AGILOS: A MIDDLEWARE CONTROL ARCHITECTURE FOR APPLICATION-AWARE QUALITY OF SERVICE ADAPTATIONS

BY

BAOCHUN LI

M. S., University of Illinois at Urbana-Champaign, 1997
B. Engr., Tsinghua University, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

Abstract

In heterogeneous network and operating system environments with a fair amount of performance variations, multiple applications compete and share a limited amount of system resources, and suffer from variations in resource availability. These complex applications, such as *Omni-Track*, a client-server based omni-directional visual tracking application, are thus desired to adapt themselves and to adjust their resource demands dynamically. Frequently, the ultimate objective of application-level adaptation is to preserve the quality of the most critical application-specific parameter, such as tracking precision, even at the cost of other non-critical parameters such as perceptual quality. On one hand, current adaptation mechanisms built within an application have the disadvantage of lacking global information to preserve fairness among all applications. On the other hand, adaptive resource management mechanisms built within the operating system are not aware of data semantics in the application, and reservation-based resource management mechanisms may need modifications to commodity off-the-shelf operating systems or network protocol stacks widely deployed today. We believe that appropriate application-level QoS adaptation decisions can be achieved with the assistance of a middleware architecture, where both application and system level dynamics can be observed and analyzed to decide when, how and to what extent adaptation has to occur.

In this work, we present *Agilos* (**Agil**e Q**oS**), a novel *Middleware Control Architecture* to enforce the best possible adaptation decisions for distributed multimedia applications, via dynamic controls and reconfigurations of their internal parameters and functionalities. Several major contributions are presented in this dissertation. First, for modeling *adaptors*, we developed a *Task Control Model* based on control theory, in order to enact graceful degradation or upgrade paths

within a required QoS range. With the Task Control Model, we are able to reason about and validate analytically adaptation attributes such as stability, adaptation agility, fairness and equilibrium values of the adaptive behavior. Second, complementary to the adaptors, we deployed a *Fuzzy Control Model* in the design of middleware *configurators*, in order to model the process of choosing among application-specific parameter-tuning and reconfiguration choices. Third, in order to optimally design the reconfiguration rules in configurators so that the best possible application QoS is achieved, we designed mechanisms and protocols for a series of Quality of Service (QoS) probing and profiling services, named *QualProbes*, for the discovery of relationships among QoS parameters and reconfiguration actions. Fourth, precise decisions on adaptation timing and scale depend on accurate observations and estimations of system states. Estimation errors and significant end-to-end delay may obstruct desired accurate adaptation measures. We devised an optimal state prediction approach to estimate current states based on available state observations. Fifth, we studied another dimension of adaptation choices in the scenario where a complex application involves multiple servers and clients in a distributed environment. We present a gateway-centric approach that facilitates on-the-fly reconfiguration of client-server mappings. Finally, the architecture is designed to feature generic applicability and ease of deployment of a wide variety of applications. We present such a design emphasizing a clean division of application-neutral and application-specific middleware components.

Evaluations and validations to our approach are not only analytical, but also experimental. We have built a prototype of the *Agilos* architecture, and validated it with *OmniTrack*, a client-server based omni-directional visual tracking system. The application features rich adaptation choices in various aspects, ranging from image quality to client-server mappings. The gateway-centric approach implemented in *Agilos* is thus applicable to this scenario. Among all QoS parameters that are application-specific, the topmost priority is to preserve the tracking precision. In this work, we show that the *Agilos* architecture effectively achieves the best possible performance with respect to the most critical QoS parameter, the tracking precision, while the adaptive actions are stable, flexible and configurable according to the needs of individual applications.

TO MY WIFE FANG

Acknowledgments

First and foremost, I would like to thank my thesis advisor, Professor Klara Nahrstedt, for her invaluable directions and support throughout my research efforts towards this thesis. Her insights and suggestions to the thesis topic enlightened me in various detailed aspects throughout the work. Her directions with regards to the development of my academic thinking and writing were inspiring and helpful for my future work.

I would like to thank the members of my thesis committee, Professor Campbell, Vaduvur and Belford, for their helpful discussions on the ideas in the work, as well as careful reading and insightful comments on my thesis.

I am grateful for the support and assistance from all current and previous members of the Multimedia Operating Systems and Networking (MONET) research group, especially Dongyan Xu, Shigang Chen, Mukul Chawla, Sergio Servetto, Hao-hua Chu and Lintian Qiao. Their insightful comments for the work are of great help.

It was also an enjoyable experience working with the tracking subgroup within the MONET research group. I am especially grateful for William Kalter, Won Jeon, Jun-hyuk Seo and Mukul Chawla, for their hard work and significant contributions to the entire OmniTrack and Agilos project. The project could not achieve the current level of completeness and results without their work.

This research was supported by the Air Force Grant under contract number F30602-97-2-0121, NASA Grant under contract number NASA NAG 2-1250, and National Science Foundation Career Grant under contract number NSF CCR 96-23867.

Last but not least, my family deserves particular recognition for their unconditional emotional

support during the past years. I am grateful to my parents for their encouragements; and to my wife, Fang, for her love, dedication and belief in my graduate studies, without which all that I have achieved was not possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The algorithmic and architectural design and implementation of *Agilos* (**Agil**e Q**oS**), our middleware control architecture, is motivated by a close examination of the advantages of adaptation-based systems operating over commodity and best-effort based environments, compared with the challenges and difficulties in traditional reservation-based systems. In this chapter, we introduce the background of the research area, discuss motivations of our work, present major features and contributions of the *Agilos* middleware architecture, and outline the rest of the dissertation.

1.1 Background

As modern computer and network technologies develop at a rapid pace to serve the growth of e-businesses that deploy mission-critical servers with multimedia contents, state-of-the-art computer systems and distributed multimedia applications become more and more *complex* and *dynamic* in their behavior and requirements. On one hand, such complexities are due to their rich functionalities and problem-solving capabilities offered to end users; on the other hand, they are due to their collaborations and interactions with other applications or software components. Such integrations are intensified by the convergence of component-based software engineering approaches and distributed computing techniques, where distributed components located on heterogeneous and interconnected end systems collaborate to complete a specific task. The Internet is a typical example of such an observation: it is a network that scales to millions of computers of heterogeneous

varieties, ranging from palmtops to supercomputers. These computers are generally referred to as *end systems*, and are connected through network links of different nature. The wide variety of applications that are running on them are started and terminated constantly, and have vastly different requirements for computational and communicational resources.

Confronting such a trend of increased heterogeneity and volatility, traditional solutions with regards to the design of applications and their supporting network protocols focus on the delivery of at least a basic level of functionalities, while operating under a range of network and end system performance behaviors. Such a design may not satisfy the needs of a resource-hungry distributed application that wishes to deliver high performance and increased complexity, often with integrated multimedia contents. The concept of *Quality of Service* (QoS) is raised to solve unique problems brought by these applications, and various QoS architectures [1] are designed to provide *QoS guarantees* and satisfy the resource requirements of these applications. Without some QoS-related measures taken, these applications are very sensitive to the performance variations of the environment, including the network connection and end system resources such as computational power. A good example is a video stream that is sensitive to bandwidth limitations, or a speech stream that may become incomprehensible under excessive jitter, i.e., dynamic fluctuations of latency.

Ideally, after a certain level of QoS guarantees are provided to these sensitive applications, critical quality requirements such as soft real-timeliness [1], precision, perceptual quality, reliability and security will be satisfied. Most of these qualities can be identified and measured quantitatively with *application-level QoS parameters*. Based on user preference, some of these parameters may be more critical for the purpose of satisfying user requirements than others. For example, in a video conferencing application, the perceptual quality may be quantified by application-level QoS parameters *frame rate* and *image resolution*. In this case, a particular user may feel that the frame rate is more critical than image resolution during adaptation, or vice versa.

[1]For multimedia applications, timing constraints of media streams are stringent for playback purposes. We refer to these timing properties as *soft real-timeliness* to differ from hard real-time applications where timing deadlines are guaranteed.

In order to deliver their rich functionalities, these complex applications need to consume resources. There are three important types of resources for these applications: CPU capacity, network bandwidth and storage. CPU capacity is mostly needed in computationally intensive applications, such as those for scientific computing. Network bandwidth is necessary in all distributed applications that need data transfers among end systems. Storage resources include main memory and secondary storage, which is the resource of concern in database applications that need an exceedingly amount of storage. Since variations in resource availability affect application-level QoS parameters in various aspects, these resources are critical to the provision of QoS guarantees to sensitive applications.

1.2 Motivation

1.2.1 Challenges in the Reservation-based Approach

In order to deal with such complex multimedia applications that are sensitive to variations in resources, the ultimate goal of any QoS-aware mechanisms is to assure that the Quality of Service delivered by the execution environment matches the one required by the applications, and vice versa. A direct solution is via a portfolio of reservation-based QoS mechanisms involving resource reservations, admission control, as well as renegotiations when the demand or supply changes. Existing research achievements that fall under this category can usually provide either strict or statistical QoS guarantees, so that whenever an application is admitted for service and allocated adequate resources, a guarantee for Quality of Service via reservations is enforced, assuming that the resource usage of the application does not exceed its agreed resource reservations.

Although the advantages of such reservation-based approaches are obvious, there are difficulties for these mechanisms to solve QoS problems with complex real-world applications. These issues are enumerated as follows.

1. It is frequently hard, if not impossible, to give precise estimates of the amount of required

resources for the lifetime of a specific complex application. Such an application may execute in a distributed environment, require simultaneous access to resources of various types, and be closely integrated with functionalities of other applications, so that the inter-application boundaries are not clear. These observations make it much more difficult to issue resource allocation estimates *a priori* for reservation purposes.

2. The required amounts and types of resources during the service of an application usually vary over time, sometimes by a significant gap between the minimum and maximum usage patterns. For example, video streaming over a network link is known to demonstrate the property of Variable Bit Rate (or VBR). Such variations in resource demands and consumption are frequently observed in other complex applications, due to the inherent nature of processing and transmitting data in these applications.

3. The QoS requirements of a specific application usually vary according to the interactions from an end user during application runtime, frequently via graphical user interfaces (GUI) of the application. Combined with variations in resource demands inherent to the application, these interactions with the end user creates further difficulties in the process of estimating resource requirements *a priori* for reservation purposes. Reservation-based approaches resort to two problematic solutions. First, to reserve the maximum amount of resources required. This may not be cost-effective and beneficial for the overall system utilization, since it reduces the degree of statistical multiplexing. Second, to make *resource renegotiations* when such variations actually occur. Such renegotiations may be costly, since it involves execution of negotiation protocols within vertical layers in both operating systems level and the network protocol stack. Furthermore, if such renegotiations fail, the QoS guarantee, one of the main advantages of reservation-based approaches, may not be offered to the application during its entire lifetime.

4. Current existing commodity and off-the-shelf operating systems (OS) and network protocols, that are actively deployed in both the end systems and within the network infrastruc-

ture of Internet and Intranets, are mostly best-effort. These protocols and OS components are designed to provide the basic services that stress correctness and reliability, even when congestions, short-term disconnections or partial failures occur. Prominent examples include the TCP/IP protocol being used for most of the Internet, or off-the-shelf Windows NT operating system found in most of the low-end personal workstations or even departmental servers. Reservation-based approaches call for an overhaul and replacement of such a best-effort infrastructure with *QoS-aware* components, with a portfolio of reservation-based mechanisms integrated. This is necessary since in order to provide QoS guarantees and to reserve resources, all OS service layers and network protocols need to be involved. If any segment or layer is omitted, that part may fail to reserve resources in order to meet QoS expectations of the application.

1.2.2 Advantages of the Adaptation-based Approach

The above challenges and difficulties in reservation-based mechanisms motivate the study of the adaptation-based approach, which either operates on a standalone basis or complements existing reservation-based approaches.

As a result of the previously discussed disadvantages of reservation-based approaches, a variety of new programming models, middleware components, operating system mechanisms and network protocols that fall under the category of *adaptation-based* approaches have been developed. Such adaptation-based solutions attempt to address the following two unique issues raised by the heterogeneity of environments and complexity of applications:

1. These adaptation-based solutions best target the heterogeneous end-to-end computing environments that support most of the mission-critical applications today. From a system's point of view, we observe that either the entire end-to-end infrastructure is best-effort and QoS-unaware, or that QoS-aware system components coexist with QoS-unaware components in end systems and networks along the end-to-end path. For example, while the backbone net-

work infrastructure may adopt QoS-aware resource reservation and scheduling protocols, best-effort network protocols are still being used in local area networks, which are parts of the end-to-end path. From the application's point of view, if services with statistical or no guarantees exist in the underlying environment, the QoS level that the application demands may not be satisfied continuously. Violations of application QoS requirements may be caused by physical resource limitations such as inherent bandwidth variations and error characteristics in wireless links, or by statistical multiplexing and concurrency introduced by a dynamic number of application tasks sharing the same resource pool in end systems and networks. These observations demand that adaptation-based solutions be deployed in the system or application levels.

2. These adaptation-based solutions best target complex distributed applications with multimedia contents integrated, residing in the above described heterogeneous and dynamic environment. Since the resource demands of these applications are dynamic due to their complexities, they must be adaptive to the QoS variations during their lifetime of execution. Apparently, applications which have strict hard real-time requirements do not fit in this environment, because the adaptation-based solutions cannot satisfy these applications with deterministic QoS guarantees that they need. Frequently, despite their complexities and dynamic behavior, complex distributed applications do not have hard real-time requirements, and they usually have very *flexible* and dynamically-changing demands within a specific range of QoS requirements between a minimum and a maximum level. This range of flexibility is an important prerequisite for adaptation-based solutions to work, since they allow room for adaptations to occur. We refer to such applications as *flexible* applications. Within such a range, these applications can accept and tolerate resource scarcity to the minimum bound, and may improve its performance if a larger share of resources is allocated. On the other hand, these applications are willing to sacrifice the performance of some quality parameters in order to preserve the quality of the most critical parameter. When QoS variations

occur and QoS cannot be maintained for all application quality parameters, it is possible and desirable to trade off less critical parameters for the interests of preserving the quality of the most critical parameter. We refer to such a critical parameter as the *critical performance criterion* of an flexible application.

OmniTrack, a client-server based omnidirectional visual tracking application, is a typical example of such flexible applications. The basic client-server relationship in such an application works as follows. A tracking server captures live video frames in real time from a video camera, and sends them over the network to the tracking client. The client executes multiple computationally intensive tracking algorithms, referred to as *trackers*, which track the coordinates of interested objects. Finally, the client visually shows the result of these tracking algorithms. Our interests focus on key application QoS parameters such as the *tracking precision* of trackers, which depends on image quality, number of active trackers being executed concurrently, covered region of active trackers, and so on. These application-level QoS parameters again depend on resource availability (such as network bandwidth and CPU). As long as the tracking precision is preserved, other parameters in the application, such as the image quality, can be dynamically tuned, adjusted and reconfigured. This shows that the critical performance criterion is to keep the tracking precision as small as possible. We will describe the implementation and capabilities of the *OmniTrack* application and all aspects of its adaptation possibilities and QoS parameters in details in Chapter 9 and 10.

1.3 Features and Contributions of the *Agilos* Architecture

Based on such a detailed comparison study between reservation-based and adaptation-based approaches, our work on the *Agilos* middleware control architecture adopts an adaptation-based approach, and operates in the middleware layer, with a strong focus on critical application-level Quality of Service parameters. In this section, we first present the major characteristics of the *Agilos* middleware architecture, followed by by emphasizing the key contributions of the work

presented in this dissertation, and concluded by a list of highlights in *Agilos*.

1.3.1 Agilos: A Middleware Solution

If we examine the adaptation efforts implemented in all layers within a distributed system, we note that there are two distinct levels that adaptation may take place: within the system level (e.g. operating systems and network protocols, usually in the kernel space) and within the application level (usually in user space). Even though the actual adaptation practices and mechanisms used may be diverse, they both attempt to adapt according to condition changes in the system. However, adaptation practices in these two categories have different targets. In the system level, as in the case of various adaptive resource management schemes, the objectives of adaptation are set at a much lower level, for example packets or cells in network protocols, and time slices with respect to adaptive soft real-time scheduling algorithms for the CPU resource. In addition, the emphasis and interests are focused on global properties such as fairness, system utility or resource utilization. On the contrary, in the application level, adaptations are more focused on higher level application-specific semantics, such as optimizations towards meeting the critical performance criterion, e.g., keeping the tracking precision in the *OmniTrack* application.

Ideally, it is best to propose a design that may achieve both of the above goals, so that both system level and application level interests would be satisfied. In reality, of course, this may be impossible to achieve. However, emerging distributed applications begin to rely heavily on specialized software facilities in between applications and operating systems, referred to as the *middleware*. Traditionally, middleware components provide a layer of programming transparency by encapsulating remote procedure calls to foreign application components and objects, so that the location and platform heterogeneity within a distributed application may be shielded from application developer. Recent research efforts note that such a notion of middleware should be extended to virtually all software facilities that may provide generic services to the applications. For example, web-based applications may need a layer of middleware to provide database connectivity from the web server to backend database repository.

We have chosen to design our adaptation-based solutions in the middleware layer, since it does not require tight integration or modifications to the best-effort services in OS kernel and network protocol stack, which is the major advantage of adaptation-based over reservation-based systems. Our goal with the *Agilos* middleware architecture is to monitor system resources and achieve some system-wide adaptation properties, and be fully configurable to the application's needs and aware of application-specific semantics as well. While it is impossible to pursue both interests fully as in the ideal case, we focus on application-specific requirements and on making the adaptation path better meet the needs of these requirements, such as to satisfy the critical performance criterion.

Naturally, since both middleware components and the actual QoS-aware applications may be reconfigured to adapt to the changing environment, two approaches exist with two distinctive focuses. One approach, as adopted by the middleware solutions in the *QoO* [2] and *Da CaPo++* [3] projects, is to dynamically reconfigure the middleware itself so that it can transparently provide a stable and predictable operating environment to the application. This approach is attractive since it does not require any modifications to the application, any legacy application can be deployed with little efforts and with a certain level of QoS assurance. However, since it can only provide a generic solution to all applications, a set of highly application-specific requirements cannot be addressed. Alternatively, the middleware may be *active*, and exert strict control of the adaptation behavior of QoS-aware applications, so that these applications adapt and reconfigure themselves under such control. This approach enjoys the advantage of knowing exactly what are the application-specific adaptation priorities and requirements, so that appropriate adaptation choices can be made to address these requirements. We take the latter alternative in the *Agilos* middleware architecture.

1.3.2 Major Contributions

The key contributions of the *Agilos* middleware control architecture presented in this dissertation are summarized as follows.

1. The *Agilos* architecture bridges the functionalities of both the traditional system-level adap-

tive resource management mechanisms, such as flow control, and application-specific adaptive mechanisms, such as frame dropping schemes in MPEG. On one hand, it is designed to focus on determining adaptation strategies by tuning the application via available application-specific adaptation choices. On the other hand, it maintains system-wide properties, such as adaptation agility or fairness, in such process of tuning the applications.

2. Unlike many resource-centric approaches that focus on managing system-level resources such as CPU and network bandwidth, the *Agilos* architecture "manages" the applications themselves by exerting strict control over its adaptive behavior. In order to achieve such a goal, the adaptive policies adopted by *Agilos* is expressed by a rule base, and is highly configurable to the application's adaptation needs. Armed with such highly configurable adaptive policies, the generic decision-making process in *Agilos* is able to satisfy the critical performance criterion of an application via the selection of a suitable adaptation strategy.

3. Unlike many application-specific adaptive approaches that focus on how and when to adapt within a specific application, such as the work presented in [4] and [5], *Agilos* is an middleware architecture designed to be completely decoupled from the applications. It features generic applicability to a wide range of applications that comply to a minimal set of requirements, and the deployment process include easy steps that are straightforward to implement by the application developer. All communications between the applications and *Agilos* are made through a standard component model, which is CORBA in our implementation.

4. The *Agilos* architecture features a hierarchical design with multiple tiers. The first tier focuses on system-wide adaptation properties. The second tier focuses on application-specific adaptation choices, and may be tuned to meet the needs of individual applications. The third tier promotes reconfigurations with respect to mappings among multiple clients and servers. Such a multi-tier architecture makes it possible for *Agilos* to be both generic and flexible.

In Chapter 2, we extend the above skeleton descriptions of our contributions and features, by presenting an design overview of the *Agilos* middleware control architecture.

1.3.3 Summary

Our work with respect to *Agilos* belongs to the category of adaptation-based approaches, and is implemented in the middleware layer. The *Agilos* architecture focuses on the following features:

1. It implements a middleware-based QoS adaptation scheme that actively controls the behavior of applications in a heterogeneous distributed environment.

2. It implements the control mechanisms necessary to make correct decisions about when, how and to what extent application-level QoS adaptations should occur.

3. It is strongly focused on achieving the best possible adaptation strategy for the interests of applications, particularly emphasizing the *critical performance criterion*, the single most critical QoS parameter in the application.

4. It features a hierarchical design with multiple tiers.

1.4 Outline of the Dissertation

In the remainder of this dissertation, we present a thorough examination of our work with the *Agilos* middleware control architecture and the *OmniTrack* application. Our work in *Agilos* and *OmniTrack* is both theoretical and experimental. In Chapter 2, we first present an overview of the overall architecture of *Agilos*, with a layered division among application-neutral and application-specific middleware components. We will also introduce the rich functionalities in the *OmniTrack* application, and illustrate its adaptation choices in various domains. In subsequent chapters, we present a rigorous treatment on the design and implementation of each type of middleware components in *Agilos*, namely, the adaptors, configurators, QualProbes, negotiators and the gateway. On the theoretical front, Chapters 3, 4 and 5 focus on the design of adaptors and configurators, namely, the Task Control and Fuzzy Control Model. Chapter 6 focus on the algorithmic design of QualProbes, which provide probing and profiling services in the *Agilos* architecture. Chapter 7

discusses outstanding issues related to the relationship between multiple servers and clients in an application such as *OmniTrack*. An application-level gateway is introduced to handle responsibilities such as server switching and selections. In addition, negotiators are introduced on all clients and servers to facilitate their interactions with the gateway. On the implementation front, Chapter 8 presents the application programming interface via which the applications interact with *Agilos*, which is designed to feature generic applicability and ease of deployment for new distributed applications. Chapter 9 illustrates problems and their solutions with respect to the implementation of *Agilos* and the deployment of *OmniTrack*. Chapter 10 presents various experimental results. Chapter 11 discusses related work, and Chapter 12 concludes the dissertation and discusses future work.

Chapter 2

Agilos: An Overview

In this chapter, we give an overview of our study on *application-aware QoS adaptations*, and present original contributions in *Agilos*, a middleware control architecture. The term *application-aware adaptation* was first introduced in the Odyssey project [6, 7], and was defined as adaptation that exists in between application-transparent system-level adaptation mechanisms, and the *laissez-faire* approach, where applications deal with adaptations completely by themselves. The functionality of the *Agilos* architecture exactly fits such definition of application-aware QoS adaptations, with the benefit of balancing the interests and advantages of adaptation in both system and application levels.

2.1 Objectives of *Agilos*

The goal of the *Agilos* architecture is to solve the following three major problems towards application-aware QoS adaptation.

1. *Application-generic Adaptation*. *Agilos* is designed to serve as a distributed and application-neutral coordinator to control the adaptation behavior of all concurrent applications in the end systems sharing the same pool of resources (e.g. CPU), so that if viewed globally, these applications do not adapt in a conflicting or unfair way.

2. *Application-aware Adaptation.* By directly controlling the adaptive behavior of a complex distributed application, *Agilos* assists the application to make more effective adaptation decisions. Such decisions include when, how and to what extent adaptation is carried out within an application. We refer to the methods used for making these adaptation decisions as *adaptation strategy*. Since the objective of all adaptation-based approaches is to provide the best possible Quality of Service to an application allowed by the current conditions of resource availability, the important issue is to find ways to qualitatively measure the optimality of an adaptation strategy. In our work, we focus on an application-specific *critical performance criterion*, which is associated with the most critical QoS parameter within the application.

3. *Probing and Profiling.* On behalf of the applications, *Agilos* is responsible to probe the current dynamics of application-level QoS parameters and resource availability, in a heterogeneous environment and on the fly. In addition to such probing services, it is also responsible for recording the probing results in QoS profiles, so that inter-parameter mappings may be derived.

Without the introduction of such a middleware architecture, applications are compelled to make application-level adaptation decisions within themselves. They are thus required to implemented QoS probing and any adaptation strategies within the application. This increases the burden with application development, the results may not be fair to other applications sharing the same pool of resources, and proprietary implementations that are tightly bound to a specific application cannot be shared with other applications, even with similar semantics and behavior.

