by the XML language, to aid in making assumptions explicit. Although it would be impossible to enumerate all assumptions, it is helpful to list some of the more important ones and enable more extensive assumption checking both at design time and run time.

5.6 Explicit Knowledge of Time and Delay

While communication and computing systems may result in “random” and large delays, which cannot be controlled, nevertheless if the Tracker is given information on the age of a sample (packet) then it can decide how to optimally utilize the information contained in it.

That is, post facto knowledge of per packet delay can be used to recover stability as well as improve performance of tracking error in control systems, as we will demonstrate in Chapter 9.

An important issue in general purpose systems is that distributed clocks are not synchronized in general. Indeed, synchronization may not even be possible in systems which cross administrative domains. We shall discuss these and other issues related to time in Chapter 10.

5.7 Virtual Collocation

Distributed control is much more difficult to design than centralized control. However, much of the difficulty is removed once we provide the abstraction of “virtual” collocation wherein components can be envisioned as being virtually collocated, except that they experience additional delays while sending messages. This virtual collocation abstraction can be used to design a system in the traditional sense, as for example when the designer designs planning, scheduling, set-point generation and feedback control layers.

5.8 Location Independence

Within any distributed system, there will be decisions as to where each function will execute. The notion of “where,” in this sense, may include physical locations as well as logical locations. For large systems, we wish to create the abstraction that the entire system is merely one collocated computational entity, in spite of the fact that components are distributed across computational nodes. Each component must be “addressed” somehow. Its logical address may be an IP address, for example. We may also be concerned with its physical location. For example, a video sensor may provide only local geographical coverage. More to the point, what is important is not the physical or logical location itself, but rather, the implications of that location, e.g., a higher failure rate, or greater delay, or lower utility. It is this set of information that is relevant to the design and operation of such systems. Thus we support semantic addressing schemes.

5.9 Semantic Addressing

Requiring a client to know the address of a server is an unnecessary dependence which makes changes in the server troublesome to the client. Instead, we can provide a mechanism whereby clients ask for a particular service semantically, and a matchmaking service finds the server which can provide the services. “Semantic” may be nothing more than “camera of type x covering location y .” Semantic addressing represents a level of indirection, which shifts responsibility for addressing from individual scattered entities into a few highly specialized entities, which are designed just for this purpose. This then allows the individual entities to evolve separably without concern for changes in the communication structure. We claim this is beneficial overall, and necessary for such nonfunctional requirements as reliability, evolution, and dependence reduction.

5.10 Migration for Self-Optimization and Reliability

General purpose control systems are assumed to have sensors and actuators in locations determined by physical necessity. That is, a sensor is placed where it can sense the information desired and an actuator is placed in such a way as to exert control authority. Except for these two types of components, all components in a general purpose control system are virtual, with the ensuing capability to exist, or rather to execute, at any location. Determination of the optimal physical location in which they should execute depends upon such issues as timing constraints, communication delays, computational loads, etc.

During normal operation of a system, loads and delays may vary. As each of these may be dynamic, we contend that execution of a component in a general purpose control system should be able to migrate anywhere within a system, provided the new location is capable of sustaining the execution properly.

To actually accomplish migration, the system must be capable of several supporting functions. The first is that the components must continue to be able to communicate after the move. This involves updating the communication information at each of the nodes or components

which communicate with the migrating component, as well as updating the communication information at the component itself.

To migrate a component, the system must be able to determine the loads on the physical resources which will exist before and after the migration. This involves estimating the current loads on participating nodes as well as projecting an estimate for the loads which will be experienced after migration. This support can and ought to be provided as part of a middleware service.

Upon migration, the loop delays experienced by the system may change. Whether the delays increase or decrease, the system must be capable of determining the new delays in order to provide interested components with current delay information. Preferably, the system can provide good estimates of “before and after” delays as part of an optimization procedure which provides an automatic migration capability to the system.

5.11 Implementation of the Principles of Design

Having presented several of the design principles necessary for general purpose networked control systems, we now present four key implementations present in the testbed which work together to implement the design principles discussed and meet the challenges of networked control. They also provide the background for the testbed design as well as the design of a hypothetical networked control system.

CHAPTER 6

A DESIGN PATTERN FOR INCREMENTAL EVOLUTION

6.1 The Need for the Incremental Evolution of Design

The first consideration is that the design of a large system is always in flux. It is never at an end. As a system is built, new features are always added.

In the early mass production of World War II aircraft, US automobile manufacturers assumed that automobile assembly-line methods would translate to aircraft manufacturing, without a strong understanding of the additional complexity of aircraft and the manufacturing precision required. Frequently, design changes were required even before the first aircraft would come off the line. Rather than change the assembly line, the fixes were often done in separate modification centers. Even then, further changes were often made at front-line bases [29].

Similarly, it is erroneous to design today's large and complex systems under the assumption that software is easy to change, and therefore adaptable. The ability of a system to adapt to changing requirements depends heavily on the overall architecture of the system and the nature of the changes. Only if the system is well-designed, with flexible architecture, can one hope that the resulting system will be adaptable.

An important driver of change is "feature bloat," though we do not use the phrase in a pejorative sense. But it must be carefully managed. One starts with a modest goal, and an eye toward future changes, and completes it reliably. Then one inserts additional functionality to make the system more useful. Indeed, this is an ever present feature of many software projects. (Successive versions of Microsoft Word are just one prominent example.)

Similarly, viewed from the usage end, customers do not always know what they want or need at the beginning of a design cycle. Upon experiencing a new capability, they may envision slight variations that would make the capability more useful. Apparently small changes can, however, have large unintended negative effects as they ripple through the design of a complex system. Systems should thus be well-designed a priori, to the extent possible, so as to be able to incorporate this inevitable feature bloat, and insulate the risk of feature failure from other parts of the system which must be reliable.

Incremental development may also be necessary from an economic point of view in the proliferation and mass adoption of a technology. A system under development for an extended period of time will not produce any financial support for the developer during the development phase. Thus, for large development efforts, it is useful to build the system in smaller increments, each of which provides an increase in functionality. This produces continuous revenue, making the proliferation phase financially viable.

Incremental development also provides useful feedback in the design and application of the system. As increments are tested, identified problems can be resolved before future increments suffer from the need for redesign. We can see this principle in the early development of our testbed. In the beginning, we simply worked to get a single car running in open loop, according to a preplanned sequence of speed and steering commands. In this phase, the cars were found to be too slow, and the motors unreliable. We did not need to have an entire system working to discover this. Moreover, this discovery led to changes in the motors and gearboxes which would have changed all of the calibration data for each car. We had not yet invested time calibrating every car; thus, early feedback helped to avoid this time-consuming task for the remaining cars. We were also initially concerned about slack in the steering mechanism and hence the repeatability of the cars performance. Several open-loop tests proved that the cars were sufficiently repeatable to meet our needs, thereby avoiding a redesign of the steering which we had thought necessary.

Of course, incremental upgrades must be relatively simple to incorporate at each stage. Moreover, it is useful to be able to “roll back” if an upgrade fails in some fashion. This ability to “undo” is a challenge to system design, but provides much needed flexibility to designers and users alike.

6.2 The Incremental Evolution Design Pattern

Design Patterns are a solution to a problem in context. According to Gamma et al. [9], “design patterns capture solutions that have developed and evolved over time. Hence they aren’t the designs people tend to generate initially. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design Patterns capture these solutions in a succinct and easily applied form.”

In keeping with traditional principles of functional programming, our focus is on functional reuse. When adding new features, our goal is to “insert” functionality rather than revamp the existing architecture. Instead of being critically dependent on correct operation of a higher performance feature, we aim to use it when it is satisfactory, but revert to a simpler and tested version when it is not. Thus large complex systems must incorporate the ability to switch between components when they fail or when increased functionality is desirable, while maintaining system integrity in the face of faults, failures, and changes in operational environments. Figure 6.1 presents the architectural construct, or design pattern [9], of “Incremental Evolution.”

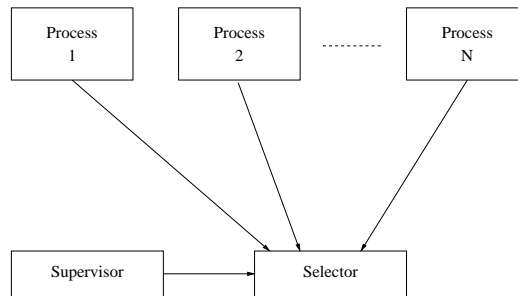


Figure 6.1 Incremental Evolution architecture.

An example concerning the incorporation of a vision data filter illustrates the method. In an early version of the testbed, with just one car running, and just one camera, the vision system was not responsible for identifying the car, but just reporting its position and orientation. A reliable system for this functionality was in place, and the real-time Tracker was able to use the raw vision data reliably for its operation. Moreover, there was no need for a centralized store of vision data; therefore, the FeedbackServer was not yet implemented. The system performed its task of following predetermined trajectories quite well in this early version. However, at a later

stage, to improve the smoothness of trajectory following, it was decided to add a Kalman filter. This was done by adding it as a parallel block to an existing communication path. During the debugging phase, the existing position and orientation information which was “reliable” but not “very accurate” was used to monitor the Kalman filter’s output. Figure 6.2 illustrates the Incremental Evolution process applied to the Kalman filter.

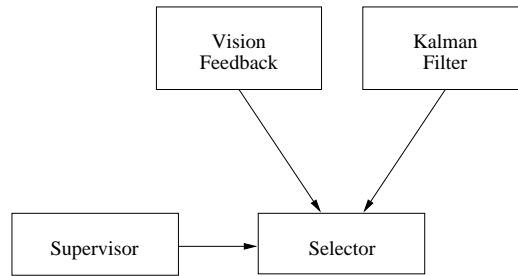


Figure 6.2 Kalman filter inserted via Incremental Evolution.

This idea of Incremental Evolution can be cast into the Simplex architecture of [12]. Sha, et al. [30] have considered the use of a simple reliable controller as a backup to a complex, unreliable controller. This method is based on using simplicity to control complexity. The key notion is that the simple controller, previously established to be reliable in some way, can always maintain stability of the system and meet certain safety parameters provided that the system state is within a well defined operating region, as for example, the basin of attraction of its Lyapunov function. Then a smaller region is defined within which the complex controller is given authority over the system. A supervisory process, which must also be reliable, observes the system state in order to determine if and when the complex controller will cause the system state to move outside the basin of stability of the simple controller. When this occurs, the supervisor switches control to the simple controller, thereby maintaining stability.

Incremental Evolution encompasses more than redundant safety systems. It extends to what we call “data fusion.” Consider multiple data sensors in a system. An aircraft avionics system may receive position information from GPS, land based beacons, and inertial navigation system, as well as manual updates from a navigator. As the fusion becomes more sophisticated, the likelihood of introducing errors grows. The Incremental Evolution process provides simple algorithms to guard the complex versions, thereby ensuring reliability, while recovering the desired high performance potential.

Another usage of Incremental Evolution lies in assessing the effects of time-delays in the incremental deployment of a more complex control system. Control systems are generally sensitive to timing. Delays introduced into a stable control loop can even render it unstable. A system designed with the ability to switch between a stable version of a process and an experimental version, can accommodate online development and testing safely. Consider, as a simple example, a filter located somewhere along a control feedback loop. When a more sophisticated filter is being entertained, the additional processing required for it may introduce additional delay, which could render it worse than the original simpler design. By applying the Incremental Evolution design pattern, we are able to first program another version of the simple filter that includes the additional delay, without any algorithmic changes, and use the Incremental Evolution pattern to switch between the original and the delayed versions, monitoring the system for undesirable effects. Once we have tested this sufficiently, we may then install the full functionality of the complex filter and run it in place of the delayed version of the original filter. Because of the supervisor, we can make these changes at run-time (in real-time) without bringing down the system. Moreover, the original filter is still in place, ready to be used in the event of undesirable behavior of the complex filter. So Incremental Evolution facilitates incremental operational testing by allowing low risk online upgrade.

Yet another place where Incremental Evolution is useful is in “planning.” Multiple plans can be generated and evaluated, and the plan with the best performance can be implemented. One example of this in the Testbed is in the midlevel Planner which continuously monitors the vision data, predicting where cars will be in the next several steps and comparing the current trajectory with those positions in order to predict future collisions. Upon detection of a potential collision, the Planner may create several alternative plans. These alternate paths are then checked for collisions, and if one is deemed successful, it is used. If not, the desired behavior is to come to a stop, and the Planner accordingly stops the low level Tracker, thereby avoiding a potential collision. Figure 6.3 shows Planning inserted via Incremental Evolution.

These examples illustrate the fundamental ability to connect to, and select among, multiple sources of data or control. Properly implemented, this functionality provides for evolution, rollback or undo, and reliability. It provides a separation of decision criteria, or rules, from the

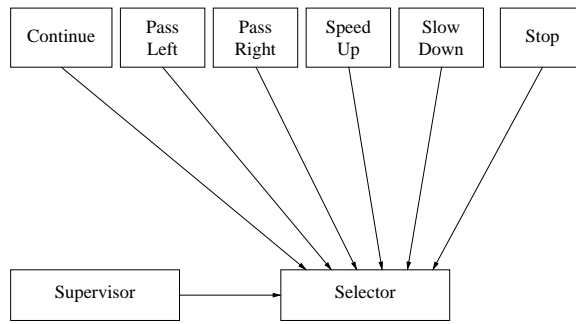


Figure 6.3 Planning as Incremental Evolution.

execution of the criteria. By implementing the Selector as a separate process from the other components, we can create the Simplex Architecture for reliable on-line system upgrade.

Incremental Evolution can be aggregated or composed hierarchically, where each element of the architecture can be a trivial one, or a very complex system of its own, or something in between. Thus, Incremental Evolution exhibits a self-similar nature useful for hierarchical construction and decomposition.

The design pattern that we call Incremental Evolution combines widely used principles into a useful architectural construct or design pattern. Whether or not a system realizes the benefits of connection to and selection among multiple sources depends upon the software design of the underlying infrastructure.

CHAPTER 7

DESIGN BASED ON LOCAL TEMPORAL AUTONOMY

7.1 Robustness to Blackout

We propose the principle that design should ensure the ability or property that individual components be capable of continuing to operate for at least some period of time, even in the absence of correct operation of neighboring components. While the autonomy may not be of indefinite duration, it should provide immunity to short failures of components upon which it functionally depends.

In addition to actually providing such local temporal autonomy, the very process of ensuring such autonomy eliminates several unnecessary dependencies. Moreover, such autonomy can then be used to enable restarts and other operations which make the system resilient to failure.

We illustrate the concept on several examples taken from the convergence testbed and attempt to tie these examples into a strong argument for “blackout” robustness.

7.1.1 Actuator autonomy

As described in Section 3.2.2, the testbed employs model predictive control. We operate the controller periodically and have the actuator hold the control outputs for the period until new commands are issued.

Initially, the controller computed commands and was responsible for sending these commands to the serial port of a laptop computer, from which the commands travel through a dedicated communication system to the motors on the cars. As mentioned previously, the com-

mands were thus dependent on the liveness of the controller process. A failed controller created a blackout in actuator commands. The dedicated microcontroller responsible for issuing commands was programmed to sample and hold. The result is that the cars would maintain the last steering and speed commands issued. This may seem reasonable, until we consider that it means that a car will continue to drive blind with a given speed and steering angle.

The solution to this unsafe condition is relatively simple. The function of sending commands to the serial port can simply be extracted from the function of creating commands, and placed into a separate process. The resulting module would be able to issue a failsafe stop command in the absence of a controller update.

We can do even better than this. Because model predictive control computes a sequence of future commands, we can pass the sequence to this new module which we call the actuator. On each subsequent update, the actuator merely flushes the last sequence and replaces it with the new sequence. However, if the controller fails to produce an update, the actuator consumes the current sequence, issuing the controls accordingly. When the short sequence is exhausted the actuator issues the stop command.

7.1.2 Controller autonomy

The vision system is not perfect. Because of lighting conditions around the track, there are areas of the track in which a car is not properly identified. A car located in such a blackout area will receive no update from the vision system. If the controller patiently waits for an update, which is not forthcoming, the controller cannot issue commands to the actuator. In a short time, the actuator will stop the car, leaving it forever blacked out. This can be overcome by inserting a `StateEstimator` before the controller which can use the last commands issued as an estimate of the future state of the car. This is often called “dead reckoning.”

For simplicity of computation, it is desirable to have a controller update its controls in a periodic fashion. Unfortunately, the vision system does not produce perfectly periodic updates, and the communication system may not deliver them in a periodic order. Thus, the controller, which would like nice periodic inputs, must be decoupled from the aperiodic behavior of the vision system feedback.

Using the StateEstimator mentioned above, we can update estimates of the state as they are produced by the vision system, whenever needed by the controller. Moreover, we can update estimates of the state according to the commands issued by the actuator whenever received. This capability includes the ability to incorporate late data packets much later than when they were produced.

7.1.3 Vision system autonomy

To establish communication with the FeedbackServer, a VisionServer can wait for a handshake with the FeedbackServer as part of its initialization. This produces the undesirable effect that when the FeedbackServer is shut down and subsequently restarted, the VisionServer will not attempt to reconnect to it. The result is that failure of the FeedbackServer cascades into a restart requirement for the VisionServers. This is correctable by establishing a separate communication component linked to the VisionServers which handles connection and reconnection messages from clients. In this fashion, the VisionServers can be started and left to run indefinitely, without concern for when the other components in the system are brought up and down.

In another example of robustness to blackout, the vision system has been designed with color redundancy for each car. In this way, color loss, which is geographically distributed, can be tolerated for up to two of the six color patches placed on each car.

7.2 Controller Blackouts

In the event that two cars are approaching each other, we desire the cars to cooperate in some fashion to resolve the potential collision. As this requires inherently unreliable wireless communication, the cars must have some reliable backup. In this case, each car produces an estimate of a bounding box within which the potentially colliding car may exist. If communication with the car is not established quickly, and the bounding box touches the current position of the “blind” car, then this car will make a worst case assumption and stop, preventing collision. As the other car operates with a similar mechanism, neither car will proceed until they have established car to car communication.

7.3 Capabilities Derived from Local Temporal Autonomy

By having some degree of autonomy at multiple locations within the system, the system becomes very robust to independent failures. Moreover, it enables additional capability. For example, as actuators will not fail in the event of a controller failure, we now have the ability to swap out controllers in real time. Similarly, we can upgrade the VisionServers in real time. This also facilitates migration capability in that the swapping can occur across nodes within the system. Of course, other capability is required to migrate, but the fundamental ability to restart a component elsewhere in the system is a major step towards migration capability.

CHAPTER 8

ARCHITECTURAL FRAMEWORK: MIDDLEWARE

Solutions to crosscutting problems may themselves need to be crosscutting. Creating a framework for the design of reliable, evolvable, and proliferable general purpose control systems requires a balanced and integrated solution to all of the issues. Because the solutions are integrated and interwoven, the presentation of the solutions, and why they are indeed solutions, is a challenge in itself. The issues are not at all linear, though we will endeavor to present a linear thread throughout them.

The framework we propose incorporates the design principles we have presented through the use of a middleware specifically designed to implement the abstractions, interfaces, and services called for in the framework. The development of such a middleware is the focus of the research of another doctoral candidate, Girish Baliga [31]. He has developed the current version of Etherware, in which the algorithms for the testbed have been implemented and tested.

We begin this discussion of solutions with the design of the middleware, primarily because other solutions fit into the context of the middleware and the framework principles are manifested in the middleware itself.

8.1 Etherware

The previous discussions have outlined the need to create a common framework for general purpose control. Such a framework would be built upon general purpose communication and computation systems. It would ideally have the capability to actively protect against certain classes of failures. The notion of an active framework has serious implications for the application

design of a system and must be carefully examined. Similarly, the choice of time-triggered [32] or event based component invocation must be properly made. A major design choice is the use of enhanced messages, using the XML message format. Several such design choices have been made in the development of Etherware; here we illustrate how these choices together meet the need of an application framework for networked control systems.

8.1.1 Active middleware

An active process running in a computational system must be given processor access by the operating system on a regular basis. That is, it must be capable of executing something under its own volition. Perhaps this is best illustrated with an example of a process which is not active. Consider an update feature for software. If the update code is started by another process, either the user or another active process, then the update code is passive. If the update code is loaded automatically by the operating system, but sits dormant most of the time, it can make its own decision as to when to perform the update, perhaps based upon reading the clock.

According to the needs described earlier, a framework for the implementation of general purpose control systems must be capable of implementing the Incremental Evolution design pattern. Specifically, the framework itself must be able to manage the execution of various components in order to guarantee that the computational system can remain stable in the presence of component errors and failures. These errors include indefinite execution as well as crash errors, such as segmentation faults or illegal instructions. To prevent hanging, the framework must have an active ability to stop execution. To tolerate crash failures, the framework must be able to catch the crash and take appropriate action. Both of these needs require active control of the application.

Therefore, to support the framework requirements, Etherware is an active middleware, maintaining control over the scheduling and execution of the components within the application at all times. This is accomplished by running the Java based middleware as a single thread of control, except for the message passing component. The middleware provides a micro-kernel which is only responsible for scheduling the execution of various components.

8.1.2 Event based communication

Although the middleware must be in control of all execution in order to provide certain guarantees, the application must also be able to control its operation to some extent. It can do this in a synchronous fashion, e.g., a time-triggered architecture [32]. Another method is to have the system operate asynchronously based upon the events. The advantage of a time-triggered approach is that it can be better analyzed, and is typically used for real time systems. Unfortunately, changes to the system require reanalysis, making the approach fragile for evolving systems. An event based approach is much more agile, though less predictable in time. The middleware can provide some of the capabilities of a time-triggered architecture through the use of a time event service. Specifically, a middleware service can provide periodic events to a component as requested in order to control its timing behavior. Components can register for regular time events (ticks), as well as a single time event, which is useful for sleeping behavior. Because all components use the time service, there is no need for a separate thread for each component.

Each component is thus a producer and consumer of events. A component is essentially an event handler. Thus, for periodic control, a tick event is sent to a controller, which is then invoked as its event handler function is given the event to process. In case the component has messages to send to other components, a prioritized list of events is generated as the component is finished. This list is then added to the current list of events to be sent, and scheduled by the middleware for “sending.”

8.1.3 Invocation

Multitasking computer architecture provides the ability to have more than one active thread of control present in a system. Ultimately, only one thread, or process, is actively executing in the processor at any single moment in time. However, the operating system has the ability to schedule the various threads and processes in such a way that all are executed for some period of time, however brief, thereby guaranteeing that each thread makes some progress.

In this scenario, the operating system maintains total control over the scheduling of the applications. In this fashion, the actions of one application cannot crash the other applications.

This then provides a degree of containment for failures. For most operating systems, the scheduling of the various processes is done on the basis of time. That is, all of the running applications are given some proportion of the available execution time. Some applications do not need much time and quickly return, allowing the scheduler to give the time to another process.

For control, some actions must be taken periodically, i.e., in a time-triggered fashion, in order to satisfy the control system requirements. However, many actions are taken in response to other actions. Implementation via a time-triggered approach would introduce unnecessary delays into the system for these aperiodic events. Thus, a purely time-triggered architecture [32] is not adequate. By incorporating a time-triggered event service into an event based architecture, the resulting hybrid architecture provides the required features of a time-triggered architectures while retaining the flexibility of an event based system. In both cases, hard real time guarantees can only be made through careful offline analysis, or perhaps online in simple cases.

8.1.4 Heterogeneous middleware

There are several reasons for making a framework portable to various hardware and software platforms. First, design and implementation time increase when an application must be individually tailored. Second, inconsistencies among platforms cause application errors. Third, configuration dependencies can be reduced.

As part of the desire for heterogeneity, Etherware is written in the Java programming language. Although many of the features of middleware could be implemented in other ways, the Java language provides a natural way to implement a number of features. For instance, Java provides strong native support for the XML language. In addition, the use of the Java virtual machine causes attempted illegal operations to be trapped as exceptions, which can be caught by the middleware in order to provide containment of failures.

The use of Java does incur a slight performance overhead. For our testbed, the Java performance penalty appears to be negligible.

8.2 Core Services

Using an active middleware as a framework for the design of a general purpose networked control system makes it possible to provide a set of core services to applications, including message passing, semantic addressing, and time translation,

8.2.1 Message passing

Interacting components must be able to pass data and control information to one another. A middleware framework can provide a service for sending and receiving messages between components, which hides the networking details of the underlying platform. For example, the application does not need to know what the physical medium is, but may be interested in the average delay of the message. More importantly, the middleware can provide a nonblocking message service wherein messages are sent and received without the senders explicitly waiting for a response, which would effectively halt the sender until the response returns.

Messages are passed in Etherware as events. These take the form of Java XML documents. The application can request that events be sent reliably, or not. Reliable messages will then be sent over an existing TCP connection, operated and maintained by the middleware between all communicating nodes. Specifically, if two nodes currently have an active exchange of messages, there is only one TCP connection needed between them. All communication between components on these two nodes will use the same connection. If the connection is idle for some length of time, the connection will be torn down. As TCP can delay messages, and some messages are perishable, the middleware also provides an unreliable service, i.e., fast message passing service which can use, for example, UDP. As these connections are shared by components, the communication overhead of setting up and tearing down TCP and UDP connections is minimized. Moreover, the loss of communication with a remote node is made apparent quickly through the TCP keep-alive mechanisms. All remote communication is handled by a core service component called the Global Event Bus. This component is responsible for being aware of the existence of other Etherware nodes, either through broadcast, or through specialized configuration files for nodes, which reside on distant networks which broadcast will not reach.

8.2.2 Semantic addressing

At the component design level, it is unnecessary to know the physical location of a component. This knowledge is only needed at run time. Therefore, an application framework which provides the ability to “discover” the address of a particular component at run time can relieve the design burden of determining the location at design time. This is clearly more flexible as well. To do this, some other semantic information about the receiving node must be made known, and a service must be provided which is capable of understanding the semantics and translating or matching it with the address of a component.

Etherware provides a profile registry whereby components which provide a service can register themselves, including their current location, in order to allow clients to find them. This is slightly complicated by the fact that components may exist in distributed nodes. Thus, the profile registry must have the ability to discover where other service components currently reside. This is accomplished through a global profile registry which is simply a designated profile registry on a particular machine, and whose existence is made robust by the fact that a node which tries to use the current global registry will simply announce itself as the new global registry.

8.2.3 Time translation

Because different nodes in a distributed system may reflect different times on their clocks, Etherware provides for an inherent time translation service wherein time stamps received from remote nodes are automatically translated into the time reference of the local clock using the Control Time Protocol 10.3.3. This provides an abstraction of a synchronized distributed system, without the administrative constraints of synchronization.

CHAPTER 9

TIME: THE IMPORTANCE OF KNOWING IT

9.1 Introduction

Delayed delivery of feedback samples in a control loop can cause an otherwise stable system to become unstable [33]. Even when the system remains stable, there are two undesirable results from delayed feedback. First, the delay in the receipt of the information can delay corrective action, and so the system response is slower. Thus, performance is poorer. The second, less obvious, problem is that the data may actually be wrong. Information is time sensitive. Unless countermeasures are employed, delayed packets will masquerade as being valid for the current time. In that sense, the data is not merely stale, it is wrong. Although there is little mitigation for the first problem, the second can be helped.

In control systems operating over computer networks, packets convey the sampled state of a system as feedback from sensors to controllers. Such packets can be time stamped to include the time at which the sample was taken. Even when a time stamped packet is delayed, the contents of the packet, consisting not only of the sensed information but also of the time at which it was measured, can still be useful in determining the state of the system at some time in the past. Through state estimation techniques, and with accurate time stamping, a controller can then extrapolate the current state of the system based upon knowledge of past control inputs. This can help in keeping the system stable or improving performance in the presence of delay.

Time stamping itself, however, poses several challenges in a distributed environment. Distributed sensors and controllers do not have instantaneous access to the same clock; hence the notion of time is different between them. Time stamps created at a sensor must be trans-

lated somehow into the time reference of the controller. If each had a perfect clock which was synchronized with the other clock, this would pose no problem. However, no two clocks have identical tick rates. The resulting drift makes regular synchronization necessary. Synchronization requires communication, which, however, involves delay. Such delay is in general nondeterministic, resulting in limitations to the precision of synchronization.

To ease the usage of control applications over networked systems, we believe it is important to solve timing problems caused by unavoidable delay and the usage of different clocks in a networked control environment. We believe a general time stamping solution, robust and easy to use, is important for the proliferation of control over networked systems. In this chapter we motivate the need for a control time protocol. In Chapter 10, we propose such a protocol for distributed control systems, which we call the Control Time Protocol (CTP), and compare it with the well-known NTP synchronization protocol.

9.2 Effect of Delay on System Performance

9.2.1 Effect of unknown delay on system performance: Experiment 1

To illustrate the effects of unknown delay on overall system performance, we present an experiment using the real testbed controller, which is a computationally intensive model predictive controller, as described in Section 11.5.1. The controller is given the goal of looping around the track following a rounded rectangle (see Figure 9.1) and is designed on the basis of no delay. For simplicity, we only show the results with a single car.

In this experiment, we intercept feedback packets received at the controller, hold them in a buffer, and pass them on after a fixed delay. Note that this delay is in addition to the inherent overall system delay. We then vary the fixed delay added to each feedback packet and observe the behavior of the car. The results for adding 300 ms of loop delay into the system are shown by the dotted line in Figure 9.2. Note also that with 300 ms of unknown delay (plus ~ 240 ms of inherent delay), the system is exhibiting instability. (The plot also shows the effect on the control system if the magnitude of the additional delay is known, an improvement which we will discuss shortly.)

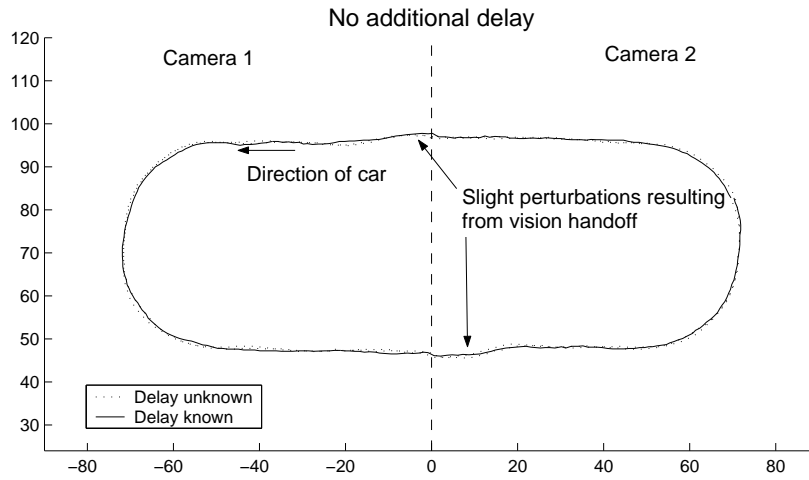


Figure 9.1 Trajectory of car with no additional delay.

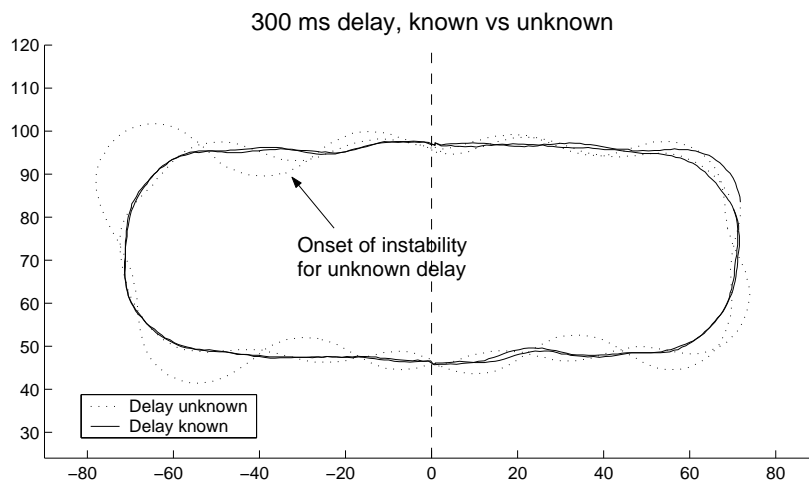


Figure 9.2 Trajectory of car with 300-ms additional delay.

The regular controller assumes a constant delay from command to observation, namely 300 ms (rounding up to the nearest 100 ms to match the 10-Hz system update rate). Thus, adding 300 ms of delay means the system has an overall loop delay of 600 ms, but the controller assumes the delay is only 300 ms (3 frames). It seems intuitive that the gain equivalent of the model predictive controller can be loosely estimated from these experiments by using the delay-gain relationship established in (9.4).

9.2.2 State estimation

Although a closed loop system relies heavily on the feedback obtained through sensors, it is possible to compute an estimate of the system state even in the temporary absence of such feedback. This is sometimes referred to as “dead reckoning.” Since the controller issued the controls to the plant, and presumably has some idea of the plant model, the controller can roughly predict the state of the plant or system as it responds to control inputs.

The testbed employs a simplified Kalman filter [34], the most common algorithm used to estimate the state of a system based on a combination of past feedback observations and control inputs. This filter is also used to smooth out noise in the feedback measurements and can also be used to reject spurious data. The algorithm assumes a state-space description,

$$x_{k+1} = A_k x_k + B_k u_k + G_k w_k, \quad (9.1)$$

$$y_k = C_k x_k + H_k v_k, \quad (9.2)$$

where $x_k \in R^n$, $u_k \in R^m$, $y_k \in R^p$, $w_k \in R^g$, $v_k \in R^h$ and A_k, B_k, G_k, C_k , and H_k are known, possibly time-varying matrices of appropriate dimension. The Kalman filter can be greatly simplified for our purposes of implementation in the testbed. The state x_k refers to the position and orientation of a car at iteration k , while y_k refers to the observation of that state. Because the vision system directly observes the position and orientation of the car in the presence of a small amount of noise, $x_k = y_k$. Hence, $C_k = I$. The linearized state space equations for the system, or car, assume that a car moves according to the commanded speed and with the commanded rate of turn. Hence, $A_k = I$. The last parameter of interest is the $G_k w_k$ term, which roughly corresponds to the internal noise of the system resulting from imperfect steering linkages and gearboxes etc.

Another source of errors is when another car or set of ghost colors masquerades as the car of interest. In such cases, the position feedback is grossly inaccurate. Also, the car may not be identified at all. Hence, we use predictions to reject spurious data.

The resulting testbed filter maintains a history of past observations and controls. Because the vision position feedback is delayed by three frames, we use the model, together with the last three controls, to estimate the state of the car for the current time. Then, when a vision packet

comes in corresponding to the current time (three or more frames from now) we update the estimate of the state for that frame. Proceeding in this way, we always have an estimate of the current state, even when vision is lost or delayed for some time. When new packets are received containing feedback data, the filter can use them, regardless of how old they are, to update the state of the system. However, the data must be appropriately matched to a particular time in the history of controls and observations. If this is not done, the filter may incorrectly apply the update by treating the position information as corresponding to the latest sample time, leading to an erroneous state estimate. Thus, old data masquerading as current data degrades the performance of the system.

To use feedback data, the filter thus needs to know the time at which the measurement was taken. In the testbed, this is the time at which the vision system captured the image from which the car position will be computed. At that point in time, the car was actually at that position. (Because the camera itself is not capable of time stamping, we create a time stamp at the moment in which the VisionServer receives the image from the frame grabber.)

9.2.3 Effect of known delay on system performance: Experiment 2

To verify that it is indeed valuable to know the delay, even when it is large, we modify Experiment 1 with one change. We add the per packet delay to the known system delay so that the filter is aware of the delay, and thus applies the appropriate feedback based on the appropriate sample time.

The effects are profound, as seen in Figures 9.2 and 9.3. The plots show the results of Experiments 1 and 2 overlaid. These plots were created by first operating the estimation with delay unknown, then suddenly switching to known delay. We conducted the experiment in steps of 100 ms of delay from 0 to 900 ms. In Figure 9.2 we show the results for 300 ms because it is at this level of delay that we can first see the onset of instability in the unknown delay case. Figure 9.3 demonstrates the onset of instability for the known delay case. This instability onset did not occur until the known additional delay reached 800 ms.

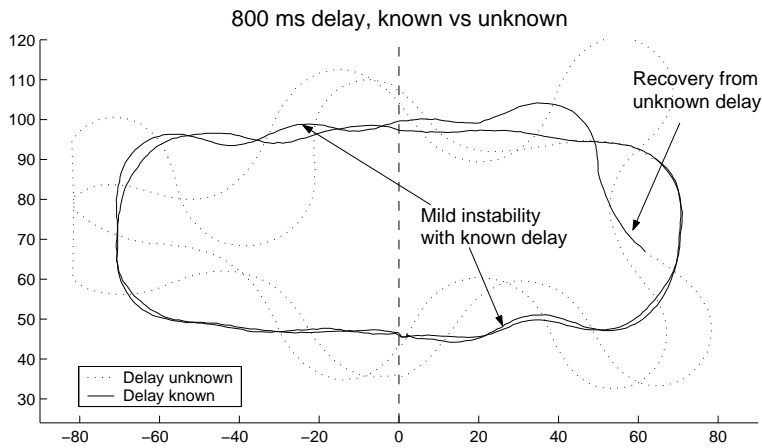


Figure 9.3 Trajectory of car with 800-ms additional delay.

Comparing these two plots we can witness the very real advantage of knowing delay. That is, by knowing the per-packet delay, the system can remain stable for an additional 500 ms of delay, roughly double the inherent system delay in this case.

These plots clearly demonstrate the value of knowing per-packet delay. They provide the practical justification for our Control Time Protocol.

9.3 Determining Overall Closed Loop System Delay

Physical systems exhibit natural delays in their operation. For example, inertia prevents instantaneous changes in the velocity of a moving object. We will refer to these natural physical system delays as plant delay. In addition to delays within the plant, a control system will experience communication and computation delay as well, which we will call processing delay. Taken together, these combine to form an overall loop delay in the control of the physical system. How to estimate each of the delays is the subject of this section

The control loop of a system includes many components, some of which may have a clock, while others do not. If a control packet could be completely circulated throughout the system, time stamped at each step, and returned to the sender with a complete history of time stamps, then we would have a basis for attempting to estimate delays and clocks throughout the system for each and every control iteration. Unfortunately, many components don't even have a clock,

let alone the ability to transmit packets. For example, in our setup the physical car units do not have a clock or a wireless transmitter. They can only receive information from the laptop through the dedicated RF transceiver.

Real control systems control physical entities. The plant is not a computational element, nor does it have communication capabilities. The control loop passes through the real physical world. In the testbed, the loop is closed by vision, wherein the car is observed by the camera. There is no means by which to send a time stamped message from the car to the vision system. Through time stamping, we can determine the processing and communication delay from sensor to controller. The plant delay is the difference between the two. That is,

$$\text{Closed loop delay} = \text{Plant delay} + \text{Processing and communication delay.} \quad (9.3)$$

9.3.1 Using stability to determine delay: Experiment 3

In this section we present a method which trades accuracy of gain measurement for inaccuracy of time measurement. It is well known that feedback delay can cause otherwise stable systems to become unstable. To motivate this, consider the stable system $\dot{x}(t) = -\alpha x(t)$, $\alpha > 0$. We will now introduce delay, so that $\dot{x}(t) = -\alpha x(t - \tau)$. We now consider the effect of $\tau > 0$.

If there exists an s with $Re(s) > 0$ satisfying $s = -\alpha e^{-s\tau}$, then the system has poles in the right half plane and the system is unstable. Instability onset occurs when $Re(s) = 0$, i.e., $s = j\omega$ for s satisfying $s = -\alpha e^{-s\tau}$. With $j\omega = -\alpha[\cos(\omega\tau) - j\sin(\omega\tau)]$, we can equate the real and imaginary parts. Equating the real parts, we have $-\alpha\cos(\omega\tau) = 0$, so $\omega\tau = \pm(2k + 1)\frac{\pi}{2}$. Equating imaginary parts, we have $\omega = \alpha\sin(\omega\tau)$. Substituting $\omega\tau$ from solving the real part gives $\omega = \alpha\sin((2k + 1)\frac{\pi}{2})$, hence $\omega = \pm\alpha$, but since we are only interested in stable systems, $\omega = \alpha$. Thus $\alpha\tau = (2k + 1)\frac{\pi}{2}$. Since we are interested in the smallest delay for which the system is unstable, i.e., $k = 0$, we have

$$\tau = \frac{\pi}{2\alpha}. \quad (9.4)$$

Using the relationship between gain and delay shown in Eq. (9.4), we can thus determine a system's loop delay if we know the smallest gain which destabilizes it. The key point here is that even though time, and thus delay, cannot be measured accurately, if gains can be

measured accurately, then we can actually determine the delay.¹ Thus we exploit the capability to accurately measure gain to estimate the delay.

We have implemented this procedure in the testbed, subject to the discreteness of control values, by building a simple controller with $\dot{x}(t) = -\alpha x(t)$ as the control law. For this test, we need only operate the car in one dimension; so we chose the x -axis. Steering is set to be as straight as possible. One practical problem that appears is the fact that our system has discrete controls, rather than continuous control. Thus, not only is the position information quantized, but the control inputs are as well. This limits the precision to which we can determine delay.

To implement this simple control law, we place a car parallel to the x -axis, directed toward $x = 0$. We compute control inputs as follows. Using the latest feedback, we subtract the x -coordinate of the center of the car from the set point to obtain the error. Then we multiply the error by a gain α and obtain a desired speed to be used as feedback. For example, with a set point of 20 in, and feedback indicating the car is at 23 in, we have a desired speed of 3α in/s. It is unlikely that the car has a discrete speed of exactly 3α in/s, so we search through the available speeds and select the speed which is closest to 3α in/s. We then send the corresponding control to the car. To better observe the stability, we introduce periodic perturbations by changing the set point as seen in Figure 9.4.

In this case, the set point changes from 20 to 30 inches every 8 s, which is enough time for a stable delay-gain pair to stabilize.

In conducting this experiment, we start with a gain α for which the system is stable, then increase the gain in steps until the system demonstrates instability. We then reduce the gain again until the system stabilizes. These two gains then bound (above and below) the precise gain for which the onset of instability occurs. The top plot in Figure 9.4 shows the changing set point. The middle plot shows the actual position of the car as it attempts to reach the set point. Although gain is the variable parameter for this experiment, in the third we plot $\frac{\pi}{2\alpha}$, which we call the “delay equivalent,” for the system. From this plot, it is readily seen that for this particular experiment, the car is stable for $\tau = 218$ ms, and unstable for $\tau = 208$ ms. Because τ is inversely proportional to the gain, and onset of instability occurs at higher gain, instability occurs at a lower τ .

¹We are grateful to Professor Raffaello D’Andrea (Cornell) for suggesting this approach.

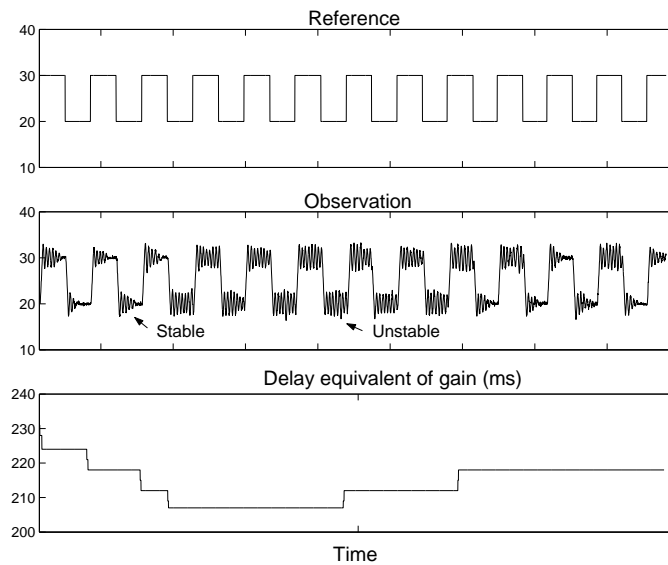


Figure 9.4 Determining delay by observing onset of instability in Experiment 1.

This experiment is subject to much noise and disturbance. Feedback delay varies. Different batteries produce different currents, so the gain that is set may not be the gain that is implemented, making readings vary from experiment to experiment. There is also variability in the cars. Some cars have a small range of speeds available. Some have a sharp cutoff from very fast to almost stopped, others are more gradual, giving better gain control. Over several experiments the delay estimate ranged from 200 to 250 ms.

9.3.2 Offline analysis of a step input: Experiment 4

System delay may also be determined by introducing large control changes, then observing the feedback for evidence of the change. The controller begins by sending stop commands. After a short time, the controller sends full speed commands, printing out time stamps on every vision data packet received. After a few moments, the controller then stops the car. Analyzing the data offline, it is easy to pick out the vision data sample which indicated the beginning of movement. Because the car had clearly begun moving at this point, we can establish an upper bound on the system delay for that iteration by simply subtracting the time at which the go command was issued from the time at which movement was observed. The results of testing four cars, each 10 times, are summarized in Table 9.1.

Delays experienced in a particular iteration vary due to sampling conditions and network delays, requiring multiple experiments to improve confidence. As shown in the table, a car occasionally began moving within 180 ms, though that was rare. The average case appears to be close to 240 ms, which agrees roughly with Experiment 3.

Table 9.1 System delay observed under stop/go control with time stamps.

Trial	Car 5	Car 6	Car 7	Car 8
1	211 ms	270 ms	240 ms	240 ms
2	210 ms	210 ms	280 ms	210 ms
3	280 ms	240 ms	269 ms	220 ms
4	240 ms	360 ms	180 ms	270 ms
5	240 ms	240 ms	180 ms	240 ms
6	240 ms	180 ms	240 ms	270 ms
7	240 ms	270 ms	318 ms	228 ms
8	240 ms	252 ms	240 ms	240 ms
9	210 ms	230 ms	240 ms	210 ms
10	281 ms	270 ms	210 ms	210 ms
Min	210 ms	180 ms	180 ms	210 ms
Avg	239 ms	252 ms	237 ms	237 ms

9.4 Estimation of Plant Delay by Temporal Alignment

We now describe a method to estimate plant delay which uses time dependent cues of dynamic motion, captured by distributed sensors, can be used to temporally align time stamped data. It treats control inputs as one “sensor,” and plant output as another. This can be used in control systems over networks of computational nodes. Such temporal alignment is in fact a general tool useful for fusing data from different sensors, and we present geometric methods of aligning temporal data, based upon real world motion, either for the purpose of synchronizing sensors, or capturing real world time shifts, such as plant delay.

Embedded in data from sensors which monitor dynamical systems is a temporal coherence. If time stamped data from independent sensors is taken from the dynamic motion of a common object, the rotational and translational movement of that object must be reflected identically in each sensor. Therefore, if the object has exhibited sufficient rotational or translational change

during the time of data capture, the time stamped data samples can be correlated to produce a temporal alignment between the sensors.

Temporal alignment can be performed, using time-dependent cues, between any two sensors that observe a common real-world motion. By applying this technique in our control setting, we can estimate the plant delay, which is the delay over the fixed unalterable part of the system under control. There necessarily is a delay between the time a controller commands an input, and the visible effect of the input in plant output, known as plant delay. Our approach to estimating plant delay is to take as the first “sensor” a StateEstimator for a plant under control, and a vision system, monitoring the motion of the plant, as the second sensor. We can then align the time-stamped data streaming from each sensor and determine the time shift between the two. Provided the clocks have been synchronized, as discussed in Section 10.3.3, the temporal shift required for alignment is precisely the plant delay.

We demonstrate the result for estimating the plant delay in the testbed.

9.4.1 Online identification of plant delay: Experiment 5

Recall that our control method is model-based, using model predictive control. The method employs a StateEstimator which takes as input the commands issued from the model predictive controller, and car positions from post processing of time stamped video images, to provide on-line estimates of car positions. We use this as a virtual sensor. The real cameras constitute the second sensor that only observes the actual behavior of the car, as opposed to the predicted behavior that the StateEstimator provides.

Our goal now is to determine the delay between issuing control commands (steering and acceleration), to the time when the reaction of the actuator (turning or speed change) are observed by the camera (more specifically, the time instant when the images are captured by the framegrabber in the computer). The time delay includes a segment of radio communication delay, the mechanical reaction time of the actuator, and the working time for the camera (e.g., shutter opening). Therefore, it is not possible to identify the delay purely using time stamps from different parts of the network. However, as noted above, we can construct a virtual camera for the controller. Using the calibration information on the actuator, for every control command

we can estimate its effect on the plant from the StateEstimator which assumes zero delay. We can thus regard the StateEstimator as a virtual camera with zero delay. The delay of the plant is then the time shift between the real camera and the virtual camera.

An experiment illustrates this technique. We direct the car along a special trajectory that exhibits an easily recognizable rotation pattern. As our testbed consists of two cameras covering both sides of the track, the trajectory includes a smooth oscillation through the first half of the track, followed by a smooth oscillation in the second half. The result, seen in Figure 9.5, has a symmetric shape about the centerline. In this figure, we plot the actual trajectory followed by a car according to the vision sensor data. The second curve is derived from the control inputs created by the model predictive controller throughout the experiment. Specifically, the control inputs are applied to a linearized model of the car, together with calibration data, to predict a trajectory of the car. Although we provide a reference point for this trajectory which is the same as the initial reference point reported by the vision system, the technique does not depend in any way on aligning the reference points.

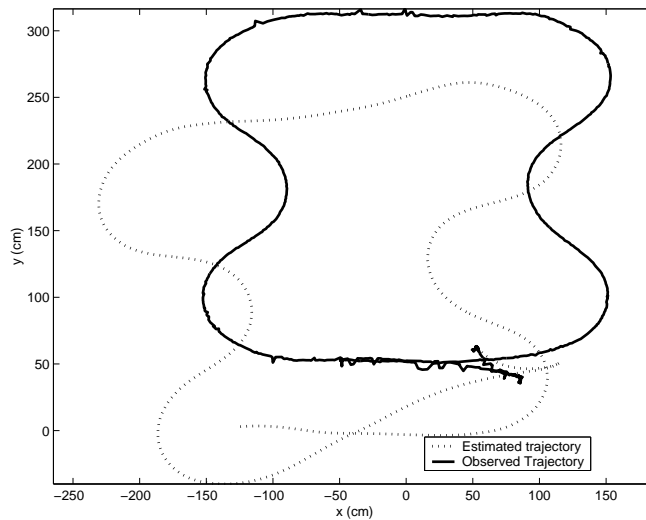


Figure 9.5 Estimated trajectory and the observed trajectory.

The large error in the translation and the relatively slow change in rotation motivate us to use the difference in rotation as the difference measure. We choose the size of the time interval for calculating the observed rotation to be $\Delta t = 500$ ms. The observation from the estimator is based on controller outputs every 20 ms, and the observation from the camera is provided about every 50 ms. In order to reconcile the two observations as well as increase precision, we

perform linear interpolation on the observed orientation (rotation angles) on both observations every 2 ms. The error of the interpolation is small due to the slow motion of the object. Then we search for the minimizing time shift over all the possible time shifts from -900 to 1800 ms.

As shown in Figure 9.6, the summed difference measure on the observed rotations attains a minimum at a time shift of 194 ms. Figure 9.7 demonstrates the effect of time shifting between the two cameras over the curve for the combined observed rotations for both cameras. As can be seen, the 194-ms time shift accurately reconciles both curves.

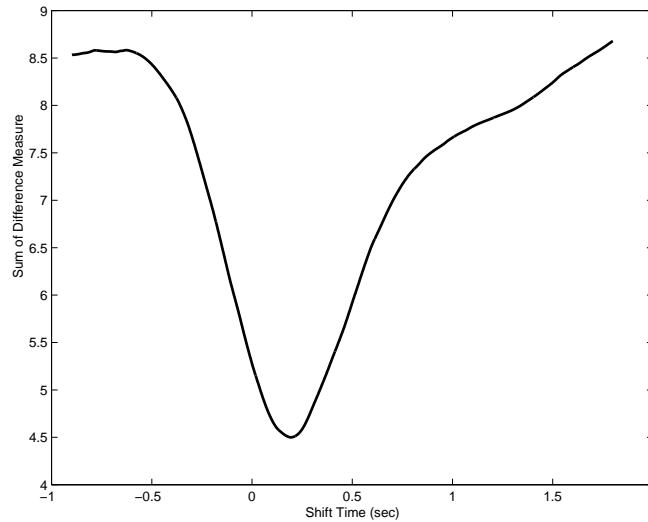


Figure 9.6 The summed difference measure attains a minimum at $r = 194$ ms.

This method was also used to estimate the closed loop system delay by using the time stamp of the vision feedback, as received by the controller. Total system delay was found by this method to be 260 ms.

These results are consistent with the results obtained from the off-line method of Experiment 1. Using time stamping, we determine the controller portion of the overall system delay to be approximately 60 ms. The experiment described above produced a plant delay estimate of 194 ms. Adding this to the observed controller delay yields approximately 254 ms for overall system delay. This closely agrees with our result of an overall closed loop system delay of 260 ms.

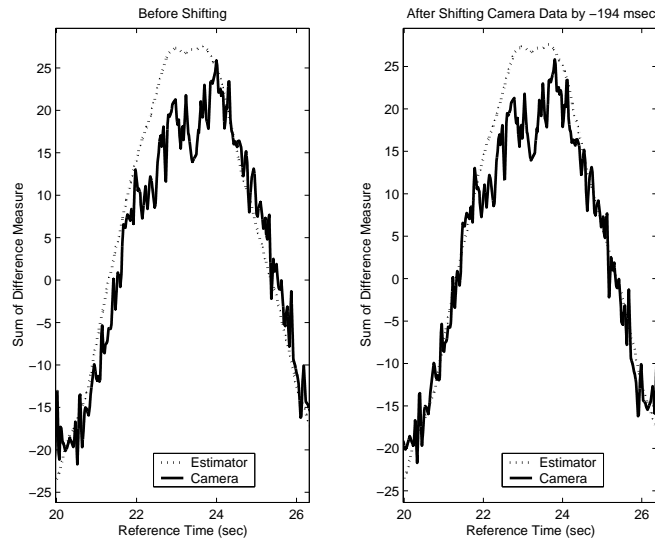


Figure 9.7 Combined observed rotations at different time instants for both cameras. Left: the combined observed rotations at each camera. Right: Shifting the curve of the combined observed rotation of the real camera backwards by the detected time shift.

A key advantage to this method is that it can be implemented online, so that it can capture dynamic changes in plant delay induced by variations in the communication network or plant dynamics.

We see above that for a visual sensor, there is a close relationship between its observed motion and the motion in the dynamical scene. This method is also a useful tool for aligning different visual sensors temporally, such as video sequence alignment, sensor clock synchronization, and plant delay estimation (see [35]).

In order to use this method, we need to reconcile clocks at the two sensors. More generally, knowledge of time is fundamental to control in general, and distributed control in particular. It is becoming increasingly important as one develops control laws for systems running over general purpose communication and computation networks. In the next chapter we describe a method for time stamping that provides valuable timing information since it makes available knowledge of the per-packet delay in information flows as well as in computations. Complementing this, as discussed in this section, is temporal alignment of distributed systems, which provides a tool to estimate plant delay online. Together they allow the design of superior controllers to stabilize systems and improve performance by providing system designers and operators insight into the timing performance of distributed systems.