The ultimate objective of *Agilos* is to control the adaptation process within the application so that under any resource conditions, it is steered towards maximum achievable user satisfaction with respect to delivered performance of a critical QoS parameter.

2.2 The *Agilos* Architecture

Agilos is designed as a *three-tier* architecture. In the first and lowest tier, application-neutral *adaptors* and *observers* support low level resource adaptation by reacting to changes in resource availability. Therefore, adaptors and observers both maintain tight relationships with individual types of resources. In the second tier, application-specific *configurators* are responsible for making higher-level functional adaptation, including decisions on when and what application functions are to be invoked in a client-server application, based on on-the-fly user preferences and application-specific rules. Furthermore, *QualProbes* provide QoS probing and profiling services so that application-specific adaptation rules can be either derived by measurements or specified explicitly by the user. In the third tier, adaptation in the highest level is performed in a distributed environment, with the assistance of a centralized *gateway* and multiple *negotiators* on both clients and servers. These components are introduced to control adaptation behavior in an application with multiple clients and servers, so that dynamic reconfigurations of client-server mappings are possible and tuned to the best interests of the application.

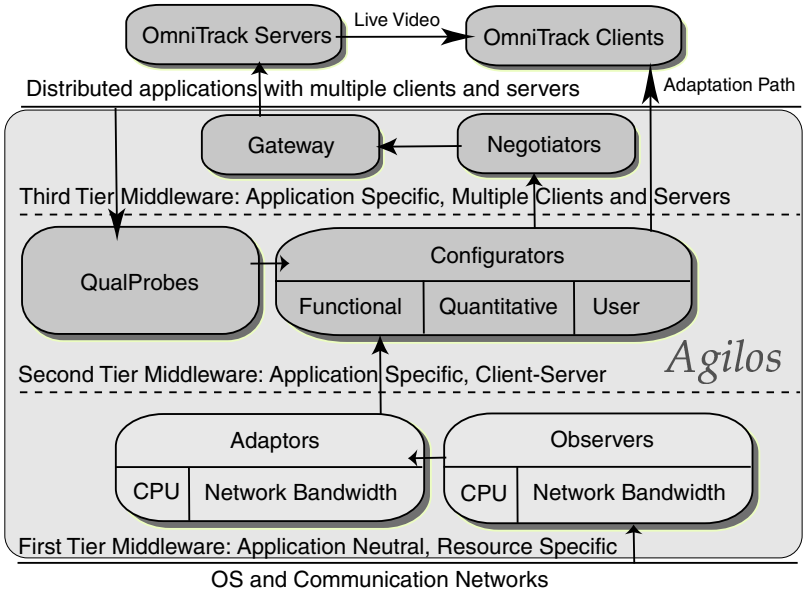


Figure 2.1: The Hierarchical Design of the *Agilos* Architecture

For the interests of generic applicability and ease of deployment of new complex applications, all communications among the middleware components, as well as between middleware layer and the application, are designed to be via a standard service enabling platform, such as the Common Object Request Broker Architecture (CORBA) [8] or the Component Object Model (COM) [9]. This three-tier hierarchical design of the *Agilos* architecture is shown in Figure 2.1.

Since the goal of middleware is to actively exert control of applications, a closed control loop is naturally formed to steer the applications so that they react to resource fluctuations. Such a control loop is formed with an integrated framework including the adaptors, configurators and observers, shown in Figure 2.2. The illustration shows a hybrid control approach adopted by *Agilos*. With the controller comprising of application-neutral adaptors and application-specific configurators, the generated control actions are both globally fair to other concurrent applications sharing the same local resources (such as the CPU) on an end system, and specific to the application's adaptation needs and critical performance criterion as well.

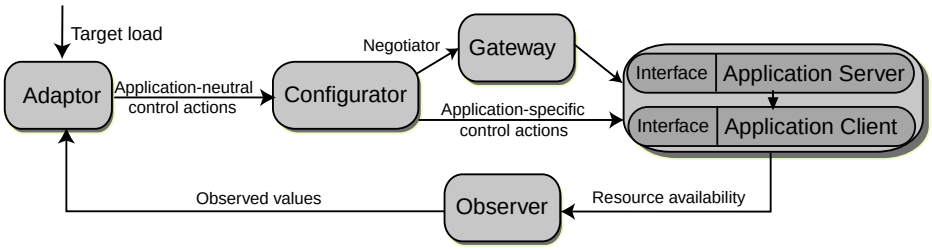


Figure 2.2: The Control Loop In the *Agilos* Architecture

2.3 The Application Model

In all subsequent discussions about application QoS parameters and resource types, we assume a *Task Flow Model* [10] for distributed applications. A complex distributed application can be modeled as several *application tasks*, each task generates output for the subsequent task, which can be measured by one or more *output QoS* parameters. Such output forms the input of subsequent tasks. In order to process input and generate output, each task requires a specific amount of

resources. An acyclic task graph, as shown in Figure 2.3 can be used to illustrate such a model.

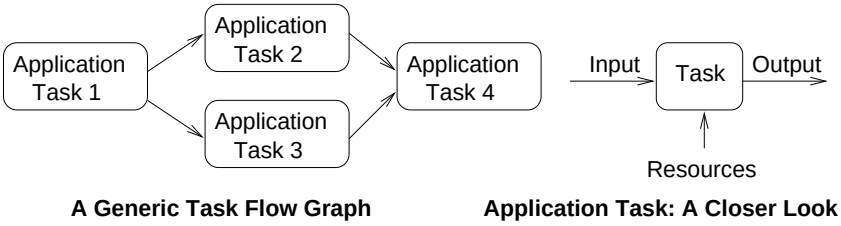


Figure 2.3: Illustration of The Task Flow Model

With such a conceptual model, we note that there may be various definitions of the concept *application task*, distinguished among themselves by the *granularity* of functional partitions in the application. Since we attempt to optimize the adaptation behavior of the application to achieve a performance goal, we divide the applications with *coarse granularity*, and demand that each task must present a one-to-one mapping to an individual executable component within the application. Static or dynamic linked library objects (such as codec or encryption modules) and individual working threads are not tasks themselves, though they may be partitioned as *subtasks*. As an example, the Task Flow Model of *OmniTrack* is shown in Figure 2.4.

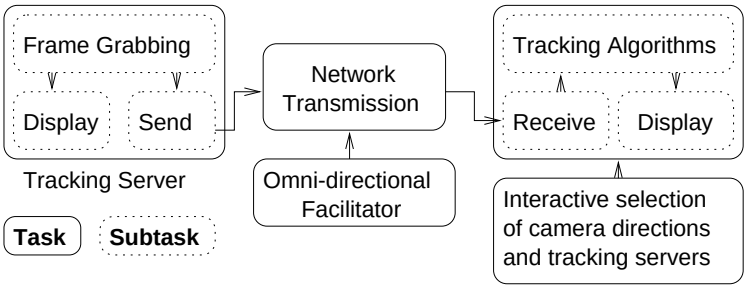


Figure 2.4: The Task Flow Model of *OmniTrack*

2.4 First Tier: Adaptors and Observers

The design objectives of the adaptors are as follows. First, they react to variations in resource availability, where the output is governed by customized algorithms derived from standard digital control algorithms, such as PID control. Second, with respect to resources, adaptors attempt to

maintain fairness among all concurrent applications within the same end system, and to control the application QoS requirements for system resources. Third, the control algorithm is parameterized and highly configurable so that a variety of different *adaptation agility* (the responsiveness of the system to changes) can be achieved. The adaptors and observers are designed to be integral part of the *Task Control Model*, discussed in detail in Chapter 3. The adaptors are based on control-theoretical methodology, using a customized PID (Proportional Integral Derivative) control algorithm.

Each system-wide adaptor corresponds to a type of resource, such as CPU and network bandwidth. Such an adaptor controls all concurrent applications sharing the resource in an end system. Three properties in an adaptor are configurable: the *target load*, *weights* of importance for each application, and *adaptation agility*. First, the target load is the load on resource consumptions at control equilibrium, shared by all applications. Should variations occur, the adaptors produce output in order to steer applications towards the target load on resources. Such a load depends on resource demands of each application and the level of application concurrency. Second, *weights* assigned to applications represent the relative importance of applications sharing the same resources, i.e., an application with a higher weight is allowed more resources following the weighted max-min fairness property. Third, *adaptation agility* expresses the responsiveness of the adaptors reacting to variations. To ensure fairness, it is configurable on a system-wide basis for all applications.

2.5 Second Tier: Configurators and QualProbes

2.5.1 Design of Configurators

The configurators determine discrete control actions based on application-specific needs and control values produced by adaptors. Their behavior is fully configurable via the definition of a *rule base*. An example of a rule in such a rule base could be "`if (cpu is high) and (rate is low) then rateaction := compress`". The configurators are based on the *Fuzzy*

Control Model, discussed in detail in Chapter 5. The application-neutral output of the adaptors is piped into the configurator as input to the inference engine based on the rule base. Any output from the inference engine is then used to directly control higher-level adaptations in applications. Figure 2.5 shows such a design within the configurator.

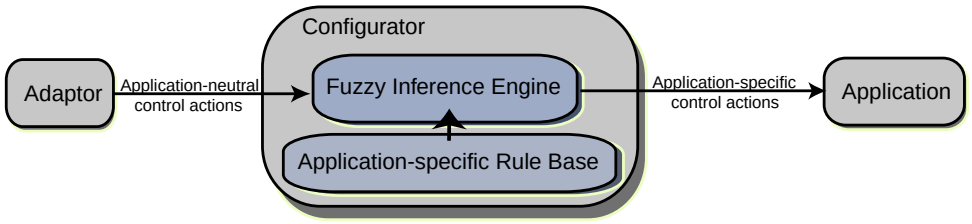


Figure 2.5: The Design of Configurator

In order to meet the needs of a wide variety of adaptation possibilities within an application, the configurators can be categorized in three groups. First, *functional* configurators attempt to reconfigure the application via the control interface, such as switching from one compression format to another. They decide which reconfiguration choice to activate via the control interface to the application. Such reconfigurations are only carried out in extreme cases where resource availability or application-specific parameters are beyond certain thresholds, defined by the rule base. Second, *quantitative* configurators tune application-level QoS parameters, such as tuning the compression ratio of a specific codec. Third, *user* configurators integrate application-specific user interfaces, with visual tuning and selection interfaces for the user to directly manipulate and adapt the application. This is necessary to fulfill the user's needs, especially when certain application behavior cannot be detected automatically. For example, in *OmniTrack*, if the object roams out of view, the user may need to interact with the camera to pan and tilt the camera or switch to a different view, so that tracking may resume.

2.5.2 Design of QualProbes

As shown in Figure 2.2, adaptors, configurators and observers form a closed loop to actively exert control of the application via its control interface. A clean division of application-neutral (e.g.

adaptors) and application-specific middleware components (e.g. configurators) ensures generic applicability to a wide variety of applications, as well as the ease of deploying new applications with minimum efforts. However, what is left unanswered is the following set of questions: (1) How do applications specify to the middleware what QoS parameter they favor? What is their critical performance criterion that any adaptation should focus on? (2) Frequently, the critical performance criterion cannot be directly measured or calculated on-the-fly, such as the tracking precision in OmniTrack or the overall user satisfaction of streamed media. In such cases, how do the changes in controllable QoS parameters relate to the critical performance criterion and resource demands?

QualProbes, a set of middleware QoS probing and profiling services, addresses these concerns. Its responsibilities are three-fold: First, it allows applications to specify to the *Agilos* middleware about its critical performance criterion, along with its relationship with other controllable QoS parameters. Such a specification is made in the form of a dependency tree. Second, the QoS probing service actively probes the application in trial runs for detailed QoS mapping functions among application QoS parameters. Such off-line probing is based on the dependency tree that the application specifies. Third, the QoS profiling service logs all probing results in QoS profiles. The actual specification of the rule base in configurators is based on these profiles.

Chapter 6 thoroughly presents a dependency tree model for capturing application-level QoS parameters and the relationships among themselves and between application and system QoS parameters, as well as internal mechanisms in *QualProbes*.

2.6 Third Tier: Gateway and Negotiators

The addition of a third tier, including the *gateway* and *negotiators* within *Agilos* architecture, removes the limitation that adaptation can only be performed within individual clients and servers, and allows for distributed adaptation control. There may be instances when the best option for a client may be to switch its data feed to a new server which better exploits available resources. For

the OmniTrack example, when bandwidth becomes restricted, local adaptations may be insufficient to provide an acceptable level of QoS. Rather, the best adaptation may be to switch to a compressed video format which requires less bandwidth, such as MPEG. However, the current server may not be able to provide video in MPEG format, either due to a software or hardware insufficiency. If there were another server providing the equivalent video in MPEG format, the best client adaptation would be to switch to the compressed video server. The gateway, when collaborated with negotiators, is designed to facilitate such adaptation choices within the third tier.

Based on these goals, the third tier is designed as a gateway-centric architecture. Figure 2.6 shows the role of the gateway and the topology of nodes within the framework. The gateway manages control connections between a client and multiple servers, and assists the client in its distributed QoS adaptive behavior. Note that the transfer of data will follow a direct path between one client and one server.

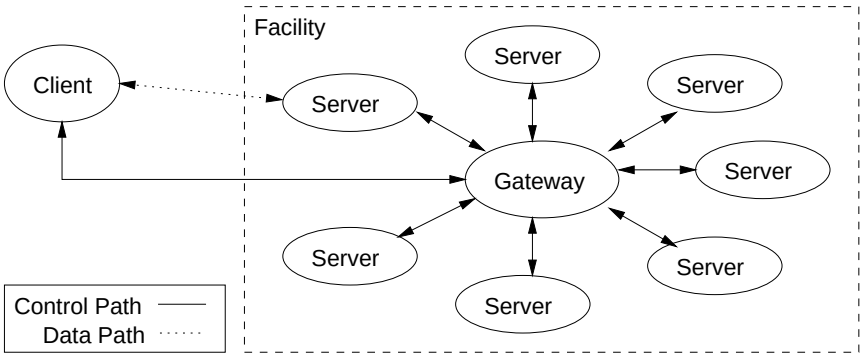


Figure 2.6: The Role of Gateway

The gateway and negotiators on servers are collectively coordinated to serve as a single *facility*. At any given time, each client is being serviced by a single server within the facility. The decision as to which server is serving the client is dynamically chosen based on server load and its ability to satisfy user requirements in the client.

Chapter 7 discusses important design issues related to the gateway and negotiators, which facilitates adaptation in a multiple-client multiple-server scenario.

2.7 *OmniTrack*: A Case Study

As a case study, we have developed *OmniTrack*, a distributed omni-directional visual tracking system, using tracking algorithms in the *XVision* [11] project. *OmniTrack* is a flexible, multi-threaded and client-server based application, which adopts complex tracking capabilities in multiple domains, such as visual object tracking, camera tracking and switching, and features full integration of user preferences. This application illustrates the coexistence of multiple adaptation possibilities, ranging from image properties, codec choices, server selections, to tracker quantities and variety. The actual adaptation choices are based on a combination of user preferences and decisions made by the underlying *Agilos* middleware control architecture. An illustration of *OmniTrack* architecture is shown in Figure 2.7.

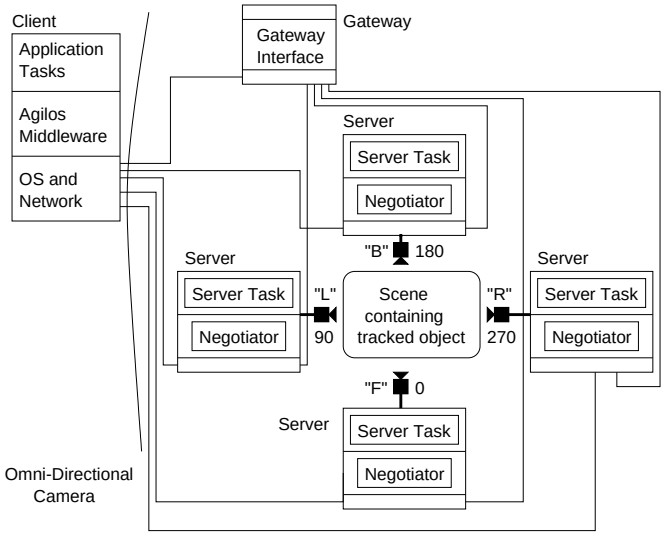


Figure 2.7: *OmniTrack*: A Distributed Omni-Directional Visual Tracking System

In a basic client-server relationship, the tracking server captures live video from its panable camera, encodes the video in a specific media format, and streams the encoded frames to the client. One or multiple tracking algorithms are executed at the client side, attempting to track the position of corresponding moving objects in the decoded video. In the scenario with multiple servers and clients, a server spawns a separate working thread to handle video streaming responsibilities associated with one particular client. On the contrary, each client only receives the video stream

from one of the servers at a time. Different servers have different characteristics with respect to the angle of views, media codec selection and server workload, thus enabling the client to switch from one to another based on its preferences. A *User Configurator* is also available on the client side, which is responsible for all interactions with the end user with respect to omni-directional views. The user may pan the camera, switch to a different view and specify advanced preferences related to the type and group of objects currently tracking.

OmniTrack is implemented in Windows NT, deployed under the control of Agilos middleware. *OmniTrack* exports a *control interface* which is clearly defined in the CORBA Interface Definition Language (IDL). All control commands made by the Agilos middleware are carried out through such a control interface via CORBA. This ensures that Agilos middleware architecture is generic and applicable to any application. Besides exporting the control interface, *OmniTrack* reports on-the-fly observations of its application-specific QoS parameters to the CORBA Property Service, so that they are always observable from the middleware's point of view.

Chapter 9 presents implementation details of the *Agilos* middleware architecture and deployment of *OmniTrack*, while Chapter 10 presents important adaptation choices available in *OmniTrack*.

Chapter 3

Task Control Model

Our objective is to rigorously model the functionalities in the first tier of the *Agilos* middleware architecture. The first tier is application-generic in the sense that it is designed to focus on resource-based and system-wide properties related to adaptation, such as adaptation agility, stability, equilibrium values, as well as fairness among applications sharing the same end system resources.

3.1 Overview

As noted in Section 2.2, the fundamental goal of *Agilos* is to actively exert control of the applications. A closed control loop, as shown in Figure 2.2, is naturally formed to steer applications so that they may timely react to resource variations. Obviously, this model corresponds identically to a typical control system. Such an analogy is explained as follows.

In a control system, there is a target system to be controlled. This target system takes appropriate actions to process the input. Such an input is determined by a *controller*, which monitors the states inside the target system, and compares them to the desired values referred to as the *reference* or *setpoint*. Both the target system and the controller can be modeled mathematically, and such a model for the controller includes the *control algorithm*. Similarly, in the *Agilos* middleware architecture, making adaptation decisions also requires observations of the current states in the target application, and such decisions are also enforced as input to the target application. The adaptors

are thus introduced to implement the functionality of a *controller*, while observers are responsible for monitoring the states of the target applications, including availability of resources.

The Task Control Model [12, 13] is derived based on such an analogy. Theoretically, the critical parts in the Task Control Model are: (1) the actual mathematical model for the target applications; (2) the control algorithms in the adaptors. The key contributions of the Task Control Model are as follows.

1. By establishing a linear mathematical model for the target applications in the Task Control Model, it is possible to utilize various proved theorems and properties available in the control theory. Thus, the model can rigorously evaluate the equilibrium, stability and adaptation agility properties of the control algorithms in adaptors, using control theoretical techniques.

2. The Task Control Model separates actual adaptation choices, such as tuning application QoS parameters, from the control algorithms used to determine the timing and scale of such adaptations. It actually makes it possible to flexibly match various application-specific adaptive mechanisms, such as media scaling, filtering and functional reconfigurations, with application-neutral control signals produced by the adaptors. On one hand, the control algorithms in the adaptors guarantee equilibrium, stability, fairness and adaptation agility properties; On the other hand, the detachment from application-specific adaptation mechanisms ensures flexibility and ease of deployment of new applications.

3. Since the control theory provides a natural mapping and a solid foundation for the Task Control Model, it is used to derive the customized control algorithms being executed in the adaptors. Along with the mathematical model derived for the target applications, such control algorithms in the adaptor complete the closed control loop, which models the control process that the *Agilos* adaptors and observers go through to adapt the application.

3.2 Review of the Task Flow Model

In order to analyze QoS adaptation using control-theoretical methodology, we need a strict mapping between traditional control systems and our system architecture to control QoS adaptations in a distributed environment. For this purpose, we consider the Task Flow Model that we briefly described in Chapter 2. In such a model, *application tasks* represent execution units that perform certain actions to deliver a result to other application tasks or the end user. The *Task Flow Graph* is a directed acyclic graph which consists of multiple application tasks, and illustrates the consumer-producer dependencies among the tasks. A directed edge from the task T_i to the task T_j indicates that the task T_j uses the output and its QoS produced by the task T_i. Tasks can be uniquely characterized by its *input quality*, *output quality* and *utilized resources*, all influencing the state space of an application task.

(a) A Generic Task Flow Graph

(b) One Application Task in the Task Flow Graph

Figure 3.1: Review of the Task Flow Model

Figure 3.1(a) illustrates a generic Task Flow Graph, Figure 3.1(b) illustrates the key characteristics of a single task in the Task Flow Graph.

3.3 Task Control Model for Adaptors

The Task Control Model concentrates on a single application task in the Task Flow Graph, referred to as the *target task*, which is the application task to be controlled by tuning or reconfiguring the target application. In addition, we introduce the *adaptor*, which performs the control

algorithm, as well as the *observer*, which observes the states of the target task and feeds them back to the adaptor.

The ideal objective of the Task Control Model is to achieve the following properties. First, dynamic changes in application QoS requirements, possibly affected by user interactions on the fly, can be accommodated in a timely fashion. This property is reflected by the characteristics of *adaptation agility* when modeling the adaptation behavior. The property of adaptation agility also affects the way that the model reacts to variations in resource availability. Second, in order to accommodate concurrency of resource access among multiple applications sharing the same pool of observable resources (often on the same end system), fairness needs to be balanced among all competing tasks accessing the shared resource. These properties will be satisfied by a careful design of a customized control algorithm in the adaptor, with the assistance of the observer. The key elements in the Task Control Model are enumerated as follows.

1. **Adaptor**: The adaptor implements the control algorithm derived from the control theory, and produces a series of *control signals* based on a specific control algorithm. These *control signals* are used by the target task to tune the controllable parameters. *Controllability* has a two-fold interpretation. First, "a parameter is *controllable*" means that it is possible to dynamically tune its values in the target task. Second, it also means that by changing values of the parameter, it is possible to affect the internal states of the target task and thus eventually affect the quality associated with the critical QoS parameter.

2. **Task states**: In order to apply the linear control theory, we need a precise analytical model to characterize the internal dynamics in the target task. We refer to the parameters in this model as *task states*. The most important task states in any application task are its parameters directly correspond to the resource usage of the application. All tasks have to consume resources in order to perform actions on input and produce output.

3. **Observer**: The adaptor needs knowledge about the current states of the target task in order to execute the control algorithm. If the task states are observable, they are observed by

the observer and propagated to the adaptor. Otherwise, if some related parameters can not be observed, observers will estimate or predict the current states, based on the estimation algorithm of its choice. We present an example of such an estimation algorithm in Chapter 4, for the purpose of estimating task states within the transmission task in a distributed environment.

Figure 3.2 summarizes the Task Control Model, as well as different roles of the target task, adaptor and observer in a typical Task Flow Graph.

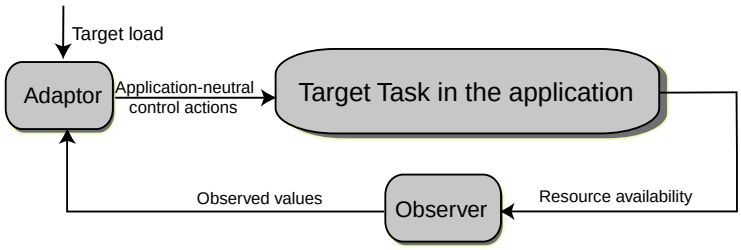


Figure 3.2: The Task Control Model

3.3.1 Generic Form of Modeling a Target Task

Once the above concepts are introduced in the Task Control Model, we are ready to present the Task Control Model and the customized control algorithm. In the most generic fashion, we use $\mathbf{x}$ for the vector of task states, $\mathbf{u}$ for the vector of controllable input parameters of the target task, $\mathbf{z}$ for the vector of observed output parameters, $\mathbf{w}$ for the vector of uncontrollable variations in the task, and $\mathbf{v}$ for the vector of the observation errors. Using the above notations, we model the target task in the Task Control Model with the following equations:

$$\frac{d\mathbf{x}(t)}{dt} \equiv \dot{\mathbf{x}}(t) = \mathbf{f}[\mathbf{x}(t), \mathbf{u}(t), \mathbf{w}(t), t] \tag{3.1}$$

$$\mathbf{z}(t) = \mathbf{h}[\mathbf{x}(t), \mathbf{v}(t), t] \tag{3.2}$$

With the above definition, the task is said to be at *equilibrium* when:

$$\dot{\mathbf{x}}(t) = 0 = \mathbf{f}[\mathbf{x}(t), \mathbf{u}(t), \mathbf{w}(t), t] \tag{3.3}$$

An equilibrium is *stable* if small disturbances do not cause the state $\mathbf{x}$ to diverge and oscillate. Otherwise, it is an unstable equilibrium. From a practical point of view, this formal definition of *stability* is identical to the definitions in previous work [14, 15], where *stability* is defined as the ability of the system to steer the target system back to equilibrium state after a disturbance has occurred.

The above stated definitions are generic and can illustrate a wide variety of adaptation capabilities of the target task. According to these definitions, the target tasks may be continuous in time, non-linear and time-varying. In this chapter we study a subset, namely, the tasks that can be approximated without loss of accuracy by discrete and linear equations as the following form:

$$\mathbf{x}(k) = \mathbf{\Phi}\mathbf{x}(k-1) + \mathbf{\Gamma}\mathbf{u}(k-1) + \mathbf{w}(k-1) \tag{3.4}$$

$$\mathbf{y}(k) = \mathbf{H}\mathbf{x}(k) \tag{3.5}$$

$$\mathbf{z}(k) = \mathbf{y}(k) + \mathbf{v}(k) \tag{3.6}$$

where $k = 1$ to k_{max}, and $\mathbf{\Phi}$, $\mathbf{\Gamma}$, and $\mathbf{H}$ are known transition matrices without an error. In later discussions, we develop a concrete analytical model based on the above generic linear model, which is frequently used within the state space approach in control systems.

3.3.2 Concrete Form of the Task Control Model

The Task Flow Model stated in Section 3.2 can characterize a wide variety of application tasks. To demonstrate a concrete example, we consider the following scenario. Let us assume multiple application tasks competing for a shared resource pool with the capacity C_{max}. Each task makes *requests* for resources in order to perform their actions on inputs and produce outputs. These requests may be *granted* or *outstanding*. If a request is granted, resources are allocated immediately.

Otherwise, the request waits in the outstanding status until it is granted. The system is granting requests from multiple application tasks with a constant *request granting rate* c.

The mapping between the abstract notation *resource requests* and the actual services that process the resource requests varies among different types of system resources. For *temporal* resources, such as central processing capability and transmission throughput, where the resources are shared in a temporal fashion, outstanding resource requests may be mapped to the waiting queue, and granted requests may be mapped to allocated temporal resources, such as bandwidth. For *spatial* resources, such as volatile or non-volatile storage capacity, outstanding requests may be mapped to the actively used and occupied capacity, such as temporal buffers in caches and real memory (swapped in pages), and granted requests may be mapped to the reclaimed capacity by the system due to inactivity, such as swapped out pages. The framework presented in this section applies to both cases.

To be more exact, we take CPU and network bandwidth as two examples of all resource types. For network bandwidth resource, from the point of view of a client-server based distributed application, outstanding resource requests can be interpreted as those data units [1] that are in transit between the server and the client. Granted resource requests, in this case, can be interpreted as the data units that have been received at the server. For CPU capacity resource, since allocation of CPU is on a time-sharing basis [2], the interpretation of the above abstract notions is more indirect. In order to achieve this, we can first construct an ideal case where full CPU capacity is available to the application, i.e., the application is running at the top possible speed that is necessary to deliver the functionalities of the application. We can thus utilize the measurements within the application in the ideal case as an ideal reference value. In actual cases, we interpret *granted resource requests* as the actual measurements in the application, and *outstanding resource requests* as the difference between ideal measurements and actual measurements.

[1] At the system level, data units may be cells or packets. However, at the application level, the actual interpretation of *data units* depend on the application-specific semantics. For example, in media streaming applications a data unit may be interpreted as a *frame*.

[2] An instruction at the assembly level of the application is either issued to the CPU or blocked for its time slice by the operating system scheduler.

Figure 3.3 illustrates the above scenario and the complete Task Control Model. Naturally, if the target task is greedy and makes an excessive number of resource requests in a short period of time, it is not fair to other tasks sharing the same resource pool. Thus, the request rate needs to be throttled by the adaptor. What the adaptor tries to control is the resource request rate, made by the target task, so that it does not exceed its fair share. The adaptor executes the control by producing control signals, which are translated to parameter-tuning actions within the application.

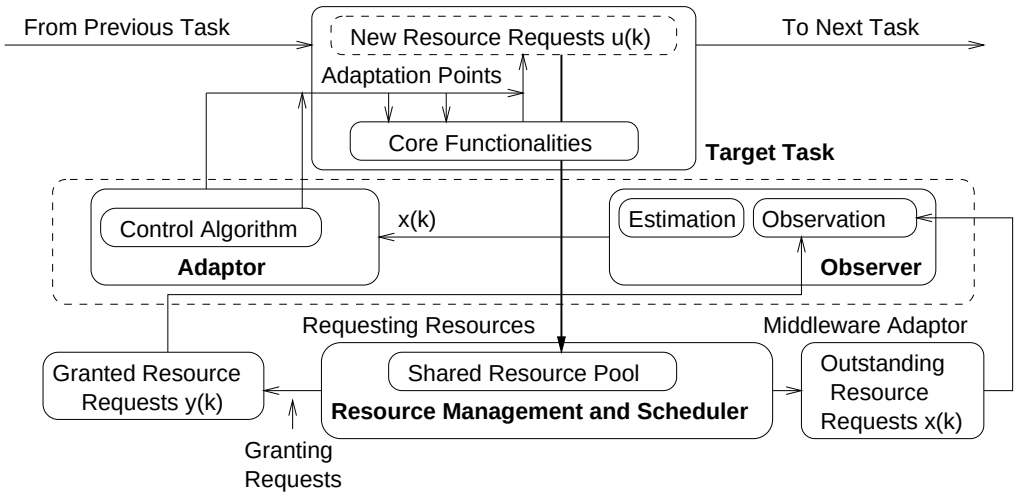


Figure 3.3: Control-theoretical Components of the Task Control Model

In the Task Control Model, we define the following variables corresponding to a target task T_i:

1. c is the constant *request granting rate* that the system grants resource requests from multiple application tasks.

2. t_c is a *constant sampling time interval*, which is the time elapsed in interval $[k, k+1]$, k being time instants;

3. $r_i(k)$ is the *number of resource requests* made by T_i during $[k, k+1]$;

4. $u_i(k)$ is the *number of resource requests* during $[k, k+1]$ allowed by the *adaptor* of T_i, which is referred to as the *adapted request rate*;

5. $x(k)$ is the *total number* of *outstanding resource requests* made by *all tasks* at time k;

6. $M(k)$ is the total number of active target tasks competing for resources in the system;

7. $A(k)$ is the set of target tasks at time k whose request rate is adapted by the *adaptor*, $N(k)$ is the set of other target tasks that are not affected by the *adaptors*;

8. $l(k)$ is the number of tasks in $A(k)$ ($l(k) = |A(k)|$), $M(k) - l(k)$ is the number of tasks in $N(k)$ ($|N(k)| = M(k) - l(k)$). We assume that both $M(k)$ and $l(k)$ stay constant within one time interval $[k, k+1]$.

9. w_i is the static weight of T_i showing its relative priority or importance compared to other target tasks.

Using the above notation, the derivative of outstanding resource requests can be described as follows:

$$\dot{x} = x(k) - x(k-1) = \sum_{i=0}^{M(k-1)} u_i(k-1) - c \tag{3.7}$$

The Difference Equation (3.7) depicts the internal dynamics of the adaptive system, where c is the constant request granting rate.

The objective of the control is to maintain the number of outstanding requests x to stay around a specific *reference* value $x^c(k)$. Under the assumption that the dynamics of the adaptive system behave according to the Equation (3.7), we can derive a control algorithm in the *adaptor* for T_i to calculate $u_i(k)$ values, which will lead to the desired values for x. For example, if a standard proportional-integral-derivative (PID) control [16] [3] is engaged, then $u_i(k)$ obeys the equation

$$u_i(k) = u_i(k-1) + \alpha[x^c(k) - x(k)] + \beta\{[x^c(k) - x(k)] - [x^c(k-1) - x(k-1)]\} \tag{3.8}$$

where α and β are configurable scaling factors. This is one of the many effective control

[3] PID control is a classic control algorithm where the control signal is a linear combination of the error, the time integral of the error, and the rate of change of the error.

algorithms illustrating the ability of the Task Control Model to capture the adaptation dynamics and map these dynamics onto a classic control model.

3.4 Control Theoretical Analysis of the Adaptor

This section continues with a rigorous analysis of the stability, fairness and adaptation agility of the Task Control Model characterized by the target task in Equation (3.7) and the PID control algorithm in Equation (3.8), in order to prove the validity of our approach using the Task Control Model.

We assume in our analysis that the controllable parameter of task T_i is the request rate $r_i(k)$. The adaptor may control $r_i(k)$ at a lower rate $u_i(k)$, namely, $u_i(k) \leq r_i(k)$. The PID control algorithm presented in Equation (3.8) becomes:

$$u_i(k) = \Psi_{r_i(k)} \ \{ \ u_i(k-1) + \alpha(x^c(k) - x(k)) + \beta[(x^c(k) - x(k)) - (x^c(k-1) - x(k-1))] \ \} \tag{3.9}$$

Where $\Psi_t(x)$ is defined as:

$$\Psi_t(x) = \begin{cases} 0 & \text{if } x < 0, \\ t & \text{if } x > t, \\ x & \text{otherwise.} \end{cases} \tag{3.10}$$

In addition, since at time k we assume that, among all target tasks, $l(k)$ tasks are adapted by their respective adaptation tasks, and $M(k) - l(k)$ tasks are not affected, we conclude that the total number of outstanding resource requests in the system shows the following dynamic property (an extension to the Equation (3.7)):

$$x(k) = \Psi_{C_{max}} \left\{ x(k-1) + \sum_{T_i \in A(k-1)} u_i(k-1) + \sum_{T_i \in N(k-1)} r_i(k-1) - c \right\} \quad (3.11)$$

$$= \Psi_{C_{max}} \{ x(k-1) + l(k-1)\bar{u}(k-1) + R(k-1) - c \}, \text{ where} \quad (3.12)$$

$$R(k) = \sum_{T_i \in N(k)} r_i(k), \text{ and} \quad (3.13)$$

$$u_i(k) = \gamma_i(k)\ l(k)\ \bar{u}(k) \quad (3.14)$$

where $\bar{u}(k)$ is the average rate for all $u_i(k)$ that satisfy $u_i(k) < r_i(k)$. In this equation, $\gamma_i(k)$ is the *dynamic weight* of task T_i which indicates priority for resource requests, and satisfies $\sum_{T_i \in A(k)} \gamma_i(k) = 1$. The dynamic weight of T_i can be derived from the *static weight* w_i of T_i, with the following calculation:

$$\gamma_i(k) = \frac{w_i}{\sum_{T_j \in A(k)} w_j} \quad (3.15)$$

Combining the Equation (3.9) and Equation (3.12), we obtain the complete characterization of the adaptation system. Associated with each task T_i, we have a *static weight* w_i, and three internal task states:

$$\mathbf{x}_i(k) = \left\{ x^c(k) - x(k), x^c(k-1) - x(k-1), \gamma_i(k-1)\ l(k-1)\ [\bar{u}(k-1) - \frac{c - R(k-1)}{l(k-1)}] \right\}^T \quad (3.16)$$

$\mathbf{x}_i(k)$ is the task state matrix of T_i, and $x^c(k)$ and $\frac{c-R(k)}{l(k)}$ are the equilibrium values of $x(k)$ and $\bar{u}_k$, respectively. The detailed analysis of the equilibrium states is given in Section 3.4.1.

One special case is when $x^c(k)$ always remains constant, which is x^c. In this case, Equation 3.9 is reduced to

$$u_i(k) = \Psi_{r_i(k)} \ \{ \ u_i(k-1) + \alpha(x^c - x(k)) - \beta[x(k) - x(k-1)] \ \} \quad (3.17)$$

which is equivalent to

$$u_i(k) = \Psi_{r_i(k)} \ \{ \ u_i(k-1) + \alpha(x^c - x(k)) - \beta[l(k-1)\bar{u}(k-1) + R(k-1) - c] \ \} \quad (3.18)$$

If we consider Equation 3.14, Equation 3.18 becomes

$$u_i(k) = \Psi_{r_i(k)} \left\{ \ u_i(k-1) + \alpha(x^c - x(k)) - \beta[\frac{u_i(k-1)}{\gamma_i(k-1)} + R(k-1) - c] \ \right\} \quad (3.19)$$

Equation 3.18 and 3.19 may be explained as follows. The u_i at the next time instant is determined by the u_i at the current time instant and two corrective terms. The first corrective term is based on the difference between the target load and current load, while the second corrective term is based on the resource demands from all active target applications. In such a control algorithm, the first term ensures the transient response of the system, and the second term (weighed by $\gamma_i(k)$) ensures fairness properties at system equilibrium. With the second term, the system is able to rebalance towards the new equilibrium when a new application enters or when existing applications are terminated. A more detailed analysis of these properties may be found in Section 3.4.1 and Section 3.4.2.

3.4.1 Equilibrium Analysis

Now that we have established the control algorithm in the adaptor, we start to analyze the exact value for which the system stays at equilibrium. The ideal case is that $x(k)$ always stays the same as the reference $x^c(k)$. Let us assume that for a specific period of time $[k_1, k_2]$, $x^c(k)$, $l(k)$, $M(k)$ and $R(k)$ are all stable and stay at constants x^c_s, l_s, M_s and R_s, respectively. Then we show the following properties:

Theorem 1: Within $[k_1, k_2]$, the number of outstanding resource requests x in the system, established by the Equations (3.9) and (3.12), will converge to an equilibrium value which equals to the reference value x_s^c. In addition, the system also fairly shares resources among competing tasks according to their static weights.

Proof: Let x_s and $\bar{u}_s$ be the equilibrium values corresponding to the system established by the Equations (3.9) and (3.12).

$$x_s = \Psi_{C_{max}}\{x_s + l_s\bar{u}_s + R_s - c\} \tag{3.20}$$

$$\gamma_i\ l_s\ \bar{u}_s = \Psi_{r_i}\left\{\gamma_i\ l_s\ \bar{u}_s + \alpha(x_s^c - x_s)\right\} \tag{3.21}$$

Ignoring the threshold cases, the solution to the Equations (3.20) and (3.21) is

$$\bar{u}_s = \frac{c - R_s}{l_s} \tag{3.22}$$

$$x_s = x_s^c \tag{3.23}$$

Equation (3.23) directly proves the first part of the theorem. Let us assume that the stable set of adapted tasks is A_s. Then the Equation (3.22) can be rewritten for task T_i at equilibrium as follows:

$$(u_i)_s = \gamma_i\ l_s\ \bar{u}_s = \frac{w_i\ l_s}{\sum_{T_j \in A_s} w_j}\frac{c - R_s}{l_s} = \frac{w_i\ l_s}{\sum_{T_j \in A_s} w_j}\ \left\{\frac{c}{M_s} + \frac{(M_s - l_s)\frac{c}{M_s} - R_s}{l_s}\right\} \tag{3.24}$$

Equation (3.24) presents the following weighted max-min fairness property. Each task T_i can be granted at least a w_i share of the resources. In addition, if $M_s - l_s$ tasks request less than their fair share, namely, only l_s tasks are adapted, then the free portion $\frac{(M_s - l_s)c}{M_s} - R_s$ can be distributed among those Target Tasks which are degraded and thus need these resources. The distribution can be done according to their static weights w_i, which identify their relative priority and importance.

This concludes the proof. □

3.4.2 Stability Analysis

The concept of *stability* has a two-fold meaning. First, in an environment of multiple target tasks simultaneously sharing the limited availability of resources, the ensemble of the adaptation activities in all tasks needs to be *stable*, which means that when the number of active target tasks is fixed, system resources allocated to each Target Task settle down to an equilibrium value in a definite period of time. This definition also implies that, if a new task becomes active, existing active tasks will adjust their resource usage so that after a brief transient period, the system settles down to a new equilibrium. Second, stability implies that with respect to variations in resource availability due to unpredictable and physical causes, for example a volatile wireless connection, adaptation activities do not suffer from oscillations. Oscillations are undesirable because they cause both fluctuations in user-perceptible qualities, and an excessive amount of adaptation attempts that may occupy too much resource to overload the system.

In order to converge to the equilibrium of the system regardless of disturbances and statistical multiplexing, we need to prove that the system is stable. Due to the nonlinear nature of the system given by the Equations (3.9) and (3.12), we are unable to derive a global and absolute stability condition, which is the case for most systems with nonlinear properties. However, formal conditions for local asymptotic stability can be addressed analytically. We present the following theorem related to local asymptotic stability conditions.

Theorem 2: The adaptation system established by the Equations (3.9) and (3.12) is asymptotically stable for the task T_i around a local neighborhood, under the condition that $\alpha > 0,\ \beta > 0$, and $\alpha + 2\beta < 4\gamma_i$.

Proof: Given the states defined in the Equation (3.16), we define

$$e(k) = x^c(k) - x(k) \tag{3.25}$$

$$\hat{u}_i(k) = \gamma_i \; l(k) \; [\bar{u}(k) - \frac{c - R(k)}{l(k)}] \tag{3.26}$$

In order to examine the asymptotic stability properties, we simplify the dynamic equations (3.9) and (3.12) in the neighborhood of equilibrium by: (1) removing the nonlinearities introduced by $\Psi_t(x)$ at both thresholds; (2) treating $l(k)$ and $R(k)$ as constants in the neighborhood of the equilibrium. Thus, Equations (3.9) and (3.12) become:

$$\hat{u}_i(k) = \hat{u}_i(k-1) + \alpha \; e(k) + \beta \; [e(k) - e(k-1)] \tag{3.27}$$

$$x(k) = x(k-1) + \frac{1}{\gamma_i}\hat{u}(k-1) \tag{3.28}$$

We perform *z*-transform on Difference Equations (3.27) and (3.28) to obtain $D_i(z)$ and $G_i(z)$, respectively, as follows:

$$\begin{aligned} \hat{u}_i(z) &= z^{-1}\hat{u}_i(z) + \alpha e(z) + \beta[e(z) - z^{-1}e(z)] \\ (1 - z^{-1})\hat{u}_i(z) &= (\alpha + \beta - \beta z^{-1})e(z) \\ D_i(z) = \frac{\hat{u}_i(z)}{e(z)} &= \frac{\alpha + \beta - \beta z^{-1}}{1 - z^{-1}} = \frac{(\alpha + \beta)z - \beta}{z - 1} \end{aligned} \tag{3.29}$$

$$\begin{aligned} x(z) &= z^{-1}x(z) + \frac{1}{\gamma_i}z^{-1}\hat{u}_i(z) \\ (1 - z^{-1})x(z) &= \frac{1}{\gamma_i}z^{-1}u(z) \\ G_i(z) = \frac{x(z)}{\hat{u}_i(z)} &= \frac{z^{-1}}{\gamma_i(1 - z^{-1})} = \frac{1}{\gamma_i(z - 1)} \end{aligned} \tag{3.30}$$

Thus, the transfer function $F_i(z)$ of the entire system is [16]:

$$F_i(z) = \frac{D_i(z)G_i(z)}{1 + D_i(z)G_i(z)} = \frac{\frac{1}{\gamma_i}[(\alpha + \beta)z - \beta]}{z^2 + (\frac{\alpha}{\gamma_i} + \frac{\beta}{\gamma_i} - 2)z - (\frac{\beta}{\gamma_i} - 1)} \tag{3.31}$$

We then consider the discrete characteristic equation of the above:

$$z^2 + (\frac{\alpha}{\gamma_i} + \frac{\beta}{\gamma_i} - 2)z - (\frac{\beta}{\gamma_i} - 1) = 0 \tag{3.32}$$

According to theorems in the digital control theory [16], in order for the system to be stable, all roots of the Equation (3.32) need to be within the stability boundary, which is the unit circle. In other words, for any root z, we need $|z| < 1$. We thus derive z as follows:

$$\begin{aligned} z &= \frac{(2 - \frac{\alpha}{\gamma_i} - \frac{\beta}{\gamma_i}) \pm \sqrt{(\frac{\alpha}{\gamma_i} + \frac{\beta}{\gamma_i} - 2)^2 - 4(1 - \frac{\beta}{\gamma_i})}}{2} && (3.33) \\ &= \frac{2 - (\frac{\alpha}{\gamma_i} + \frac{\beta}{\gamma_i}) \pm \sqrt{(\frac{\alpha}{\gamma_i} + \frac{\beta}{\gamma_i})^2 - 4\frac{\beta}{\gamma_i})}}{2} && (3.34) \end{aligned}$$

Let $d = \frac{\alpha}{\gamma_i} + \frac{\beta}{\gamma_i}$ and $f = \frac{\alpha}{\gamma_i}$, we have

$$\begin{aligned} z &= \frac{2 - d \pm \sqrt{d^2 - 4f}}{2} && (3.35) \\ &= 1 + \frac{-d \pm \sqrt{d^2 - 4f}}{2} && (3.36) \end{aligned}$$

If stability is desired, let $|z| < 1$, which is equivalent to

$$|1 + \frac{-d \pm \sqrt{d^2 - 4f}}{2}| < 1 \tag{3.37}$$

We divide the proof into two cases.

1. When $d^2 - 4f > 0$, that is $f < \frac{d^2}{4}$, real roots exist. Corresponding to the Equation (3.37), we thus have

$$-2 < \frac{-d \pm \sqrt{d^2 - 4f}}{2} < 0 \tag{3.38}$$

Which leads to

$$
\begin{aligned}
-2 < \frac{-d-\sqrt{d^2-4f}}{2} \text{ and } \frac{-d+\sqrt{d^2-4f}}{2} < 0 \\
d+\sqrt{d^2-4f}-4<0 \text{ and } -d+\sqrt{d^2-4f}<0 \\
d-4<0 \text{ and } d^2-4f<(4-d)^2 \text{ and } d>0 \text{ and } d^2-4f<d^2 \\
d<4 \text{ and } 2d-4<f \text{ and } d>0 \text{ and } f>0 \\
0<d<4 \text{ and } \max\{0,2d-4\}<f<\frac{d^2}{4} \\
0<\frac{\alpha}{\gamma_i}+\frac{\beta}{\gamma_i}<4 \text{ and } \max\{0,2(\frac{\alpha}{\gamma_i}+\frac{\beta}{\gamma_i})-4\}<\frac{\alpha}{\gamma_i}<\frac{(\frac{\alpha}{\gamma_i}+\frac{\beta}{\gamma_i})^2}{4} \\
\alpha>0 \text{ and } \beta>0 \text{ and } 0<\frac{\alpha}{\gamma_i}+2\frac{\beta}{\gamma_i}<4 \text{ and } 4\frac{\alpha}{\gamma_i}<(\frac{\alpha}{\gamma_i}+\frac{\beta}{\gamma_i})^2
\end{aligned} \tag{3.39}
$$

2. When $d^2-4f<0$, imaginary roots exist. We thus have:

$$
\begin{aligned}
|\frac{2-d}{2}+\frac{\sqrt{d^2-4f}}{2}| &< 1 \\
(\frac{2-d}{2})^2+(\frac{\sqrt{-(d^2-4f)}}{2})^2 &< 1 \\
\frac{4-4d+d^2}{4}+\frac{-d^2+4f}{4} &< 1 \\
\frac{4-4d+4f}{4} &< 1 \\
f &< d
\end{aligned} \tag{3.40}
$$

We thus have:

$$
\frac{d^2}{4}<f<d \text{ and } 0<d<4
$$

$$0 < \frac{\alpha}{\gamma_i} + \frac{\beta}{\gamma_i} < 4 \text{ and } \frac{(\frac{\alpha}{\gamma_i} + \frac{\beta}{\gamma_i})^2}{4} < \frac{\alpha}{\gamma_i} < \frac{\alpha}{\gamma_i} + \frac{\beta}{\gamma_i}$$
$$0 < \frac{\alpha}{\gamma_i} + 2\frac{\beta}{\gamma_i} < 4 \text{ and } \alpha > 0 \text{ and } \beta > 0 \text{ and } 4\frac{\alpha}{\gamma_i} > (\frac{\alpha}{\gamma_i} + \frac{\beta}{\gamma_i})^2 \quad (3.41)$$

If we merge the results from the Equations (3.39) and (3.41), we have:

$$\alpha > 0,\ \beta > 0,\ \text{ and } \alpha + 2\beta < 4\gamma_i \quad (3.42)$$

Equation (3.42) concludes the proof. □

It is obvious to see from Theorem 2 that the asymptotic stability of the adaptation for task T_i is determined by an appropriate choice of α and β. It then follows that in order to guarantee that the entire system is stable, we need to choose α and β so that for any task T_i with any static weight values w_i, stability is ensured.