9.5 Concluding Remarks

Experiments 1 and 2 show how knowledge of delay can provide a method to improve the performance or stabilize systems which would otherwise be unstable in the presence of unknown delays. Experiments 3 and 4 present a method for determining plant delay offline, while Experiment 5 presents a method for determining plant delay online. Using an online “delay meter” such as that proposed in Experiment 5, together with time stamps, can make per packet delay known, thus making the system robust to timing delays and improving its performance. As this relies on the accurate implementation of distributed clocks, we now turn attention to the problem of time stamping and estimating times on clocks.

CHAPTER 10

THE CONTROL TIME PROTOCOL

As noted in the previous chapter, knowing the delay in information packets can greatly improve the performance of control systems [33].

10.1 Challenges of Time in Distributed Systems

Time is measured by clocks, which are imperfect. Distributed clocks can synchronize through communication, but that is subject to delays. Abrupt changes in time caused by users, or automatic updates such as daylight savings and NTP, further complicate the process. We now explore some fundamental limits on the precision of estimating time in a distributed system.

10.1.1 Clock fundamentals

No two clocks tick at the same rate. Thus, over a long period of operation, the displayed time will drift with respect to a reference. It should be noted that even a single clock is not perfectly constant in its tick rate. Environmental conditions such as temperature, pressure, or even humidity, may affect the behavior of the oscillator, causing it to speed up or slow down slightly. The ratio of the tick rates between two clocks is commonly called *skew* or drift. Because drift can refer to both the rate of change of a clock with respect to a reference and the accumulated difference in displayed time with respect to an reference, we will not use the term drift. Instead, the difference in tick rates will be called *skew*, and the difference in displayed time will be called *offset*.

At any instant, two clocks will display different times, and the difference between them is called *offset*, which changes over time. To determine the *offset* we must provide communication between the clocks in the form of time stamped messages. If such communication itself had no delay, then we could instantaneously know the time of one clock at the other. In real systems, however, there is an unpredictable delay, which can prohibit one clock from ever knowing the time as measured by the other clock. Moreover, variations in communication delay also limit the precision of estimating the time on imperfect clocks.

10.1.2 Simple clock model

For purposes of this discussion, we assume that clocks are linear, i.e., *skew* is constant. Thus the displayed time on one clock is an affine transformation of the displayed time on a reference clock, with relative *skew* α , and some *offset* at some point in time, usually time $t_{local} = 0$ on the reference clock. Thus,

$$t_{remote}(t) = \alpha t_{local}(t) + offset(0), \quad (10.1)$$

where $t_{local}(t)$ is the time as measured at the reference clock, and t_{remote} is the time at the other clock.

Real clocks maintain reasonably accurate time. However, due to environmental variations such as temperature change, the oscillator in a clock can speed up or slow down slightly, causing the clock to drift. In addition, the clock can be changed by external sources, such as NTP, which will be discussed in Section 10.3.1.

10.1.3 Representation

The precision of time representation depends on the number of bits allocated by the programming language and operating system to represent a time or date stamp. Programs written in C represent the time of the clock using the number of seconds which have elapsed since 1 Jan 1970 at 12:00AM. This value is stored as a 32-bit integer. It also provides a microsecond measure which must be added to the seconds when comparing times with subsecond precision. The Java programming language provides one 64-bit number which represents the number of milliseconds elapsed since 1 Jan 1970, 12:00AM.

Computer systems rely on quartz crystal oscillators which have a frequency near 32 kHz, allowing for a clock resolution of about 31 ms. Hence, the resolution of time available on a computer system varies with programming language, operating system, and hardware clock constraints. Most common programming languages and operating systems today can provide precision to milliseconds at reasonable cost. Thus, widespread usage of our timekeeping protocol may be limited to systems which need time stamping precision no finer than milliseconds. Since communication network delays are often on the order of milliseconds themselves, we would expect higher performance systems to require specialized communication or computation systems anyway.

10.2 Determining Offset with Communication Delay

The first question that arises is the following: What is the fundamental constraint on how well we can determine the *offset* of one clock with respect to another, given that they communicate, but with some unknown delay? To examine this, we will begin with the simple case involving perfectly linear clocks and fixed, noiseless delay. We assume that the two components are distributed, with no means to communicate outside of the network channel. We make no assumption of symmetry on the fixed delays. That is, the delays incurred by packets from node A to node B can be different from the delays incurred by packets from node B to node A. But we will assume delays in a given direction are always the same.

10.2.1 Indeterminate offset in the presence of asymmetric delay

We shall now show that even with perfect clocks and fixed noiseless delay, it is impossible to uniquely determine the *offset* between two clocks. Consider a repeating sequence of time stamped messages such as in Figure 10.1.

The time stamps S_1, R_2, S_3, R_4 , etc., are in terms of the local clock, while R_1, S_2, R_3, S_4 , etc., are in terms of the remote clock. We will use the local clock as a reference; hence we use the notation LR_1, LS_2, LR_3, LS_4 for the virtual local reference times for the remote times

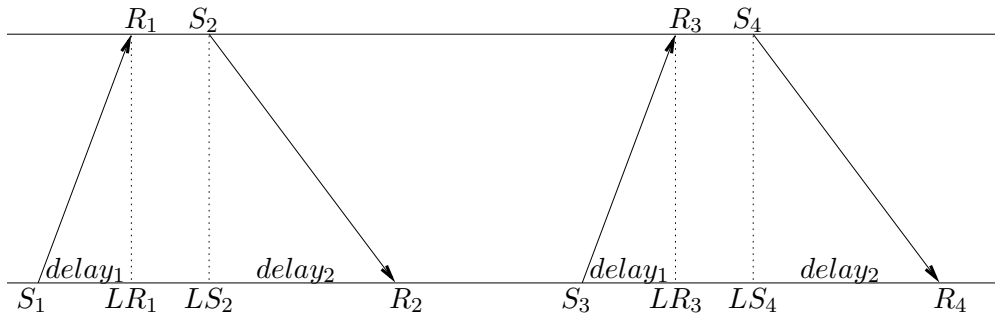


Figure 10.1 Message exchange for offset determination.

R_1, S_2, R_3, S_4 , respectively. We assume fixed but asymmetric delay, so

$$\begin{aligned}
 LR_1 &= S_1 + delay_1, \\
 LR_3 &= S_3 + delay_1, \\
 &\vdots
 \end{aligned} \tag{10.2}$$

while

$$\begin{aligned}
 R_2 &= LS_2 + delay_2, \\
 R_4 &= LS_4 + delay_2, \\
 &\vdots
 \end{aligned} \tag{10.3}$$

and so on. Note that both delays are measured in terms of the reference clock. Translating the local times into remote times gives

$$R_1 = LR_1 * skew + offset(0) = S_1 * skew + delay_1 * skew + offset(0), \tag{10.4}$$

$$R_3 = LR_3 * skew + offset(0) = S_3 * skew + delay_1 * skew + offset(0), \tag{10.5}$$

$$\vdots \tag{10.6}$$

while

$$S_2 = LS_2 * skew + offset(0) = R_2 * skew - delay_2 * skew + offset(0), \tag{10.7}$$

$$S_4 = LS_4 * skew + offset(0) = R_4 * skew - delay_2 * skew + offset(0), \tag{10.8}$$

$$\vdots \tag{10.9}$$

and so on, These relationships can be captured in matrix form as follows.

$$\begin{bmatrix} R_1 \\ S_2 \\ R_3 \\ S_4 \\ \vdots \end{bmatrix} = \begin{bmatrix} S_1 & 1 & 0 & 1 \\ R_2 & 0 & -1 & 1 \\ S_3 & 1 & 0 & 1 \\ R_4 & 0 & -1 & 1 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} skew \\ skew * delay_1 \\ skew * delay_2 \\ offset(0) \end{bmatrix}. \quad (10.10)$$

The leftmost vector and middle matrix each contain known information, assuming the time stamps of all transactions are pooled between the nodes. This could be done, for example, by exchanging them in later packets. Thus, this system of equations can be used to solve for the four values: $skew$, $skew \times delay_1$, $skew \times delay_2$, and $offset$. However, the middle matrix has rank 3, since the fourth column is the difference between the second and third columns, regardless of how many additional messages are exchanged. Therefore, we can never uniquely determine all four values, even with an infinite exchange of messages. To illustrate this, we provide two different remote clocks in Table 10.1 which have identical skew 1, but different $offsets$. Their delays are fixed. Yet both result in exactly the same sent and received times. As seen from Table 10.1, it is impossible to differentiate between $offset$ and delay effects precisely.

Table 10.1 Two different offset/delay combinations which produce identical time stamps.

Time stamps	Example 1	Example 2
$trueoffset$	100	93
$delay_1$	5	12
$delay_2$	10	3
S_1	0	0
R_1	105	105
S_2	110	110
R_2	20	20
S_3	30	30
R_3	135	135
S_4	140	140
R_4	50	50

Even though the $offset$ cannot be determined precisely, it can be bounded, as we show in Section 10.2.5. We should note that in the special case that the delays are indeed symmetric, or if either delay is known, it is possible to uniquely determine the $offset$. It is also possible to

uniquely determine the *offset* in the noisy case if the delays in both directions have the same means.

10.2.2 Least squares estimation of *skew* and *offset*

Suppose we have two perfectly linear clocks between which we desire to characterize the skew and *offset*. Assuming noisy communication delays, but with symmetric means, we wish to compute the linear relationship between the two clocks, i.e., α and $offset(0)$ in $t_{remote}(t) = \alpha t_{local} + offset(0)$.

To estimate the *skew* and *offset*, we may capture a large set of data over a long time period and make estimates. As real systems have noise and varying delays, we wish to use algorithms which suppress the noise through some sort of averaging or filtering. A reasonable choice for this is the simple linear regression model implemented with a linear least squares estimation algorithm [36].

We collect data in a series of short pings. These are messages which are time stamped, sent, and then immediately returned after being time stamped as, say, y , by the recipient. Upon receipt by the original sender, the returned message is time stamped a third time as shown in Figure 10.2. From causality, we know that the time at which the message was returned by

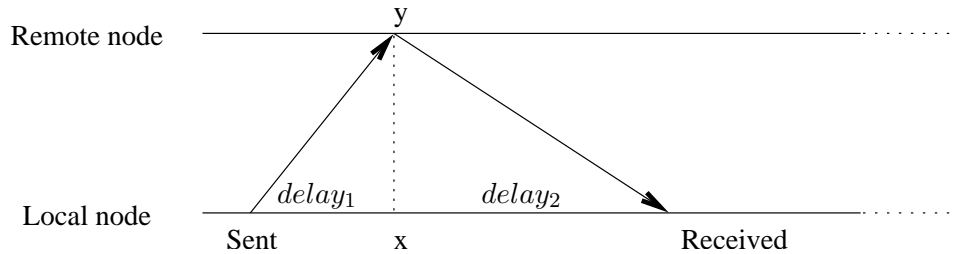


Figure 10.2 Ping message exchange for offset determination.

the remote node must have occurred between the sent and received times of the sender. As a result, we can bound the offset between the nodes above and below. Because it is impossible to determine asymmetric delays anyway, and because it yields the minimum error, we will assume that the reply of the responding node occurred at the midpoint between the sent and received times, or $x = (Sent + Received)/2$ in the local nodes reference frame.

We now collect the data from a number of such pings. After receiving a sufficient number of samples to reduce the effect of the noise in delays, we can apply the least squares algorithm. Let the pair (x_i, y_i) refer to the local and remote times as constructed above from the i^{th} ping. That is, y_i is the ping time stamp, and x_i is the midpoint between the Sent and Received time stamps. Let

$$\begin{aligned}\bar{x} &= \frac{1}{n} \sum_1^n x_i, \\ \bar{y} &= \frac{1}{n} \sum_1^n y_i, \\ \sigma_x^2 &= \frac{1}{n} \sum_1^n (x_i - \bar{x})^2, \\ S_{xy} &= \frac{1}{n} \sum_1^n (x_i - \bar{x})(y_i - \bar{y}).\end{aligned}$$

The best linear fit to the data $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ is

$$y - \bar{y} = \frac{S_{xy}}{\sigma_x^2} (x - \bar{x}), \quad (10.11)$$

where $\frac{S_{xy}}{\sigma_x^2}$ represents the estimate of the *skew* α between the clocks, and \bar{y} represents the *offset*(\bar{x}) at local time \bar{x} .

From the estimated values for α and *offset*(\bar{x}), we can then compute an estimate of the time y on the remote clock at any time x on the local clock.

Unfortunately, the least squares algorithm has one problem. It weights all data equally. If the clocks experience abrupt changes, the algorithm will not adapt to the change for a very long time.

10.2.3 Numerical issues in recursive least squares estimation

To avoid the excessive memory requirements of storing a long history of data, we turn to the recursive form of the least squares algorithm. We begin with the equation for the model,

$$y_{n+1} = \phi_n^T \theta, \quad (10.12)$$

where

$$\phi_n = \begin{bmatrix} x_n \\ 1 \end{bmatrix}, \quad (10.13)$$

$$\theta = \begin{bmatrix} \alpha \\ \text{offset}(0) \end{bmatrix}. \quad (10.14)$$

The recursive updates are formed using the matrix P_n , computed at every iteration, which is the inverse of the sum of the outer products of the regression vector ϕ_n .

$$P_n = P_{n-1} - \frac{P_{n-1}\phi_n\phi_n^T P_{n-1}}{1 + \phi_n^T P_{n-1}\phi_n}, \quad (10.15)$$

and

$$\hat{\theta}_{n+1} = \hat{\theta}_n + \frac{P_{n-1}\phi_n}{1 + \phi_n^T P_{n-1}\phi_n} (y_{n+1} - \phi_n^T \hat{\theta}_n). \quad (10.16)$$

Recall that in this relationship, the second element of $\hat{\theta}$, namely $\text{offset}(0)$, refers to the *offset* at time $t_{ref} = 0$. While this algorithm has taken into account the long history of observations, it has not yet taken care of the equal weighting problem which can prevent adaptation of the algorithm to abrupt clocks shifts. To eliminate this, we may introduce a forgetting factor $0 < \lambda < 1$ into the recursive least squares algorithm as follows:

$$P_n = \frac{1}{\lambda} P_{n-1} - \frac{1}{\lambda} \frac{P_{n-1}\phi_n\phi_n^T P_{n-1}}{\lambda + \phi_n^T P_{n-1}\phi_n}, \quad (10.17)$$

and

$$\hat{\theta}_{n+1} = \hat{\theta}_n + \frac{P_{n-1}\phi_n}{1 + \phi_n^T P_{n-1}\phi_n} (y_{n+1} - \phi_n^T \hat{\theta}_n). \quad (10.18)$$

For rapid adaptation, λ may be chosen as 0.995. For very slow adaptation, perhaps 0.9999 may be appropriate.

Unfortunately, this algorithm is not stable due to the ill-conditioning of the matrix P . Considering that many representations of time begin on Jan 1, 1970, the numbers involved get very large. At the time of this writing, the Java representation of time is on the order of 1.07×10^{12} . We can compare this large (and growing) number to typical values of skew. Although not perfect, standard clocks keep time quite well. Hence the skew between them is very close to unity. The significant digit in skew is typically the 7th or 8th digit of decimal precision.

The resulting matrix P has disproportionate eigenvalues. For example, in one sample data set, at one update, the eigenvalues of P were found to be -0.1 and 4.4×10^{-29} . Such huge disparities arise from the large time numbers and the relationship to the precision required in the

skew. The disparities in the eigenvalues lead to numerical instabilities. Double floating point representations provide 53 binary bits of precision, which corresponds to 15 decimal places. The large numbers involved in the recursive updates exceed the precision capabilities of the double floating point representation. That precision is crucial to correct implementation of recursive least squares.

While it is theoretically possible to normalize the times to obtain better numeric conditioning, experiments with real data sets revealed that the instability remains. Intuitively, this is because although we may reduce the sizes of the numbers involved in the time data sets (x, y) , we do not reduce the precision required by those representations. In fact, what is of most interest to us in these computations are the least significant digits. Thus we turn to a stable algorithm which sacrifices some implementation complexity, and a bit of memory, for a stable accurate result.

10.2.4 Windowed least squares

We will continue to use the least squares approach as the basis for our proposed Control Time Protocol. However, instead of fitting our estimates to the entire history of observations, we will only fit the data to a moving window of the data. This will allow for rapid adaptation of the algorithm. In practice, we have used a history of 1000 data points with good success. This algorithm is further augmented by a few practical filters to improve its robustness to outliers in delay and sudden shifts in *offset*.

For robustness to the possibility of large delay, we simply ignore data that returns with a large round trip time. If a packet's delay exceeds a threshold — say a multiple of the average delay — we do not use it in the computations. As the *skew* does not change very quickly, we can easily afford to ignore data packets that are likely to be outliers.

To overcome the shifts in *offset*, we recall from Section 10.2.2 that the actual *offset* during any exchange of packets can be bounded above and below. Therefore, when our estimate of the *offset* at the current time exceeds these bounds, we hold it to the bound rather than using the estimate. We now show how these bounds may be determined.

10.2.5 Offset bounds

Consider the ping exchange of Figure 10.2. Because this system is causal, we know that the ping reply time y occurs after the *Sent* time (since, $delay_1 > 0$). Hence, we know that $offset(Sent)$ is less than $y - Sent$. (This quantity can be computed offline after the message is returned.) Similarly, $delay_2 > 0$, so $offset(Received)$ is greater than $Received - y$. We know the elapsed time on the remote clock between *Received* and *Sent*.

Now we make the assumption that $offset(Sent) \approx offset(Received)$, which is a good approximation because the *skew* is very nearly 1. From this, we bound the *offset*. For $t \approx x$, $Received - y \leq offset(t) \leq y - Sent$.

To use these bounds, we simply run the windowed least squares (WLS) algorithm to obtain our best guess as to the current skew and *offset*, then check the resulting *offset* against the bounds computed above. If the result of the windowed least squares algorithm exceeds either of the bounds, we simply use the corresponding bound as the current *offset*. This may continue to be used at each iteration until the LS algorithm returns to within the bounds. Note that just as the offset shifts over time, so too the offset bounds must shift over time. This is easily accomplished by extrapolating using the current estimate for skew, which is $\frac{S_{xy}}{\sigma_x^2}$.

10.3 Time Translation Instead of Synchronization

Using the windowed least squares algorithm, incorporating a check for violation of offset bounds, is good architecture, following a use versus depend relationship that can simultaneously provide high system performance on average, while maintaining system robustness to large perturbations to the system. This then forms the basis of the Control Time Protocol to be discussed further in Section 10.3.3. Before doing so, we will look at an alternative, the Network Time Protocol, which we find unacceptable due primarily to operational considerations.

10.3.1 Synchronization using the Network Time Protocol (NTP)

For some systems, particularly business systems, the goal is to synchronize the clocks such that the *offsets* are always small. This is typically accomplished through NTP and facilitates

record keeping across multiple systems. Thus the purpose of NTP is not control, but book-keeping. However, NTP serves as a good example and comparison for the protocol we propose. We now present the applicable NTP algorithms.

The NTP model [37] is a master-slave hierarchical synchronization model. A few highly accurate national and international clocks serve as the top level source of time information. Beneath these clocks are a relatively small number of secondary time servers which use the primary clocks to keep themselves accurate. These servers are then used to keep large networks of systems accurate.

Synchronization is accomplished through infrequent exchange of time stamp information. A client will time stamp a request, and send it to an NTP time server. The server will then add its current time stamp and reply. The client then time stamps the reply when it arrives. The NTP algorithm then aligns the midpoint between the sent and received times at the client machine with the reply time of the server, in exactly the same way and for the same reason that we chose to do it in the Control Time Protocol. This alignment is natural in the absence of additional knowledge such as the existence of asymmetric delay. The client will then adjust the clock of the client host machine to reduce the *offset*. The accuracy of NTP is thus limited by minimum round trip message delay times, just as the Control Time Protocol.

Over time, ping exchange times are observed and the client can determine the skew, or drift, of the local machine with respect to the server in much the same way as the windowed least squares estimate, with additional filters. As confidence in the estimate of skew grows, the client then begins to adjust the local clock to drive the offset to zero. (Note that because the clock is continually being adjusted, there is no notion of a constantly growing offset.) With a good estimate of skew, even without an exchange of messages, the client can continue to adjust the clock, providing accurate time service even in the event of temporary network outages, with minimal loss of accuracy. The NTP service allows for a client to synchronize with many servers, thus increasing the reliability through redundancy. The filters improve the robustness and performance of NTP.

10.3.2 Control issues when using NTP

For systems which employ the NTP service, time stamp translation is simple. Remote time stamps are simply assumed to be the same as local time stamps. This however has some errors, since clocks may diverge for a while before being reset. This depends on the particular implementation of NTP, and the capability of the underlying operating system. For example, Unix based systems provide a mechanism to slowly change clocks, whereas Microsoft Windows based systems only provide a facility to change the time. Thus, with respect to a reference time, such systems running NTP exhibit a sawtooth pattern as seen in Figure 10.3,¹ which may be unacceptable for certain control applications.

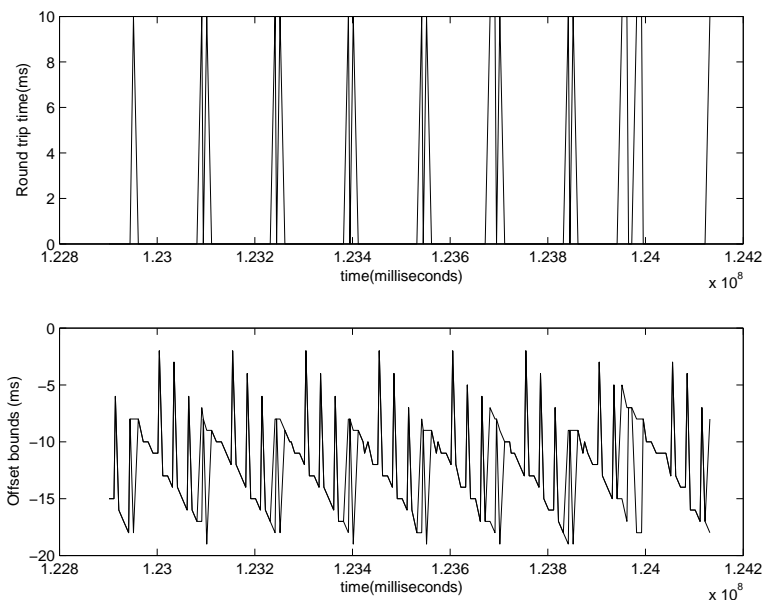


Figure 10.3 Jitter in offsets between two NTP enabled machines, studied through the Control Time Protocol.

Another problem is that systems employing it are now open to failures in the NTP service, which is an independent process running on each machine. NTP can fail for many reasons. It may not be installed, or may not be started at all. If servers are unavailable for a long time, the drift may be significant enough that when they become available, the NTP algorithm will reject them as being too far off.

¹This study of the behavior of NTP is only possible because we have another tool, our own Control Time Protocol, to study NTP's time performance.

The bottom plot in Figure 10.3 contains the upper and lower offset bounds computed from captured time stamps over a period of 100 s. The top plot is included as a reference to show the round trip delay times experienced by packets exchanged between two Windows 2000 machines running NTP. The sharp spikes are attributable to the fact that the data was collected at the application level of the operating system. Thus, if the replying process on the remote machine is swapped out when a packet is sent, that packet shows a delay of 10 ms, which is the minimum time to schedule processes in most operating systems. To read these plots, one must ignore the sharp spikes and focus on the underlying sawtooth pattern that exists. Note that the offset bounds are nearly zero, as expected in a pair of systems running NTP.

Another problem is that in a distributed control system, all the sensors and controllers may not use the same NTP server to synchronize the clocks. Thus, while each is synchronized to the NTP server of its choice, two machines have an *offset* as a result of the two NTP servers having their own *offset*. Moreover, if one or both become unsynchronized somehow, the system has no way of detecting this and will continue to assume the time stamps are equal. Thus, undesirable dependency on external entities is introduced.

To use NTP, the NTP client must have permission (usually root) to change the local clock. In some applications, this may not be acceptable. Perhaps the sensor and controller belong to different organizations which each have their machines synchronized to their own organizational time, and changing the time on a particular machine is not permitted. Closing a control loop between the systems will thus cause time stamp problems.

Using the NTP service also requires that each machine have an NTP process running at all times. This process must be started and configured independently from the control software. This produces an architectural dependence on NTP, which is out of the control of the system designer. This can be troublesome for widespread proliferation of networked control systems.

10.3.3 Control time protocol

The time stamp collection of the Control Time Protocol (CTP) functions much like NTP in that aligned *offset* pairs, the (x_i, y_i) of Section 10.2.4, are stored, and within a sliding window are fit to a line using the Least Squares method. This line can then be used to determine the

offset at any point in time. CTP is thus a time translation service providing pairwise time translation between communicating nodes. CTP does not change the time displayed on the clock, it only interprets it. CTP maintains a history of *offsets* as a look up table, in order to translate old time stamps, as well as a capability to translate future time stamps.

10.3.4 Differences between NTP and CTP

There are three fundamental operational differences between NTP and CTP. First, CTP does not change the time on any clock. Second, CTP runs within a control application, not as a separate process on a machine. Therefore, if a control application with an integrated CTP functionality is installed in a computer, the control time protocol would be present in the application, rather than requiring a separate installation and configuration as with NTP. Architecturally, this reduces the dependence of the control application on outside functions. Lastly, while a system running NTP is synchronized to a small set of time servers through pairwise message exchanges with the servers, CTP requires pairwise communication between any nodes that are communicating. This could prove to be a bottleneck if the number of sensors and controllers became large. This is mitigated somewhat by the fact that the time-keeping is required only on sensor-controller pairs that are currently communicating, and by piggy-backing time message exchanges, the additional bandwidth consumed by CTP would be negligible.

As an optimization of CTP, consider how CTP might operate when NTP is also operating. Because NTP is constantly changing the clock time, the Control Time Protocol is not likely to have accurate estimates. A reasonable approach, therefore, is to allow NTP to run, and do no time translation. However, should NTP cease to function, CTP should again resume estimation and translation. This is easily accomplished. The key observation is that the *offsets* between two machines should be small. If they are not, either NTP is not working, or the two machines are slaved to separate machines which themselves have some large *offset*. This motivates the following algorithm. When the *offsets* observed by CTP are smaller than some threshold, assume NTP is functioning well and do no time translation. In other words, assume the times are the same on both machines. But continue to compute *offsets* via CTP. If the *offsets* exceed the threshold, resume translations. This then provides the ability to operate alongside NTP without depending on it.

10.3.5 Measuring the performance of NTP using the Control Time Protocol

In fact, CTP can be used to monitor the behavior of NTP. Figure 10.4 shows actual *offset* data captured between two windows machines, using a simple Java program which exchanged time stamps. One of the machines was using NTP at the time. Note that if both machines were using NTP, the *offset* data would be fairly close to zero.

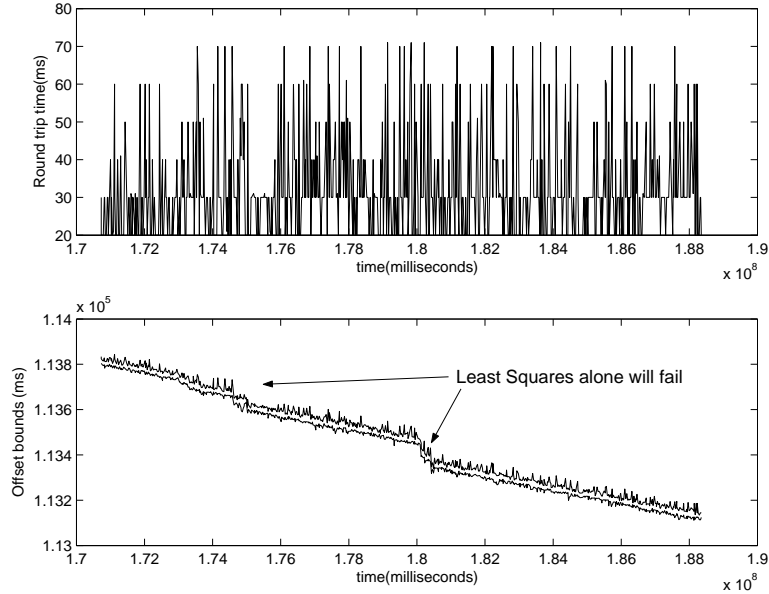


Figure 10.4 Linear skew with infrequent step changes.