Corollary: There exist appropriate values of parameters α and β so that all the tasks in the system are stable, for any pre-determined static weight w_i for task T_i.

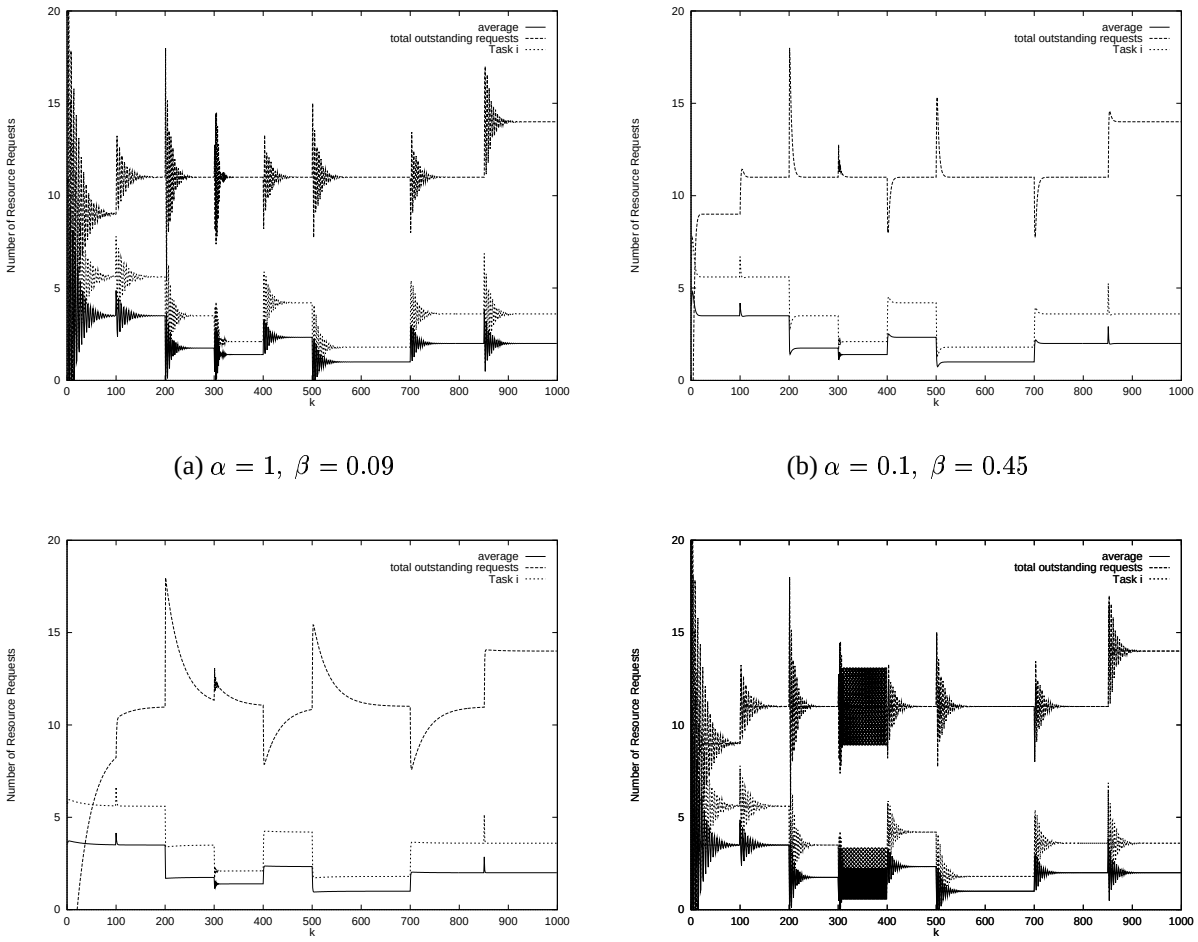
Proof: Assume w_{min} is the minimum value among all pre-determined w_i. α and β can be chosen to satisfy

$$\alpha > 0,\ \beta > 0,\ \text{ and } \alpha + 2\beta < 4\ \frac{w_{min}}{\sum_{\forall i} w_i} < 4\ \frac{w_i}{\sum_{T_j \in A(k)} w_j},\ \forall i,\ k \quad (3.43)$$

It follows from Theorem 2 that if these conditions hold, the system will be stable for any task T_i with a static weight w_i. □

3.4.3 Configuration of Adaptation Agility

In addition to stability requirements, it is also desired that the system responds quickly to changes in both resource availability and QoS requirements of the application tasks. First introduced in the Odyssey project [6], *adaptation agility* is defined as the speed and accuracy of an adaptive system with respect to detecting and responding to changes in resource availability and



(a) $\alpha = 1,\ \beta = 0.09$

(b) $\alpha = 0.1,\ \beta = 0.45$

(c) $\alpha = 0.015,\ \beta = 0.5$

(d) $\alpha = 0.7,\ \beta = 0.25$

Figure 3.4: Illustrations of Configurable Agility and Dynamic Responses

application QoS requirements. In short, it is the *responsiveness*, or the *sensitivity*, of an adaptive system to dynamic variations. This notion is similar to the notion of sensitivity introduced in [15], though the work focuses on ABR flow control algorithms in ATM networks.

It is desirable for the system to configure adaptation agility in each adaptor corresponding to different types of system resources. For example, a network-centric system (such as a web server) may be much more sensitive to changes in network bandwidth than to changes in CPU availability.

In our Task Control Model, adaptation agility is determined by the configurable parameters in the control algorithm. In the specific case of a PID control algorithm in the Equation (3.9), α and β are configurable as long as the stability conditions in Theorem 2 hold. The actual configuration

is tailored to the needs of the system.

Four illustrations are given in Figure 3.4(a) - 3.4(d) to show the effects of different configurations on the system. In all four graphs, we simulate the system established by Equation (3.9) and (3.12) for 1000 time intervals. The changes of $l(k)$, $R(k)$ and $x^c(k)$ are given in Table 3.1:

Time k	*0-100*	*100-200*	*200-300*	*300-400*	*400-500*	*500-700*	*700-850*	*850-1000*
$l(k)$	2	2	4	5	3	3	3	3
$R(k)$	3	3	3	3	3	7	4	4
$x^c(k)$	9	11	11	11	11	11	11	14

Table 3.1: The values of $l(k)$, $R(k)$ and $x^c(k)$ used in the illustration

It can be seen from these illustrations that the dynamics of the system is affected significantly by different configurations of the adaptor. In Figure 3.4(d), it even starts to oscillate in the interval $[300, 400]$. The configuration in Figure 3.4(b) reaches equilibrium much faster than Figure 3.4(c), and both of them do not show the oscillating transient response shown in Figure 3.4(a). Overall, the illustrations show that the adaptor is configurable according to the needs of the system. For the PID control algorithm in Equation (3.9), there are two parameters, α and β, to configure. However, for more complex control algorithms, there will be more dimensions for finer tuning of the transient responses.

3.5 Summary

Our work in this chapter has made two major contributions. First, we established the Task Control Model based on the Task Flow Model presented in Chapter 2 and rigorously defined the mapping between the classic linear control systems and the Task Control Model. Second, we utilized the PID control algorithm in the control theory, applied the algorithm in the Task Control Model, and analyzed fairness properties, asymptotic stability conditions, and adaptation agility of the adaptation behavior. Obviously, control theory itself is not new; our contribution is to propose a model that successfully applies the control-theoretical methodology to the practice of modeling application-aware QoS adaptations, so that from a systems point of view, stability of adaptation in

any active applications are guaranteed, agility can be tuned, and application concurrency is fairly promoted with respect to QoS adaptations.

Chapter 4

Observations in Distributed Adaptation Control

4.1 Overview

In the first tier of the *Agilos* architecture, in order to adjust the application appropriately, and to decide when, how and to what extent for the application to adapt, accurate identification of current system states is critical, based on all or a subset of observable parameters. As noted in Section 2.2, this *observe and control* process forms a closed control loop, which is the nature of a control system. In digital control theory, the control signals in such a process are determined by a *controller*, based on the current state estimates. In Chapter 3, we developed a Task Control Model that takes advantage of the control theory to model this process, and gave theoretical results to prove stability and fairness properties in the model. Such a design is implemented in the adaptors within the *Agilos* middleware. The objective of this approach is to optimally adjust the internal parameters and semantics of flexible applications, so that their adaptive behavior conforms to system wide properties, such as fairness among all applications sharing the same pool of end system resources.

In addition to a rigorous definition of the mathematical model for the target task, which has been elaborated in Chapter 3, it has been noted that the effectiveness of a control algorithm executed in the adaptor depends on accurate observations of system states. For the states associated with some of the end system resources such as CPU availability, an accurate observation may be easily

achieved. However, associated with resources in a distributed environment with the presence of end-to-end delays, such system states are not directly observable in the end points of the network, thus they need to be estimated. A typical example that applies to the *OmniTrack* application is the throughput (in terms of volume per time unit) that data is transmitted among end systems. In such a scenario, in order to control the application and dynamically adjust the data rate transmitted from server to client, we need to obtain system states such as quantities of data in transit in the networks or arrived at the client. Significant end-to-end delay may obstruct the desired accurate observations, when estimations are used instead.

The chapter presents the design of an optimal state estimation mechanism [17] executed in the observers of *Agilos*, that estimates data throughput in a distributed application. The key contributions of this mechanism are the following. (1) *An extended Distributed Task Control Model*: The Task Control Model is extended from a model focusing on local resources such as CPU, to a distributed model focusing on bandwidth availability in transmission tasks. Since it is impossible to observe network traffic sharing the same links, the tradeoff is that the fairness property that was previously proved can no longer be guaranteed. (2) *A linear model for transmission tasks*: To characterize the transmission tasks in a distributed application, we develop a linear model with concrete coefficients, on which state estimation techniques are based. (3) *Optimal state prediction mechanisms*: Accurate control signals are based on precise estimates of system states. With the presence of significant end-to-end delay that obstructs accurate state observations, we present an optimal prediction approach to estimate current system states based on available observations in history. We adopt optimal estimation theories such as the Kalman Filter in our approach. Assisted by optimal state prediction techniques, the application can be appropriately controlled to adapt to dynamic variations.

4.2 Modeling Transmission Tasks

As introduced in Chapter 2, we view each application as a series of connected *tasks*, with each application task as a concrete component that performs operations on the input, generates output and consumes resources. With such a view of an application, a distributed application has one or more *transmission tasks* in its Task Flow Graph, which transmits application data between two end systems. We observe the current system states within the observer, such as available bandwidth in the transmission task. These observations are delivered to the adaptor, which calculates control signals according to the control policy. Similar to Figure 2.1, the distributed view is shown in Figure 4.1 in the example of a client-server based application.

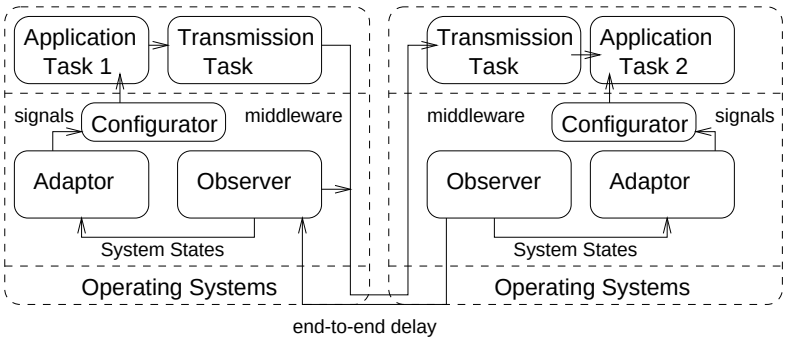


Figure 4.1: An Distributed View of the *Agilos* Middleware Architecture

Presented in Chapter 3, the customized control algorithm in the adaptor is the outcome of applying the control theory to the theoretical analysis of adaptation behavior. In such an algorithm, we were able to prove that, if priority weights are given to each application task, the system fairly allocates resources among competing tasks according to the weighted max-min fairness property. We also proved that the system converges to the equilibrium, and stability of control is preserved around a local neighborhood. In this chapter, with an appropriate model for the transmission task, we can extend the model to apply to a distributed environment, with the presence of significant end-to-end delays.

4.2.1 State Observation in the Transmission Task

The accuracy of control signals calculated by the adaptor relies on precise observations of system states. However, in the distributed environment where observing system states in a transmission task is necessary, end-to-end propagation delay poses serious difficulties to observe and capture such information.

An important state to observe is available bandwidth within the transmission task T_i, with i being an index in the series of tasks within the application. We take a client-server application as an example, and assume that the observer located on the client can observe the number of received data units[1] during the time $[k, k+1]$ (k being discrete time instants), $z_i(k)$. In reality, we assume that $y_i(k)$ is the number of data units actually received during $[k, k+1]$. At the server side, the actual number of data units sent by T_i during $[k, k+1]$, denoted by $u_i(k)$, is controlled by the adaptor. Finally, $x_i(k)$ is the number of data units *in flight* in the transmission task T_i. Note that the transmission task T_i itself is distributed at both client and server side, therefore, both $x_i(k)$ and $y_i(k)$ are internal states in T_i, while $y_i(k)$ is also the output of T_i. The above scenario is illustrated in Figure 4.2.

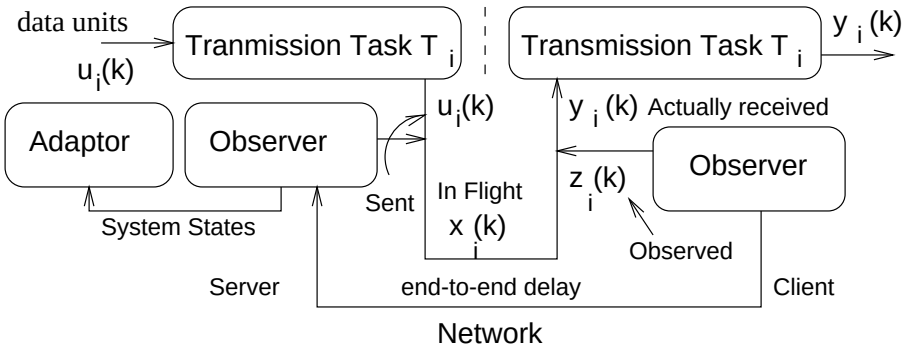


Figure 4.2: States in the transmission task

The challenge is the following. Since the *Agilos* middleware architecture is deployed separately in each of the end systems, if end-to-end delays are present, at any particular time instant k, the server observer can only obtain previously observed states by the client observer, with the lag

[1]Data units are defined based on application-specific semantics. For example, in a video-on-demand application, a data unit may be defined as a video frame.

equivalent to the end-to-end delay. This calls for an estimation algorithm in the server observer to compensate the observation error, and to predict the states for the current time instant k.

4.2.2 Linear Model for the Transmission Task

As a preparation for later applications of analytical techniques in the optimal estimation theory to estimate system states in the distributed transmission task, we present a precise analytical model to characterize the internal dynamics of the transmission task T_i.

Review of the Generic Target Task

As previously presented in Section 3.3.1, in order to control any target task, we identify several key parameters in this task, referred to as *task states*. When using the vector $\mathbf{x}$ to denote task states, $\mathbf{u}$ to denote input to the task, $\mathbf{y}$ to denote task output, $\mathbf{w}$ to denote system noise within the task, $\mathbf{z}$ to denote observation, and $\mathbf{v}$ to denote observation error, we examine a linear and discrete-time model described by the following form, repeating the model we presented in the Equations (3.4), (3.5) and (3.6):

$$\mathbf{x}(k) = \mathbf{\Phi}\mathbf{x}(k-1) + \mathbf{\Gamma}\mathbf{u}(k-1) + \mathbf{w}(k-1) \tag{4.1}$$

$$\mathbf{y}(k) = \mathbf{H}\mathbf{x}(k) \tag{4.2}$$

$$\mathbf{z}(k) = \mathbf{y}(k) + \mathbf{v}(k) \tag{4.3}$$

where $k = 1$ to k_{max}, and $\mathbf{\Phi}$, $\mathbf{\Gamma}$, and $\mathbf{H}$ are known transition matrices without an error. In later discussions, we develop a concrete analytical model for the transmission task in a distributed environment, based on the above abstract model.

Concrete Model for the Transmission Task

In order to develop a concrete model for the transmission task T_i in a client-served based distributed application, we consider two types of noise in the system. First, the data units in transit in the network from server to client, $x_i(k)$, may suffer from random and unpredictable variations and disturbances $w_i^x(k)$, caused either by physical unstable conditions (in the case of wireless links) or statistical multiplexing of network connections. Consequently, the received quantity of data units during $[k, k+1]$, $y_i(k)$, may also suffer from random disturbances $w_i^y(k)$. These are obviously *system noises* caused by the external dynamics in the transmission task. Second, the observer itself is also subject to random errors, which can be characterized as the *observation noise* $v_i(k)$. Assume that the observed value is $z_i(k)$, we have

$$z_i(k) = y_i(k) + v_i(k) \tag{4.4}$$

If we compare the Equation (4.4) to the Equation (4.1), we notice that $\mathbf{u}(k)$ is actually a scalar $u_i(k)$, $\mathbf{v}(k)$ is a scalar $v_i(k)$. and $\mathbf{y}(k)$ is a scalar $y_i(k)$. From the Equation (4.2), we have $y_i(k) = \mathbf{H}\mathbf{x}(k)$. Since $\mathbf{x}(k)$ needs to contain $x_i(k)$, we assign

$$\mathbf{H} = [1 \quad 0] \text{ , } \mathbf{x}(k) = \begin{bmatrix} y_i(k) \\ x_i(k) \end{bmatrix} \tag{4.5}$$

In addition, from the definition of $x_i(k)$, $y_i(k)$, $w_i^x(k)$ and $w_i^y(k)$, we have

$$x_i(k) = x_i(k-1) - y_i(k-1) + u_i(k-1) + w_i^x(k-1) \tag{4.6}$$

$$y_i(k) = y_i(k-1) + w_i^y(k-1) \tag{4.7}$$

It follows from Equations (4.6), (4.7), (4.1) and (4.5) that

$$\mathbf{\Phi} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}, \mathbf{\Gamma} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ and } \mathbf{w}(k) = \begin{bmatrix} w_i^y(k) \\ w_i^x(k) \end{bmatrix} \tag{4.8}$$

This concludes the complete state space representation of the linear model for transmission task T_i.

4.2.3 Extending Control Algorithms to Distributed Environment

In Chapter 3, a customized PID control algorithm was given to model the adaptor, and a weighted max-min fairness property was proved. A prerequisite for the fairness property to hold is that the observer has the ability to observe complete global system states. For example, if the resource being observed is CPU usage in the same end system, global states can be observed for all competing tasks.

However, this is normally not the case in a distributed environment, when observing task states within the transmission task. Since the observers act as *Agilos* components in the end system, they do not have the ability to obtain states corresponding to all other connections sharing the network. In such cases, the only observable states are the parameters used or allocated by the transmission task itself, such as occupied bandwidth. Therefore, while the control algorithm still adapts to variations in resource availability and shows stability and convergence properties, it lacks crucial observations to guarantee any global fairness properties.

In distributed applications, the PID control algorithm adopted in the adaptor may be modified as follows:

$$\begin{aligned} u_i(k) = u_i(k-1) + \alpha[x_i^c(k) - x_i(k)] + \\ \beta\{[x_i^c(k) - x_i(k)] - [x_i^c(k-1) - x_i(k-1)]\} \end{aligned} \tag{4.9}$$

where x_i^c is the reference value expected at equilibrium, $u_i(k)$ is the actual number of data

units sent by T_i, $x_i(k)$ is the number of data units in transit from server to client, and α and β are configurable scaling factors. The stability and convergence proofs still hold as in the previous chapter.

However, with the presence of end-to-end delays, it is inherently not trivial to accurately estimate $x_i(k)$ directly at the server, since the number of data units in transit in $[k, k+1]$ is not directly observable.[2] $y_i(k)$, the number of data units received, is directly observable, but *only* at the client side. Therefore, at the server side, the available values for $u_i(k)$ computation in the control algorithm (Equation (4.9)) are imprecise, as the $y_i(k)$ values (needed for $x_i(k)$ computation) previously observed by the client are received by the server only after an end-to-end delay from the time of observation. This leads us to the following approach. Instead of deriving $y_i(k)$ using only the available observed values transmitted from the client to the server with an end-to-end delay, we will adopt optimal state prediction techniques to estimate $y_i(k)$ at the current time instant, which forms discussions in the next section.

4.3 Optimal Prediction of Task States In Transmission Tasks

In this section, we present an optimal prediction approach to optimally predict the current task states in the transmission task, based on observed task states in previous time instants before the end-to-end delay. The optimal prediction algorithms are implemented in the server observer, while the actual observation is made in the client observer. Optimality in the prediction algorithms guarantees that the relative error between the prediction and actual values of task states is minimized, i.e., a *best possible guess* is obtained. We adopt the optimal control and estimation theory [18] to develop the proposed algorithms, and associate the theoretical solutions with the practical cases in complex distributed applications, focusing on the transmission task.

[2] Equation (4.6) is part of the linear model of the transmission task, but it can not be easily utilized for the estimation of $x_i(k)$ since it is not observable directly.

4.3.1 The Need for Prediction

It is obvious from Equation (4.4) that the client observer is able to observe $y_i(k)$ as $z_i(k)$, with an observation noise $v_i(k)$. However, from the control algorithm expressed in Equation (4.9), we note that $x_i(k)$ is actually used in the adaptor. In order to derive $x_i(k)$ on the server from the observed values $z_i(k)$ on the client, we assume that the client acknowledges all received data units to the server, and that the server observer has the knowledge of the total number of data units *unacknowledged* at the server up to the time instant k, denoted by $y_i^s(k)$. Then, we have $z_i^s(k) = y_i^s(k) + v_i^s(k)$ as the observed values of $y_i^s(k)$ with an observation noise $v_i^s(k)$. Naturally, $y_i^s(k)$ represents the total number of unacknowledged data units which are either in flight from server to the client, which is $x_i(k)$, or received by the client, but acknowledgments not yet received by the server. We thus have

$$x_i(k) = y_i^s(k) - \sum_{t=k-\lceil \frac{d_i}{t_c} \rceil}^{k-1} y_i(t) \tag{4.10}$$

where d_i is the end-to-end transmission delay from client to server, t_c is the sampling time interval between $[k-1, k]$, assuming $d_i \geq t_c$. Ideally, if $y_i(t), \forall t \in [k - \lceil d_i/t_c \rceil, k-1]$ is known, $x_i(k)$ can be computed and then used in the control algorithm of Equation (4.9). However, the end-to-end delay, represented by d_i, prevents the knowledge of $y_i(t),\ \forall t \in [k - \lceil d_i/t_c \rceil, k-1]$. The last available observation is $z_i(k - \lceil d_i/t_c \rceil)$. The need of predicting these values of $y_i(t)$ in the server observer before calculating $x_i(k)$ arises from this lack of knowledge. Figure 4.3 illustrates the above scenario.

We use $\hat{y}_i(k)$ to denote the predicted values of $y_i(k)$. Assuming that $\hat{y}_i(k)$ is already obtained optimally, we can estimate $x_i(k)$ by the following Equation:

$$x_i(k) = z_i^s(k) - \sum_{t=k-\lceil \frac{d_i}{t_c} \rceil}^{k-1} \hat{y}_i(t) \tag{4.11}$$

The problem then shifts to the development of appropriate mechanisms to obtain $\hat{y}_i(k)$.

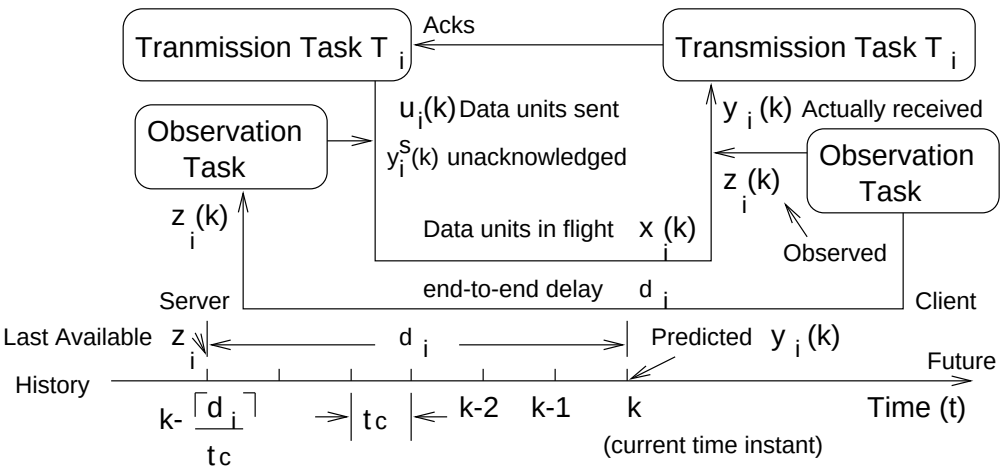


Figure 4.3: State Prediction in transmission tasks

4.3.2 Mechanisms for Optimal Prediction

Definition of Optimality

Based on the Separation Principle [18], for a linear stochastic system where an observer is used to estimate the system state, the parameters for the observer and controller are determined separately. Informally, this means that we can develop an optimal prediction algorithm for $y_i(t), \forall t \in [k - \lceil \frac{d_i}{t_c} \rceil, k-1]$ in the server observer, while still retaining complete freedom for adopting alternative control algorithms in the server adaptor.

With regards to the prediction accuracy, we prefer to design an *optimal* prediction algorithm that minimizes the sum of squared errors between the predictions and values being estimated, i.e., a least-squares estimate. More precisely, if $\epsilon(k) = y_i(k) - \hat{y}_i(k)$, where $\hat{y}_i(k)$ is the predicted values of $y_i(k)$ at time k, we try to minimize the quadratic error cost function $J(\mathbf{y}) = J(y_i(k)) = \frac{1}{2}\epsilon^2(k)$. The optimal prediction approach, e.g., Kalman Filter, presented in this section is designed to minimize $J(\mathbf{y})$.

Requirements of an Optimal Solution

The optimal prediction problem is generally hard if the linear stochastic system is in its generic form. However, it is proved in optimal control theory [19] that simplified prediction algorithms

can be adopted as an optimal solution in a special case, with two prerequisites. First, the system random disturbances $\mathbf{w}(k)$ and observation noises $\mathbf{v}(k)$ are uncorrelated white Gaussian-Markov sequences with zero mean. This can be interpreted that: (1) random vectors in the same stochastic sequence are independent of each other; (2) they can be uniquely characterized by a joint Gaussian probability density function; (3) this density function has zero mean expectation, and (4) random vectors in different stochastic sequences are uncorrelated with each other. Second, the initial system state vector $\mathbf{x}(0)$ is also a Gaussian random vector with zero means.

We assert that the system states and noises in the transmission task T_i observe such nature. This assertion is based on the following characteristics.

(1) The states in the observer and the transmission task are not correlated, since the observers are implemented separately in the middleware level, while the transmission task is part of the application. This observation guarantees that the observation noise $\mathbf{v}(k)$ and the system noise $\mathbf{w}(k)$ are uncorrelated.

(2) Within the transmission path, when the number of simultaneous connections N sharing the same physical communication channel (statistical multiplexing in intermediate switches) is large, we expect the changes in N in t_c is very small compared to N. This leads to the fact that changes in $x_i(k)$ due to activities of other connections will be small. Thus, when we model $x_i(k)$ as a process given by Equation (4.6) $x_i(k) = x_i(k-1) - y_i(k-1) + u_i(k-1) + w_i^x(k-1)$, the term $w_i^x(k-1)$, which represents the dynamic disturbances caused by activities of other connections, can be modeled as a zero-mean Gaussian white noise [14]. Even though when $x_i(k)$ is small and the connection is in a starting stage, the possibility of an increase is larger than a decrease, this assumption of zero mean is justifiable when $x_i(k)$ is sufficiently far from 0. The same observation also applies to $y_i(k)$ and $w_i^y(k)$. This concludes that the random noise $\mathbf{w}_i(k)$ is a white Gaussian-Markov sequence with zero mean.

We conclude that random disturbances of the transmission task satisfy the requirements of applying the simplified prediction algorithms, such as the Kalman Filter prediction algorithm that follows.

Parameters in the Kalman Filter

We now apply the frequently used optimal estimation algorithm, Kalman Filter, to solve the prediction problem of task states in the transmission task.

Equation (4.8) shows that both $\mathbf{\Phi}(k-1)$ and $\mathbf{\Gamma}(k-1)$ in Equation (4.1) are constants without error. In addition, $\mathbf{u}(k-1)$ are also known in the server observer without error in the interval $0 \leq k \leq k_{max}$. We introduce the definition of the following terms:

(1) The *expected values*, or *expectations*, $E(\mathbf{x})$ of any random vector $\mathbf{x}$ is defined as the mean vector of $\mathbf{x}$. Formally, $E(\mathbf{x}) = [E(y_i(k)), E(x_i(k))]^T$, where $E(x)$ for a random variable x is defined as $E(x) = \int_{-\infty}^{\infty} xp(x)dx$, if $p(x)$ is the probability density function of x.

(2) The *error covariance matrix* $\mathbf{P}$ of $\mathbf{x}$ in the transmission task is defined as:

$$\mathbf{P}(k) = E[[\mathbf{x}(k) - \hat{\mathbf{x}}(k)][\mathbf{x}(k) - \hat{\mathbf{x}}(k)]^T] \tag{4.12}$$

(3) The dynamic system disturbance $\mathbf{w}$ is a white, zero-mean Gaussian random sequence showing the following properties, where $\mathbf{Q}(k)$ is the system noise covariance matrix:

$$E[\mathbf{w}(k)] = 0 \tag{4.13}$$

$$\mathbf{Q}(k) = E[\mathbf{w}(k)\mathbf{w}(k))^T] \tag{4.14}$$

$$E[\mathbf{w}(k)\mathbf{w}(j))^T] = 0, (j \neq k) \tag{4.15}$$

(4) Similarly, in Equation (4.2) and (4.3), $\mathbf{H}$ is a constant and the observation noise is modeled as a white, zero-mean Gaussian random sequence that is uncorrelated with the system disturbance:

$$E[\mathbf{v}(k)] = 0 \tag{4.16}$$

$$\mathbf{R}(k) = E[\mathbf{v}(k)\mathbf{v}(k))^T] \tag{4.17}$$

$$E[\mathbf{v}(k)\mathbf{v}(j))^T] = 0, (j \neq k) \tag{4.18}$$

$$E[\mathbf{n}(k)\mathbf{w}(j))^T] = 0(\text{all } j \text{ and } k) \tag{4.19}$$

where $\mathbf{R}(k)$ is the observation noise covariance matrix. According to Equation (4.5), $\mathbf{v}(k)$ is a scalar $v_i(k)$, it follows that $\mathbf{R}(k)$ is the variance of $v_i(k)$, $var(v_i(k)) = \sigma^2$, when $v_i(k)$ is a Gaussian distribution (m, σ).

In practice, it is necessary to determine $\mathbf{Q}(k)$ and $\mathbf{R}(k)$ offline. These covariance matrices indicate the level of confidence in the system model and observations, respectively. If one were to increase $\mathbf{Q}$, this would indicate that stronger noises are driving the dynamics. Consequently, the rate of growth of the elements of the error covariance matrix $\mathbf{P}(k)$ will also increase, which increases the filter gain $\mathbf{K}(k)$, thus weighing the measurements more heavily. Therefore, by increasing $\mathbf{Q}$, we in effect put less confidence in the system model. Similarly, increasing $\mathbf{R}$ indicates that the observations are subject to a stronger corruptive noise, and therefore should be weighed less by Kalman Filter.

Operations of Kalman Filter

Based on these definitions, Kalman Filter operates recursively in a *predict-update* manner. Informally, we may describe the operations in the following phases. Formal descriptions are postponed to the next section.

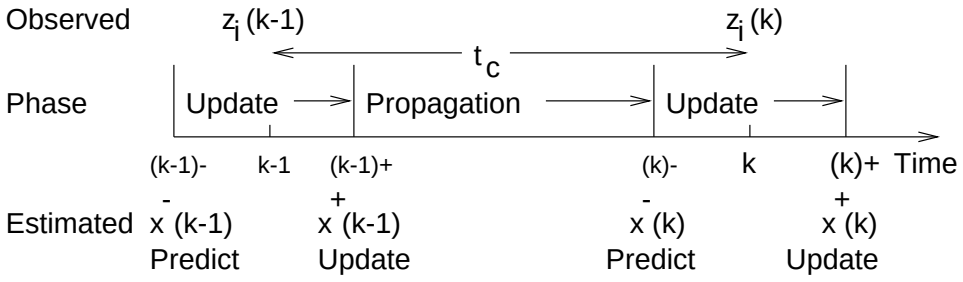


Figure 4.4: The Kalman Filter in Operation

(1) *Prediction Phase* occurs at time k^-, that is, before observations are made at time k. State predictions $\hat{\mathbf{x}}^-(k)$ are made for states $\mathbf{x}^-(k)$, and error covariance predictions $\mathbf{P}^-(k)$ is also made.

(2) *Kalman Filter Gain Computation Phase* occurs between k^- and k^+, where $k+$ is the time after k. The Kalman Filter gain matrix $\mathbf{K}(k)$ is computed to be used later in the Update Phase.

(3) *Update Phase* occurs at time k^+. The Kalman Filter gain matrix $\mathbf{K}(k)$ is used along with the new observation $\mathbf{z}(k)$. The error covariance matrix $\mathbf{P}^+(k)$ is also updated from previously predicted $\mathbf{P}^-(k)$ in the Prediction Phase.

These phases are executed repetitively till the time when the latest observation is available from the client observer. After this time instant, we can deploy a linear-optimal predictor to predict the state and its error covariance on the basis of all information that are available without observation. Denoting the time of latest available observation on the client as $k - \lceil \frac{d_i}{t_c} \rceil$ for transmission task T_i, where k is the present time instant on server, the linear-optimal predictor starts with the latest state estimate update phase using Kalman Filter, i.e., $\hat{\mathbf{x}}^+(k - \lceil \frac{d_i}{t_c} \rceil)$, and then recursively applies the state prediction phase to calculate $\hat{\mathbf{x}}(t) = \hat{\mathbf{x}}^-(t),\ \forall t \in [k - \lceil \frac{d_i}{t_c} \rceil, k-1]$. According to Equation (4.5), we have the following for transmission task T_i:

$$\hat{y}_i(t) = \mathbf{H}\hat{\mathbf{x}}(t),\ \forall t \in [k - \lceil \frac{d_i}{t_c} \rceil, k-1] \tag{4.20}$$

Equation (4.20) concludes our prediction mechanisms utilizing the Kalman Filter. When the estimated values of $y_i(k)$ are applied to Equation (4.10), $x_i(k)$ can be obtained and thus applied to the control algorithm in the adaptor as presented in Equation (4.9).

Formal Steps in the Kalman Filter

In the following equations, we distinguish between estimates made before and after the updates. $\hat{\mathbf{x}}^-(k)$ is the state estimate that results from the prediction equation (4.21) alone (i.e. *before* the observations are considered), and $\hat{\mathbf{x}}^+(k)$ is the corrected state estimate that accounts for the observation made. $\mathbf{P}^-(k)$ and $\mathbf{P}^+(k)$ are defined similarly. For completeness, the initial conditions

are $\hat{\mathbf{x}}^+(0)$ and $\mathbf{P}^+(0)$.

- State Estimate Prediction Phase:

$$\hat{\mathbf{x}}^-(k) = \mathbf{\Phi}\hat{\mathbf{x}}^+(k-1) + \mathbf{\Gamma}\mathbf{u}(k-1) \tag{4.21}$$

$$\mathbf{P}^-(k) = \mathbf{\Phi}\mathbf{P}^+(k-1)\mathbf{\Phi}^T + \mathbf{Q}(k-1) \tag{4.22}$$

- Kalman Filter Gain Computation Phase:

$$\mathbf{K}(k) = \mathbf{P}^-(k)\mathbf{H}^T[\mathbf{H}\mathbf{P}^-(k)\mathbf{H}^T + \mathbf{R}(k)]^{-1} \tag{4.23}$$

- Update Phase:

$$\hat{\mathbf{x}}^+(k) = \hat{\mathbf{x}}^-(k) + \mathbf{K}(k)[\mathbf{z}(k) - \mathbf{H}\hat{\mathbf{x}}^-(k)] \tag{4.24}$$

$$\mathbf{P}^+(k) = [\mathbf{P}^-(k)^{-1} + \mathbf{H}^T\mathbf{R}(k)^{-1}\mathbf{H}(k)]^{-1} \tag{4.25}$$

4.4 Summary

In this chapter, we focus on the development of an optimal estimation strategy when some of the system states are not observable, and must be estimated, due to end-to-end delays between end systems in a complex distributed application. We have extended the Task Control Model to the distributed environment, modeled the transmission task in a state-space representation, and presented an optimal state prediction mechanism to overcome end-to-end delay in distributed state observations. The optimal prediction mechanism proposed in this chapter is integrated in the observer, as a middleware component and part of the *Agilos* architecture in a larger scale.

Chapter 5

Dynamic Reconfigurations

5.1 Overview

Chapter 3 and 4 discussed control and estimation problems in the Task Control Model, focusing on the adaptor and observer in *Agilos*. The major contribution of the Task Control Model is that it addresses system-wide global adaptation properties, such as stability guarantees, adaptation agility and fairness. However, the issues related to the mapping between system-wide adaptation and application-specific adaptation choices still remain to be addressed. This chapter discusses the design of the configurator, located in the second tier of the *Agilos* architecture. We present the *Fuzzy Control Model* [20] in a top-down fashion, which forms the theoretical basis of designing both functional and quantitative configurators. The goal of the Fuzzy Control Model is to appropriately represent application-specific adaptation choices in an application-neutral processing model, so that it is flexible enough to be aware of the adaptive capabilities of a wide range of applications.

Within the scope of the Task Control Model, the configurator may be recognized as an extension to the "controller" component of a closed-loop control process. Before introducing the configurator, such a component is represented by the adaptor in *Agilos*. The role of the configurator is to bridge between the application-neutral output of the adaptors and the application-specific adaptation actions. Particularly, the configurator utilizes the Fuzzy Control Model to translate the control signals generated by the adaptors into actual adaptation choices, which are invoked in the application. Note that this translation mechanism differs and supersedes the QoS translation

between different categories of QoS parameters, in the sense that it translates the control actions, rather than parameter values. In Figure 5.1, we illustrate the role of configurator in the Task Control Model.

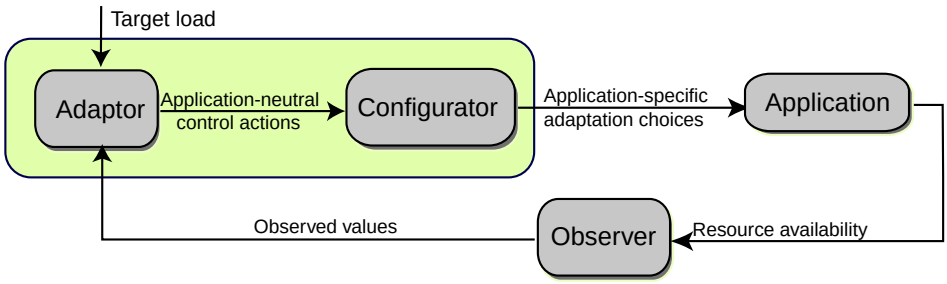


Figure 5.1: The Role of Configurator in the Task Control Model

The key contributions of this chapter are the following. (1) We present strong motivations to the introduction of the Fuzzy Control Model, by showing why fuzzy control theory is chosen rather than alternative models. (2) We show that our approach is feasible when dealing with non-linearities of different control choices, such as a hybrid combination of parameter-tuning actions and reconfiguration choices. These choices are naturally nonlinear and mostly discrete, while the rules that guide the decision-making process are mostly intuitive and heuristic. These rules are application-specific and determined manually with the assistance of QualProbes (presented in Chapter 6) to best fit the interests of the particular application. (3) We illustrate the application of fuzzy control theory in the design of the configurator, so that the adaptation choices and preferences for different applications can be expressed explicitly in the *rule base* and *member functions* for each linguistic value. The *rule base* provides linguistic rules that the *inference engine* is based on, and the *fuzzy inference engine* generates manipulating signals that control the actual application. (4) We show in details the internal works of the fuzzy inference engine, which implements the decision-making process that maps from application-neutral control signals to application-specific adaptation choices.

5.2 Motivations behind the Fuzzy Control Model

Similar to the role of the Task Control Model in the design of adaptors, we utilize the rich semantics and features in existing fuzzy logic and fuzzy control theory to design the configurators, which we refer to as the *Fuzzy Control Model*. In this section, we discuss the reasons that motivate the adoption of the Fuzzy Control Model.

5.2.1 Division between Application-Neutral and Application-Specific Components

The objective of designing the configurator is to design a mapping process, which translates from application-neutral control signals generated by the adaptors, to application-specific adaptation choices, including parameter-tuning and reconfiguration choices. Let us assume that we focus on one specific application, such as the *OmniTrack*. Intuitively, the ad-hoc approach is to hard-code a specific configurator that caters to the needs of this application only, and hand-tune the mapping process implemented by such a configurator. However, such an approach is obviously not flexible, since a different application requires re-implementing the configurator.

Consequently, a more flexible design is to extract the application-specific adaptation policies and specify them in a separate input file, while keeping the mapping process itself generic to all applications. The design of the Fuzzy Control Model follows this principle with a clean division between the generic fuzzy inference engine and the application-specific rule base, that "fuels" the fuzzy inference engine. With the fuzzy control theory, the inference engine precisely models the application-neutral mapping process, yet flexibility is offered by switching the rule base for different applications. The rule base is specified with a input file that the inference engine uses when the mapping process is activated.

The flexibility requirements in the design of the configurator rule out of the possibility of adopting some alternative approaches that are less flexible, such as using non-linear control theory. Since it is hard to specify the precise mathematical models for the diversely different target application

to be controlled, non-linear control theory is not feasible to be utilized to design our configurator.

5.2.2 Advantages of the Fuzzy Control Model

The advantages of adopting the Fuzzy Control Model are the following:

1. Taken the fact that multiple reconfiguration options and parameter-tuning possibilities exist in a typical complex application, the controllable regions and variables within the application are in most cases discrete, non-linear and complex. In these applications, the model for the target task is nonlinear in nature. On the other hand, a control system based on fuzzy logic can be conceived as a nonlinear control system, in which the relationships between controller inputs and outputs are expressed by using a small number of *linguistic rules* stored in a *rule base*. The nonlinearity of the fuzzy controller matches naturally with the inherent nonlinearities with respect to controllable regions and adaptation possibilities within an application. Such a fuzzy controller is the foundation of the design of configurators.

   The reason why we have chosen to use fuzzy logic rules rather than a plain rule-based system is as follows. First, the inference process for a plain rule-based system is based on binary logic, which is a subset of fuzzy logic. We believe that the fuzzy logic theory is able to provide better facilities to express richer semantics and a more complicated mapping process. Second, a plain rule-based system is only able to react on comparisons with threshold values, which eliminates the need for membership functions. Threshold values are sufficient if there are only a few adaptation alternatives to choose from. However, if there are many alternatives and each have overlapping resource needs, attempting to make correct adaptation decisions by only threshold values is hard. On the contrary, fuzzy rules, along with the fuzzy inference process, are able to provide a natural solution to such a highly complex scenario. Finally, if the membership functions are specified as simple shapes such as trapezoids or triangles, a system based on fuzzy rules presents equivalent performance overhead with respect to the mapping process, if compared to a plain rule-based system.

2. The Fuzzy Control Model is inherently generic and highly configurable. Both the rule base and the definition of membership functions for linguistic values can be configured to be application-specific. The Fuzzy Control Model offers a common design for configurators suitable for all applications, without loss of generality and configurability.

3. The Fuzzy Control Model includes the fuzzy inference engine (with its linguistic rule base), which represents the decision-making process and resembles natural human communication and reasoning. For this reason, it is natural and straightforward for the application to specify its own adaptation preferences and decisions in the form of linguistic values and rules. The merits of the simplicity, however, do not affect the flexibility and power of fuzzy control systems to define the most complicated nonlinear multiple-dimensional control surface.

4. Nowadays, we are faced with applications that have increasingly complex adaptation mechanisms and behavior, for which different modeling representations (e.g., piecewise linear models, radial-based function models) may be difficult to obtain. With an appropriately defined rule base, the Fuzzy Control Model may lead to models that describe the adaptation behavior of applications sufficiently well. Thus, such a fuzzy modeling approach may turn out to be a useful complement to traditional modeling and control approaches, such as those used in adaptors, when both the complexity and uncertainty about the application increases.

5. The Fuzzy Control Model is able to quickly express the control structure of a system using *a priori* knowledge, and to depend less on the availability of a precise model of the target task being controlled. This is of great practical significance, since precise modeling is usually the bottleneck for the application of effective model-based control systems. In most cases, the rule base in the Fuzzy Control Model leads to compact descriptions of application-level adaptation behavior, as well as natural handling of any inherent nonlinearities in the closed control loop. Without the assistance of such a fuzzy modeling method, implementation of such a controller could be difficult, if not impossible.

Both quantitative and functional configurators are designed with the Fuzzy Control Model. The only difference is that they have a different defuzzification process focusing the parameter-tuning actions and reconfigurations, respectively. We discuss such a minor design difference in Section 5.3.4. The user configurator does not enlist the assistance of the Fuzzy Control Model. Instead, it presents a user interface so that the user may interact with the application, in order to activate reconfigurations and tune the parameters dynamically during application runtime.

5.2.3 A Comparison with Other Alternative Approaches

There are other alternative models that the configurator may take advantage of in its design. In this section, we compare the Fuzzy Control Model with three examples of alternative models in details. Such comparisons strongly motivate the selection of the Fuzzy Control Model.

Threshold Values

The most straightforward approach of the mapping process is to activate certain adaptation choices whenever the output of resource adaptors increases or decreases beyond certain *threshold values*. This is the approach adopted by many previous research work [6] [21]. Essentially, this model resembles the behavior of step functions, in the sense that reconfiguration options are either turned on or off based on a comparison with threshold values. The obvious advantage of this model is that the mapping process is easy to implement when there are only one or a few application-specific parameters to adapt. The disadvantage is that this model lacks the expressive power to model a complicated mapping process that is beyond a clear-cut solution. In fact, if we specify the membership functions in the Fuzzy Control Model as step functions that the only possible values are 0 and 1, they are identical with the specification of threshold values. This explains that the Fuzzy Control Model is a superset and it is capable in modeling a more complex mapping process.

Discrete-Parameter Markov Chains Model

An alternative approach is to use the discrete-parameter Markov chains model. Each individual state in the Markov chain is used to model the different execution or fidelity levels within the application. A transition matrix that contains single-step transition probabilities. These transition probabilities are capable of expressing a more complicated conversion process and transitional conditions. However, there are two intrinsic drawbacks to this approach. First, the specification of all the conditional probability values in the transition matrix is inherently hard, since they may not be intuitively derived from the adaptation policies. In comparison, the specification of fuzzy rules are straightforward since the linguistic values correspond naturally to the semantics of the application QoS parameters. Although the specification of membership functions are less straightforward, the *QualProbes*, which are introduced in Chapter 6, provide important insights into the adaptive behavior of the applications, and are able to significantly help such a design process. Second, since each state in the Markov chain corresponds to a different execution or fidelity level in the application, such a level consists of a set of parameter values and reconfiguration options. Because of this, such a model restricts adaptation possibilities to a set of predefined levels, and does not allow each parameter and reconfiguration option to be tuned freely and independently. These drawbacks are exactly the advantages of the Fuzzy Control Model, where adaptation choices are specified with a set of rules, which are intuitive for the application developer to specify and tune. Further discussions of the advantages and problems of applying the Markov chain model are also found in previous work on video transmissions over IP networks [22].

Non-linear Control Theory

There are alternative domains in the control theory that are particularly applicable to the needs of non-linear time-variant target systems. Examples of these domains are adaptive control and robust control theory. Robust control theory attempts to establish a linearized model for non-linear systems, while adaptive control theory attempts to change the control algorithms by on-the-fly

system identification.

One of the major requirements to apply these models is to have an initial mathematical model for the control target to start with. In the adaptive control theory, for example, such a basic analytical model for the control target is required for the system identification process to work. However, it is almost impossible to come up with a precise analytical model for a specific target application, due to the various adaptation choices and alternatives available.

Compared to the above approaches, modeling the mapping process with the fuzzy control theory does not need an analytical model for the target application in order to design the control algorithms, since the controller is designed as a set of linguistic rules. Relaxing this requirement shows a clear advantage of adopting the Fuzzy Control Model.

5.3 Design of the Configurator

As stated in the previous section, the configurator takes the output of the adaptor as input, and generates actual manipulating and control actions to activate reconfiguration or parameter tuning within the application. The Fuzzy Control Model, shown in Figure 5.2, is used for designing both quantitative and functional configurators. The model comprises of five components built within the configurator. The *fuzzy inference engine* implements particular fuzzy control algorithms defined in the application-specific *rule base* and *membership functions* for linguistic values. The *input normalizer*, *fuzzifier* and *defuzzifier* prepare input values for the fuzzy inference engine, and convert fuzzy sets (the decisions made by the inference engine) to the actual real-world control actions for the applications.