The abrupt steps in Figure 10.4 were approximately 90 min apart, possibly induced by NTP which will be discussed in Section 10.3.1. The larger step occurred over 8 min, shifting the *offset* by a total of 150 ms. These correspond to actions taken by NTP.

10.4 On-Line Measurement of System Delay

Some delays can be determined through time stamping, i.e., delays between nodes with clocks each with two-way communication capability. As it is impossible to use time stamps at the radio controlled car, we cannot isolate individual delays from command to execution. Physical delays include those due to inertia, clock shutter, and propagation delay. When these delays are in series, they cannot be isolated. In our system, the first meaningful time stamp

can be taken at the moment the vision system receives a frame. From there, through the DataServer, to cars, the delay can be obtained through the use of time stamps, to within some margin of error. If the entire closed-loop system delay can be determined through some other means, such as those presented in earlier sections, we can then determine the combined total of physical or unobservable delays.

Using a time translation capability such as that provided by CTP, it is possible to provide an on-line “meter” showing current delays experienced in a system. We are currently developing such a meter, giving system operators a tool to analyze system performance, both in system development and testing, as well as in the operational environment.

CHAPTER 11

TESTBED DESIGN

11.1 Interfaces

As argued previously in Chapter 2, the essence of systems design is the establishment of interfaces. While it is relatively straightforward to modify code within a single component, changing interfaces creates a cascade of changes into multiple components. Therefore, a primary consideration has been the establishment of the core set of interfaces crucial to evolution and reliability. Although we present these in the particular context of our testbed, our approach is quite general and can be applied to many other networked control systems, and in Chapter 12 we sketch out an example application.

Since interfaces exist between components, we will present them in conjunction with the component pairs that require an interface. Components requiring external interfaces include controllers, sensors, actuators, StateEstimators, planners, and supervisors.

11.1.1 Controller to actuator interface

The interaction between a controller and an actuator is effectively one way, as shown in Figure 11.1. The controller provides controls which the actuator carries out. While an actuator may provide feedback to the controller, we consider such potential feedback to come from a sensor and therefore do not model it in this interface. One reasonable exception to this view is to have the actuator provide an acknowledgement or otherwise indicate its willingness or ability to perform the requested controls.

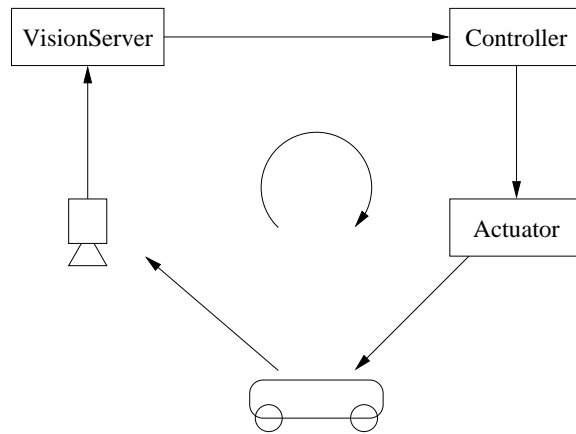


Figure 11.1 Initial components.

We assume that the controller may, but does not have to, provide more than the current controls. That is, a timed sequence of controls can be provided to compensate for communication losses or, equivalently, brief controller outages. This provides the requisite local temporal autonomy.

As actuators are likely to differ in the controls they can receive, as well as the formats of the signals for these controls, it is expected that the control signals themselves are application specific. The radio controlled cars used in the testbed have been designed to receive a pair of controls, speed and steering. In particular, the controls are provided as letters from A to V, with speed indicated by uppercase characters and steering indicated by lowercase characters. Crucial to our implementation of the model predictive controller is the assumption that the speed and steering control labelings are monotonic. While not explicitly necessary for proper operation of a generic model predictive controller, monotonically labeled controls simplify the search routines used to create control sequences.

The interface from controller to actuator may be described in the following format: One way, perturbed periodic messages containing control commands to be enacted by the actuator. Controls range from uppercase A-V and lowercase a-v, with l indicating straight ahead, L indicating stopped, A maximum reverse, V maximum forward, a maximum left turn, v maximum right turn. Messages contain sequences of control pairs, with associated dwell times per pair, i.e., hold straight “l” for 200 ms, then turn at angle (command) d for 300 ms, followed by straight “l” for 400 ms. The actuator will not block waiting for updates. That is, if updates

have come in, old controls are discarded and new ones fill the command buffer; otherwise, the actuator merely performs according to previously provided commands. In the event that the actuator command buffer is exhausted, the actuator will stop the car.

Not mentioned in the interface is any assumption on how fast the actuator is able to provide updates to the car. We assume that can be derived as part of the overall system delay loop according to Section 10.4.

We now consider how this interface allows for evolution. We assume that evolution in this case applies to the controllers, as replacement or upgrade of actuators renders the controller status temporarily irrelevant. In contrast, upgrading or otherwise evolving a controller can be done without effect on the actuator. We will assume that the upgrade must be done online and reliably. Thus, the upgrade will be performed in three stages. The first stage swaps the current controller for a modified controller with a wrapper around it that includes a monitor. This monitor merely evaluates the state of the system and determines whether a new controller is safe. The second stage involves dynamically adding another controller into the monitor's system and starting it properly. When the new controller appears to be ready, the monitor will switch control to it, leaving the second controller operating as a reliable backup. Provided the new controller functions properly, the system will operate with the new controller for some time. In the event of trouble, the monitor reverts control back to the previous controller. Ultimately, the monitor may shut down the backup controller and remove itself from the data and control path. However, it is also possible that the monitor would continue to remain in place, and call the new controller the reliable backup controller in anticipation of another upgrade. Exactly what rules to follow for switching can be application specific. For noncritical applications, perhaps it is enough to simply swap in the new controller.

Such online upgrade has been accomplished in the testbed as has thus far performed as expected.

11.1.2 Sensor to controller

It is assumed that any communication from a sensor to a controller traverses a general purpose network. Therefore, hard real time guarantees may not be feasible. Also, late updates

may be obsolete if more recent observations have arrived. Therefore, sensor communication in the testbed is sent via a communication protocol that does not provide reliable delivery, which in fact can only be provided at the price of delaying future updates. This data stream is provided as a basic service of the Etherware and is called an event pipe. It is based upon the User Datagram Protocol (UDP). Updates are not assumed to arrive in order, or in a perfectly periodic fashion. We have called this data stream a “perturbed periodic” stream. An event pipe can also provide reliable delivery, and exact requirements for communication can be listed when requesting an event pipe.

In general, a system composed of many sensors requires data fusion from many sources. In the testbed, the data from two VisionServers is composed together in a component called the FeedbackServer. The interface between the FeedbackServer and the VisionServers is identical to the interface between the FeedbackServer and the controllers. These interfaces follow a client-server type architecture wherein the FeedbackServer requests data from the VisionServers, and the VisionServers make the data they have available to the FeedbackServer.

The format of the data in the above two interfaces is slightly different. The controller need only receive the data for its own specific car, whereas the FeedbackServer must receive all of the data in order to disseminate it.

However, in both cases, the data is sent as an Etherware event, in the form of a well formed XML document, capable of describing the data, and therefore distinguishing between the two cases. As a result, a controller can make a request of a VisionServer just as well as from the FeedbackServer. If the VisionServer has data for the car, it will provide it to the controller.

The advantage of this interface structure is that upon failure of a FeedbackServer, a controller is capable of obtaining the data autonomously. Moreover, if another data service becomes available which can advertise its services, the controller can obtain that data as well, without any change to the communication structure of the system. It must be understood, however, that the controller may require an upgrade in order to utilize the additional or modified data. That is possible via online upgrade as described previously.

11.1.2.1 StateEstimator

All updates from sensors, whether via a data fusion element or other filter, can be regarded as information in the form of a new observation. In the case of the testbed, the observation includes an x - y position and an orientation. Because time stamps make delay known, it is assumed that all sensor updates will be time stamped by the system.

In order to use the time stamp information, a StateEstimator component, capable of maintaining a small window of past updates and properly ordering them, must intercept the sensor updates to the controller. Moreover, the StateEstimator may also receive the history of controls issued to the actuator in order to reduce sensor noise impact as well as buffer against sensor loss or brief sensor errors.

It is assumed that the StateEstimator can provide current estimates on demand. Therefore, the StateEstimator must be collocated with the controller to avoid any delay in obtaining an update. Should the layers above the controller desire to have access to the results of the StateEstimator, they may subscribe to the services of the StateEstimator as needed or allowed by the system.

In the testbed, sensors are called VisionServers and they provide observations to the FeedbackServer. This interface is essentially one way, with observations provided from the VisionServers to the FeedbackServer as fast as possible, though with no periodicity guarantee.

The interface from the FeedbackServer to the StateEstimator is typically one way, with the sensor providing updated observations regarding the current state of the physical system. Therefore, the sensor to StateEstimator interface consists of periodic (or possibly perturbed periodic) messages containing the latest observations on the state of the system.

The interface from the StateEstimator to the controller is inherently two way. In one direction, the StateEstimator provides current estimates of the state of the system. In the case of the testbed, this includes the position and orientation of the car of interest at the time indicated by the request. In the other direction, the controller provides two types of signals. One is a request to receive an estimate corresponding to a particular time, and the other includes the current commands sent to the actuators.

Note that the interface expected by the Controller from a sensor is contained in the above design interface. If no StateEstimator is provided, the Controller simply asks for an update from the FeedbackServer directly. If no FeedbackServer is provided, the Controller can connect directly to one or both VisionServers. In that case, when an update arrives, it is simply assumed to be current, or perhaps delayed by some predetermined and fixed amount of time. This flexibility allows the system to evolve according to the need. Some applications have no need for state estimation; therefore, it should not be absolutely required. Having the ability to fold it in later, however, gives the system flexibility for evolution, as well as improving performance.

11.1.3 Actuator to sensor interface

This connection is made through the real world. Although not part of the software architecture, the physical interface has an influence on the evolvability of certain portions of the system. In our case, the interface includes the placement of color patches on the roofs of the cars. It includes assumptions on vision system reliability, lighting brightness, and color patch arrangement uniqueness, etc. Although this interface does not exist in software, it is critical and should be explicitly stated in the design. If cars which have other color patterns are placed on the testbed, they will clearly not be capable of receiving updates. The color configuration changed several times over the development of the testbed, each time requiring a simultaneous change to the vision sensor software. Note that these changes have no fallback or undo capability. Running the older version of the vision sensor code simply will not work, illustrating that not all changes to a system can be handled through techniques such as the Incremental Evolution pattern.

In the testbed, this interface carries several assumptions. Namely, no two cars have the same set of colors. The loss of one or two color patches does not render the identification of the car impossible. The color schemes must be separated by a reasonable Hamming distance. In our system, even if two colors out of six are lost, a car can still be distinguished from other cars. It is assumed that the sample rate of the sensor is fast enough for the application.

The design further assumes that cars will behave according to the commands provided by the controllers via the actuator module. If this assumption is violated, perhaps due to weak

batteries, or some obstruction, various application services must be made aware to prevent further system damage. For instance, if the car will not move, but its desired setpoint or target position is indeed moving, the cars will be asked to go faster and faster, perhaps outside of the safe and tested dynamic range of the system. With adaptive calibration or control, this can cause the adaptation to err, resulting in incorrect behavior of the controller.

11.2 Interface Layering

The previous interfaces have all been concerned with the lowest level of control. They form the basis of a single control loop, perhaps being used by multiple controllers. It is important to note that these interfaces are sufficient to describe, design, implement, and operate nearly any control loop. However, control loops can exist in higher hierarchical layers too. In fact, those layers are self similar, with lower level controllers appearing as actuators to the controllers above. The important abstraction is that the “actuator” is given setpoints to achieve, and provides a “best effort” to achieve them. If it cannot do so, the higher layers can be designed to adjust the setpoints accordingly.

To illustrate this in the testbed, we can provide a trajectory to a controller which follows a particular route on the “streets” of the testbed. Given no obstructions, the controller, with feedback from the sensors, is able to “drive” the cars along this route, according to the times indicated and reach the destination. Let us suppose, however, that another car happens to become stalled somewhere in the route. As the route is now blocked, the previous trajectory is no longer safe. But it should not be the responsibility of the controller to replan. This is properly the job of the higher layer of planning and scheduling. Given such a higher level of planning, the controller is thus an “actuator” to the planner.

We now continue the description of testbed interfaces, taking into account the fact that these layers are self similar to the previously mentioned interfaces.

11.2.1 LocalPlanner

We provide to each controller its own planner. The planner takes as its input goals which it must then translate into a trajectory to provide to its controller. In doing so, the planner must honor various constraints, such as avoiding collisions. Because of the safety requirements, it is requisite that this planner be collocated with the controller, just as the StateEstimator. This avoids the internode communication failure modes.

Because of the global or regional knowledge required by the LocalPlanner, it is able to perform several types of planning. One involves following a particular car in a formation. In this mode, the planner takes the input from the planner StateEstimator described below, which is an estimate of the future positions of other cars, and formulates a trajectory based upon those positions. This trajectory can be reformulated for every controller update.

Another mode is to accept a preplanned trajectory from another source and then monitor its execution for potential collisions. In the event that a potential collision is detected, various alternate trajectories can be formulated, such as speeding up or slowing down, passing on the left or right, etc. Each of these alternatives can be evaluated for potential collisions as well, and the planner can choose one which does not, or simply stop the car.

Note that a key aspect of this maneuver is that the LocalPlanner must be able to get the car back on the previous trajectory somehow, or risk violating higher level assumptions such as a global schedule. In this event, the LocalPlanner must request an update to the global schedule to avoid a cascade of potential collisions.

11.2.1.1 LocalPlannerStateEstimator

To avoid collisions, the planner must have knowledge of the states of other cars. This is obtainable via the FeedbackServer, or perhaps from the cars themselves. In either case, the LocalPlanner requires an entity to determine the global (or regional) state of the system in order to make good decisions. Thus the planner level StateEstimator may communicate with the FeedbackServer in addition to the other cars. The interface from the StateEstimator to the LocalPlanner is simply a future horizon of positions and orientations of the various cars in the

system. The StateEstimator may filter out the cars which are too far away to be of concern for collisions.

Note the self similarity available in this level. The LocalPlanner's StateEstimator includes the trajectory provided to its car controller as part of the input to the estimator. Moreover, in the case of car to car communication, each planner may provide its desired trajectory to neighboring cars. Given this additional insight, the estimator can more accurately predict the future state of the system and thereby perform better. This behavior is similar to the ability of a Kalman filter to obtain state estimates by taking into account the actual controls sent to the plant.

11.2.1.2 LocalPlanner to Tracker interface

Regardless of which mode it operates in, i.e., leader-follower, collision avoidance, or merely passing on trajectories from higher layers, plans are provided to the lower controller exactly the same way, as a series of time waypoints to follow.

In addition to the trajectory interface, the LocalPlanner has the ability to send a “stop” and subsequent “restart” command to the controller in order to guarantee immediate stopping of the car. Note that this ability bypasses the real function of the controller, effectively sending the “stop” command straight to the actuator.

11.2.2 Supervisor

Local plans may not be sufficient to ensure global behavior. Therefore, a global Planner is required to satisfy global properties. While this may be implemented in the testbed in either a centralized or distributed fashion, we have implemented it in a centralized fashion for now, primarily because distributed algorithms for collision free traffic scheduling have not yet been developed. The bottleneck is algorithmic rather than architectural.

11.2.2.1 SupervisorStateEstimator

We assume that a global entity can provide performance superior to that of a group of local entities. To do so, the global entity must have more knowledge. In the testbed, this is accomplished via a `SupervisorStateEstimator`, which can use the global plan provided to each of the cars as input, just as the Kalman filter utilizes knowledge of actual control inputs. Because the estimator has access to this additional information, it can provide better estimates of the global system state.

This becomes especially important in the testbed while accomplishing maneuvers such as parking. Because the traffic scheduling is discretized, the `StateEstimator` must evaluate the current position of cars as lying in one of a set of bins. Without the insight of which bin the car was trying to achieve, it is possible for the estimator to misclassify the bin a car is in, causing a large deviation in the replanning for that car. In some cases, the replanning can oscillate as two wrong decisions toggle the car from one plan to another.

The interface from the `FeedbackServer` to the `SupervisorStateEstimator` is exactly as in the case of the `LocalPlanner`. The interface from the `SupervisorStateEstimator` to the `SupervisorPlanner` is exactly as in the case of the `LocalPlanner`. Moreover, the interface from the `SupervisorPlanner` to the `LocalPlanner` is also the same as the `LocalPlanner` to the controller, with the exception that this communication is assumed to traverse the network because the `GlobalPlanner` is not collocated with any of the `LocalPlanners`.

11.3 Self Similar Hierarchy

This hierarchy can be continued, with `SupervisorPlanners` merely serving as “actuators” to other “supervisors,” etc., as shown in Figure 11.2. This layering and abstraction serves well in cases where the time constants of control at the layers are increasing. That is, if each layer can afford to take more time, and perhaps even requires more processing time due to the complexity of computations, an autonomous hierarchy is the appropriate architecture.

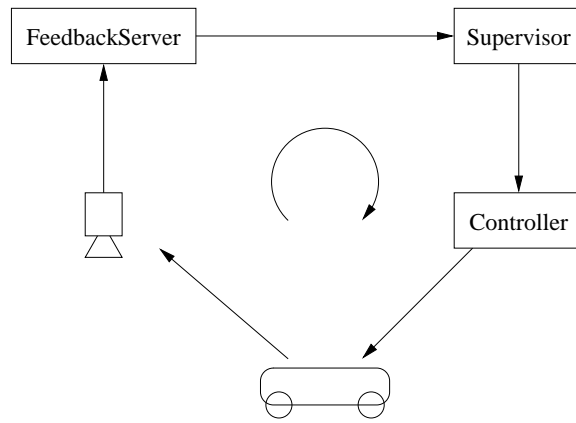


Figure 11.2 Self similar hierarchy of control.

11.4 Local Temporal Autonomy

We have alluded to the local temporal autonomy built into the testbed throughout this dissertation. To envision the extent of it, we summarize each case in this section.

11.4.1 Sensor autonomy

The VisionServers are capable of operating in the absence of any other components. Also, other components which are brought online later can connect to them as needed, at run time. Moreover, the failure and subsequent restart of a VisionServer is perceived merely as a brief outage in its service to components. The VisionServers are also not dependent on the presence or absence of cars in their view to operate. The ability to add or remove cars from the testbed is essential, and does not affect the stability of the VisionServers.

11.4.2 FeedbackServer autonomy

Because each controller connects to the FeedbackServer, it is essential that cars be able to connect to or disconnect from the FeedbackServer without creating the need for restarting any components. This is possible. In fact, the FeedbackServer can be terminated and restarted without the need for restarting any other components in the system. The loss of the FeedbackServer is simply perceived as an outage. This was accomplished through the use of robust communication protocols residing within Etherware. The Etherware can cache the communi-

cation state for a time, allowing a client to attempt to reconnect to the restarting component automatically.

The current implementation of this design has proved stable, removing all dependencies beyond the inherent functional dependencies.

11.4.3 Controller autonomy

The existence of the StateEstimators (Tracker and LocalPlanner) makes the Controller autonomous with respect to the sensors. As the Supervisor to Controller dependencies are merely functional, the Controller is autonomous with respect to the Supervisor. The Controller has no dependence on the Actuator beyond functional as well. Therefore, the Controller can operate in complete autonomy to the rest of the system. Of course, without the services provided by the other components, the Controller may not perform any useful function.

11.4.4 Actuator autonomy

Receiving a horizon of future commands makes the Actuator autonomous with respect to the Controller.

Because the system is implemented in a protected environment such as Java, component failures merely generate exceptions, which are caught and handled by terminating the component and restarting it. Therefore, even though all the components are operating in a single operating system process, they are independent of each other, leading to strong autonomy for all components. We believe this property of Local Temporal Autonomy to be crucial for the proliferation of general purpose control systems.

11.5 Algorithms

11.5.1 Model predictive control

The testbed employs model predictive control as the low level controller. Advantages include robustness and simplicity for a nonlinear system. The controller essentially searches a large

number of potential control sequences, evaluating each according to a least squares cost function, and choosing the sequence with the lowest cost. We use a horizon of six control changes, with each step equaling 300 ms. This seems to provide reasonable forecasting without sacrificing the near tracking performance. In practice, we have to limit the control sequence search space in order to compute in the time allotted. This is done by assuming that controls cannot change dramatically from one step to the next. We also fix the speed for each search. To allow for speed changes, we perform a search according to the current speed — one faster, and one slower — as well as a special fast forward speed which enables better starting performance. This is similar to the strategy used in the Sendai city train system controller [38]. Because the controller only adjusts one speed level per frame (10 Hz), it can require a full second to come up to speed otherwise.

11.5.2 Actuator

The actuator stores a buffer of future commands in order to tolerate brief controller outages. The actuator always terminates this buffer by appending the “stop” command, thereby enforcing fail-safe in the event of controller failure. Because the actuator utilizes the serial port, it must be run on a particular laptop. Its controller, however, has no such limitation. Therefore, the actuator must be able to receive commands from anywhere, and indeed, from any controller. To provide for a measure of security, the actuator will not receive commands from a new controller unless the old controller has disconnected from it.

11.5.3 FeedbackServer

The FeedbackServer simply combines the data received from the VisionServers and provides it to other components. There is no data processing at the FeedbackServer.

11.5.4 VisionServer

Vision processing is made simpler by the use of image libraries available. We use the Matrox framegrabber, and hence use the Matrox Imaging Libraries to interface with the framegrabber. Images are captured in the background every 16 ms. When the VisionServer is ready for the

next image, an image is copied from this buffer and is processed. The detail on how this is accomplished can be found in Appendix C. For purposes of this section, we note that the physical layout of the color patches used to identify the cars and determine their position and orientation was designed for fault tolerance. Up to two color patches out of the six on a car can be lost without misidentifying the car. This robustness to color loss is required to combat the fluctuations in color processing. The geometric algorithms to extract the position and orientation information were custom built for this application.

11.5.5 LocalPlanner

The LocalPlanner has several modes of operation. The most common is to defer planning to the Supervisor. However, if directed, the LocalPlanner can perform limited collision avoidance capability. This ability is not useful in the citywide traffic scheduling problem as it tends to conflict with the replanning done at the Supervisor level. Moreover, the LocalPlanner has no concept of roadways and boundaries; hence, the cars do not honor the roadway constraints. For free driving, the LocalPlanner can create a formation following trajectory wherein the LocalPlanner estimates the future positions of a leading car, and then plans a route to follow with some offset.

11.5.6 Supervisor

The role of the Supervisor is to translate goals into collision-free scheduled routes. To accomplish this, the Supervisor receives a current estimate of the state of the system, determines which portion of the street network a car is in, plans a route for that car to achieve certain objectives, beginning with its current location, and then schedules this route together with all other routes for all the other cars. This is done using a discretized representation of the roadway. This representation takes the form of a graph. For more information on how this algorithm functions, see [25].

11.6 Evolution View

We have argued for the importance of evolution in the proliferation of general purpose control systems. We have claimed that our proposed framework, implemented in Etherware, is capable of enabling evolution. To support this claim, we now present the evolution of the testbed as it would have been given the existence of Etherware from the beginning. In the subsequent chapter, in which we present a hypothetical design of a power demand response system, we show a similar evolution.

11.6.1 Testbed version 1

We will start with a set of hardware at a point in time at which it does not yet function together. At this point, we assume the existence of cars, with some way to send commands to the cars. In this version we would create an Etherware Actuator component capable of receiving Etherware actuator command events, and ultimately causing the cars to move accordingly.

We also assume the existence of a feedback system. Specifically, this version must be capable of “seeing” the cars on the testbed track, identifying them, and determining their positions and orientations. As with the Actuator component, we would build an Etherware VisionServer component which is capable of sending sensor updates via Etherware events.

The last component in this version is the controller, see Figure 11.3. We create an Etherware component capable of receiving sensor updates in the form of Etherware events and sending actuator commands in the form of Etherware events. At this point, the controller should be very simple and self contained. The controller may simply drive the car in a circle, telling it to turn left if the car is outside of the circle and right if it is inside the circle. The point of this controller is to prove the connectivity of the Etherware system.

To make this version execute, the Vision Sensor must register its services with the ProfileRegistry, and the Controller must register its service with the ProfileRegistry. Hardware connectivity must exist, i.e., the network must be available. Thus, this version closes the loop with a do-nothing configuration.

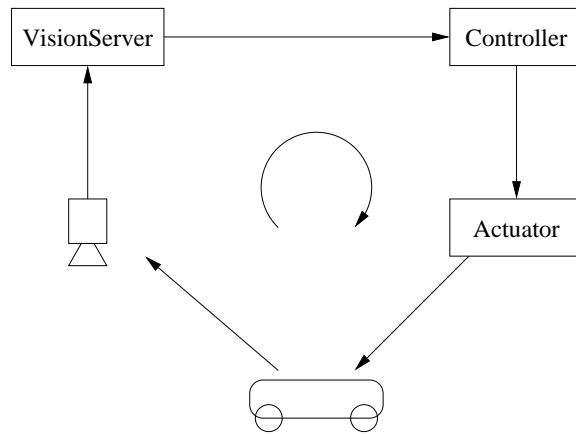


Figure 11.3 Closing the loop.

11.6.2 Testbed version 2

In this and subsequent versions, we “grow” the testbed into a more elaborate and robust system. We could add components in several different orders; we will present just one feasible order. In this version we will provide placeholders for all of the additional components required for completeness. In anticipation of multiple sensors, we will first insert a FeedbackServer between the VisionServer and the car controller, which we will call the Tracker to avoid confusion with the overall controller, see Figure 11.4. The primary function of the FeedbackServer is to provide a common source of data as sensors are added to the system. It also serves to isolate the VisionServers from the possibility of servicing multiple clients. Because the FeedbackServer represents a single point of failure, it would be feasible to replicate the FeedbackServer and have cars connect to both sources. We will not do so in this example.

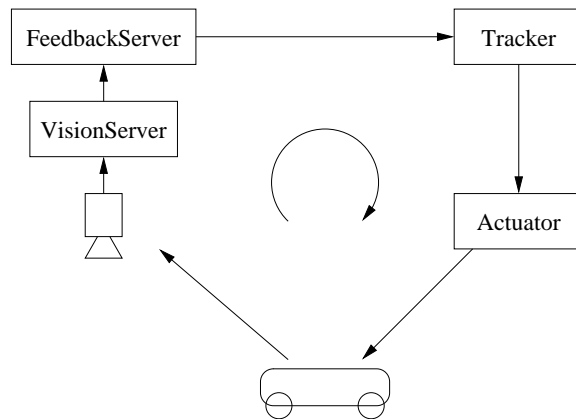


Figure 11.4 Inserting a FeedbackServer.

The first “upgrade” will be to provide a StateEstimator for a car, as shown in Figure 11.5. This version of the StateEstimator will provide a history of past observations for this car received from the VisionServer, or rather from the FeedbackServer since it is now in place, as well as control events sent from the controller. To do this, the StateEstimator must make a request for vision data covering its location, as well as implement a simple Event Handler to receive such data. The Event Handler in this first StateEstimator may simply “sample and hold” rather than implement a Kalman filter. It must also implement an Event Handler to receive estimate request events from the Tracker and subsequently send updated estimate events back to the Tracker.

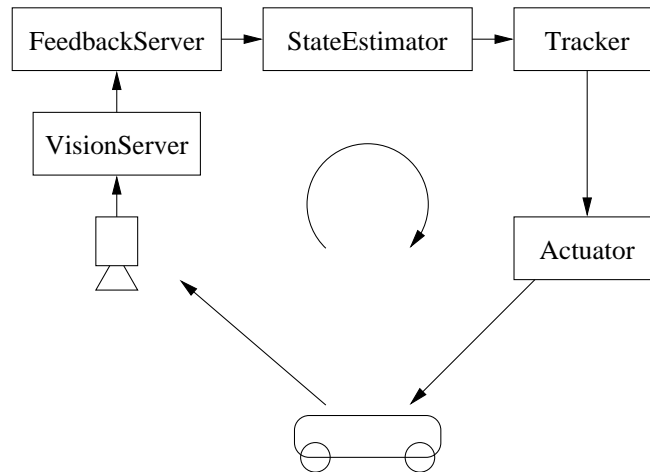


Figure 11.5 Inserting a StateEstimator.

The second “upgrade” will be to provide for a LocalPlanner, as shown in Figure 11.6. The first version of the LocalPlanner simply reads a trajectory from a file and sends it as an Etherware event to the Tracker.

A StateEstimator (see Figure 11.7) is then created for the LocalPlanner which requests sensor data from the FeedbackServer for all cars for which the FeedbackServer has information. This StateEstimator will ultimately provide a good estimate of future positions for each visible car, but for now this estimate will simply be a “sample and hold” estimate. That is, this version will assume that the cars are not moving. The StateEstimator will also have an Event Handler to receive plans from the LocalPlanner regarding future desired positions for its own car. The LocalPlanner is then configured with an Event Handler to receive this dummy estimate from its StateEstimator. This Event Handler does not do anything just yet. The LocalPlanner is

also configured to send a copy of its Plan to the StateEstimator in conjunction with sending it to the Tracker.

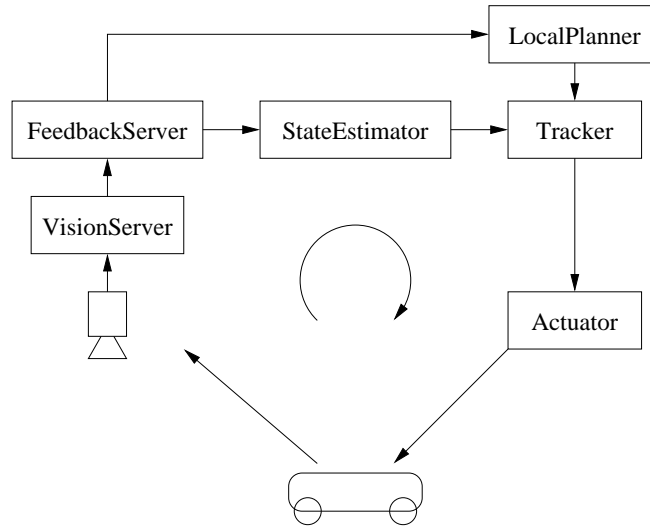


Figure 11.6 Inserting a LocalPlanner.

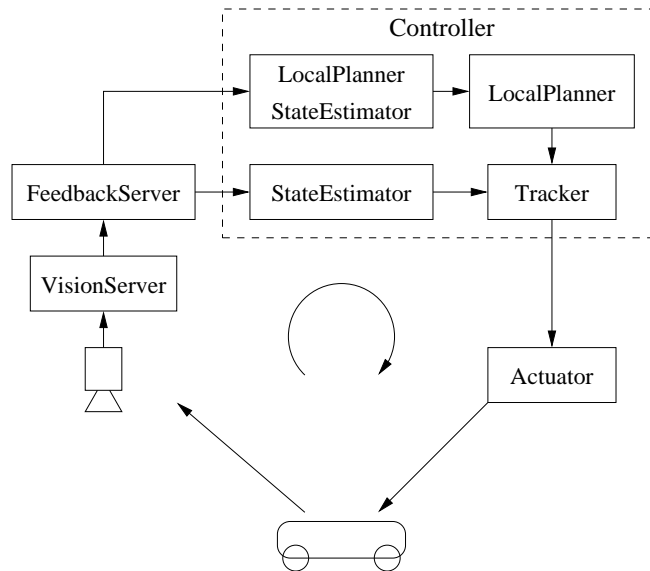


Figure 11.7 Inserting a StateEstimator for the LocalPlanner.

This completes this version in which all components for an individual car are created, albeit minimally, and any configuration problems can be discovered and corrected. This version prepares the system for algorithm insertion into the StateEstimator and the Tracker.

11.6.3 Testbed version 3

We are now ready to insert functionality into the newly created StateEstimator component. In the testbed, the StateEstimator which provides estimates of the state of a particular car is only concerned with the vision feedback for its own car. In addition to this information, the StateEstimator receives controls sent to the Actuator. It retains a window of past observations and controls in order to capitalize on any available sensor information, even if it is slightly delayed. Since all sensor and control updates are time stamped, the observations and controls can be properly ordered and accounted for in time. Thus, for every estimate, the Estimator begins with the earliest observation available and recomputes estimates using a time update and a measurement update for each observation and a time update for each control according to the Kalman filter algorithm [34]. The result is that all incoming data can be useful, even if it arrives out of order or late.

When integrating this new StateEstimator we can follow the pattern of Incremental Evolution by using the raw vision feedback to determine if the Estimator is grossly in error, see Figure 11.8. If so, a log entry is created and the Estimator is switched out. The log files help to troubleshoot the errors and improve the Estimator. Note that because the Estimator is likely to be more precise than the raw vision data, the monitor must be turned off and the Estimator used exclusively once the gross faults are removed from the Estimator. Otherwise, raw vision data errors would cause the Estimator, which corrects for such errors, to be switched out because the difference between the vision data and the Estimator would be too large.

11.6.4 Testbed version 4

This evolutionary step will upgrade the simple controller to a slightly more complex controller. To do so reliably, we will first insert a monitor and a switch according to the Incremental Evolution pattern (Figure 11.9). We then code a simplistic Model Predictive Controller and operate the testbed. If the new controller causes the car to deviate excessively from the desired trajectory, the new controller is switched out and the old controller operates the car. Meanwhile, log files are added which indicate the conditions under which the switching out occurred for troubleshooting purposes. This utilization of the switch for logging assists greatly in debugging

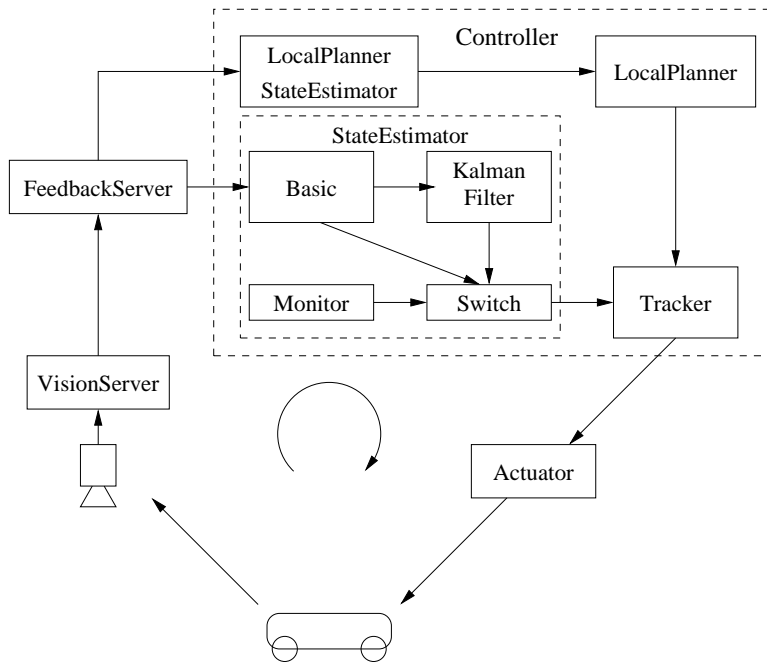


Figure 11.8 Upgrading the StateEstimator using the Incremental Evolution pattern.

because it can capture a lot of information at precisely the moments of interest, as opposed to logging vast amounts of information all along.

When the first controller is satisfactory, we use it as the “old” controller and continue improve the controller as a “new” controller. These iterations thus evolve the controller into a reliable complex controller.

11.6.5 Testbed version 5

Given a better controller, we can now ask it to do more precise activities. For this, the LocalPlanner must be fleshed out and include greater capability. Here we implement the ability to load preplanned trajectories, follow other cars visible in the system, or avoid collisions.

For operations which require awareness of other cars in the system, we rely on another StateEstimator. This Estimator was outlined in the second version of the testbed, and receives global information. It maintains a history of global observations, giving it the ability to predict future positions of cars using dead reckoning algorithms. These estimates can then be tested

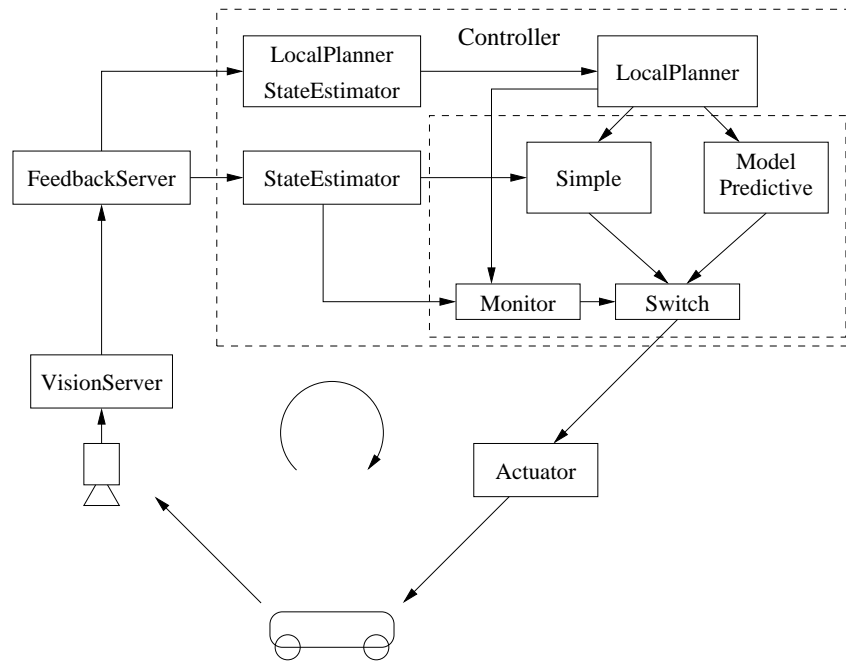


Figure 11.9 Upgrading the Tracker using the Incremental Evolution pattern.

by providing them as desired waypoints to the controller below. Thus, a manually driven car can be “driven” in front of the car, causing the controlled car to follow it.

Once the algorithms required for global state prediction have been implemented and tested, we can then begin to create collision detection algorithms, followed by collision avoidance algorithms. Eventually, we would like to incorporate sophisticated algorithms which can coordinate between cars to avoid collisions. Each of these additional functions can be incorporated using the pattern of Incremental Evolution as shown in Figures 11.8 and 11.9.

11.6.6 Testbed version 6

So far, we have limited the testbed to a single sensor. However, we have all of the structure in place to incorporate another sensor. We will do that in this version. A second VisionServer will be installed covering another section of track. The FeedbackServer will be informed of its existence because it registers with the Profile Registry. Thus, this particular upgrade can occur online, and without service disruption. Of course, to use its services, new trajectories must be created to operate the car in the new area of the track. These changes are reflected in Figure 11.10.

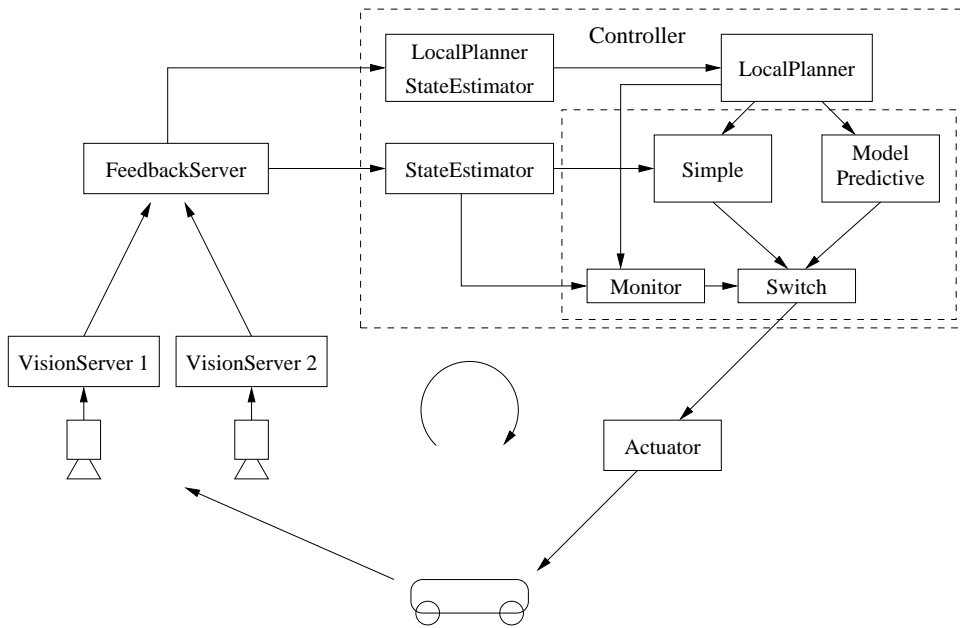


Figure 11.10 Adding another sensor.

11.6.7 Testbed version 7

At this point, we may also multiply the cars. The controller architecture for the new cars will be a copy of that for the old cars. For simplicity, we have collapsed the detailed drawings of the controller down to a single block, as shown in Figure 11.11. Because the vision system was designed to discriminate between cars already, and the FeedbackServer can provide car identification as part of its feedback service, integrating multiple cars is straightforward.

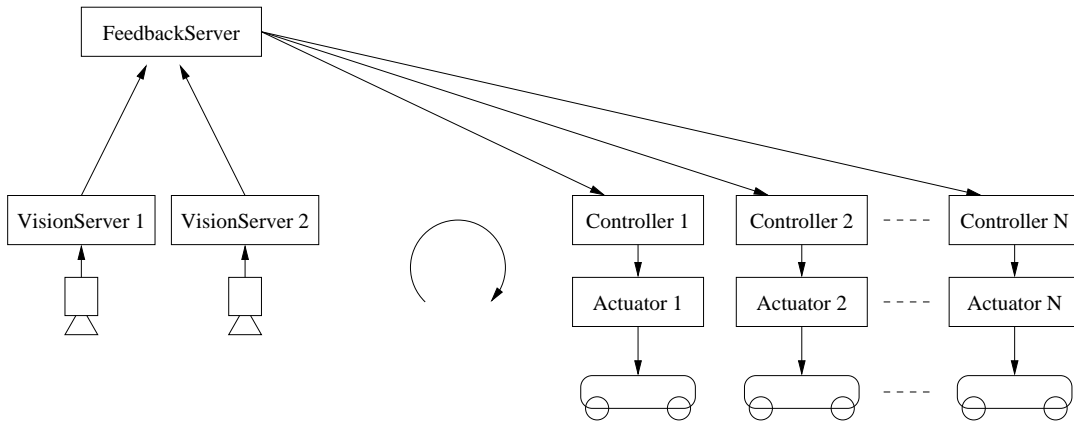


Figure 11.11 Incorporating additional cars.

11.6.8 Testbed version 8

With multiple cars, it now becomes important to have deconflicted trajectories. We will begin to insert this functionality by first creating the components which will be responsible for this operation and connecting them in a do-nothing fashion. These components include the Supervisor and the Supervisor StateEstimator, shown in Figure 11.12.

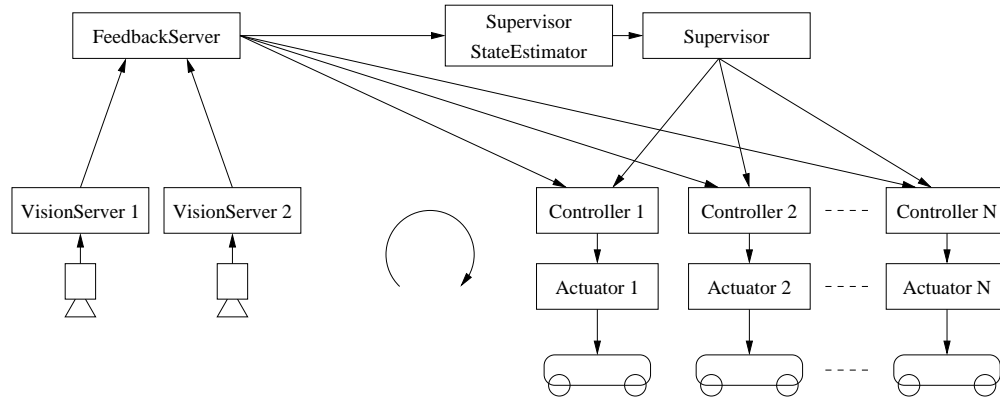


Figure 11.12 Inserting a Supervisor and its StateEstimator.

We must now begin to insert supervisory functionality into these newly created components, which is discussed in Section 11.6.9.

11.6.9 Testbed version 9

Mapping a roadway into discrete sections, we will create a discretized track which can use an algorithm devised to prevent gridlock and successfully route all cars to their destinations. This component resides in the Supervisor. In addition to creating collision-free schedules, the Supervisor may iterate such planning, and therefore must be able to determine the current state of the system with respect to the roadway operation. Therefore, we provide a Global StateEstimator for the Supervisor, which can provide a global picture of where the cars are, but in terms of the roadway. Because small position differences in the roadway may lead to large changes in planning, we require very accurate positioning from this Global StateEstimator. Thus, the Global StateEstimator will require the output from the Supervisor as input, using the additional information of where the cars were directed to go in order to bias the Estimates toward proper operation.

This completes the development of the current testbed configuration. Using similar evolution, we can include additional sensors, for example, by upgrading the cars to have onboard sensors. To utilize these sensors, the StateEstimator for the tracker would need to be upgraded. We can include car-to-car communication for improved collision avoidance capability. This would primarily involve the StateEstimator for LocalPlanners. Each StateEstimator could receive future plans for neighboring cars. The algorithms within the StateEstimator would then need to be upgraded to utilize the additional information for better collision avoidance. The key to evolution in each of these cases is that the changes induced are contained to only those components of the system which must produce or consume the additional functionality. For example, implementing better collision avoidance through car to car communication does not require a new Tracker component. Rather, the only upgrades required are to first, the LocalPlanner, which must coordinate with the remote car, and second, the StateEstimator for the LocalPlanner, which must improve its estimate for future positions, taking advantage of the additional information provided by the remote car (which presumably knows its own future plans and can provide them to the Local StateEstimator.)

11.7 Designing with Virtual Collocation

To illustrate the usefulness of the Virtual Collocation abstraction, consider how this abstraction was used in Version 5 of the testbed. This version provided for enhanced local planning, such as collision avoidance. The algorithm is centralized by nature, but the information needed to avoid collisions is not. For instance, while the vision system can provide a global picture of the current state of the system, it is inherently delayed. Moreover, it cannot predict where cars might be in the future. To enhance the current and future estimate of the state of the cars near a particular car, it would be useful to have these cars provide their future plans to the StateEstimator serving the LocalPlanner. Using this abstraction, namely that the StateEstimator retains responsibility for providing good estimates of future car positions, the collision avoidance algorithms can focus on planning a collision free path in the presence of nearby cars. The algorithm is thus free to solve only its portion of the problem, as if all information were locally available.

Even the LocalPlanner’s StateEstimator utilizes the abstraction of Virtual Collocation. It may use the vision data provided by the FeedbackServer in order to determine which cars might be close enough to be concerned about collisions. The estimator can then initiate a dialog with each of these cars to request that they provide future plans to the estimator. The estimator itself need not know how to communicate with the cars. It merely sends an event to each of them using the Semantic Addressing capability of the Etherware. Such a request may look like “Send “Plan Update” request to car number 4.” The Profile Registry will find where car 4’s controller is located and forward the request to that node. This controller has an event handler to receive this event and produce a reply which contains its current plan from now until a short future horizon. This event is then sent via the Etherware back to the StateEstimator for the LocalPlanner serving the local car. Thus the estimator never knew where car 4’s controller was located within the Etherware system. It may as well have been on the same node as the local car doing the collision avoidance planning.

The LocalPlanner can follow a similar operation to create a Leader Follower scenario. Here the StateEstimator for the LocalPlanner issues the same request for a plan from the leading car. If that car responds, that plan is what is passed to the LocalPlanner. If not, perhaps in the case that a car is being manually controlled and therefore does not have an Etherware controller, then the StateEstimator merely estimates future positions of the car based upon past observations.

11.8 Methods for Reliability

We have argued for the importance of reliability. Reliability is enhanced in the testbed through application of the abstractions and patterns presented here as well as through custom solutions designed to address specific problems. We attempt to summarize these efforts to show how reliability can be influenced by the principles presented in this theses.

Reliability is the property that the system, or portions of the system, function as desired, even in the presence of errors. From the bottom up, we can look at the reliability efforts in the sensors and the actuators.

The actuator gives steering and speed commands to its car. If the actuator fails, the behavior of the car is not predictable. In the testbed, the microcontroller sending signals to the radio control subsystem will merely hold the last command given. Thus, a failed actuator will result in a car continuing to move at the previous speed, which is not safe. If the actuator is integrated into the same component as the controller, a failed controller will result in a failed actuator resulting in an unsafe condition for the car. Thus, the actuator has been designed as a separate component, capable of directing the car on its own. The importance of this reliability is better understood when considering the complexity of control commands versus the simplicity of merely executing them. The actuator has very little complexity and is therefore more reliable. Moreover, a failed controller can be rapidly restarted. If the actuator was provided with a sequence of future controls, the brief outage of the controller may not be missed. The result of this is that while the controller may fail, the system does not.

The actuators are composed of a radio control link as well as a custom built model car with a stepper motor for steering control and a variable speed motor and gearbox combination for speed control. The original motors were inexpensive and unreliable, burning out regularly. New motors were acquired together with new gearboxes to adapt the speed of the cars to a more suitable range. This merely points out that the reliability of the underlying hardware contributes to the overall system reliability and must be addressed in any reliability review.

The sensors are required to provide position and orientation information for each of the cars. This can be done in many ways; we chose to assist the vision system by placing precoded color patches on the cars. The particular choice of patterns was an optimization between having large enough patches for increased probability of the patch being properly identified, and having enough patches for redundancy in the event that a patch is not found. As the size of the “roof” of the car is fixed, an increase in color patch size results in a decrease in the number of patches which can be placed on the car. Because the probability of finding a particular color patch varies according to brightness and other variations throughout the track, it was decided that the vision algorithms must be able to identify a car even if two color patches were not found. For details on how this was accomplished, see Appendix C.

Part of the process of finding color patches in an image involves scanning the image pixel by pixel for possible matches to the color of interest. This is a time consuming operation. A

performance enhancement could use the past position of a car to narrow the search space for the colors, thereby potentially improving the operating speed of the sensors. This scheme, however, has an impact on the reliability of the system and must be understood. If a vision sensor must have seen a car recently in order to know where to look for it, new cars cannot be added to the system. Moreover, a car which is reintroduced in a new location will not be found. Thus, if possible, it is preferable from a reliability perspective to have the vision system scan the entire image on every update.

11.9 Concluding Remarks

Throughout this evolution, the pattern of adding hollow components, without any functional addition, facilitates the reliable evolution desired. For always on systems, this procedure can be accomplished using a middleware such as Etherware. We believe this microevolution approach to systems design and implementation will be key to the proliferation of general purpose control systems.

CHAPTER 12

A SKETCH OF A POSSIBLE LARGE SCALE APPLICATION FOR POWER DEMAND REGULATION

To illustrate the architecture and principles presented in this dissertation, it is useful to create a hypothetical design of another system based upon general purpose control.

The national power grid is a system faced with decentralized control, limited sensing, and long reaction time constants. The result is that large deviations from normal operations can create sudden and drastic power outages. When the loads of the power system cannot be met, the system is forced to resort to load-shedding to prevent catastrophic failure. As load-shedding has drastic consequences, it would instead be desirable to reduce these loads voluntarily. In this chapter, we will sketch a very preliminary outline of the design of a system to accommodate that desire which is founded upon the principles of a general purpose control system. Our purpose here is only to illustrate by an example that general purpose control systems may possibly have widespread usage in the era of system building that we are now in.

12.1 Background

Power generation facilities produce large amounts of electrical power that cannot be stored directly. Power production must equal power demand. Unfortunately, the time constants involved in bringing generators online or shutting them down are large, perhaps several hours. Accommodating even slight changes in power demand may require several minutes to achieve through manipulating tap positions etc [39, 40]. Some system changes are instantaneous, such as when circuit breakers trip, causing abrupt spikes in voltage or current. Utility companies

have little or no control over how much power is demanded at any moment in time. Physical principles dictate voltages and current flows as customers increase or decrease their power demands. Moreover, when demand exceeds forecasted supply, expensive forms of power must be made available including short term gas driven turbines or purchasing electricity under an expensive short term contract. Thus even if demand can be met, it may become unreasonably expensive to do so.

In this chapter, we will consider the hypothetical problem where utility companies have the ability to request that customers temporarily reduce their power requirements [41]. For example, if a few large industrial consumers or thousands of homeowners voluntarily shut off their air conditioning systems in a timely manner, perhaps an imminent blackout could be averted. To do this, we must assume that the utility company has the ability to sense the current state of power supply and demand. Moreover, to give the utility company the ability to influence the customer demand for power, we will install devices in the homes of customers, or in the plants of industrial companies, which can be configured to “voluntarily” cut off certain non critical, or “elastic” loads.

These capabilities represent sensors and actuators (the load shedders) in this load balancing system. The final requirement is that automatic controllers are capable of receiving and properly interpreting the sensed inputs, then determining the load shedding requirements and communicating them to the actuators which will then shed the load.

12.2 Home and Industry Participation in Power Regulation

The challenge for including consumers in the regulation of power flow is two-fold. First, the individual consumers, or rather their appliances, must be informed in real time about current conditions in order to assist in real time. The second issue is how to incorporate voluntary demand reduction into the regulation schemes. The result is a federated system whereby individual users can preprogram how they wish to participate in this setup, but enable them to override those decisions at any time. Given the potentially large number of users, on average there may be sufficient control authority to achieve temporary change as needed.

We now consider the problem of how to inform individual consumers about current conditions. We assume the existence of a general purpose control box in the home of participating consumers. This box, which we will call the Energy Control Box (ECB), shown in Figure 12.1 will require a communication link between it and the providing utility company. In addition, the ECB must have the ability to communicate with various appliances throughout the home in order to ask them to back off at certain times. For example, the heating and air conditioning control box may communicate with the ECB to schedule the heating and cooling cycles.

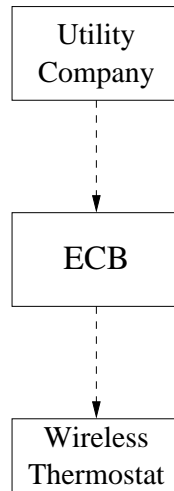


Figure 12.1 Basic configuration.

The ECB could communicate with the utility company through various means. One scenario involves a communication network directly over the incoming power wires [42, 43]. As this scheme involves the difficulty of transmitting such signals through power transformers, as well as requiring the setup of a new communication network, it may be more feasible to use the Internet for such communication. This too may prove troublesome as it would require an “always on” Internet connection in the home of each participant. A third alternative could use the telephone network wherein each ECB would be able to receive a telephone call originating from the utility company indicating the current demand requirements. The ECB could also occasionally call the utility to send meter readings and other information. A primary advantage with this alternative is that the telephone network already exists, is managed by another entity, and has independent power supply. The cost to use this network should be minimal. We note that satellite dish television receivers already employ the telephone network to communicate

usage of PayTV movies to the dish network provider. A disadvantage with using the telephone network as the downlink from the power utility company to possibly thousands of homes is that it may not be able to carry the traffic without excessive delay. A possible solution is to use radio broadcasts similar to weather radio systems. On the uplink, the telephone can still be used.