While the architecture of the Fuzzy Control Model is generic and can be applied to any applications by configuring the rule base and membership function definitions, we use the *OmniTrack* application as a concrete example to elaborate our design of the configurator.

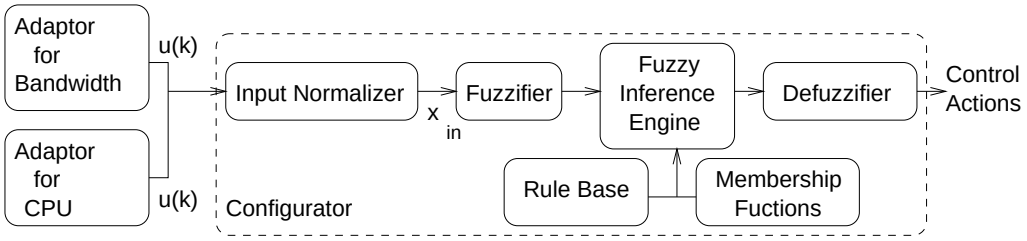


Figure 5.2: The Architecture of the Fuzzy Control Model

5.3.1 The Rule Base and the Inference Engine

The decisions of selecting linguistic values and rules in the rule base are based on a combination of human expertise and the output of QualProbes (presented in Chapter 6) operating on the particular application. The tradeoff is to decide on a minimum number of linguistic rules, while still satisfying the desired critical performance criterion in order to achieve an acceptable adaptation performance. All of the linguistic values used in the rule base should use words of a natural or synthetic language, such as `moderate` or `low` for the linguistic variable `cpu`. These values are modeled by fuzzy sets. In most cases, this form of representation leads to a compact description of the adaptation behavior within the application.

The design of the rule base is a two-phase process. First, the linguistic rules are determined. Second, membership functions of the linguistic values are set. In the configurator, the first phase of design generates a set of conditional statements in the form of if-then rules. The generic form is:

$$\begin{aligned} R^{(1)}: &\quad \text{if } X_1 \text{ is } A_1^{(1)} \text{ and } \ldots \text{ and } X_n \text{ is } A_n^{(1)} \text{ then } Y \text{ is } B^{(1)} \\ \ldots & \qquad\qquad \ldots \\ R^{(m)}: &\quad \text{if } X_1 \text{ is } A_1^{(m)} \text{ and } \ldots \text{ and } X_n \text{ is } A_n^{(m)} \text{ then } Y \text{ is } B^{(m)} \end{aligned} \tag{5.1}$$

where $X_1 \ldots X_n$ and Y are linguistic variables, $A_1^{(k)} \ldots A_n^{(k)}$ and $B^{(k)}$ ($k = 1, \ldots, m$) are linguistic values, defined by fuzzy sets $\tilde{A}_1^{(k)} \ldots \tilde{A}_n^{(k)}$ and $\tilde{B}^{(k)}$ ($k = 1, \ldots, m$), respectively. These linguistic values are also characterized by their membership functions, $\mu_{A_l^{(k)}}(x)$ and $\mu_{B^{(k)}}(y)$ ($l = 1, \ldots, n$),

respectively, with x and y being the elements of universal sets U and V. Each rule defines a fuzzy implication that performs a mapping from fuzzy input state-space to a fuzzy output value. After the defuzzification process, the fuzzy output value directly corresponds to a particular control action within the application.

The fuzzy inference engine operates by using the dual concepts of generalized modus ponens and compositional rule of inference [23]. In our implementation of the fuzzy inference engine, we adopt the C-FLIE inference engine implementation [24] as well as its input format for specifying the rule base and membership functions. For the mathematical completeness of this chapter, we summarize the internal mechanisms as follows.

The concept of generalized modus ponens is derived from the operation of modus ponens in binary logic. Modus ponens is the operation to draw a conclusion from two premises. Assume that we have the proposition p : "x is A" and the implication if-then rule $p \rightarrow q$: "if x is A then y is B" as true, we can conclude that the proposition q : "y is B" has to be true. In fuzzy logic theory, Generalized modus ponens extends the above operation in the following manner. If we have propositions p : "X is A" and q : "Y is B" where X and Y are linguistic variables and A and B are linguistic values, when both the if-then implication rule $p \rightarrow q$: "if X is A then Y is B" and proposition p^* : "X is A^*" is valid, where A^* is not necessarily the same as A, we can perform the generalized modus ponens and conclude q^* : "Y is B^*". The membership function μ of B^* is calculated by using the $\sup - *$ compositional rule of inference and Larsen's product operation rule:

$$\mu_{B^*}(y) = \sup_x [\mu_{A^*}(x) \star \mu_A(x)\mu_B(y)] \tag{5.2}$$

where $\star$ is a t-norm operator. An usual selection is the intersection definition of t-norm: $u \star w = \min(u, w)$.

When multiple input linguistic variables exist in the rule, inference can be extended by interpreting the fuzzy set of $A^{(k)}$, which is $\tilde{A}^{(k)}$, as the product of fuzzy sets $A_1^{(k)}, \ldots, A_n^{(k)}$. Its

membership function is defined as:

$$\mu_{A_1^{(k)} \times \ldots \times A_n^{(k)}}(x_1, \cdots, x_n) = \mu_{A_1^{(k)}}(x_1) \star \mu_{A_2^{(k)}}(x_2) \star \cdots \star \mu_{A_n^{(k)}}(x_n) \tag{5.3}$$

where $\star$ is the previously defined t-norm operator and $k = 1, \ldots, m$.

If a rule base contains multiple rules, overall decision of the inference engine is obtained by taking the union of $\tilde{B}^{(k)*}(k = 1, \ldots, m)$, which is the fuzzy sets of linguistic values $B^{(k)*}$ calculated by Equation (5.2) and (5.3). The calculation is as follows:

$$\mu_{B^{(1)*} \cup \ldots \cup B^{(m)*}}(y) = \mu_{B^{(1)*}}(y) \otimes \cdots \otimes \mu_{B^{(m)*}}(y) \tag{5.4}$$

where $\otimes$ represents the s-norm operator for defining disjunctions in approximate reasoning. A usual selection is $u \otimes w = \max(u, w)$.

5.3.2 Rule Base for *OmniTrack*

We consider the *OmniTrack* application as an example for designing the rule base and membership functions used in the configurator. As we noted, the ultimate objective and most critical application-specific quality parameter in the application is the *tracking precision*. If the precision is compromised, the objects lose track and other parameters are not meaningful.

In this application, the adaptation possibilities can be classified into two categories. First, control actions may occur in order to adapt to transmission bandwidth variations, so that bandwidth requirements within the application are adjusted to maintain tracking precision. Second, adaptations may take place to adapt to varied availability of CPU cycles, so that CPU requirements are adjusted. For other complicated applications, memory or secondary storage requirements are also taken into consideration.

Within the above categories of adaptation possibilities, there are two different kinds of adaptation choices. First, parameter-tuning actions try to tune quantitatively continuous parameters, such

as image size, to meet adaptation goals. These actions are controlled by the quantitative configurator. Second, application-level reconfiguration choices are available so that the resource demands of applications may be adapted by choosing among available configuration options, each having diverse requirements for resources. This process sometimes involves an alteration in the Task Flow Graph of the application, and is controlled by the functional configurator.

As an example, we have identified some of the adaptation possibilities in the basic client-server relationship of *OmniTrack* as follows, divided into two major categories. A more elaborate discussion of all adaptation choices in *OmniTrack* is presented in Section 10.2.

- **Adaptation of Communication Bandwidth Requirements**. Since the application is client-server based, sufficient bandwidth is required for preserving tracking precision. First, the following options exist for parameter-tuning actions for uncompressed image transfer. (1) The *image size* can be enlarged or reduced to adjust bandwidth requirements, by *chopping* the edges. The tradeoff is that the smaller the image, the higher the probability that the objects move out of range. (2) The *frame rate* of live video streaming can be increased or decreased. (3) The *color depth* can be altered. Existing choices for coding one pixel are 24 bits RGB, 16 bits packed RGB, 8 bits grayscale or 1 bit black-and-white. Second, if we consider reconfiguration choices, *compression* and corresponding *decompression* can be activated, using available choices such as Motion-JPEG and streaming MPEG-2 among others. Bandwidth requirements are reduced dramatically at the expense of increased CPU load.

- **Adaptation of CPU Requirements**. The tracking algorithms, referred to as *trackers*, are inherently computationally intensive. In the current implementation, there are three frequently used trackers. *Line* tracker and *corner* tracker are edge based algorithms, the *SSD* tracker is a region based algorithm. These algorithms present diverse computational requirements. In addition, the application can run multiple trackers simultaneously tracking multiple objects. The tradeoff is increased computation load. These facts motivate the following reconfigu-

ration choices: (1) Add additional trackers to utilize idle CPU; (2) Drop active trackers to decrease CPU demand; (3) Replace existing trackers by less or more computationally intensive trackers. Finally, parameter-tuning adaptation may also be applied by modifying the size of the *tracked region* of a specific tracker, effectively tuning the computational load of the tracker. The tracked region is defined as the searching range of the tracker in the feature detection stage of computation.

The adaptation measures described above make it possible to design the rule base for the basic client-server pair in *OmniTrack*, following the generic form given in the Equation (5.1). As Figure 5.2 shows, the Fuzzy Control Model takes the output of multiple adaptors as input, each of which corresponding to one type of resource. In the particular case of *OmniTrack*, we focus on two types of resources: CPU cycles and transmission bandwidth. In our rule base, the linguistic variable `cpu` corresponds to the values $u(k)$ generated by the adaptor with respect to the CPU resource, and the linguistic variable `rate` corresponds to the values $u(k)$ generated by the adaptor with respect to transmission bandwidth. The range of measuring linguistic variable `cpu` is $[0, 1000]$ with an unit of 0.1% of CPU load, and the range of measuring linguistic variable `rate` is $[0, 2000]$ with an unit of kilobytes transmitted per second. Before processing in the inference engine, the numerical crisp values u_k are first linearly normalized to the above ranges and units, then mapped to a fuzzy set by the fuzzification process, of which the mathematical details are documented as follows for the mathematical completeness of this chapter.

A fuzzy inference engine calculates fuzzy sets as results, taking fuzzy sets as inputs. In the above equations, the calculated union of fuzzy sets $\tilde{B}^{(k)*}(k = 1, \ldots, m)$ is the output of the inference engine, while the inference rules and the fuzzy set $\tilde{A}^*$ are the inputs.

However, we do not normally have the fuzzy set $\tilde{A}^*$ in advance, since we normally deal with numerical crisp values. The fuzzification process takes the numerical crisp value x_{in} as input, and generates a fuzzy set $\tilde{A}^*$. If there is no uncertainty in the numerical values, a simple fuzzification process can be:

$$\mu_{A^*}(x) = \begin{cases} 1, & \text{if } x = x_{in} \\ 0, & \text{if } x \neq x_{in} \end{cases} \tag{5.5}$$

Otherwise, if there is some uncertainty in the numerical value x_{in}, the membership values of the elements of $\tilde{A}^*$ can be selected such that, $\mu_{A^*}(x)$ is taken as 1 if $x = x_{in}$, and $\mu_{A^*}(x)$ decreases linearly from 1 as x moves farther away from x_{in}.

In the former case where no uncertainty is involved, since $\tilde{A}^*$ will contain only a single element with membership value equal to 1, calculation in Equation (5.2) will become

$$\mu_{B^*}(y) = \mu_A(x_{in})\mu_B(y) \tag{5.6}$$

In the case of multiple input variables, we substitute Equation (5.3) in (5.6) and obtain

$$\mu_{B^*}(y) = \min[\mu_{A_1}(x_{in}), \cdots, \mu_{A_n}(x_{in})]\mu_B(y) \tag{5.7}$$

to compute the output of one inference rule. Finally, we compute an overall decision by applying Equation (5.4) to aggregate the calculated $\tilde{B}^{(k)*}, k = 1, \ldots, m$. This shows that the simple fuzzification process shown in Equation (5.5) simplifies the inference process in the inference engine.

There are two inference outputs using the rule base, corresponding to the bandwidth adaptation and CPU adaptation actions, respectively. The linguistic variables used for the output are `rate-action` and `cpuaction`, respectively. Each linguistic value that the variables `rateaction` and `cpuaction` corresponds to a particular control action, such as `size` for image size in the quantitative configurator, and `compress` for the functional configurator. With respect to the input, the linguistic values used for both `cpu` and `rate` are `low`, `moderate` and `high`.

We present an example of the rule bases for control actions related to the basic client-server relationship in *OmniTrack*, using the input format in the C-FLIE implementation of fuzzy inference

engine. Particularly, an example of the rule base defined for the configurator may be:

```
if (cpu is low) and (rate is moderate) then rateaction:= size;
if (cpu is moderate) and (rate is moderate) then rateaction:= size;
if (cpu is high) and (rate is moderate) then rateaction:= size;
if (cpu is high) and (rate is low) then rateaction:= compress;
if (cpu is high) and (rate is high) then rateaction:= compress;
if (cpu is moderate) and (rate is high) then rateaction:= raw;
if (cpu is low) and (rate is high) then rateaction:= raw;
if (cpu is low) and (rate is low) then rateaction:= blacknwhite;
if (cpu is moderate) and (rate is low) then rateaction:= blacknwhite;

if (cpu is moderate) and (rate is high) then cpuaction:= adjustregion;
if (cpu is moderate) and (rate is moderate) then cpuaction:= adjustregion;
if (cpu is moderate) and (rate is low) then cpuaction:= adjustregion;
if (cpu is low) and (rate is moderate) then cpuaction:= droptracker;
if (cpu is low) and (rate is high) then cpuaction:= droptracker;
if (cpu is low) and (rate is low) then cpuaction:= droptracker;
if (cpu is high) and (rate is low) then cpuaction:= addtracker;
if (cpu is high) and (rate is moderate) then cpuaction:= addtracker;
if (cpu is high) and (rate is high) then cpuaction:= addtracker;
```

5.3.3 The Design of Membership Functions

In normal design practices of fuzzy control systems, Gaussian, triangular or trapezoidal shaped membership functions are used to define the linguistic values of a fuzzy variable. Since triangular and trapezoidal shaped functions offer more computational simplicity, we choose them to define all membership functions for linguistic values used in the rule base.

The particular design of these membership functions is largely application-specific. In our visual tracking application, we have defined the membership functions as shown in Figure 5.3, in four universal sets for variables `cpu`, `rate`, `cpu_demand` and `rate_demand`, respectively.

5.3.4 The Defuzzification Process

Since the decision of the inference engine is expressed in fuzzy sets, in order to be able to use it as a control signal for applications, it has to be mapped to reconfiguration options or crisp numerical values of parameter-tuning actions. The defuzzification process produces a non-fuzzy output, y_{out}, whose objective is to represent the possibility distribution of the inference. There is no

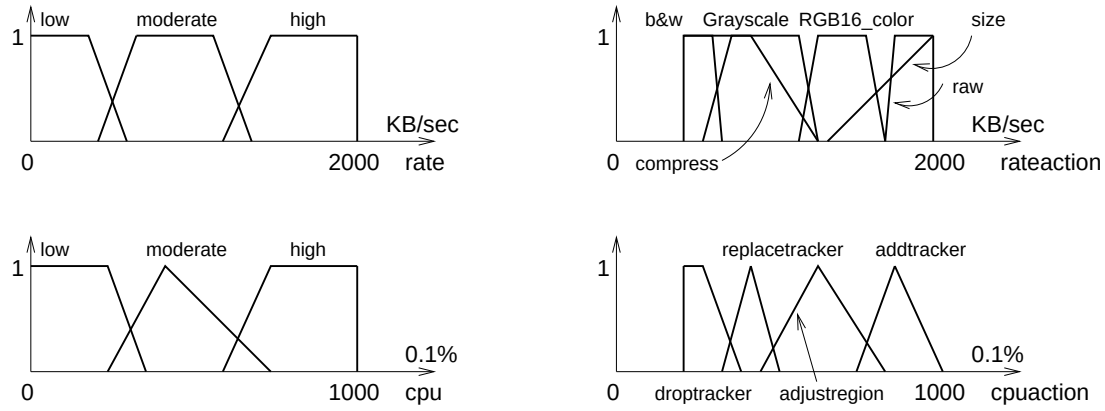


Figure 5.3: Membership Functions of the Linguistic Values

single method for performing the defuzzification. In fuzzy control systems, because of the ability of generating smoother control surfaces, the *Center of Gravity* method is frequently used. Detailed mathematical definition of the *Center of Gravity* method is presented as follows.

The *Center of Gravity* method is a frequently used method for the defuzzification process. This method divides the integral of the area under the membership function of the output fuzzy set (Equation 5.7) into half, and the defuzzified value y_{out} marks the dividing point. Formally, in the continuous case this results in

$$y_{out} = \frac{\int y \mu_{B^*}(y) dy}{\int \mu_{B^*}(y) dy} \tag{5.8}$$

Once y_{out} is obtained, the mapping method to the actual control actions varies in different types of configurator. In functional configurators, if in the rule base, $\tilde{B}$ is a fuzzy set corresponding to a reconfiguration option (e.g. `droptracker`, etc.) and $\mu_B(y_{out}) \neq 0$, the corresponding reconfiguration is activated. In the quantitative configurator, if $\tilde{B}$ is a fuzzy set corresponding to a parameter-tuning action associated with the parameter p (e.g. `size` associated with *image size*) and the tuning range $[p_{min}, p_{max}]$, then the modified value of p is set at

$$p = (p_{max} - p_{min}) * \mu_B(y_{out}) + p_{min} \tag{5.9}$$

when $\mu_B(y_{out}) \neq 0$.

In our specific implementation of quantitative configurators, the tuning range $[p_{min}, p_{max}]$ is specified for the parameter p in a separate configuration file, the details of which is presented in Chapter 9.

5.3.5 An Example of the Inference Process

To summarize the internal fuzzy inference process presented mathematically from Section 5.3.1 through Section 5.3.4, we visually illustrate the mathematical inference process from the input to the output control actions in Figure 5.4.

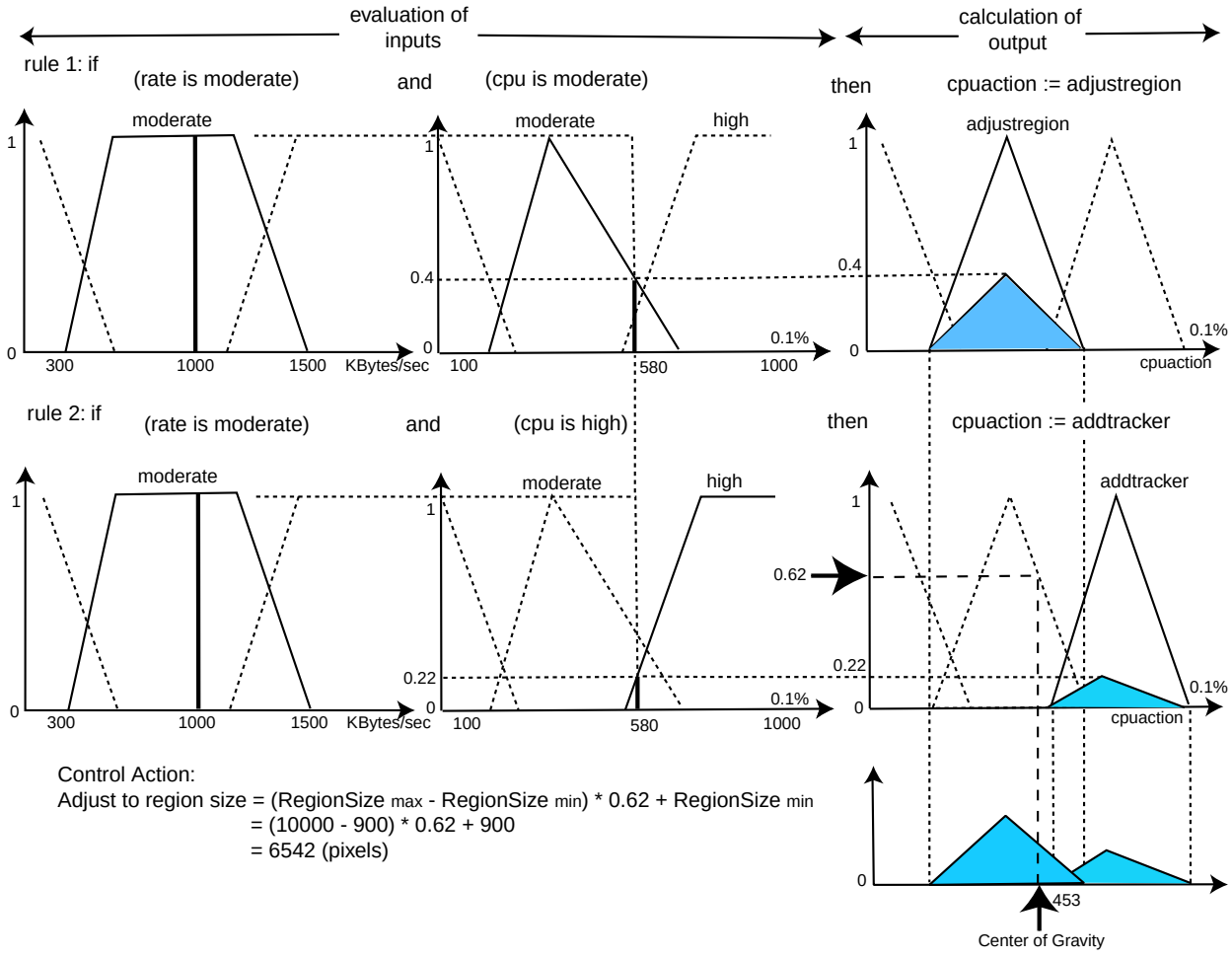


Figure 5.4: An Example of the Inference Process to Compute Control Actions

In Figure 5.4, we assume that the output of CPU and network adaptors is 58.0% and 1000

KBytes/sec respectively. If we assume a unit of 0.1% for CPU load `cpu` and 1 KBytes/sec for network throughput `rate`, this leads to an input `cpu` value of 580 and a `rate` value of 1000. From the definitions of membership functions specified in Figure 5.3, we note that the current `rate` value is a member of the fuzzy set "moderate" to the full extent (membership value is 1), while the `cpu` value is a member of the fuzzy sets "moderate" and "high", with membership values of 0.40 and 0.22, respectively. Their membership values of any other fuzzy sets are zero.

Once the above is acknowledged, each rule in the rule base is visited, and the minimum of membership values of the inputs to the corresponding linguistic values are found according to the Equation 5.3 and then the corresponding (for each rule) output linguistic value is scaled according to Equation 5.2. To be more specific, with the membership values of input variables, only the following two rules yield nonzero scaling factors and thus contribute to the calculation of the output:

```
if (rate is moderate) and (cpu is moderate) then cpuaction := adjustregion
if (rate is moderate) and (cpu is high) then cpuaction := addtracker
```

After applying the inference process shown in Equation 5.3 and 5.2, these two rules yield 0.40 and 0.22, respectively, as visually illustrated in Figure 5.4. If we consider the first rule, according to the linguistic rule, the output (`cpuaction`) is "adjustregion", whose membership function is adjusted by a value of $\min(1, 0.40)$. Similarly, in the second rule, the output is "addtracker", whose membership function is scaled by a value of $\min(1, 0.22)$. The results of membership function adjustments are shown as shaded regions in the top two curves of the "calculation of the output" section of the figure.

For the calculation of the actual output, we take the union of all scaled output fuzzy sets according to the Equation 5.4, then by using the Center of Gravity defuzzification method shown previously in this section, a numerical value for the output variable `cpuaction` is calculated as 453 in this example (with the unit being 0.1% of CPU load). This value is then mapped to the actual parameter values to be tuned according to the Equation 5.9. According to the membership function of "adjustregion", 453 corresponds to a membership value of 0.62. On the other hand, ac-

cording to the membership function of "addtracker", the corresponding membership value is 0. In Equation 5.9, only "adjustregion" is activated since "addtracker" does not satisfy the condition that $\mu_B(y_{out}) \neq 0$. The result, as shown in the figure, is to activate the adaptation choice "adjustregion" and tune each of the tracker region size of the SSD trackers in the application to 6542 pixels.

5.3.6 The User Configurator

The user configurator is a completely application-specific component that is designed to interact with the user, so that the user may participate in the decision-making process of application-aware adaptation. Visual tuning and selection interfaces are provided for the user to directly manipulate and adapt the application.

The introduction of the user configurator complements the functionality of the quantitative and functional configurators with the Fuzzy Control Model. The assumption is that, in most cases, completely automatic adaptation control is not the optimal solution with respect to achieving the highest possible degree of user satisfaction, since it deprives the user's ability to participate in the decision-making process when automatic adaptation is not sufficient to achieve the goals.

For the *OmniTrack* application, the user may interact with the user configurator to pan or tilt the camera on the tracking server so that the object does not roam out of bounds. In other cases, it may prefer to switch to a server with a different angle of view, in order to capture a better shape of the object for tracking. Both adaptation activities enhance the quality of tracking and the user satisfaction. However, since they involve interpretation of semantics that is not easily observable, they can not be activated automatically by *Agilos*. In these cases, user interaction is critical to achieve the overall satisfaction.

The implementation of such an user configurator is dependent on specific semantics of a particular application. Such implementation for the *OmniTrack* application is presented in Chapter 9. The user configurator also interacts with the third tier of *Agilos*, since it may trigger negotiations and server switching activities on user demand. Further details of such interactions are postponed to Section 7.2.

5.4 Summary

In this chapter, we focused on flexible distributed multimedia applications that need to adapt their behavior to variations of the resource availability and assure quality of critical QoS parameters. In this work we presented a Fuzzy Control Model, incorporated in the design of the middleware configurator. The configurator maps numerical values from the *Agilos* adaptors to actual control actions of parameter tuning or reconfiguration choices. The design of the rule base and membership functions within the Fuzzy Control Model was also shown in the context of a *OmniTrack*. Chapter 9 will present a full spectrum of various characteristics of *OmniTrack* in details, and experimental results are postponed to Chapter 10.

Chapter 6

The Design of QualProbes

6.1 Overview

As introduced in Section 1.2, the *Agilos* architecture is designed to be an *active* middleware layer, in the sense that it exerts strict control of the adaptation behavior of QoS-aware applications, so that these applications adapt and reconfigure themselves under such control. The goal is to provide the *best possible* QoS with the current resource availability in a swiftly changing environment.

The first and second tier of *Agilos* consist of application-neutral *adaptors* and application-aware *configurators*, which reflect the first two tiers of middleware control in *Agilos*. In the application-neutral level, each adaptor corresponds to a single type of resource, e.g., CPU adaptors or network bandwidth adaptors. Though the adaptors are specific to resources, they are not aware of the semantics of individual applications. In contrast, the *configurators* in the application-specific level are fully aware of the application-specific semantics, and thus each configurator only serves one application. This hierarchical design of the *Agilos* architecture was illustrated in Figure 2.1. Chapters 3 to 5 have presented the design of adaptors and configurators, and addressed the problem of how to make such application-specific adaptation decisions in the *Agilos* middleware. However, there is still an important open problem: how to appropriately define the application-specific rule base and membership functions within the configurator, so that the *best possible* adaptation results are achieved by such an architecture?

This question brings forth a fundamental problem of how to appropriately choose a criterion that can assist the judgment of "What is best?". Most applications have more than one QoS parameters that are application-specific, and any changes in these parameters contribute to an increase or degradation of the delivered quality. In this chapter, we focus on the *critical performance criterion*, which concentrates on the satisfaction of requirements related to the most critical application QoS parameter. The quality of other non-critical parameters can be traded off. For example, in our case study of *OmniTrack*, the tracking precision is the most critical QoS parameter in the tracking application. The *critical performance criterion*, therefore, is to keep the tracking precision accurate and stable. The ultimate objective of the *Agilos* architecture is to control the adaptation process within the application so that it is steered towards the satisfaction of application-specific *critical performance criterion*.

Even after the critical performance criterion is determined, an accurate mapping between application QoS parameters and their resource demands still needs to be discovered in order to devise the optimal adaptation strategy, and eventually the rule base and membership functions, which we collectively refer to as the "fuel" in the configurator. In this chapter, we present *QualProbes*, a set of middleware QoS probing and profiling services, that are uniquely designed to address the following problems: (1) How do changes in non-critical application QoS parameters relate to the critical QoS parameter, and thus the critical performance criterion? Frequently the critical QoS parameter is not observable on the fly, and not directly controllable also. The relations between non-critical and critical parameters need to be discovered in order to study the indirect effects of critical parameters. In some occasions, more than two parameters will be involved in a multi-dimensional relationship. (2) How do the changes in application QoS parameters relate to changes in resource demands or consumption? (3) How do the solutions to the previous problems assist making appropriate definitions of rules and membership functions in the middleware configurator, so that the critical performance criterion, e.g., a stable tracking precision, are satisfied and maintained? Once we have solved these problems, we are able to control the adaptation process within the application from the middleware, so that under any circumstances in a best-effort en-

vironment and with fluctuating resource availability, the application is able to maintain the *best possible* Quality of Service, in the sense that the critical performance criterion is always satisfied.

QualProbes are designed to assist controlling the applications so that control actions are generated with better awareness of application's behavior and resource demands. To achieve this goal, the results of QualProbes are utilized in configuring the "fuel" of the configurator. As detailed in Chapter 5, the configurator is designed as a rule-based fuzzy control system. It can be partitioned into three parts: the fuzzy inference engine, membership functions and rule base. While the fuzzy inference engine is application-neutral, the "fuel", namely the rule base and membership functions of related linguistic values, are application-specific. Such a model guarantees that different adaptation choices and a wide variety of resource/application QoS mappings can be expressed easily with a set of new rules and membership functions in the rule base.

Rules in the rule base are written using linguistic variables and values. In *OmniTrack*, examples of variables are `cpuaction` and `rateaction`, and examples of values are `compress` and `addtracker`. These values are uniquely characterized by *membership functions*, so that the inference engine can have exact definitions of these values. The design of the rule base involves the generation of a set of conditional statements in the form of if-then rules, such as *if (***cpu** *is high) and (***rate** *is low) then* **rateaction** *:= compress*.

Apparently, the role of QualProbes is to capture the run-time relationships between application QoS and their resource demands, so that the above rules are activated with appropriate timing. The key contributions of QualProbes [25] are the following. First, via on-the-fly measurements in benchmarking runs, they are able to precisely discover and profile the relationship between changes in non-critical application QoS parameters, such as the image quality or frame rate in *OmniTrack*, and changes in the critical parameter, such as the tracking precision on the client. The results of such discovery are important to the satisfaction of the critical performance criterion. Second, via similar probing mechanisms, they are also able to obtain and profile the relationship between changes in controllable application QoS parameters and related changes in resource demands or consumption. Finally, we present a series of solutions so that these profiles obtained by Qual-

Probes may assist defining the appropriate rule base and membership functions to properly adapt the application.

6.2 *QualProbes:* Investigating Application-Specific Behavior

Since the ultimate objective is to steer adaptations towards satisfaction of the critical performance criterion, the primary goal of QualProbes services is to devise mechanisms that best facilitate such optimal steering of adaptation decisions. To achieve this goal, QualProbes need to address the following issues. First, QualProbes need to accurately capture the relationships between the most critical application QoS parameter, such as the tracking precision, and other non-critical ones. This is crucial to perform tradeoffs of non-critical parameters. Second, QualProbes need to capture the resource demands of each non-critical QoS parameters. Both of the above are achieved via run-time probing and profiling mechanisms. Finally, such profiling results should be used to assist the generation of application-specific control rules, which are integrated in the configurator.

We address the above issues in the following sections. We illustrate our solutions with actual examples derived from *OmniTrack*.

6.2.1 Relations Among QoS Parameters and Resources: The Dependency Tree Model

As previously noted, the application-specific QoS parameters can be classified as *critical* (usually one parameter such as the tracking precision) and *non-critical*. In addition, the changes of each parameter in the *non-critical* collection may cause and be dependent on the changes of zero, one, or multiple types of resources.

Assume that we study m different resource types, and the current observation of consumed resources are $R_1, R_2, \ldots, R_m$, measured with their respective units. Typically in *OmniTrack*, $m = 2$, and R_{cpu} is measured with the CPU load percentage, while R_{net} is measured with bytes per second.

In addition, assume that there are n unique non-critical QoS parameters that may influence the critical parameter, p_c, in the application. These parameters are p_i, $i = 1, \ldots, n$. For $p_i, \forall i$, there are l of resource types related to p_i, where $l \leq m$. In the *OmniTrack* example, if p_i is *frame rate*, its changes correspond to R_{net} and R_{cpu}. In contrast, if p_i is the *object velocity*, it does not directly correspond to any resources, though p_c, the tracking precision, depends on its variations.

A Dependency Tree for Application QoS Parameters

Although each p_i corresponds to resources R_i, $i = 1, \ldots, l$, we observe that such dependencies are generally hard to capture directly. We take the parameter *frame rate* in *OmniTrack* as an example. Naturally, the frame rate of video streaming depends on network bandwidth availability. However, the nature of such dependence is non-deterministic: For the same available bandwidth, the frame rate varies diversely for compressed video versus uncompressed video; different CPU load may limit the capacity that trackers can consume the frames, thus limiting the frame rate. Similar situation applies to other parameters.

Such observations illustrate that each p_i, in addition to being *directly* dependent on resource types, depends directly on a subset of p_j, $j \neq i$, and via its dependence with this subset of parameters p_j, *indirectly* corresponds to resources. We define that if p_i is *dependent* on p_j, then changes in p_j can cause changes in p_i. Ideally, a generic model for capturing the dependencies is by using an acyclic directed dependency graph, with the critical parameter p_c as the source, and resources R_i, $i = 1, \ldots, m$ as the sink. For simplicity reasons, we only consider a special case that all but the bottom levels of such a dependency graph is a directed *binary tree*, with p_c as the root of the tree, and resources as the leaves. Each p_i depends on zero, one or two other parameters or resources.

There are two key characteristics in such a dependency tree[1]. First, the resource types R_i, $i = 1, \ldots, l$ are always leaf nodes of the tree. This is based on a simplified assumption that the changes of each resource type never depend on any other resources, i.e., that resource types are

[1]To be exact, it is only a binary tree without considering the bottom level related to resources. Otherwise, it is more of a lattice.

independent with each other. Second, we note that in addition to demanding resources of certain types, the changes of an application QoS parameter may change the resource availability of some other resource types, without demanding them. For example, while changing the *compression ratio* in *OmniTrack* demands CPU resources, its changes will have significant effects on available network bandwidth also, since less data is necessary to be transmitted. This case is presented by a directed arrow from the resource node R_i to the QoS parameter node p_j, showing that the availability of R_i relies on p_j, rather than the usual case that p_j demands and relies on R_i. The direction of such a directed arrow is dependent on specific application QoS parameters, and specified by the user or application developer. An illustration of our directed dependency tree model and an real-world example with *OmniTrack* is given in Figure 6.1.

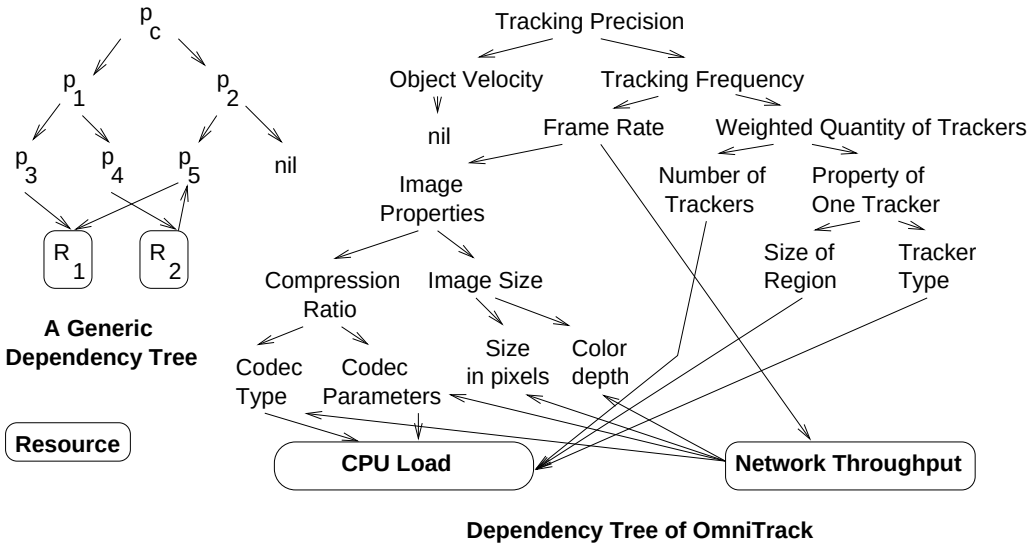


Figure 6.1: The Dependency Tree for Application QoS Parameters

Characterizing the Relationship Between Dependent Nodes

Once we have established the dependency tree of QoS parameters for an application [2], the relationship between dependent nodes needs to be characterized appropriately. We assume that for $\forall i, \forall t$, there exist $\{p_i\}_{min}$ and $\{p_i\}_{max}$ such that $\{p_i\}_{min} \leq p_i(t) \leq \{p_i\}_{max}$, any values beyond

[2]Such establishment is application-specific, and may be derived based on knowledge of a specific application.

this range is either not possible or not meaningful. For example, the *frame rate* may vary in between $[1, 30]$ fps. Assume the parent node p_i depends on two descendant nodes p_x and p_y. The dependency can thus be characterized by a function $f_{i,x,y}$, defined as:

$$\begin{aligned} \Delta p_i &= f_{i,x,y}(\Delta p_x, \Delta p_y) \\ p_k &= \{p_k\}_{min} + \Delta p_k, \text{ with } k \in \{i, x, y\} \\ 0 &\leq \Delta p_k \leq \{p_i\}_{max} - \{p_i\}_{min} \end{aligned} \tag{6.1}$$

where Δp_k is a normalized value of p_i based on $\{p_i\}_{min}$. Function $f_{i,x,y}$ defines the dependence relationship between the parent node p_i and its descendant nodes p_x and p_y. If p_i only depends on one node p_x, then $f_{i,x,y}$ is equivalent to $f_{i,x}$, where $\Delta p_i = f_{i,x}(\Delta p_x)$.

Similarly, we may define ΔR_x as $\Delta R_x = R_x - \{R_x\}_{min}$. If one or two of the descendant nodes are resource types R_x and R_y, then we define f_{i,r_x,r_y} so that $\Delta p_i = f_{i,r_x,r_y}(\Delta R_x, \Delta R_y)$. Note that for the special case that the availability of resource type R_i depends on changes in p_j, i.e., there is a directed link from R_i to p_j, we define $f^r_{r_i,j}$ such that $\Delta R_i = f^r_{r_i,j}(\Delta p_j)$. Figure 6.2 visually shows the above characterization.

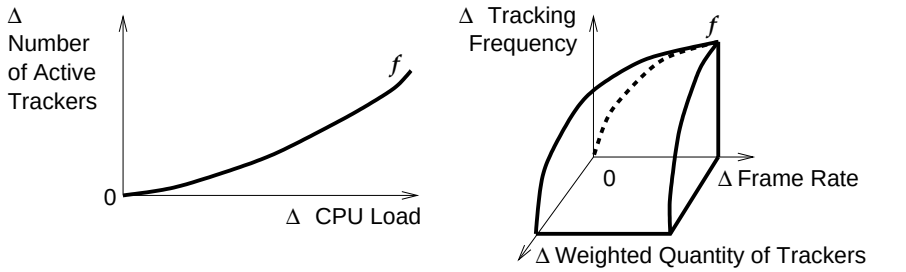


Figure 6.2: Characterization of Dependencies among QoS Parameters

If we obtained all $f_{i,x,y}$ in the dependency tree via probing and profiling services, the relationship of any application QoS parameter p_i and its related resources can be characterized by a series

of substitutions. As an example, for the generic dependency tree in Figure 6.1, we have

$$\begin{aligned}
\Delta p_c &= f_{c,1,2}(\Delta p_1, \Delta p_2) \\
&= f_{c,1,2}(f_{1,3,4}(\Delta p_3, \Delta p_4), f_{2,5}(\Delta p_5)) \\
&= f_{c,1,2}(f_{1,3,4}(f_{3,r_1}(\Delta R_1), f_{4,r_2}(\Delta R_2)), f_{2,5}(f_{5,r_1}(\Delta R_1)) \qquad (6.2)
\end{aligned}$$

and

$$\Delta R_2 = f^r_{r_2,5}(\Delta p_5) \qquad (6.3)$$

which characterizes the relationship between p_c and resources R_1 and R_2. Note that the probing algorithm illustrated in the next section provides point-wise estimation of the functions $f_{c,1,2}$, $f^r_{r_2,5}$, and others.

6.2.2 QualProbes Services Kernel: The QoS Profiling Algorithm

QualProbes services are responsible for run-time capturing of the relationships f and f^r between dependent nodes in an application-specific dependency tree, and for properly storing the results in profiles. QualProbes services are middleware components, and implement a *QoS Probing and Profiling* algorithm as the kernel in each component. While the probing and profiling services provided by QualProbes are application-specific, the kernel algorithm is designed to be application-neutral, thus we require that all related application QoS parameters should present the following properties:

1. *Observable*. Their run-time values at any instant can be obtained in a timely manner.

2. *Tunable*. They should be either directly or indirectly tunable from outside of the application. Note that all the QoS parameters that are directly linked to the resource nodes (CPU and throughput) in the dependency tree are directly tunable parameters.

The online observation of application QoS parameter values ("reading" access to the parameter) is achieved with the assistance of the CORBA Property Service in our implementation. Applications report values of their QoS parameters as CORBA properties to the Property Service when initializing or when there are changes, while QualProbes services kernel retrieves these values from the Property Service when necessary. In order to properly report parameter values in the applications, they need to be instrumented at the source level with calls to member functions in the *Agilos* C++ class `TMetrics`, which encapsulates all necessary mechanisms to communicate with the CORBA Property Service. Implementation details related to the reporting and retrieval of these values are further discussed in Section 8.2.2.

In order to tune application QoS parameters ("writing" access to the parameter), the application needs to export an Application Control Interface to the *Agilos* quantitative configurator. QualProbes services only need to reuse the functions in this interface to control directly tunable QoS parameters in the application. Details about the definition of such an application control interface is presented in Section 8.3.

Having ready "read/write" access to the application QoS parameters, QualProbes services execute a *QoS Profiling* algorithm in the kernel. The algorithm traverses the dependency tree starting from the leaves, and upwards to the root, while attempting to discover the functions f and f^r previously defined by tuning the values in descendant QoS parameters or resource types and measuring those of the parent QoS parameter. If f is three-dimensional, a nested loop involving both descendant parameters is executed. Figure 6.3 demonstrates the QoS profiling algorithm in the pseudo-code form. In this algorithm, function **tune** executes recursively in order to tune an application QoS parameter indirectly. The result of executing the function **tune**(p_j, k) is to "write" the value k to the parameter p_j. If p_j is directly tunable, then **tune** simply assign the value k to p_j by the application control interface; otherwise, it takes advantage of recursion to eventually tune the parameter p_j indirectly.

After the execution of this kernel algorithm, all functions f and f^r, that represent the relationship between any non-leaf node p_i and their descendant nodes, are profiled in the form of a

for each resource leaf node R_i in the dependency tree:
  if link($R_i \rightarrow p_j$) or link($p_j \rightarrow R_i$) exists
  for $k = \{p_j\}_{min}$ to $\{p_j\}_{max}$ step $\{p_j\}_{increment}$
    tune(p_j,k);
    log the tuple $\{\Delta p_j, \Delta R_i\}$ as $\{k - \{p_j\}_{min}, R_i - \{R_i\}_{min}\}$;
for each non-leaf node p_i in the dependency tree (nodes on descendant levels first):
  if p_i has one descendant parameter node p_x
  for $k = \{p_x\}_{min}$ to $\{p_x\}_{max}$ step $\{p_x\}_{increment}$
    tune(p_x, k);
    log the tuple $\{\Delta p_x, \Delta p_i\}$ as $\{k - \{p_x\}_{min}, p_i - \{p_i\}_{min}\}$;
  else if p_i has two descendant parameter node p_x and p_y
  for $k_1 = \{p_x\}_{min}$ to $\{p_x\}_{max}$ step $\{p_x\}_{increment}$
    for $k_2 = \{p_y\}_{min}$ to $\{p_y\}_{max}$ step $\{p_y\}_{increment}$
      tune(p_x, k_1); **tune**(p_y, k_2);
      log the tuple $\{\Delta p_x, \Delta p_y, \Delta p_i\}$ as $\{k_1 - \{p_x\}_{min}, k_2 - \{p_y\}_{min}, p_i - \{p_i\}_{min}\}$;

tune(p_i, value)
  if p_i is directly tunable via exported interface
    call the application control interface to set p_i = value;
    return;
  else
    assume descendant nodes of p_i are p_x and p_y
    for $k_1 = \{p_x\}_{min}$ to $\{p_x\}_{max}$ step $\{p_x\}_{increment}$
      for $k_2 = \{p_y\}_{min}$ to $\{p_y\}_{max}$ step $\{p_y\}_{increment}$
        tune(p_x, k_1); **tune**(p_y, k_2);
        if ((observed p_i) == value) **return;**

Figure 6.3: QualProbes Services Kernel Algorithm

series of tuples, with the variables in the function spanning the complete range of in between their minimum and maximum possible values. Having these profiles in the form of tuples of values, we may then utilize a two-dimensional or three-dimensional plotting tool to visualize the functions.

As an concrete example, Figure 6.4 illustrates the results of tuning the QoS parameters *object velocity* and *tracking frequency* in order to measure the tracking precision. The output of the inner loop (by only tuning tracking frequency) is shown as bold dotted lines.

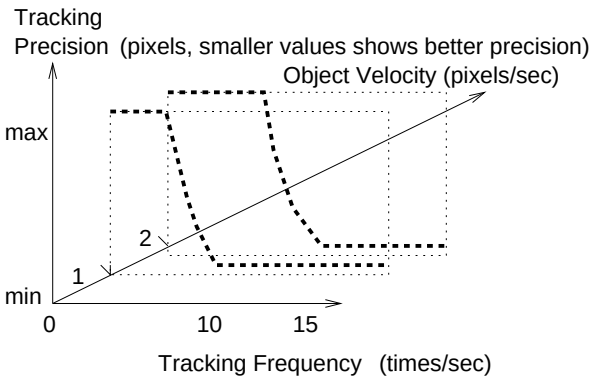


Figure 6.4: QualProbes Services: An Example

6.2.3 Towards Better Middleware Control

The design of QualProbes services in previous sections addresses the problem of discovering relationships between the application QoS parameters and resource demands of an application. In order to complete the solutions provided by QualProbes, we need to address the issue of bridging the obtained profiles (namely, functions f and f^r) with actual membership functions and inference rules in the configurator.

Towards Better Rule Bases

Based on our extensive experiences with the *OmniTrack* application, we believe that the inference rules inside the rule base cannot be generated automatically. Such rules need to be written by the application developer for a specific application. The reasons are two-fold: First, a rule base customized by the application developer is best in exploiting all available adaptation choices and best optimizes the rich semantics of these choices, naturally integrating the relative priorities of different application QoS parameters. In other words, the application developer should decide the set of QoS parameters to be traded off in the event of quality degradation. Second, the rules are not constant. It should be tuned towards the needs and user preferences in different occasions where the application is executed.

That said, the dependency tree for application QoS parameters is able to provide important knowledge on how the rule base should be specified, once it is appropriately established for an

application based on its semantics. The dependency tree and the profiles may facilitate the specification of the rule base in many ways, enumerated as follows.

First, since each of the tunable QoS parameters found in the dependency tree corresponds to a particular control action, the output linguistic values in the inference rules should have a one-to-one mapping with these tunable parameters. For example, as shown in Figure 6.1, the parameter "Codec Type" should be mapped to a linguistic value "compress", which activates the control action that changes the codec type from uncompressed video to Motion JPEG. The parameter "Size in pixels" should be mapped to another linguistic value "size", which changes the frame size of the streamed video. Configuration files are used to specify the mappings among the control interface functions, tunable parameters and linguistic variables. The details of these files are shown in Section 8.3.2 and 9.3.4.

Second, the availability of the dependency tree and relationship profiles promotes more detailed understanding with respect to application adaptive behavior. In other words, it illustrates how one application QoS parameter will change if its descendant nodes change.

For the *OmniTrack* example whose dependency tree is shown in Figure 6.1, we examine the behavior of the application starting from the CPU resource. Starting from the relationship between compression ratio and the leaf node "CPU load", after the execution of *QualProbes* services kernel algorithm illustrated in Figure 6.3, we obtain a profile that characterizes the relationship between compression ratio and CPU load. From the profiles, we learn that a lower compression ratio will lead to a lower CPU load. Further up the tree, similarly to the above process, we learn that with the same network throughput, the frame rate is lower if the compression ratio is lower. This by itself will lead to a lower tracking frequency with the same quantity of trackers. Finally, we analyze from the relationship profiles that a lower tracking frequency will eventually cause unstable tracking precision.