A promising alternative for communication involves the use of the cellular telephone network. As the cellular solution is a radio solution, it is inherently broadcast. Thus, a single signal could be picked up by thousands of receivers at the same time. Of course, the receivers would have to be modified slightly from the traditional cell phone receivers, essentially all listening on the same channel. For the uplink, the ECB's cellular transceiver would have to be able to function as a traditional cell-phone.

Produced in mass, a cellular equipped ECB should not cost much to manufacture. It is essentially a modified cell-phone with a slightly larger memory capacity and the ability to connect to a home via a wireless network as well as via a wired connection such as Ethernet or USB. Note that in a basic configuration (e.g., with an ECB and a single wireless thermostat), an ECB would not need any physical wiring connection to the home except for a power source. Only the thermostat would have to be wired, and it would merely replace an existing thermostat.

We now explore how this system could be used by a consumer. The basic function would be to schedule brief power demand changes to assist the utility company. Whether there is excess power which needs to be used or a shortage which needs to be mitigated, the ECB can turn the air conditioning unit on or off, respectively. This basic functionality, referred to as "Demand Response," may be a useful capability to the utility company. What remains to be seen is whether it can be accomplished in a cost effective and socially acceptable manner.

12.3 Peripheral Functionality

We now turn attention to how this basic function could spawn other useful functionalities. Perhaps the electric meter can be made accessible to the ECB, i.e., the ECB has access to the current power consumption as well as the current meter reading of the home junction box. Furthermore, we assume that the ECB can be accessed by a personal computer via the

same wireless interface used to communicate with the thermostat. See Figure 12.2. As each component in the system runs an Etherware program, the personal computer can exchange events with the ECB.

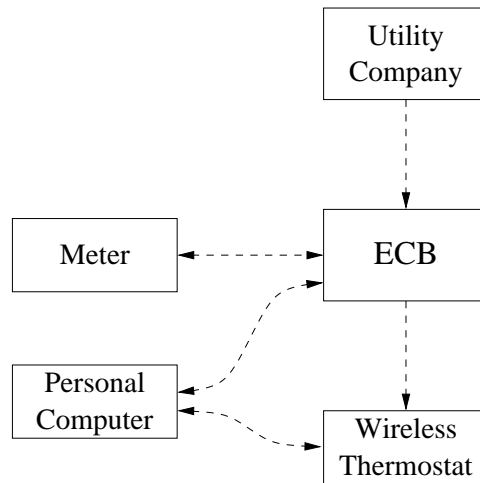


Figure 12.2 Communicating with a personal computer and/or the utility meter.

Special purpose Etherware components could be created in order to allow a homeowner to analyze energy usage with much more resolution than currently available with monthly or bi-monthly meter readings. In fact, such a system could be used to analyze home energy patterns and perhaps warn of problems. For instance, an air conditioning system low on freon may run excessively. If the software is aware of the inside and outside air temperatures over a period, it can determine that the air conditioning unit has lost efficiency, and can then alert the homeowner. We can carry this idea further and assume that the ECB can monitor the current being fed to the air conditioning unit and monitor its energy consumption in real time, comparing it to the cooling load and giving the homeowner options to schedule a more efficient cooling schedule. Perhaps a whole house fan can be operated by the software so as to draw in cooler outside air in the evenings instead of running the air conditioning unit.

Other appliances could also be included. For instance, the dishwasher could have a communication interface running a small Etherware program. The dishwasher could communicate to the ECB or the PC what its cycle is like, or the ECB could ask the dishwasher to postpone an energy consuming cycle temporarily to adjust for the utility company's current load shedding

profile. Appliances could be controlled via wireless links, or perhaps by simply controlling the circuit through special purpose circuit breakers as shown in Figure 12.3.

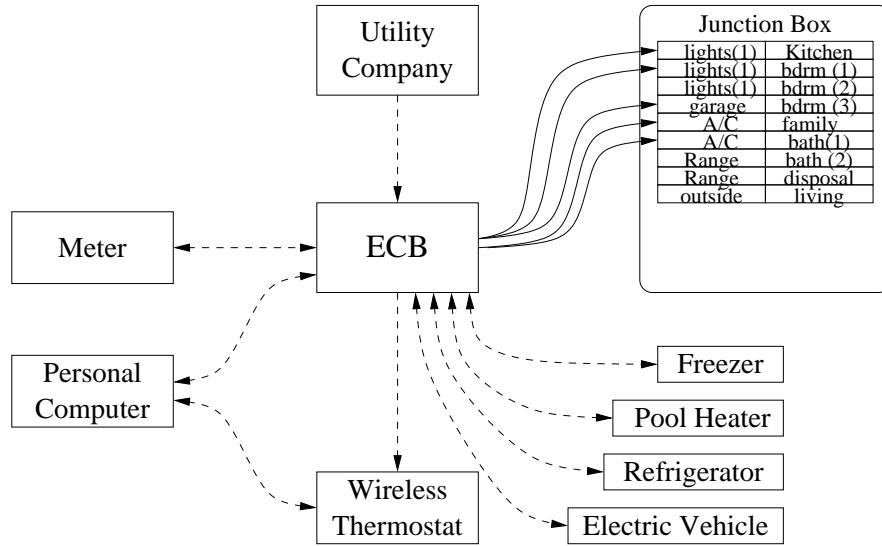


Figure 12.3 Incorporating control of other appliances.

An additional benefit would be to provide to the homeowner analysis of actual energy consumption throughout the home. Many consumers have little appreciation for which appliances consume the large electrical costs. Incorporating electric current sensors into circuit breakers could provide the information of which circuits are using how much power at which times. Although such analysis may not be meaningful to the average consumer, the cost conscious or environment conscious consumer may be willing to spend the extra money to save electricity and help the environment. Additionally, software programs to optimize consumption may be disseminated over the Internet to homeowners.

By providing such automation throughout the home, we may be able to provide security services as well. For instance, a vacationing homeowner may wish to be alerted if nonessential electrical usage is detected. Perhaps a concerned consumer wondering if the iron or the stove was left on can log on to the home network and “look,” or even shut it off.

Looking further into the future, it may be economical for consumers to operate electric cars by scheduling charging time according to pricing schedules given by utility companies. Perhaps the electric cars are connected in the evening, but an interrupting box can be used to schedule

the charging. This scheduling can be used by the utility company to schedule generation as needed.

Finally, such mechanisms may be used to reduce the peak-to-average demand ratio that is a significant concern to utility companies.

12.4 Incentive Pricing

This entire scenario assumes some incentives for the consumer. The obvious choice is to have variable billing rates, e.g., time of day rates, such that participation carries an economic benefit. Another benefit is that the utility company may be able to manage special conditions as well as reduce peak-to-average demand ratios, though that would require appropriate control laws.

12.5 Local Autonomy

The utility company planning component functions as a limited controller. It is important for customer satisfaction that the utility company not have complete control over the operation of the appliances in the home of the consumer. The overarching principle is that given the expected size of the customer base, it may be reasonable to assume that a sufficient number of homes will honor the request to voluntarily reduce load. The utility company may even have access to the settings of ECB boxes, and sample them to determine and predict response to its requests to reduce power consumption. The reduction can and should be able to span a large dynamic range. In order to create this dynamic range, we assume that the reduction requests carry additional information to allow the homeowner's ECB to make a decision regarding its load.

We assume that the utility company will need to iterate requests. A first request may be issued and the system may then be monitored for the resulting consumer response. As all this involves delay, care should be taken to avoid large fluctuations in the system which can itself induce instability into the system. This iteration of staged requests can even lead to gradual and smooth reduction in demand.

One way to reduce destabilizing gains is to have ECBs comply according to some random delays or random amounts of backoff. Clearly, these effects must be analyzed in simulation as well as actual tests before a utility company Planner can be used effectively.

In keeping with the desire to provide the homeowner with the ability to control his or her own compliance with requests, we propose that each request contain the following information: severity level, desired per home load change, and a time window. If the desired window is very short, we would expect greater compliance and a faster response. If the time window is long, responses should lengthen, and compliance may vary. The homeowner retains the right to designate via software on the PC how the compliance for her home will behave. Indeed, each homeowner will essentially formulate a cost/reward function for compliance, which may well include financial incentives. For instance, the request may indicate that excess power is available and that participating homes may get the additional power for a much lower cost. Therefore, it may be in the interest of the homeowner to run the air conditioning unit or the heating system a little longer at a lower rate. Or alternatively, perhaps an electric vehicle could be charged during the night, but at a time which is convenient to the utility company; therefore the cost to the consumer could be lower. Obviously, this requires more accurate recording of energy usage throughout the home, which may well be valuable in its own right.

The aforementioned capabilities may, and probably do, need to be automated. This provides yet another advantage: automated and enhanced meter reading. Because of the business implications of meter reading, we assume that the ECB must be provided by the utility company running billing software created for the utility company, while the ECB must also be capable of executing home control commands. Such a mutual operation must co-exist properly within the hardware and software framework of this device. Or, perhaps the home security or home control applications would be a consumer add-on, capable of attaching to the ECB for power and battery backup as well as using the wireless link.

Because of this, one can envision the use of Etherware inside the ECB. Core components can be designed and built by the utility company, with peripheral components uploadable by the consumer. The Java protections for component faults must protect the core components from crash failures of the nonessential consumer features.

We now turn our attention to the next logical component in importance, the electronic thermostat. Ultimately, the requests from the utility company which are digested by the ECB must be able to influence the air conditioning unit in order to have any effect. This may be accomplished with special purpose circuit breakers which contain a semiconductor thyristor switch which can be operated by a hard wire connection from the ECB to the circuit breaker in the junction box. It may also be done over a wireless connection from a wirelessly enabled thermostat to the ECB. In this configuration, the ECB occasionally directs the action of the thermostat. In the absence of ECB (or PC) inputs, the thermostat simply follows a known schedule, such as tracking the current temperature set-point. The most important aspect of the thermostat is that any currently available programmable thermostat will do, with the necessary addition of the wireless or wired link to the ECB or the home network.

12.6 Design of a Power Demand Response System

The previous section outlined several factors which may play a role in this type of large system. Several of these factors coincide with requirements for general purpose networked control systems. For instance, proliferation is necessary for mass adoption and subsequent reduction of costs. The power demand response system can only be effective if a sufficient number of homeowners adopt it.

There are many socioeconomic factors that enable or prevent mass adoption. One is incentives. Variable pricing would be essential in this system. Implementing variable pricing requires the ability to have local record keeping at each meter. This in turn requires that the record keeping portion of an ECB be inaccessible to consumers. As a result of these forces, an ECB would necessarily have to be produced for and distributed by the utility companies. In this model, the ECB could also be designed with whatever custom interface is needed to communicate via a cellular network. Moreover, the ECB could communicate with, or even replace, the electric meter for a home.

Note the architectural implications of such a model. The ECB serves as the interface from utility company to the home. By analogy, currently the meter and junction box traditionally serve as the interface to the home.

We now consider the autonomy required by the components within the home. The ECB clearly must function at all times. It must be autonomous with respect to failures of any other part of the system. It serves a central role to many potential services and has no backup. In this case, there is no additional need for redundancy because the complexity of the ECB does not warrant it. Moreover, failure of the ECB is not absolutely critical. It simply means that only one particular home cannot respond to power demand change requests, or optimize other services within the home. The ECB should have a power backup, however, to assure operation during a blackout, perhaps placing a cellular call to a vacationing homeowner. Another use of the ECB box is that when the power supply returns after a temporary blackout, it can delay turning on certain circuits in the house until power has stabilized to prevent spikes in these circuits.

The home heating and cooling control system must be capable of operating autonomously with respect to the ECB. There must be no dependency, other than the functional dependency of the desired optimized operation. Thus, the heating and cooling control system functions just as the Tracker in the convergence testbed. The ECB represents the LocalPlanner. Note one functional difference between the power demand response system and the testbed. In the testbed, the LocalPlanner is assumed to be collocated with the Tracker for safety reasons. In the power demand response system, there is no safety requirement, and fail safe is provided by the thermostat itself; thus, the LocalPlanner can be separated from the tracker by an unreliable communication link.

The LocalPlanner may be utilized in scenarios other than load balancing for the utility company. As an always on entity, it may be capable of security monitoring for the home. In such a role, the homeowner may desire to have the ability to “program” a portion of the ECB to behave as desired. Thus, the ECB is a dual entity, providing proprietary interface to the utility company and an open interface to the consumer. We envision that any customized software added to the ECB would run in a protected area of the ECB using the Etherware monitoring capabilities. The programs themselves could be developed on a PC and downloaded and upgraded at run time into the ECB to perform these additional functions. Perhaps the ECB could have hardware add on modules that accomplish these roles. Note that in these roles,

other components in the home serve as sensor inputs to and control outputs from the ECB, which again serves as the LocalPlanner.

The utility company is of course the global planner. It is the responsibility of the utility company to monitor system conditions, establish power demand change needs, set temporary prices accordingly, and broadcast the resulting incentives to the various ECB's, which then implement responses in accordance with the rules the homeowner has programmed into the ECB. Note again that the homeowner retains the autonomy to comply or not. Of course, the ECB monitors compliance for billing purposes. Compliance though, carries an incentive.

Other components in an evolving version of this system can be added as additional controllers with associated sensors and actuators. Charging an electric vehicle would involve a charging controller which allows charging anytime through the night that the prices are sufficiently low. Of course, if the prices do not drop sometime through the night, the charging controller may autonomously decide that the car must be charged in time for the homeowner to get to work. Therefore, at 5 a.m., the car could begin charging regardless. At a neighbor's house, this charging may not occur until 8 a.m. because they are retired and may prefer to capitalize on the possibility of lower prices later in the morning. In each case, the charging controller would be programmed via a wireless link to a PC which runs electric vehicle charge controller software. Thus, the charger itself need not have any significant user interface, keeping hardware prices low. We do envision a switch which disables program modification to prevent drive-by wireless tampering. This again represents the necessity of providing to the user the autonomy to operate according to personal desire.

Through these examples, we have discussed the possibility of programming a particular component in the system. This is possible using the Etherware system by simply sending a new component as a Java class embedded in a well-formed XML document. The receiving Etherware node interprets the event as a request to upgrade components and loads the new component into memory somehow. In some cases, the new component may be monitored for safety or correctness via the Incremental Evolution design pattern.

In summary, the design of a power demand response system is the proper establishment of interfaces. We have shown how important properties of autonomy, evolvability, reliability, and proliferation can be satisfied using a design based upon general purpose networked control

systems. It may be possible that this exercise could be readily realized into a working and evolvable system.

12.7 Concluding Remarks

The hypothetical system described above may not prove feasible or be realized for various reasons. However, opportunities abound for using the IT revolution to increase efficiency of several large man-made systems. Thus, general purpose control systems of one sort or another are, we believe, sure to become commonplace over time, just as computing and communication have become an integral part of our lives. As that time approaches, we expect that the challenges outlined here, as well as strategies to meet those challenges, will be an integral part of their development. We plan to continue this endeavor, developing other example systems, stressing the designs illustrated here, and creating tools for facilitating the design of such systems.

CHAPTER 13

CONCLUDING REMARKS

In this dissertation we have identified and addressed the requirements for the widespread proliferation of general purpose networked control systems. We have examined in detail numerous crosscutting considerations into the design and operation of general purpose networked control systems. This includes the issues of design patterns for incremental evolution, the role of knowledge of time and per packet delays in distributed systems, the need for reliability, and the abstractions of virtual collocation and local temporal autonomy. We have identified critical bottlenecks and proposed preliminary solutions to these bottlenecks.

As part of this work, an actual working example of a general purpose control system has also been developed as a research testbed, which allows the identification of relevant issues and permits testing of the suggested solutions. This testbed has also served well as an architectural proof of concept, evolving continuously to incorporate new features.

We hope that continued efforts in this direction will in time produce additional insights into the requirements of such systems, develop standards and tools for their production, and ultimately produce useful products with widespread usage. That is the cycle that we hope this dissertation has catalyzed.

APPENDIX A

GLOSSARY OF TERMS

Actuator: An actuator is a device which can act upon the physical world. For instance, a motor can create a force to turn a physical object, such as a wheel. In Etherware, we also refer to the Etherware component sending signals to the motor as an actuator.

Autonomy: The ability of a component to operate despite the failure of components with which it interacts.

Cluster: A group of nodes on the same local network. I.e., a broadcast message can reach every node in a cluster.

Collocation: Collocation is the property that system components are located on the same node. Virtual collocation is an abstraction such that components appear collocated when in fact they are distributed.

Compile time: The point at which computer source code is translated into executable form.

Component: A computational entity responsible for some processing within the control system. An Etherware Component is an Event Handler composed of three basic functions. The first function is the Initialize function, the second is the ProcessEvent function, and the last is the Terminate function. Each of these functions may return one or more Events for processing by other Event Handlers in the system. Thus, a component returns an event list. Multiple components can reside, or execute, on a single Etherware node.

Computational Stability: The property that the computational system is itself stable. This implies that the system be robust to certain errors, such as programming errors, failures, deadlocks, etc.

Configuration data: Configuration data refers to that data which is not part of the design interface, but is required to “complete” the design for a particular customized application. Examples include such details as IP addresses, filenames, initialization sequences, calibration data, etc.

Containment: The property that failure of one component does not induce failure in another component.

Controller: Any component which takes as input a state estimate and goals, and computes control inputs which are designed to move the state of the system under control toward the goals. Controllers are often hierarchical, taking advantage of timescale decomposition and abstractions.

Current time: This is the time displayed on a given clock at a particular moment in time.

Design time: The development phase in which architecture and interfaces are considered, and decisions are made regarding the final structure of a system.

Dynamic: Dynamic operation is the interaction that occurs on the system while the system is in operation, in particular during system integration, operation, and test.

Elapsed time: The amount of real time that has transpired between two distinct moments. In practice, this information must be captured as the difference between clock measurements taken on the same clock at the two distinct moments.

Error Model: Error models define each particular class of errors which a system is designed to tolerate. For example, we claim that networked control systems must tolerate delay and loss in the communication network. Thus, delay and loss are part of our error model.

Etherware: Etherware is middleware targeted for the domain of networked control systems. It is also an architecture for the software design of such systems.

Event: A message passed between components. An Etherware Event consists of three basic pieces of information. First the event must have a profile that indicates the recipient of the event. Second, the event must have an information payload of some sort, i.e., the event content. Third, the event has an originating time stamp.

Evolution: Making changes to a system. This can occur while the system is or is not running. It can occur in large or small steps.

Exceptions: When processes in an operating system behave in an illegal manner, the operating system can throw an exception indicating the misbehavior. The exceptions are ultimately caught and processed by either the application, or the operating system. In the case of the operating system, the default action is to terminate the process. We prefer to catch the exceptions within the middleware and terminate only the offending component, taking a moment to save the essential current state of the component for restarting.

Filter: A device which receives sensor data and processes it, or aggregates it, passing the information on, is called a filter.

GlobalEventBus: An Etherware component responsible for sending and receiving events to components not located on its own node.

GUID: For unambiguous event delivery, each component in an Etherware system must have a globally unique identifier (GUID).

Hierarchy: The property that control can exist at multiple layers of abstraction. For example, while setpoints represent a goal to a low level tracking controller, they represent the controls output by a higher level controller.

Interface: The principal function of an interface is to simplify and specify the interaction between components. Properly done, either side of the interface can change independently, provided each conforms to the interface. In managing complexity, the idea is to make interfaces as simple as possible, thereby containing the complexity within known boundaries.

Invocation: The manner in which components become computationally active.

Kernel: A kernel is the core of an operating system. An Etherware kernel is the core of the Etherware framework. Following a microkernel architecture, the kernel is extremely simple, serving as the system postal service. As events are generated, the kernel serves as the system router, sending events to appropriate Event Handlers, whether on this Etherware Node, or on a remote Node. There is exactly one kernel per Etherware node. The kernel is kept simple by placing all complexity into offboard replaceable components. The kernel only knows how to deliver events to EventHandlers residing on this node, with GUID's in the profile. For events that do not have a GUID in the profile, the kernel defaults to the ProfileRegistry. For GUID's which do not correspond to this node, the kernel merely passes the event to the remote bus, which delivers internode Etherware events.

Latency: With real systems, there is delay involved in every action. For communication, it may be the result of the propagation of the electromagnetic energy through a channel, or the time to form a packet or process it. For a computer, it may represent the time required to execute the instructions within the processor. For physical systems, there is an inertial delay or transfer lag between when the control signal is applied and when the forces cause sufficient movement in the physical devices such that it can be detected. While these delays may be small, they exist, and in aggregate represent a problem in many real time control systems. We refer to this natural delay as latency.

Local time: The time displayed at one particular location.

Location independence: Within the Etherware framework, there is no fundamental difference between local and remote functions. Services are addressed semantically, without regard for which node they physically reside on. The Etherware provides the network connectivity to allow clients and servers to reach each other without knowing anything more than the type of service being provided. This removes the location dependence associated with hard wired IP addresses.

Middleware: Distributed software that resides between the operating system and the application. Middleware may be active or passive. A passive middleware provides services to

a client, but the client is in complete control of its own operations. An active middleware provides the complete operating environment for an application.

Migration: Moving an executing component from one computing location to another. This typically is realized by copying some state from the current component, initializing another identical component in the new location, then terminating the original.

Mobility: The ability for a distributed component to move from one computing location to another.

Node: We define a node to be a single computing entity with at least one network address and only one clock. An Etherware Node consists of all of the hardware and software available in a single physical location. A node has exactly one clock available to it and runs exactly one Etherware Kernel.

Physical Stability: The property that the physical system under control is itself stable. This implies that the the state of the physical system remain within safe boundaries regardless of the inputs to the controller.

Profile: Designed to support semantic addressing, a profile is service specification. The profile indicates the type of service desired, and perhaps parameters which narrow the selection of a server. For instance, the type of service may be vision position feedback, and the parameters would be the geographic coordinates of a car for which the position information is desired. When a request for service is filled, the client and server are able to address each other directly, without requiring the service of the ProfileRegistry. This is accomplished by simply placing the globally unique identifier (GUID) in the profile. In this way, event delivery is streamlined.

ProfileRegistry: The ProfileRegistry is the matchmaker of Etherware. It translates between semantic addresses and GUID's. That is, the function of the ProfileRegistry is to look up servers which can provide the service requested by the client.

Runtime: The times at which the system software is actually executing. In this phase, the executable code is considered to be fixed and all behavior is manipulated by inputs, not

by redesign. In practice, run time can involve online modification of executable code. To distinguish this case, we will call this dynamic upgrade.

Scheduler: A scheduler is responsible for determining the order and timing of activities in a system. The Etherware Scheduler schedules Etherware events and threads for processing on this node. That is, given a list of events to be processed, the scheduler sorts out which event should be processed first.

Sensor: A sensor is any device capable of providing information about the physical world in a digital format. For our purposes, we often include in this definition the Etherware component which first receives information from the physical sensors.

StateEstimator: A component used to identify the state of a particular portion of the system under control. We employ state estimators at multiple levels in the testbed. Some utilize a Kalman filter, taking advantage of knowledge of control inputs. Others maintain a history of observations and extrapolate future positions for collision avoidance. In each case, the state estimator is optimized and tailored to reject particular sources of error. That is, each state estimator must have an implicit error model against which it is trying to defend.

Static: Static activity refers to design and compile time activities, specifically operations on the software rather than the system.

Timing Service: Etherware employs the Control Time Protocol to translate remote time stamps into the time reference of the local node. The effect is the same as synchronization, however, without actually modifying the clocks.

APPENDIX B

ETHERWARE - CONTROL ORIENTED MIDDLEWARE

Middleware is a software infrastructure for integrating distributed applications, which resides above the underlying operating system. Middleware may be active or passive. A passive middleware provides services to a client, but the client is in complete control of its own operations. An active middleware dictates the complete operating environment for an application. In this way, an active middleware is capable of providing additional guarantees to the application.

With many different computer processors with unique operating systems available, a system designer has to worry about many low-level integration details. In addition to differences among systems, there are issues of configuration, such as IP addresses that are not really pertinent to the design, but which must be dealt with in order to make a system operational. Middleware can bridge the gap between computing resources (including the operating system) and system design. In essence, middleware presents an abstraction of collocation to the application above.

Increasingly, it is becoming clear that middleware can effectively relieve much of the design burden. Indeed, limited automatic configuration can relieve much of the design burden by not only creating a useable configuration initially, but optimizing it over time in response to actual workload, and adapting to meet current needs. For instance, a migration capable middleware can monitor a large system for bottlenecks in computing or communication resources, moving load from high stress to underutilized resources. Middleware can also log a history of activity in order to justify useful upgrades.

In our vision of assisted design and system integration, middleware running at each of the nodes in the system has system stability as the highest priority. The middleware is capable of monitoring components as they execute, catching exceptions when components fail, restart-

ing them when feasible, or diverting the inputs and outputs to other components capable of continuing to provide system stability, at the expense, perhaps, of optimal performance.

As described earlier, control systems require certain guarantees of stability in order to maintain safety. However, the complexity of systems gives rise to faults and associated failures which are nearly impossible to eliminate. Therefore, control systems must be capable of surviving errors and maintaining safety. A middleware oriented towards the control domain must have the ability to provide stability in the presence of errors.

Because of its position in the application stack, middleware is capable of hiding much detail which is unnecessary and a source of distraction to a systems designer. As design time must be reduced in order for proliferation, middleware can be an important contributor to the proliferation of networked control systems.

B.1 Etherware

Etherware is middleware targeted for the domain of networked control systems. It is also an architecture for the software design of such systems. Striking a balance between generality and specificity, Etherware is designed for a large class of systems, but not all.

For portability, Etherware is implemented using the Java programming language. This design choice also provides a mechanism for reducing failure propagation, to be discussed later. While in some cases the Java Virtual Machine incurs a slight performance overhead, the focus of Etherware is on minimizing the development time of networked control system applications.

Etherware is an event-based architecture. Applications are comprised solely of event handlers. That is, all communication in the system is expressed in terms of events. There are events, and event streams in order to accommodate different communication requirements.

The middleware is in constant control of the system and thus can provide certain safety and reliability guarantees. Indeed, these guarantees ameliorate a large class of system integration hurdles. The event based paradigm leads to a significant reduction in unnecessary execution and timing dependencies introduced during implementation.

B.2 Using Etherware

We now address how Etherware assists in system integration. Because Etherware constitutes a design framework, there are several properties which are provided by using the Etherware framework. These properties are derived from various abstractions.

B.2.1 Collocation abstraction

The first design abstraction hides the details of a distributed system, wherein components need not know a priori what each others physical addresses are. Indeed, this abstraction makes the system appear to be collocated at a single node. This abstraction not only reduces configuration errors; it also allows for run time flexibility in a very natural way. A server can be brought down, and as long as another similar server exists, the client need not even be aware of the change. This also allows for a restart of the server without the client being aware of the restart, provided it is accomplished quickly enough.

B.2.2 Evolution

The upgrade of components can be accomplished by bringing new components online, followed by retiring existing components. This can even be done at the same time, without the need for bringing down the entire system or rebooting it in any way.

B.2.3 Migration

Beyond the issue of stopping a component and restarting it somewhere else, Etherware provides a mechanism for encapsulating the current state of a component in order to forward that state to the new component, and initializing the new component with the forwarded state. This then represents a true migration of components from location to location within the Etherware network.

Each component has a terminate function, which is designed to save the component state into another data structure, and then terminate the component. The same state saving function can be used without actually terminating the component. Thus, we can replicate a component

or migrate a component by saving its state and sending that state to another location and initializing a copy of the component using the saved state.

B.2.4 Containment of cascading failure

In a typical operating system process, executing an illegal instruction, such as divide by zero, or attempting to access memory outside of the allocated memory for that process will result in termination of the process by the operating system, in order to protect other process from the misbehaving process. This has the unfortunate side effect of making all components within a process completely dependent on the proper execution of all other components in that process. Instead of causing the termination of all components, it would be preferable to terminate the offending component alone, and perhaps restart it if applicable. In this fashion, the failure is contained from propagating into other components.

One feature of Java which is exploited for containment is exception handling. When a component throws an exception, such as a divide by zero or other improper operation, the exception can be caught by a components exception handler. This exception handler can be designed to save the component state into another data structure and then terminate.

The important feature here is that although the component dies, the Etherware process does not. Provided such exceptions are properly caught, the system is inherently immune to component execution failures. Additionally, the termination of a component can trigger the reinstantiation of the component with the saved state, effectively restarting the failed component without bringing down an entire process. This quick restart can mask subsequent system failures which could result from the loss of the component.

These features combine to provide a stable development platform for system integration; even in the case of dynamic system upgrades, Etherware can maintain stability of the control system as well as the physical system under control. Although integration errors can still occur, they can be experienced and logged without causing any damage and often without even stopping test. This facilitates much faster development and operational testing, more aggressive testing, and ultimately better test data as it is not lost as a result of system failure.

B.3 Application Design within Etherware

The design abstractions provided by Etherware facilitate the design and implementation of distributed systems in general. However, the primary focus of Etherware is to facilitate networked control systems. Therefore, in this discussion, we restrict the applications to those systems that consist of at least one sensor, one actuator, and one controller. As these names may invoke multiple ideas, we will define each of them explicitly.

B.3.1 Design principles

It is assumed that a networked control system must be safe and available. These two priorities drive the desire for robustness to many classes of system failure. While we are concerned about the failure of individual components, the focus is on maintaining operation and safety of the entire system. This is often referred to as the stability of the system. We now discuss some issues and associated terminology.

B.3.1.1 Stability challenges

While control system theory is replete with robust solutions for many classes of system failures, networked control introduces several specific classes of failures which must be handled appropriately. Evolutionary systems also provide a rich set of failure modes.

B.3.1.2 Error models

An important class of errors in a networked system is that messages can be lost or delayed. The delay can be erratic as well.

B.3.1.3 Component failures

Given the distributed nature of a networked control system, individual components can fail due to hardware failures, software faults, communication disruptions, and many other intermittent disturbances.

Particularly in wireless communication systems, message loss is an ever-present source of failure. For real-time systems with safety guarantees, this implies that actuators must be able to operate safely in the absence of any communication. This may include shutting down, or reverting to some other minimal safe mode of operation. In this sense, an actuator can be thought of as having a minimal controller built in.

B.3.1.4 Invoked components

The first paradigm shift within Etherware is that all application components are primarily passive. Indeed, each component is truly nothing more than a consumer and producer of events. Events can be very expressive, thus allowing for a large design space. But they are nevertheless events. This simple interface makes the interconnection elements of Etherware very manageable, fast, reconfigurable, and efficient.

By maintaining complete execution control over the application, the middleware is able to provide monitoring and configuration services such as watchdog timers, migration and optimization, failure containment, and so forth.

The way in which a component gains active execution is that an event must be sent to it for processing. This model allows for single or periodic timing events to be requested as a service, thus providing a method by which the component can be invoked at a particular time or on a regular periodic basis.

B.3.1.5 Interactions

Given this processing model, the designer begins by determining what actions within the physical world are desired, and what sensors and actuators can produce the information required and create the physical reactions within the real world. Then the designer can “connect” a controller to the sensors and actuators in order to close the feedback loop.

B.4 Design Abstractions and Philosophy

We now present some of the design abstractions in Etherware, as well as the philosophy underlying it.

B.4.1 Event-based application

A component in Etherware is a producer and consumer of events. When an event designated for a particular component arrives, the component is invoked by sending the event to the component's `ProcessEvent` function. The component is now actively running on the particular node and the result of its computation and processing can be expressed as an event, or perhaps a list of events. These events are then sent to appropriate event handlers of other components and the cycle continues.

B.4.2 Local scheduling of events

For control applications, timing is often crucial. Indeed, some functions must be executed in a periodic fashion. This is accomplished by having components register (with a Ticker service) to receive periodic events. The Etherware thus functions as the sole timing unit for the system. Clearly, this places a burden on the Etherware to schedule the processing of events. However, this burden is appropriately placed on the middleware as only a centralized entity can accomplish the inherently centralized task of scheduling on a centralized resource.

For systems which execute on multiple nodes or resources, the scheduling task is accomplished on a per node basis. That is, no matter how many Etherware “applications” are running at a single node, all scheduling of local event processing is the responsibility of a single Etherware kernel. There is no coordination between remote Etherware kernels for scheduling local event handling.

B.4.3 Location independence

Within the Etherware framework, there is no fundamental difference between local and remote functions. Services are addressed semantically, without regard for which node they

physically reside on. The Etherware provides the network connectivity to allow clients and servers to reach each other without knowing anything more than the type of service being provided. This removes the location dependence associated with hard wired IP addresses.

B.5 Core Etherware Services

Although the entire Etherware structure is designed for evolution and replaceability, a few critical Etherware components are not run time upgradable. While they are not run time upgradable, they have been reduced to the simplest possible tasks and are rarely, if ever, required to upgrade.

B.5.1 Kernel

At the heart of the Etherware system lies the kernel, which has three primary responsibilities. The first is to dispatch events, the second is to manage the components residing in this kernel, and the third is to schedule events and threads. The function of delivering events embodies all of the scheduling and prioritization tasks, the routing tasks, and the timing tasks to be accomplished by the middleware. To enable run time upgrade of the complex portions of these tasks, the Etherware kernel is extremely simple. It is effectively nothing more than a postal service. For all events that have a globally unique identifier in the profile which are recognized by the kernel to be for event handlers on this node, the kernel merely routes the events to the event handler. For remote identifiers, the kernel defaults to what is called the GlobalEventBus, which knows how to route to remote locations. For connection events, in which the sender does not yet know the identifier of the recipient, a profile is created which essentially requests a type of service. As the kernel cannot yet route this event, it must default to the local ProfileRegistry which may or may not be aware of this service. If it is aware of the service, it returns the event to the kernel with the identified of some server capable of providing the service. The kernel can then route the event to an appropriate server. This explanation is not yet complete as we have not discussed how the ProfileRegistry and GlobalEventBus acquire knowledge. This will be discussed in Section B.6.

B.5.2 ProfileRegistry

The ProfileRegistry is the matchmaker of Etherware. A client in need of a service composes an event to send to the server. As it does not yet know of the server, it creates a profile of the service which is desired, and that profile is in essence a form of addressing. The client then sends the event to the kernel for routing. The kernel is simple. Since it does not understand semantic profiles, the kernel defaults to the local ProfileRegistry. Thus, all initial requests will be routed to the ProfileRegistry to make a client server match.

If the ProfileRegistry is aware of a server which can provide services matching the requested service profile, the ProfileRegistry forwards the event to the server by placing the Etherware address (IP address, component ID etc) of that server as the profile of the event. This is then given to the kernel which is now capable of routing the event to the server. Note that the GUID of the requesting client is included as part of the event such that when the server responds to the client, it does so by placing the client's GUID in the profile of the response event. Future exchanges of messages now avoid the overhead of the ProfileRegistry. Should the server or the communication to the server fail in any way, the client merely requests service again by placing the service profile in the profile of a service request event. In this manner, a server can be replaced, upgraded, or relocated with minimal disruption in service. The default profile is the globally unique identifier (GUID).

B.5.3 GlobalEventBus

If all globally unique IDs corresponded to the local node only, the kernel would be entirely aware of how to route events. As the purpose of Etherware is to enable networked control, it is clear that some entity must be capable of discovering and interpreting remote addresses. This is the job of the GlobalEventBus.

The GlobalEventBus must be configured to know how to communicate with other nodes. For nodes on the same network, this is accomplished automatically via simple broadcast network messages. For more complicated networks, such as those traversing the Internet, simple broadcast is not acceptable or sufficient. Therefore, some limited configuration data must be provided to the Etherware, possibly as configuration files which contain the IP addresses of

remote nodes. This information need only be known by one GlobalEventBus per Etherware cluster. Note that the system need not know which servers are running at each location, only that a remote node or cluster exists and how to reach it. The rest of the discovery process is automated.

B.5.4 NetworkTimeService

As Etherware is operating over a distributed system, there is no guarantee that individual nodes display the same clock time at the same moment. Although there are synchronization schemes (such as NTP) that can minimize clock differences, these are very restrictive, requiring administrative privileges to modify each of the clocks. Moreover, these are unnecessary for Etherware. Instead, Etherware employs the Control Time Protocol, described in Section 10.3.3 to translate remote time stamps into the time of the local node. The effect is the same as synchronization, however, without actually modifying the clocks. Note that there is virtually no interaction between CTP and a synchronization protocol. The two can run side by side with no dependence or undesirable interaction caused by CTP on the synchronization protocol. In the reverse direction, if the synchronization protocol abruptly adjusts a clock, then CTP may require an interval of time corresponding to an adaptation time constant to learn of this change.

As we assume that each node has exactly one clock, it should be apparent that the CTP algorithm is run on a per node basis. As the GlobalEventBus is responsible for all internode communication, it can include time stamps on all messages, and read time stamps from incoming messages, passing them all to another component responsible for implementing CTP.

Etherware includes a time stamp on every event, and it is assumed that the application will make all references to time within an event relative to the timestamp of that event. Note that the time stamp is generated locally, i.e., according to the local clock. For intranode events, the time stamp needs no translation. By collecting internode time stamps, provided by the GlobalEventBus, the NetworkTimeService can monitor the current offset between its clock and each of the remote clocks to an error no larger than half the round trip times experienced by network packets. Using these offsets, the NetworkTimeService merely translates the time stamp of each message arriving from a remote location, via the GlobalEventBus, into the time reference of the current node and forwards the event to the kernel. This then represents an abstraction

of time such that individual nodes can operate synchronously without concern for actual clock offsets. Moreover, abrupt changes in remote clocks can be compensated for correctly, without cascading the disruption throughout the rest of the system.

All methods of synchronization are fundamentally limited by the round trip delay experienced by internode messages. CTP is therefore optimal, providing equivalent performance of a synchronization scheme, but without unnecessary dependence.

B.6 Configuration and Startup

As an illustrative example of how Etherware operates, we consider how the system is initialized. First we treat an isolated node, then the entire network.

B.6.1 Initialization of a single Etherware node

For this example, we assume a working subset of a networked control system. By “working,” we assume that some components have been written and tested. Thus, this example merely steps through the process of initializing an Etherware based system.

First, the Etherware kernel is started on the node. Upon initializing, the kernel instantiates a Scheduler and a ProfileRegistry. It also instantiates a GlobalEventBus, a NetworkTimeService, and a Ticker service. Next, the kernel consults a configuration file to discover which components must be created and initialized. Let us assume there is a server and a client to be started. Presumably the server would come first in the initialization sequence. The Etherware would first instantiate the server and initialize it appropriately. Part of the initialization procedure would involve having the server create a profile registration event, which would then have to be delivered by the kernel. The profile of this event would be the ProfileRegistry. That is, the server does not know the Etherware address of the ProfileRegistry, only that a RegistrationEvent needs to be created. As the kernel cannot deliver events without an Etherware address, the kernel defaults to the ProfileRegistry, which was where the event was destined anyway. So the ProfileRegistry recognizes the profile as matching itself, and processes the event itself, rather than looking for an available server. As the server includes its own Etherware address in the

event, the ProfileRegistry has access to the server's Etherware address, and now associates the services offered by the server with the server's Etherware address. As additional servers register, the ProfileRegistry's knowledge of the system grows.

We now consider the client. After initializing the server, the kernel instantiates and initializes a client. As part of initializing the client, the client may request services or connection to services of some type. This is embodied as an event, with the profile of the desired service. The kernel processes this request as it does all others. It looks for an Etherware address, which it will not find, and so it defers to the local ProfileRegistry. As the server has previously registered, the ProfileRegistry knows about it and can make a match between the client and the server. To complete the request, the ProfileRegistry forwards the event by placing the Etherware address of the server in the profile field and returning it to the kernel. Upon receipt, the kernel knows what to do with the event, and thus sends it to the appropriate component, in this case the server. Upon processing the request, the server will send an acknowledgment or some sort of data directly to the client using the Etherware address of the client, which was part of the original service request event. The acknowledgement is an event, which includes the server's Etherware address. Now the client has the server's Etherware address and vice versa. If the request was for repeated service — perhaps for a data stream, for example — future events will be sent directly from client to server and back from server to client, using the Etherware address of the client or server in the profile.

B.6.1.1 Error handling through exceptions

If the server should fail or be shut down, an attempt to deliver an event to the server will fail. Etherware catches such exceptions and forwards the information to the sender. Thus the client will be aware of the failure of the link and can take appropriate action, such as requesting the service again, perhaps to be serviced by another similar server which did not fail.

B.6.2 Initialization of Etherware on multiple nodes

If no clients requested service from remote servers, the isolated instantiations of Etherware could remain isolated. As this is not how Etherware is intended to be used, we now examine how the islands of Etherware become connected.

For this example, we assume Etherware has been started on multiple nodes within a single closed network. Each of these nodes thus has a kernel running, with a ProfileRegistry and a GlobalEventBus also available.

Upon initialization, a GlobalEventBus will broadcast a request via a well-known port to any other Etherware GlobalEventBus instantiations capable of hearing the request. Upon receiving such a message, a GlobalEventBus will broadcast a reply with its own network address, as well as a list of all other GlobalEventBus addresses that it has. These messages are not necessarily the same as Etherware events, but represent another form of internode communication. Because this is a broadcast over a shared channel, other GlobalEventBus components will receive the message and update their internal cache of GlobalEventBus addresses. Etherware provides garbage collection of such data by using a time stamped cache. If an element in the cache has not been used for some long period of time, it is removed.

Through broadcast, all GlobalEventBus instantiations can become aware of each other. They then share information about which GUID numbers they can provide connection to, so that each GlobalEventBus can now know exactly how to route to each Etherware address.

We now examine how a local Etherware kernel utilizes the GlobalEventBus. Assume there is a client which already has the Etherware address of a remote server. The client creates an event with the Etherware address of the server in the profile and “sends” it to the kernel. The kernel looks at the Etherware address and does not recognize it as one of its own. As such, the default is to give it to the GlobalEventBus. As the GlobalEventBus has been initialized, and is aware of other nodes and which Etherware address’s exist on which nodes, it is therefore able to forward the event to the appropriate node using traditional networking. Upon receipt, the remote node’s GlobalEventBus will pass the event to the local kernel, which recognizes the Etherware address as one of its own and the event is delivered.

The GlobalEventBus provides a conduit for sending events to other Etherware nodes. Knowing what services exist on other nodes and their respective Etherware addresses, is yet another service provided by the ProfileRegistry. A local ProfileRegistry which cannot find any other ProfileRegistry will assume the role of a global ProfileRegistry. Subsequently, a ProfileRegistry will discover the global ProfileRegistry by sending events with a default GUID understood by the GlobalEventBus. As the kernel cannot understand a GUID unless it has been registered in the kernel, the kernel will default the GUID to the GlobalEventBus. Assuming the GlobalEventBus does not know where the global ProfileRegistry is, it will broadcast the message, including the GUID of the sender, to the network of GlobalEventBusses previously discovered. Each GlobalEventBus will then pass the message on to its local ProfileRegistry. If any local ProfileRegistry is currently the global ProfileRegistry, it will respond to the originating local ProfileRegistry, thus making itself known.

Subsequently, each server which becomes available will register with its local ProfileRegistry, which will in turn register the server with the global ProfileRegistry. Each local ProfileRegistry will periodically update its cache of profiles by querying the global ProfileRegistry for any new profiles.

Note that none of these services depends in any way on the configuration of the application. Also the application does not depend in any way, except for delays in delivery, on how the Etherware executes these services.

B.7 Steps of Design Using Etherware

By providing the infrastructure and services of a networked control focused middleware, we free designers from some of the trivia, while at the same time obligating the designers to adopt a perhaps slightly different paradigm of design. This paradigm is not restrictive, but instead offers a simple programming model, which must be adhered to.

Step 1. Actuator interface Determine the actuators based upon the physical effect desired in this control system. That is, what will this system “do?” When actuators are selected, or at least the kind of actuators which may be used, we then design the software interface to each of these actuators. For instance, in the testbed, the actuators are cars. Without determining

everything about the cars we can quickly surmise that the cars will have speed and steering commands. We can develop a simple linearized model of car behavior wherein a car proceeds forward for a short distance, followed by a change of rotation. At this point, this is all that is required in order to establish the interface.

The result of interface design for the actuator is the understanding that two types of commands must be given to the actuator. We may assume for robustness that the commands may be given in a sequence together with applicable command times.

Step 2. Sensor interface. As we wish to provide closed loop control, it is necessary to obtain feedback on the “plant,” or car in this case. Various sensors are available for this. Wheel encoders, inertial systems, and GPS type sensors can all be placed on board the car. In addition, external sensors may be located offboard, as is the case in the testbed. We then create the interface for each sensor. In the case of the cameras, we assume a camera is capable of identifying an individual car and providing position and orientation information. We may also require additional information such as quality estimates. We also require time stamps. Although the clocks are automatically translated in Etherware, the relevant time stamps are application specific, such as when an image is grabbed versus when it is processed.

The final sensor design decision is to determine whether the sensor provides information for multiple “plants.” We will do so in the testbed. Therefore, the sensor interface includes the latest sensor data for all cars visible by a given camera.

Although not currently employed in the testbed, we could also specify onboard sensors. In this case, the sensor may provide rapid readings of wheel position, or instead integrate the readings and provide elapsed distance traveled over a time interval. Such decisions may be repeated for inertial systems, etc. The end result is a set of sensor interfaces.

As sensors provide sensing services to other components, it is necessary to establish a profile for each sensor in order for the system to be able to connect clients to this service.

StateEstimator. As mentioned previously, StateEstimators provided a much needed buffer between the unpredictable behavior of a communication system and the required predictability of a controller. As the sensor and actuator interfaces have been specified, we simply reuse those specifications to connect the sensors to the StateEstimator and to provide command

inputs to the StateEstimator. The last input is the output of the StateEstimator which is simply the current estimate of the position and orientation of this particular car. Note that for predictability, we assume that the StateEstimator will execute on the same node as the controller. Indeed, we wish to specify this explicitly within Etherware. As the StateEstimator is to function as part of the controller, we will not specify a profile for it independent of the controller.

Controllers. Having specified the StateEstimator and actuator interfaces, the only thing remaining to specify within the controller is the command interfaces. In the testbed, we have chosen to use trajectories, or timed waypoints as command interfaces. In addition, we specify another command interface for mode changes. Specifically, we may wish to stop the cars immediately, or start them all at a certain time. This is accomplished via a mode control command interface.

In addition to these interfaces, a controller must have a profile in order to allow an actuator to obtain the “services” of the controller.

This completes the basic design of the testbed. Further interfaces will be required in order to allow for planners and schedulers, etc. However, the basic structure of the system is now laid out, with evolution available to add the higher level functions over time.

B.8 Steps of Implementation Using Etherware

Having specified the interfaces, we can begin implementation which can proceed in parallel as needed. The key steps of implementation will be to begin with an example provided within the Etherware framework, and then to execute the Etherware system on each of the sensor and actuator nodes. Ideally, simple versions of sensors and actuators can be quickly created and brought online for rapid development and testing.

In future work, an integrated development environment will be created to facilitate design and implementation within the Etherware framework.

B.9 Steps of Execution Using Etherware

To run the Etherware based system, one must start the basic Etherware service on each of the participating nodes in the system. The system will execute broadcasts and queries as it comes online in order to establish basic services.

After basic initialization, the system is ready to begin bringing sensors and actuators online, followed by controllers. Fortunately, order is not particularly relevant as the system can operate safely in the absence of components. It may not be able to accomplish anything, but system stability is preserved. Therefore, if a particular component is unavailable for any reason, we simply work to bring it online.

B.10 Conclusion

This appendix is a brief introduction for designing within the Etherware framework. As a relatively new middleware, Etherware will continue to evolve and design tools for its use will have to be developed. This appendix merely serves as a snapshot into its current state of development in order to facilitate understanding of the testbed.

APPENDIX C

VISION SYSTEM

C.1 Cameras

As we were interested in multiple sensors from the beginning, the track was designed for two ceiling mounted cameras which together cover most of the track. (The edge of the track is not visible, which permits the hiding of cars where that is of interest.) The fields of view overlap slightly in the center, allowing for exploration of “handoff” where cars are visible in both sensors. At present, handoff is handled by having the latest vision update overwrite previous updates. Thus, when a car gets a vision update from the FeedbackServer while in the overlap region, the update could come from either camera. With proper calibration, the resulting jitter is minimal.

The cameras were originally set to a shutter speed of $1/120$ s (8.3 ms) in an effort to keep the overall system delay to a minimum. This exposure captures only half of a lighting cycle (60 Hz, or 16.6 ms). If the exposure happens to cover the brightest portion of the cycle, the image is brighter. If not, the image is darker. When the phase of the overhead lighting fixtures varies independently from the phase of the camera shutter, this produces a slowly varying oscillation in brightness resulting, which introduces a great deal of noise in the image processing algorithms. While subtle, the effect is also noticeable to the human eye.

This is one of many examples in the testbed where unforeseen coupling may exist between components of the system, causing unintended and undesirable dependencies. Robust systems must either tolerate such coupling, or provide decoupling solutions. In this case, the problem is readily solved by setting the shutter speed of the cameras to $1/60$ s (16.6 ms), giving one full

cycle of exposure, regardless of the phase. The alternative would be to set up a complicated workaround wherein the cameras are synchronized to the building power frequency.

The cameras capture light on charge coupled devices (CCD). The picture element, or pixel, of a standard camera has four color elements: two greens, one red, and one blue. This is done because the human eye is better able to discriminate green colors. (Newer technology called Foveon [<http://www.foveon.com>] is now available which captures all three color components in one picture element, effectively quadrupling the resolution of a CCD.)

The cameras are operated by a vendor proprietary control box which converts the camera images to an NTSC video delivery format. The NTSC video format was designed to compress analog video images. The compression enhances brightness differences at the expense of some color degradation. These two features combine to produce an image which is very sensitive to brightness variations and green colors, while producing low quality red and blue color representation.

A coaxial cable connects the control box to a Matrox-Meteor II frame grabber mounted in a PC. The frame grabber then converts the NTSC signal back into a digitized frame or bitmap which can then be processed by the image libraries. An ideal camera system would not convert images from an essentially digital pixel format into analog and then back to digital through a frame grabber. Rather, it would process pixels right at the camera, without shipping images along cables. This represents another trade-off in communication and control systems. Where should the processing be done? Is it better to transport the pixels from one component to another, or process and then transport the results? Such optimizations are difficult to do today, but we will discuss a general method to accomplish this through middleware, in the future work section of this proposal.

C.2 VisionServers

The two desktop PCs are responsible for extracting information regarding the cars from the video images. Hence, we call these machines the VisionServers. Both are Dell machines running the Pentium 4 processor at 1.4 and 1.6 GHz. (The machines are different because they were purchased at different times. That has proven useful because by having slightly different

processors, we are able to determine which vision processing operations can be improved with a faster clock.) Each machine has 256 MB of memory. The operating system is Windows 2000 Professional. They are connected via a network interface card to the lab's wired network.

The PCs each have a frame grabber which communicates with the Matrox image libraries to capture images for processing. The libraries provide support for finding pixels which fall within certain color ranges. (Careful tuning is required to ensure that the particular colors used on the cars can be found consistently without including phantom colors. That is, there is a strong correlation of type I and type II errors in the calibration of the color parameters.) The libraries further process the discovered colors by grouping them into "blobs." The resulting list of blobs is then provided to the feature extraction routines for processing into cars.

For automatic color extraction, the system need not conform to the perception of the human eye. Rather, the color response should be uniform over the color spectrum. Brightness variations affect all values of color in the Red/Green/Blue (RGB) image format which is produced by the frame grabbers. To isolate colors, we convert the RGB image into another color space, called Hue/Luminance/Saturation (HLS). This is equivalent to choosing a different and better suited coordinate basis. In this format, color is contained in the hue, color depth or richness is represented by the saturation value, and luminance indicates brightness. To find colors in an image, we are not interested in brightness; so we largely ignore the Luminance value.

Detecting, locating, and identifying cars from the blobs requires geometric reasoning as well as color coding techniques. We use six colors on a car, chosen from an alphabet of eight colors without repetition. For robustness, we wish to allow up to two colors to be missing from a car and still be able to properly identify the car. To simplify the otherwise very complex geometric reasoning, we have chosen two colors to be common in the center of every car. Two were chosen so as to allow one center color to be missing. With this simplification, we cannot lose both center colors and still identify the car.

The identity of a car is encoded in the remaining four color patches. With two colors used in the center, there are six colors remaining in the color alphabet. Four colors are chosen for each car, without repetition, to be placed on each of the four corners. Order matters, thus we have 360 possible car encodings. Because we wish to have the ability to lose up to two colors and still identify the cars, we search the space of car encodings to find a set of encodings with

maximum difference amongst the members. The largest such set had a Hamming distance of three, with 22 members. This means that with two colors lost, we can still uniquely identify a car. From this list, we chose 15 codewords, and these “codes” represent the cars.

From the color patches we find the center of each car and identify the orientation. This is the feedback information required by the car controllers for closed loop feedback control.

C.2.1 Performance

The processing time per sample required to extract car information varies with the number of colors found on the track, which is proportional to the number of cars. A fully loaded track has 15 cars, but each side of the track rarely has more than 10. With 10 cars to be found, the 1.4-GHz VisionServer can complete a frame in 56 ms, while the 1.6-GHz machine completes in 51 ms. As these are multiprocessing machines, these numbers occasionally jump to around 60 ms while some other process has access to the CPU momentarily. Displaying the results to screen consumes around 5 ms, which we can turn off. Thus, the VisionServers operate very close to 20 Hz, slightly faster when there are fewer cars. This is more than sufficient for our purposes, but can be sped up with faster machines.

C.3 Other Positioning Possibilities

A vision system inherently has a precision on the order of one pixel, which, on our track, with 640×480 resolution, corresponds to 0.18 in. After allowing for noise, we end up with roughly 0.5 in resolution, more than sufficient for our purposes. While other possibilities for position feedback exist, each has its drawbacks, and few have precision even on the order of an inch. Lab visitors frequently ask if we have considered other sensors, so we will briefly discuss the differences amongst possible positioning sensors, including the impact on precision, accuracy, speed, noise, and cost.

The Global Positioning System (GPS) has become a well-known source for nearly worldwide positioning. It was established by the U.S. military for targeting and vehicle navigation. So why don't we use GPS for our track? The GPS system is based upon radio signals received

from overhead satellites. Inside of a building, particularly in the basement, attenuation and multipath fading greatly reduce the strength of such signals, rendering them almost useless. Although the U.S. military has discontinued the use of selective availability, in which civilian use of GPS is degraded to roughly 100-m accuracy, there are still atmosphere conditions which typically limit the precision of GPS to meters. Our application requires precision on the order of an inch. Moreover, using GPS would require receivers on each car, with some communication capability from the cars to the rest of the system. All of this adds weight and cost to each individual car, as well as undesirable complexity. The place for using GPS sensors is outside, in large area experiments, particularly when communication issues arising from mobility are a main research focus.

Acoustic signaling has also been suggested. Under a MURI contract, a group in the Computer Science department here at the University of Illinois is demonstrating acoustic tracking using small computers called Motes, which have limited sensing, computation, and communication capabilities. On our track, these devices demonstrated precision to within 6 in at best. With cars 5 in wide, this level of precision would not allow two opposing cars to approach each other in a two-lane road scenario. Acoustic sensors are very susceptible to noise, particularly echoes, and require very well-tuned, computationally demanding signal processing to extract even minimal precision. They also require on-board processing and some communication from cars back to the rest of the system.

A laser bar coding scheme would probably work to identify vehicles, but its positioning capabilities are somewhat limited. Laser sweeps used in bar coding are one-dimensional. Covering the two-dimensional track would require a ceiling mounted laser scanning in both dimensions. Unfortunately, the scan would provide identification capabilities, but no information on position or orientation. Coupled with the risks of reflected laser light, we did not pursue this possibility.

C.4 Reducing Functional Dependence on the Sensor

The original controller design was tightly coupled to the vision output. A controller waited (blocked) on vision input. Upon receiving a vision update, the controller computed a new sequence of controls, then blocked again. Thus, the controller was functionally dependent on

the VisionServer. This is not a problem if the VisionServer is reliable. But if the VisionServer does not recognize a car in some area of the track (a common problem that we will discuss shortly), then the controller receives no input and creates no output. Without new commands, the car follows the previous commands, which typically included a stop command at the end for failsafe reasons. Thus no input means the car stops. But the reason for no input is that the lighting conditions and such at that location prevent the car from being seen properly. Therefore, if the car stops, and is still in the troublesome area, it is effectively deadlocked. As this was a serious problem, we worked to overcome it in many dimensions.

C.4.1 State estimation

First, we wanted the car to be able to tolerate vision losses for some time and continue to operate. This reduces the functional dependence from constant updates to infrequent updates. If a car is operating under open loop control, clearly the accuracy of its trajectory tracking deteriorates. Even with sporadic updates, the car can correct the estimate of its state, realizing a graceful degradation. As mentioned in Chapter 2, the testbed employs a Kalman filter capable of using vision data without depending on it, to output some estimate of the state of the car.

C.4.2 Lighting dependence

When the testbed moved on to multiple cars, in order to report the position of multiple cars, some process was needed to distinguish individual cars among a set of identified cars. This can be done at the VisionServer, at some other centralized entity, or a controller could receive position information for all cars, then reason about its last position and choose the position update which most closely matched the position the car thought it was in. The second and third solutions are context dependent, and architecturally unsound. The clean solution is to have the vision system itself distinguish between them somehow. This required some method of encoding the identity of cars along with the coding for position. This led to increased color identification requirements and other associated issues which will now be discussed.

There are several ways to encode the identity of a car; we began with three color patches. A common color of orange was used in the back with two colors in front, chosen from a set of

four colors without repetition. This theoretically gave us 12 codes to identify cars, which was not enough for our fleet, but it was a starting point. In practice, this scheme suffered from many misidentification errors such as locking onto pieces of cars which were not color patches, interpreting the background track as a color, or simply not identifying a color patch at all. That is, introduction of this new requirement drastically reduced the reliability of the sensor system.

Having introduced the above solution to have more colors identified, we began to experience reliability problems with the vision system. The first problem identified was that of large lighting intensity variations over the track, which was detected by examining the distributions of pixel values for the same color placed randomly on the track. Lighting variation is unavoidable, arising from the fact that light sources are discrete, illuminating the track in patterns rather than uniformly. Intensity variations are further increased through the transformation of image formats, as discussed below.

C.4.3 Image format and transformations

Image processing begins when the image is captured by a video camera using a charge coupled device (CCD). Because the human eye is more sensitive to green than red or blue [44, 45], typical cameras have four elements per pixel: two greens, one red, and one blue. (There are other technologies such as the Foveon CCD which actually capture all three colors in a single pixel.) The resulting bitmap of red/green/blue (RGB) values is then transformed into an NTSC signal which is sent from each camera to a frame grabber located in each of the vision signal processing desktop machines. The NTSC format was developed to reduce bandwidth requirements for video images, and capitalizes on the fact that the human eye is more sensitive to brightness changes than color changes [46]. Hence, more bandwidth is used to represent brightness than color in the NTSC standard. The received NTSC signal is captured by the frame grabber and transformed back into an RGB bitmap, which is then available for image processing. It should come as little surprise that the resulting image is very sensitive to green and brightness changes, since those two aspects were emphasized, while red and blue are suppressed.

While there is more to be discussed regarding vision processing, we will mention at this point that the ideal scenario would be to eliminate the encoding in NTSC format altogether. One can conceive of capturing an image in RGB, sending it to the computer digitally, and processing it directly. Even better would be to have a hardware transformation from one color space to another color space as will be discussed below. Each of these steps would be desirable as vision processing matures, enabling more sophisticated vision sensing.

C.4.4 Color space

Now we consider the problem of color segmentation in image processing, whereby we determine what “color” a pixel is. Thus to classify an orange patch, we must check individual pixels and classify them as orange or not. First we must have some idea of the RGB values orange patches produce in the image. We can place orange patches around the track, save an image to file, then analyze the red, green, and blue component values of orange pixels from various patches. With 8-bit color, the range of values is 0-255. Suppose for an orange patch that the red value ranges from 50-150, the blue from 25-50, and the green from 60-200. We then code the system to filter out all pixels which do not fall within this range. The problem with this scheme is that the color orange has now consumed a very large portion of the available color space, with very little room remaining for other colors. Iterating, we now tighten up the values to restrict orange and allow for other colors. Now we have great difficulty finding orange patches because fewer pixels will be classified as orange. Perhaps we then reduce the number of pixels required to call it a patch, but that has the effect of allowing more “ghost” patches to appear, thereby requiring additional filtering. This is the familiar problem of tradeoff between Type 1 and Type 2 errors in detection [47].

The root of this problem is variation in brightness across the track. The reason for the wide range of RGB values in each color is that brightness affects the values of all three colors (see Figure C.1). What we really need is another color space in which the “color” component is orthogonal to the brightness component. Several such color spaces exist; we have chosen to use the hue-saturation-luminance (HLS) color space which is supported by our frame grabber (see Figure C.2).

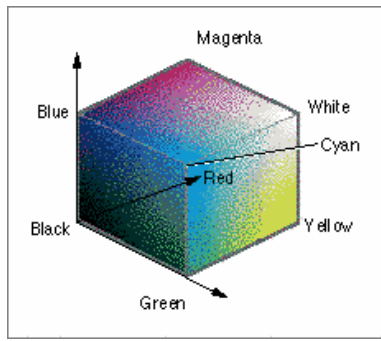


Figure C.1 RGB Color Space.

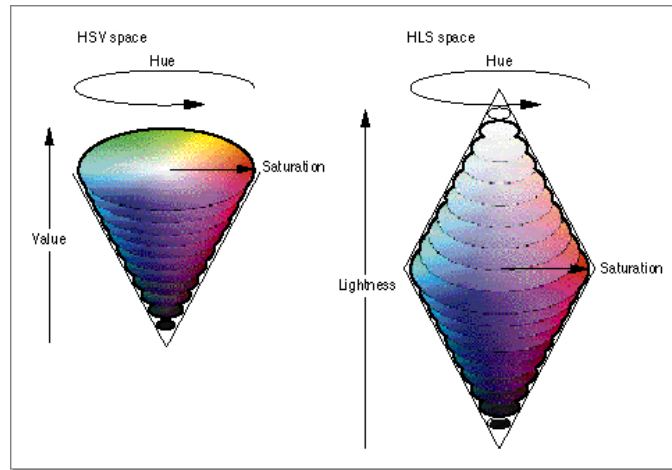


Figure C.2 HSV and HLS color spaces.

In HLS, “color” is captured in the hue component (H), while brightness is captured in the luminance (or lightness) component (L). Saturation (S) is the least understood, but is some measure of the color richness, or depth. The imaging library used in conjunction with our frame grabber transforms an RGB bitmap into an HLS bitmap through a nonlinear transformation. This transformation consumes ~ 13 ms on our 1.4-GHz machine.

Consider segmenting colors in the HLS color space. To accommodate a wide range of lighting conditions, we allow a wide range of values in luminance (L) for each color. Since color is what we are discriminating, we would like narrow ranges in hue (H) for each color. In practice we have found that pixel samples from colors which are close to green have very narrow distributions, while colors closer to red and blue have wider distributions. This is as expected, given that pixels have two green elements each.

Segmenting in RGB, with careful tuning, we were able to distinguish six colors, one of which was black, although the reliability of each individual color was not high. Using HLS we can discriminate up to eight colors (not including black) fairly well, without extreme tuning. Also the reliability is much higher than with RGB. Note that reliability refers to positively identifying a particular color in a particular location. Because the lighting conditions change more with geographic location around the track than with time, it is really a measure of the percentage of the track in which a particular color patch can be found, rather than a probability of being found in each sample.

C.4.5 Reducing patch dependence

Even after tuning the color identification system, there are still residual errors. Thus, to enhance reliability, we desire robustness to color patch loss in order to improve the overall reliability of the vision system. Because patch loss depends on location, we decided not to repeat colors on any car. That way, if orange is not visible in some area, it only omits one patch, rather than two. As previously noted, we have eight colors to choose from. We can place three color patches on a car and choose 15 of the possible color combinations ($8 \times 7 \times 6$) and thereby “decode” which car it is, where it is, and what direction it is facing. However, this scheme depends on finding each and every one of the three colors. Thus, proper selection of colors and patch size to use on a vehicle is an optimization problem in geometry as well as lighting issues.

We now consider in what pattern the color patches should be placed on the roof of the car, and how many colors are to be used per car. There are geometric issues to be considered. For example, if a color patch is too small, the likelihood of it being found and properly segmented is low. We have found that a patch needs to be bigger than 4 in^2 to be found reliably (better than 90%), but that a patch of 9 in^2 is sufficiently reliable, $\sim 98\%$. With cars which are 5 in wide by 9 in long, we chose to use six color patches, each $2.5 \times 3 \text{ in}$, thus using up the entire roof of the car with a simple geometric structure which can be exploited in identifying cars, as shown in Figure C.3.

With a color pattern established, we can consider how to design a decoder robust to color losses. The first step in this is to determine what errors we plan to be robust against. The



Figure C.3 Color patch layout.

probability of losing all colors is very low $\sim 1\%$. Unfortunately, the probability of finding all colors is also relatively low, $\sim 80\%$. But the probability of finding five out of six colors is pretty high, $\sim 95\%$, and the probability of finding four or more color patches is very high (roughly 99% of the track). Thus we aimed to make the system robust to the loss of up to two color patches. As a practical simplification we established the two center colors as common to all cars, thus simplifying the decoding process. Our error model then has to be modified slightly. We will tolerate the loss of one or the other of the center colors, but not both. This is done so that we always have a reference point on a car. Putting our two most reliable colors, pink and orange, in the center makes the likelihood of losing both colors quite small. With this change, we now have six colors which can be placed in four positions around the four corners of the car. This is equivalent to a four-digit code word with letters taken from an alphabet of six letters. Now we must devise a code in which the loss of up to two of these letter can be tolerated.

This can be done with simple Grey codes in the following manner. First we enumerate all the possible code words with no repetition, $6 \times 5 \times 4 \times 3 = 360$. Then we construct a 360×360 matrix of these codes and find the Hamming distance between each codeword. Two code words have a Hamming distance of 2 if they can still be distinguished even if two of the letters of a codeword are changed. Within the set of 360 codewords, there will be pairs of codewords with hamming distance of 1, 2, and 3. Grouping the pairs of codewords, we find particular sets whose members all have a Hamming distance of 3 from each of the other members. We desire the largest such set. In our special case, it turns out that we can find a set of 22 code words whose Hamming distance is 3. Since we have 15 cars to operate, the solution sufficed. We are

currently looking into the possibility of geometric Hamming codes as a generalization of our particular problem.

C.4.6 From patches to positions: Reliable coding

With codes chosen, we now must design an algorithm to interpret groups of color patches as cars. The imaging libraries report blobs of color according to our rules of segmentation. We then have a list of blobs with a center of mass for each blob. We then pair up all of the center colors which are close. This accounts for most of the patches in the scene. For each pair, we search for blobs which are close to the center of these two colors. Note that we may not have four neighbors. If there are at least two neighbors, we assign locations to each of the neighbors and test the configuration to see if it is a viable code word. If at least two of the colors agree with a code, we declare it to be a car. Using the color centers, we then establish the location of the car and its orientation. This is trivial when all six colors are found. The coordinates of the center, (x, y) , are computed as follows, where (x_i, y_i) is the x, y -coordinate of the i^{th} patch. Patch 1 is the front left patch. Patch 2 is the front right. Patch 3 is the left center patch, and so on.

$$x = \frac{\sum x_i}{6}, \quad (C.1)$$

$$y = \frac{\sum y_i}{6}, \quad (C.2)$$

The orientation θ is

$$\theta = \begin{cases} \text{atan}(\frac{dy}{dx}) & dx > 0, \\ \text{atan}(\frac{dy}{dx}) + \pi & dx < 0, \\ \frac{\pi}{2} \times \text{sgn}(dy) & dx = 0, \end{cases} \quad (C.3)$$

where

$$dx = (x_1 - x_3) + (x_3 - x_5) + (x_2 - x_4) + (x_4 - x_6), \text{ and} \quad (C.4)$$

$$dy = (y_1 - y_3) + (y_3 - y_5) + (y_2 - y_4) + (y_4 - y_6). \quad (C.5)$$

When colors are missing, we lose a center of mass location as well. A simple solution is to drop symmetric patches.

The next case to consider is the loss of one center color. Then we consider only those center colors which have not been paired up already. Without the second center color, we have no reference for orientation until we have established the locations of neighboring patches. To do this, we search for neighbors within an appropriate distance. Then we determine the angle from the one center color to each of the neighbors and order the neighbors based on their angle. This provides a circular ordering around the center patch. Now we split this into two cases. We can have four or fewer neighbors. If we have fewer than three, we discard this sample. With more than four, we simply pick the first four. Since we rarely have extra colors (ghost colors), this is satisfactory. Otherwise, we would need to try out all possibilities of colors combinations.

Now we consider the subsequent computations after four neighbors have been discovered around a single center color. Because the neighbors are on opposite ends of the car, we can pair them up as two pairs of nearby neighbors. Then we find which patch in each pair is closer to the center color, and since we now have an estimate of the position of each color patch on the car, we can “decode” the car. In practice we have noise in the patch locations, and so we can make mistakes with such a simple algorithm. However, these mistakes are typically error corrected by the coding scheme and are relatively rare, so we accept the possibility, rather than pay a penalty in processing time for the common case. With the locations now known, we find the center of each car and its orientation as mentioned before.

When only three neighbors are discovered around a single center color, the computations proceed much the same as with four neighbors, except that we have only one nearby pair.

To demonstrate the improvement made possible by the combination of geometric reasoning and error correction for missed color patches, we conducted some experiments and collected the data. It was discovered that the previous improvement, involving color space transformations, was actually sufficient. In fact, the data showed that almost everywhere, all six color patches were found almost all of the time. To demonstrate the robustness of error correction, we reduced the tolerance on the two center colors, thereby forcing frequent losses of one or both center colors. In Figure C.4, the arrows indicate where a car was found using all six color patches. The dots indicate where it was found in spite of lost color patches. The strong geographic correlation may be noted.

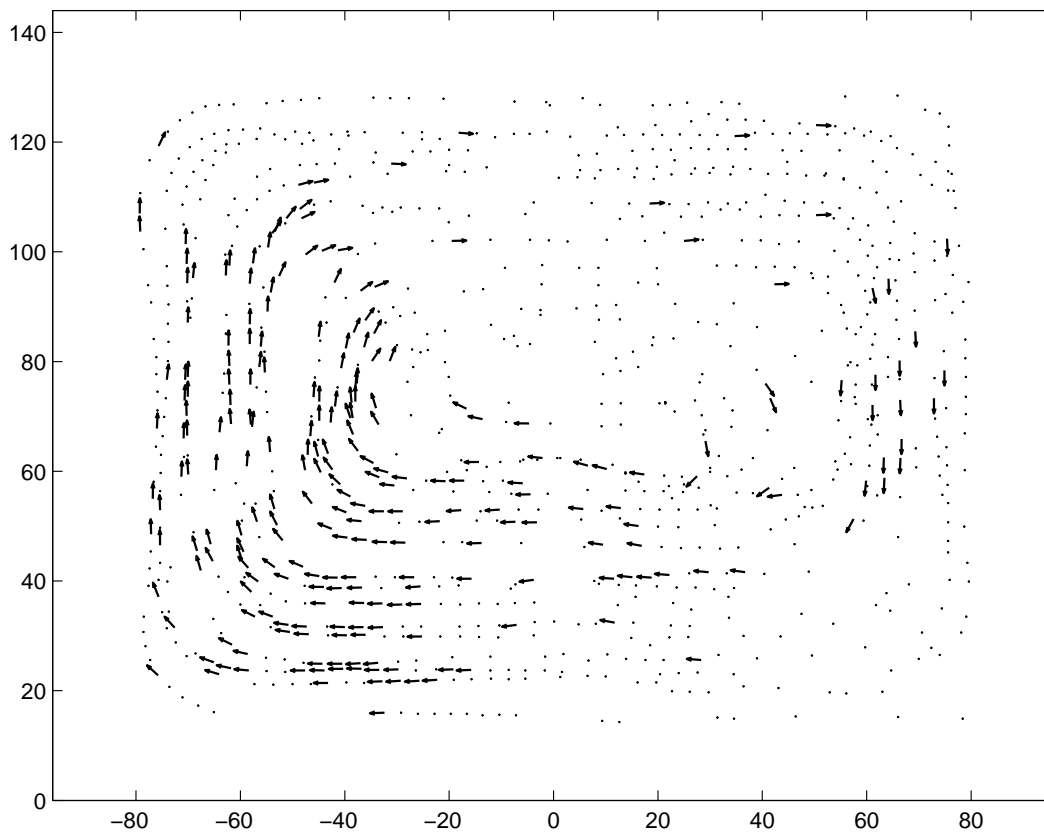


Figure C.4 Improved reliability when missed color patches are tolerated.

As shown in Figure C.4, even with poor tuning a car can thus be sensed almost anywhere on the track. In poor areas a car is seen at least 50% of the time, and on about 95% of the track it is seen about 95% of the time. This represents a reliable vision system, and a welcome improvement over the original system which had large blind spots in which a car could not be sensed, and which covered roughly 25% of the track. For good areas, a car was sensed only about 50% of the time. Moreover, we have been able to optimize using different libraries than the original system, making a 20-Hz sample rate possible on the two Pentium 4 machines running at 1.4 and 1.6 Ghz.

C.5 Robust Vision through Dependency Reduction

Through these examples we see demonstrations of how functional dependence can be reduced. Through state estimation, the controller reduces dependence on the individual updates,

relying instead on average performance. Through tolerating the loss of particular patches, as well as utilizing a more robust color space, the vision system reduces the dependence on lighting conditions.

The resulting system has proven very robust, providing accurate and reliable position and orientation throughout the entire track. We believe that, in general, evolution of a complex system requires minimization of functional dependence wherever possible, creating some degree of autonomy, which in turn enables future evolution.

REFERENCES

- [1] Internet Software Consortium, “Internet domain survey,” January 2003. <http://www.isc.org/ds/WWW-200301/index.html>.
- [2] M. Katevenis, “Reduced instruction set computer architectures for VLSI,” ACM Doctoral Dissertation Award, MIT Press, 1984.
- [3] J. Stankovic, “Vest: A toolset for constructing and analyzing component based operating systems for embedded and real-time systems,” University of Virginia, Tech. Rep. TRCS-2000-19, July 2000.
- [4] K. Carter, A. Lahjouji, and N. McNeil, “Unlicensed and unshackled: A joint OSP-OET white paper on unlicensed devices and their regulatory issues,” May 2003, http://hraunfoss.fcc.gov/edocs_public/attachmatch/DOC-234741A1.doc.
- [5] V. Lipset, “In-car bluetooth to grow beyond telephony, study says,” May 23 2003. <http://www.thinkmobile.com/Everything/News/00/67/32/>.
- [6] CrossBow Technology, Inc., “Wireless sensor networks,” Jun 2004, http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.
- [7] S. Graham, G. Baliga, and P. R. Kumar, “The convergence of control with communication and computation: Proliferation, architecture, design, and middleware,” submitted to IEEE Conference on Decision Control, December 2004.
- [8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1995.
- [10] B. Boehm, “A spiral model of software development and enhancement,” ACM SIGSOFT Software Engineering Notes, August 1986.
- [11] R. E. Kalman, “On the general theory of control systems,” in *Proceedings of the 1st IFAC Congress*, vol. 1, Moscow, pp. 481–492, 1960.
- [12] L. Sha, R. Rajkumar, and M. Gagliardi, “The simplex architecture: An approach to building evolving industrial computing systems,” in *Proceedings of the International Conference on Reliability and Quality in Design*, Seattle, WA, Anaheim, CA, pp. 122–126, March 16–18, 1994.
- [13] G. Baliga, S. Graham, L. Sha, and P. R. Kumar, “Service continuity in networked control using etherware,” to appear in IEEE Distributed Systems Online, 2004.
- [14] W. R. Stevens, *TCP/IP Illustrated*, Boston, MA: Addison-Wesley, 1996.

- [15] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, pp. 103-111, August 1990.
- [16] V. Kawadia and P. R. Kumar, "A cautionary perspective on cross layer design," University of Illinois at Urbana-Champaign, Tech. Rep., June 28 2003. http://black1.csl.uiuc.edu/~prkumar/ps_files/Cross_Layer.ps.
- [17] G. C. Goodwin and K. S. Sin, *Adaptive Filtering, Prediction and Control*. Englewood Cliffs, NJ: Prentice-Hall, 1984.
- [18] G. C. Goodwin and D. Q. Mayne, "A parameter estimation perspective of continuous time adaptive control," *Automatica*, vol. 23, pp. 57-70, 1987.
- [19] N. Wiener, *The Extrapolation, Interpolation and Smoothing of Stationary Time Series with Engineering Applications*, Cambridge, MA: The Technology Press, Wiley & Sons, 1949.
- [20] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, pp. 379-423, 1948.
- [21] R. E. Kalman, "Contributions to the theory of optimal control," *Bol. Society Mat. Mexicana*, vol. 5, pp. 102-119, 1960.
- [22] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, p. 297, April 1965.
- [23] S. S. Pradhan and K. Ramchandran, "Distributed source coding using syndromes (DISCUS): Design and construction," in *Proceedings of the IEEE Data Compression Conference (DCC)*, Snowbird, Utah, March 1999, pp. 158-167.
- [24] R. Shukla, P. L. Dragotti, M. N. Do, and M. Vetterli, "Rate-distortion optimized tree structured compression algorithms," submitted to *IEEE Transactions on Image Processing*, January 2003.
- [25] A. Giridhar and P. R. Kumar, "Scheduling traffic on a network of roads," submitted to *IEEE Transactions on Vehicular Technology*, April 17 2003, http://black1.csl.uiuc.edu/~prkumar/ps_files/trafficpaper.ps.
- [26] C. E. Perkins and D. B. Johnson, "Mobility support in IPv6," in *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, Rye, New York, November 1996, pp. 27-37.
- [27] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, pp. 114-117, April 19 1965. <http://www.intel.com/research/silicon/mooreslaw.htm>, <ftp://download.intel.com/research/silicon/moorespaper.pdf>.
- [28] "Ariane 5 flight 501 failure," Report by the Inquiry Board, Paris, July 19, 1996, <http://www.cert.fr/francais/deri/adele/DOCUMENTS/ariane5.html>.
- [29] J. Rumerman, "The American aerospace industry during World War II," U.S. Centennial of Flight Essay, 2003, <http://www.centennialofflight.gov/essay/Aerospace/WWIIIndustry/Aero7.htm>.
- [30] L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving dependable real time systems," in *Proceedings of the 1996 Aerospace Applications Conference*, vol. 1, Aspen, Colorado, February 3-10, 1996, pp. 335-346.

- [31] G. Baliga and P. R. Kumar, "Middleware architecture for federated control systems," June 2003, <http://dsonline.computer.org/0306/f/bal.htm>.
- [32] C. Scheidler, G. Heiner, R. Sasse, E. Fuchs, H. Kopetz, and C. Temple, "Time-triggered architecture (TTA)." *Advances in Information Technologies: The Business Challenge*, 1997. IOS Press, ISBN 90 5199 385 4 (Word 97 document).
- [33] S. Graham and P. R. Kumar, "Time in general-purpose control systems: The control time protocol and an experimental evaluation," submitted to IEEE Conference on Decision and Control, December 2004.
- [34] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Trans. ASME. (J. Basic Eng.)*, vol. 92D, pp. 34–45, March 1960.
- [35] K. Huang, S. Graham, and P. R. Kumar, "Temporal alignment of distributed sensors with an application to characterization of plant delay," submitted to IEEE Conference on Decision and Control, December 2004.
- [36] H. Flanders, *Calculus*. New York, NY: W. H. Freeman and Company, 1985.
- [37] D. L. Mills, "Internet time synchronization: The network time protocol," *IEEE Transactions on Communications*, vol. COM-39, pp. 1482–1493, October 1991. Also in: Yang, Z., and T.A. Marsland Eds., *Global States and Time in Distributed Systems*, Los Alamitos, CA: IEEE Press, pp. 91-102.
- [38] S. Yasunoby, S. Miyamoto, and H. Ihara, "Fuzzy control for automatic train operation system," *4th IFAC/IFIP/IFRS Conference on Transportation Systems*, 1983, pp. 33–39.
- [39] M. A. Pai, P. W. Sauer, and K. Khorasani, "Singular perturbations and large-scale power system stability," in *Proceedings of the IEEE 23rd Conference on Decision and Control*, Las Vegas, NV, 1984, pp. 173–178.
- [40] P. W. Sauer and M. A. Pai, "A comparison of discrete Vs continuous dynamic models of tap-changing under-load transformers," in *Proceedings of the 12th Bulk Power System Voltage Phenomena-III*, Davos, Switzerland, August 1994, pp. 643–650.
- [41] S. D. Braithwait and A. Faruqi, "Demand response – the ignored solution to California's energy crisis," *Public Utility Fortnightly*, March 15, 2001. <http://dsm.iea.org/NewDSM/Prog/Library/Upload/116/Call.doc>.
- [42] M. H. Shwehdi and A. Z. Khan, "A power line data communication interface using spread spectrum technology in home automation," *IEEE Transactions on Power Delivery*, vol. 11, pp. 1232–1237, July 1996. ISSN: 0885-8977.
- [43] R. G. Olsen, "Technical considerations for wideband powerline communication – a summary," *IEEE Transactions on Power Engineering Society Summer Meeting*, vol. 3, pp. 1186–1191, July 21-25, 2002, ISSN: 7554752.
- [44] B. Goldstein, *Sensation and Perception*, 4th ed., Boston, MA: Brooks/Cole Publication Company, , 1996.
- [45] S. E. Palmer, *Vision Science*. Cambridge, MA: MIT Press, 1999.
- [46] C. A. Poynton, *A Technical Introduction to Digital Video*. New York, NY: John Wiley and Sons, 1996.
- [47] J. S. Milton and J. C. Arnold, *Introduction to Probability and Statistics*, 3rd ed., Boston, MA: McGraw Hill, 1995.

VITA

Scott Graham was born in Salt Lake City, Utah, USA, in 1969. He received the bachelor of science degree in electrical engineering from Brigham Young University, Provo, Utah, in 1993, and the master of science degree in electrical engineering from the Air Force Institute of Technology, Dayton, Ohio, in 1999. He has served on active duty in the U.S. Air Force as an engineer since 1994, currently participating in the AFIT/CI program to pursue a doctoral degree at the University of Illinois at Urbana-Champaign. His research interests include architecture, networking, control systems, and system integration.