From the above analysis, we learn that if any adaptation actions are to be taken, in order to maintain the tracking precision, the goal is to maintain the tracking frequency. Without changing the quantity of trackers, this translates to maintaining the frame rate. Let us now assume that the

output of the adaptors demands a higher value of `cpu` and a lower value of `rate`. Since the adaptors dictate that the application should request less network bandwidth, the frame rate should be lower. However, since the frame rate needs to be maintained, the compression ratio should be higher, so that the frame rate may remain the same with less requests for network bandwidth. This leads to the inference rule:

```
if (cpu is high) and (rate is low) then rateaction := compress
```

In the above rule, `cpu` is assigned to be "high" when the CPU adaptor suggests that the CPU load should increase, i.e., that the current CPU utilization is low.

Finally, a careful examination of the dependency tree for application QoS parameters may lead to more complex rule bases such as two or more rules organized *hierarchically*. We take the *OmniTrack* example again to illustrate this concept. Assume that we wish to find an appropriate inference rule to decrease the number of active trackers by the control action `removetracker`. We first learn from the profiles that when the number of trackers decreases, the tracking frequency will increase as a result. On the other hand, we find that with the same image properties, if the CPU usage increases to a very high load (e.g., near 100%), and a significant number of trackers still remain active, then the tracking frequency will decrease. The above analysis leads to the following conceptual rules:

```
if (cpu is very_high) then tracking_frequency := low
if (tracking_frequency is low) then cpuaction := removetracker
```

Obviously, these are equivalent to **if** *(cpu* **is** *very_high)* **then** *cpuaction := removetracker*, which is actually used in the rule base. However, the two conceptual rules above are important to have, since they are beneficial to the specification of membership functions with the assistance of profile analysis, which is elaborated in the next section.

To conclude, the specification of inference rules in the rule base is still largely ad-hoc and heuristic, with an analyzing process that cannot be done automatically. However, the availability of the dependency tree and profiling results are of significant assistance in such a specification process, so that iterations of trial-and-error runs are reduced to the minimum.

Thresholds: Towards Better Membership Functions

Even though the rules can not be generated automatically, the profiles discovered by QualProbes services are of significant assistance in the process of determining the membership functions of linguistic values in the inference rules. In order to demonstrate such assistance, we take one inference rule in *OmniTrack* as an example:

if *(cpu* **is** *high)* **and** *(rate* **is** *low)* **then** *rateaction := compress*

This inference rule operates as follows. First, it takes the output of CPU adaptor and network bandwidth adaptor in the application-neutral level as input. When the CPU is idle, the CPU adaptor will apply its application-neutral control algorithm and suggests that the application under its control demands more CPU resources. This yields a high `cpu` value. Similarly, when the network is congested and there is very low bandwidth available, the network bandwidth adaptor suggests that the application demands less network bandwidth, thus yielding a low value in `rate`. Second, the inference rule decides that if `cpu` is high and `rate` is low, the application should reconfigure itself and add compression to its video streaming. Third, the actual definitions, made via the membership functions, of linguistic values `high` and `low` decide the activation timing of such reconfiguration choice.

The question is: How "high" is `high` for this specific rule? As we have observed in our experiences with OmniTrack, very frequently the discovered profiles by QualProbes services are non-linear, and contain certain *threshold* values. For example, by switch codec type from "uncompressed" to "Motion JPEG", we observe that ΔR_{cpu} steps up abruptly by a certain amount, e.g., 60%, while ΔR_{net} steps down by about 90% of the original value. The *threshold*, thus, can be determined from the profiles obtained by QualProbes services. For example, `high` can be defined as higher than 60%, while `low` can be defined as lower than 90% of $\{R_{net}\}_{max}$.

As another example, let us examine the profiles obtained related to the top level of dependency tree, the tracking precision. Such profiles are illustrated in Figure 6.4. One of the corresponding inference rule is:

if *tracking_frequency* **is** *low* **then** *cpuaction := removetracker*

As illustrated by Figure 6.4, QualProbes services have discovered an approximate threshold value for tracking frequency at respective object speed levels. If the tracking frequency drops below such threshold values, we could speculate that tracking precision may degrade. In order to keep the tracking precision, which is the critical performance criterion for OmniTrack, we define the membership function of linguistic value `low` to cover the values lower than the threshold value that we have discovered, e.g., 10 iterations per second. When this definition is applied to the above inference rule, the configuration choice of `removetracker` will be activated when the tracking frequency falls below the critical threshold value. This ensures that the tracking precision is kept stable at all times.

6.3 Summary

This chapter has presented new mechanisms with respect to investigating the behavior of the application, for the purpose of generating best control actions for the application to adapt itself to the environmental variations. A detailed analysis of *QualProbes* is presented, including the application model, the dependency tree model for application QoS parameters, and the QoS profiling algorithm implemented in the QualProbes services kernel. The key contribution of this chapter is that we have provided a unique approach to ”see through” the behavior of the application, especially when environmental or requirement changes may occur. By generating the “fuel” of the configurator, QualProbes indirectly contributes to the overall effectiveness of the configuration process of steering applications towards maintaining its critical performance criterion.

Chapter 7

The Gateway and Negotiators

In previous chapters, we presented the first and second tier of the *Agilos* architecture, focusing on a basic client-server based application. However, in recent years there has been a trend that led to a paradigm shift in the types of applications being developed. As computing power gets less expensive and more pervasive, complex distributed applications tend to distribute servers to multiple hosts over a local or wide area network, serving multiple clients. This introduces a new dimension of adaptation choices including on-the-fly switching among servers. We observe that in an application involving multiple servers and clients, there may be instances when the best option may be to switch to a new server which better suits the available resources. For example, consider the *OmniTrack* application in which the client receives and displays an uncompressed video stream from a server. When bandwidth becomes restricted, local adaptations such as reducing the frame rate may be insufficient to provide an acceptable level of QoS with respect to the critical application parameter. Rather, the best adaptation may be to switch to a compressed video format which requires less bandwidth, such as Motion JPEG or MPEG. However, the current server may not be able to provide video in Motion JPEG format, either due to a software or hardware insufficiency. If there were another server providing the equivalent video in MPEG format, the best client adaptation strategy would be to switch to the compressed video server.

In this chapter, we examine the third tier of the *Agilos* architecture [26]. This tier allows for performing adaptation by reconfiguring mappings between multiple clients and servers, by switching off from one server to another which best suits the needs of the client. This tier of

adaptation is made possible by the introduction of a series of negotiators on the clients and servers, as well as a gateway which facilitates the selecting and switching of servers on behalf of the client. The third tier is designed to be generic to all applications involving multiple clients and servers, with an open interface for application-specific extensions. Such an interface between the third tier and the application is defined in Section 8.4.

7.1 Overview of A Gateway-Centric Architecture

Figure 7.1 shows a typical layout model of the *Agilos* middleware architecture working for a single client. On one end, the model consists of a client with a single connection to the gateway. On the other end, an array of servers are connected in a star topology to the gateway. We refer to such a collection of servers and gateway as a *facility*. A facility consists of multiple servers and a gateway, and collectively serves a single client. At any given time, the client is being serviced by one of the servers in the facility. However, which server is servicing the client is dynamically chosen by the gateway. The model can be extended to serve multiple clients by creating an additional connection from each new client to the gateway. This gives the gateway a central role of the entire architecture, mediating the mappings among multiple clients and servers at any time.

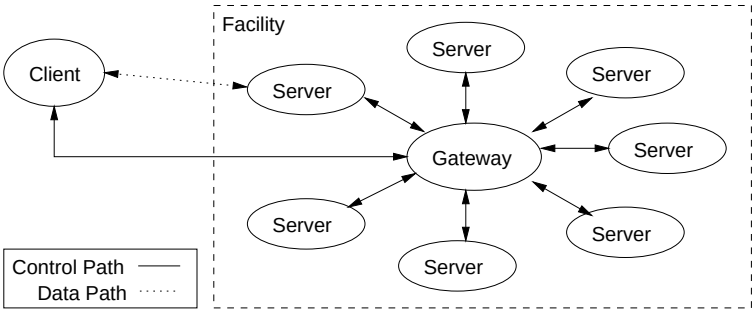


Figure 7.1: A Facility Serving a Single Client

All of the arrows shown in Figure 7.1 represent control paths. The actual application data does not follow the same paths as the control connections. All application-specific data is streamed directly from the server and the client, bypassing the gateway, and thus reducing the performance

overhead introduced by the gateway-centric architecture. This is an important distinguishing characteristic between a typical star network topology, such as a 10BaseT Ethernet, and our gateway-centric architecture. Another advantage of such a design is that, although the gateway and the servers are conceptually tightly grouped, this need not be the case geographically. The servers may be either on the same subnet as the gateway or sparsely dispersed over a wide area network, depending on the application's needs.

The purpose of the facility is to merge multiple services from multiple servers into a single interface. Entry into the facility and coordination between the clients and the servers is performed through the gateway. An example facility is the set of multiple tracking servers in *OmniTrack*, shown in Figure 7.2. In this application, each server is equipped with a video camera, and each camera has a different view of a scene. Although a client may only receive video from one server at a time, the client can dynamically switch cameras depending on the angle of view, the server workload and the video format served.

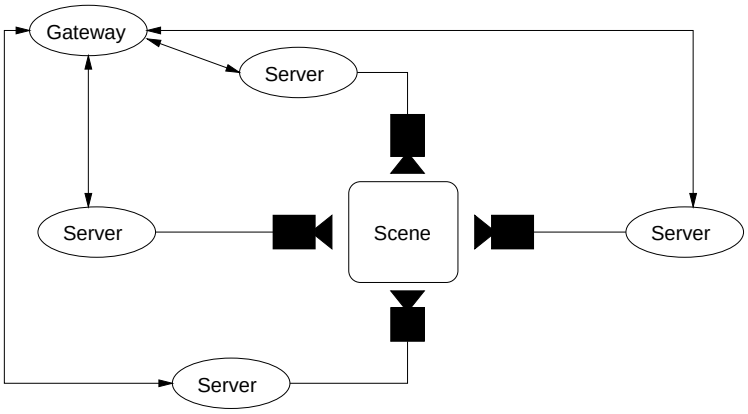


Figure 7.2: The Gateway-Centric Facility in *OmniTrack*

There are several objectives that the gateway-centric design of *Agilos* attempts to achieve. These goals are the following:

1. *Flexible*. The facility should not be restricted to a predetermined static layout of servers. Rather, the gateway should be able to handle a dynamic environment which can potentially differ radically from one moment to the next. It should be able to handle dynamic additions

and removals of servers, as well as unexpected server failures, either due to a server host failure or a network failure.

2. *Responsive*. The amount of time taken to switch servers for a client should be kept minimal. Furthermore, the amount of time for the facility to service a client's request should grow no worse than linearly in the number of servers within the facility. The gateway should not become a bottleneck in overall system performance.

3. *Generic*. The architecture should not incorporate any application-specific knowledge into its protocols. Rather, it should be highly flexible to allow new applications to be easily extended from the existing application-neutral design of the architecture. Any application-specific information should be efficiently handled by the architecture and then interpreted using an application-specific knowledge base.

4. *Transparent to Application Development*. There should be very little difference between developing an application with a highly reconfigurable server and one with several non-reconfigurable servers which can be dynamically switched by the facility. All of the client-side server switching protocols should be performed within the middleware with only a minimum of code required by the application to switch servers, such as code to switch from one active socket to another. For example, consider a facility with two types of servers: one that provides uncompressed video and one that provides compressed video. When the client switches between formats, the application developer should be able to program as though there is only one single server at all times, regardless of the server switching taking place in the middleware. From the developer's perspective, there exists one server which is capable of switching from one format to the other.

To demonstrate how the gateway-centric architecture works to incorporate these objectives, the next three sections present the design of the client and server negotiators, as well as the internal mechanisms in the gateway.

7.2 The Client Negotiator

The assumption made in the previous chapters was that a single client-server relationship is capable of providing the adaptation choices that are required to meet the critical performance criterion. However, as previously stated, this is not always a realistic assumption. To handle reconfigurations which require switching servers, an additional component is added to the *Agilos* middleware called the negotiator.

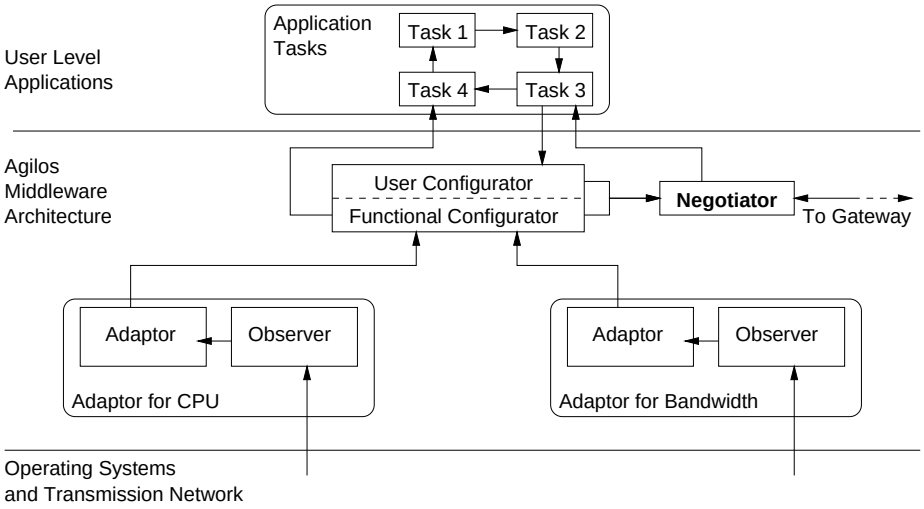


Figure 7.3: Client Architecture

The role of the negotiator in *Agilos* on the client side is shown in Figure 7.3. The negotiator interacts with both the configurator in *Agilos* on the client side, and the centralized gateway. The interaction between the configurator and the negotiator can be seen in the pseudocode in Figure 7.4a. This pseudocode represents the logic performed when the configurator needs to perform an application reconfiguration that may involve interactions with the centralized gateway. Via the inference engine, the configurator generates an control action to be performed within the application. If the action can be performed locally, and upcall is made through the application control interface via CORBA to perform such an action. If the action requires a switch of servers and thus involves the gateway, it determines the conditions required by the new server to fulfill the action and sends the action and conditions to the negotiator. For example, if the action is to switch to a server with a particular video format, the condition would be the format the new server should provide.

Upon receipt of the configurator's request, the negotiator performs the pseudocode in Figure 7.4b. The wrapping process converts the application-specific QoS request into a generic form which can be transmitted by the negotiator to the gateway. Depending on the application, some requests may not require a reply from the gateway. If the request does require a reply, the negotiator waits for a reply from the gateway. When it arrives, the negotiator unwraps the reply to transform it from a generic reply to an application-specific action. This action will include the server to use and the precise QoS conditions met by the server. These conditions are necessary so that the negotiator can perform the correct reconfiguration for the application. For example, if the client requests compressed video, this may include any one of a number of acceptable formats, such as MPEG and motion-JPEG. The gateway will reply with a server to use and the specific format with which that server is transmitting. The negotiator performs the server switch and then reconfigures the application to use the new server by an upcall through the application control interface.

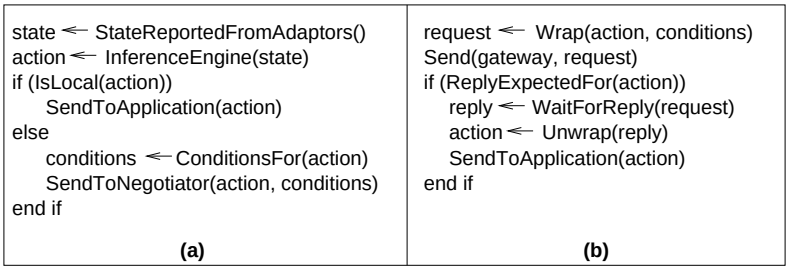


Figure 7.4: Reconfigurations Assisted by the Client Negotiator

The negotiator and configurator are independent asynchronous components within the *Agilos* architecture. Once the configurator sends the request to the negotiator it resumes execution without blocking. This allows for the client to continue operating while the negotiation process takes place in the background asynchronously. This is done because the negotiation process may require more time than a single client-server interaction. By performing negotiation in the background, the QoS degradation in data continuity caused by switching can be minimized.

All of the protocols described thus far fall into the category of functional configurations, which is performed by the functional configurator. However, another level of reconfiguration is required to handle a degraded degree of user satisfaction, which usually is related to the perceived QoS

from the user's point of view. To this end, the configurator also includes the user configurator. As introduced in Section 5.3.6, this component performs reconfigurations when perceived QoS is degraded for reasons not detectable by the *Agilos* adaptors. For example, consider a facility which is providing a live broadcast of a football game. Each server within the facility provides a different view of the football field. At some time, the view from the client's current server may be blocked, thus giving the end user a poor view of the game. Although the video itself may be displayed at peak quality, the user's perceived QoS is degraded by the video content. In this scenario, the user reports his dissatisfaction to the application which then sends a message to the user configurator. The user configurator then determines what action to take and passes that along to the negotiator. In this way, the configurator is capable of providing the best available QoS regardless of the cause of degradation. Furthermore, the negotiator can perform equivalently for all requests whether the request is triggered by a functional or user-initiated reconfiguration.

7.3 The Server Negotiator

As previously stated, each server can be designed as though it were the only server in the standard single client-server model. To make the server perform within a facility should require only a minimal set of additions to the server. This is accomplished by including the server negotiator.

Unlike the client, even there are observers that actively monitors resources on the server side, there are no server side adaptors. In addition, the server side configurators are functionally simplified in the current implementation of *Agilos*, to the extent that they only receive adaptation decisions from the client and execute these decisions. We label these server configurators as *passive*, since they do not implement internal mechanisms to actively make decisions to adapt the application. Instead, they follow the decisions made on the client side and execute those control actions on the server side of the application. This is so for two reasons. First, our model assumes a smart client and a dumb server. Thus, the server may be incapable of performing dynamic reconfiguration changes. Second, a single server may be serving multiple clients. An adaptation which

is optimal for one client may degrade the QoS for another client. Trying to find the optimal server configuration over all clients leads to tractability issues. Thus, all application-specific adaptations should be performed by the clients. An illustration of the simplified implementation of the *Agilos* architecture on the server side is shown in Figure 7.5.

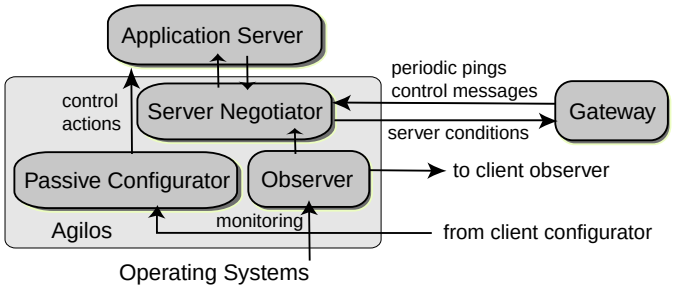


Figure 7.5: The Implementation of *Agilos* on the Server

The server negotiator performs a similar task as the client negotiator: to communicate between the server and the gateway. The negotiator serves four purposes. First, when the server comes online, it announces the server's existence to the gateway and provides the gateway with a list of all conditions this server meets. For example, a video server may tell the gateway the format with which it is transmitting. Second, it keeps the gateway informed of dynamically changing server conditions. For example, the server may periodically inform the gateway of the CPU load reported by its observer. This would aid the gateway in performing load balancing. Third, the negotiator responds to periodic pings from the gateway, with ten seconds as the interval between consecutive pings in the current implementation. This is done so the gateway will know when a server has unexpectedly terminated. Fourth, the negotiator handles control messages which arrive from the gateway. These messages fall into one of two categories: client updates or client service requests.

Client updates are performed to provide and retract authentication of clients to the server. One potential pitfall of our model is the ability for an application to circumvent the gateway to receive service from one of the facility's servers. For example, an application developer may discover through experimentation that the gateway always assigns the application to a specific server. The developer may then try to code his application to take advantage of this fact and bypass the gateway.

However, this leads to several problems. First, if the facility configuration should change, the client would no longer be suited for using the facility. Second, bypassing the gateway circumvents the facility-wide QoS configuration capabilities of the gateway. For this reason, the servers will only service clients which have been authenticated in advance by the gateway. Similarly, servers will discontinue service to a client which has been unauthenticated by the gateway. Authentication removals typically occur as part of the server switching process. As a client is switching from one server to another, the client must be authenticated on the new server and unauthenticated on the old one. However, because the server switching process takes time, an unauthenticated client is granted a short grace period during which it can still use the server. This allows the server switching process time to complete while in parallel the client can still receive uninterrupted data from the current server.

7.4 The Gateway

The gateway serves as the centerpiece in the connection between each client and the facility. It is responsible for receiving client reconfiguration requests and processing the requests for the client. These requests can be satisfied either through switching servers or through server reconfiguration as described in the previous section.

The gateway's architecture can be seen in Figure 7.6. The gateway maintains two state tables, one for the clients and one for the servers. Between these is a mapping table which maintains which clients are being serviced by which servers. This table is updated by the client/server matching reconfiguration component.

The matching reconfiguration component is invoked whenever the client requests a new server. As described in Section 7.2, the gateway receives a request from the client when a server switch is necessary. This request consists of an application specific action and a set of conditions. The matching component assigns a ranking to each server based on the action, conditions, client state, and server state. This is done by using an fuzzy inference engine identical to the one used by the

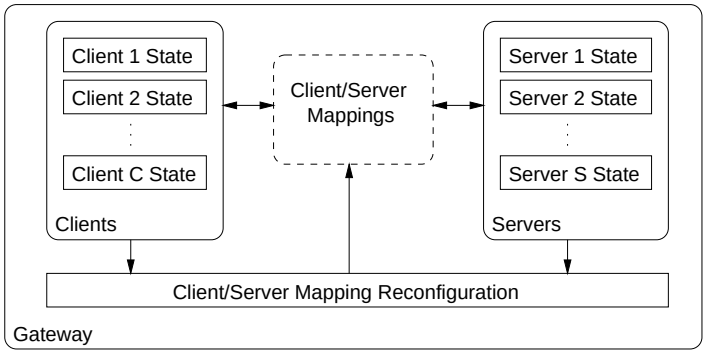


Figure 7.6: The Architecture of the Gateway

configurators but with a different rule base. The fuzzy inference engine takes in a preprocessed representation of the server state and returns a ranking for the server. For example, a rule for moving left within an *OmniTrack* application may look like this:

```
if (server_direction is left) and (server_angle is close)
   then server\_ranking is high;
```

In this example, *server_direction* and *server_angle* are preprocessed values derived from the actual location of client's current view and the server's view and the distance between them. The better a server matches the criteria for the action, the higher the server's ranking will be. Additional information can be encoded into each rule to create a more robust rule base. For example, such information may include the server's workload and available bandwidth. Figure 7.7 shows the logic performed by the gateway when switching servers for a client.

Conceptually, one can think of the gateway's role in a server switch as follows: The client uses a fuzzy inference engine to decide what action to take. The inference engine state is then sent to the gateway which completes the inference engine logic where the client left off. This approach has several advantages over performing all of the logic within the client. First, all state concerning the facility need only be maintained in one place, the gateway. The client does not need to dynamically keep track of what servers are available or how to best rank each server. Second, the client is spared the burden of negotiating a connection with the new server since all of the server negotiation is handled by the gateway. The client need only wait for a reply from the gateway then switch to the server indicated by the reply. Third, it allows for smart reconfiguration by the server without

```
<action, conditions> <= Unwrap(request)
ranked_servers = {}
for each server S
        state <= ProcessState(action, conditions, client state, S state)
        ranking <= InferenceEngine(state)
        ranked_servers <= InsertOrdered(S, ranking, ranked_servers)
end for

U<=CurrentServer(client)
accepted<= false
while (not accepted and ranked_servers <> {})
        S<= First(ranked_servers)
        accepted<= SendAuthenticationRequest(S, client)
        if (accepted)
                SendUnathentication(U, client)
                conditions<=ConditionsForServer(S)
                reply <= Wrap(S, conditions)
                Send(client, reply)
        else
                ranked_servers <= RemoveFirst(ranked_servers)
        end if
end while
```

Figure 7.7: Gateway Server Switching Protocol

client initiation. For example, if the client's server should become overloaded or unexpectedly shut down, the gateway can automatically move the client to a different server without waiting for a switch request from the client.

7.5 End-to-End Negotiation Protocol

When a reconfiguration that requires switching servers is initiated, the negotiators on both the server and the client side are used to negotiator the new server from the gateway. The end-to-end negotiation protocol between the negotiators and the gateway is shown in Figure 7.8. This protocol is described in the following six steps:

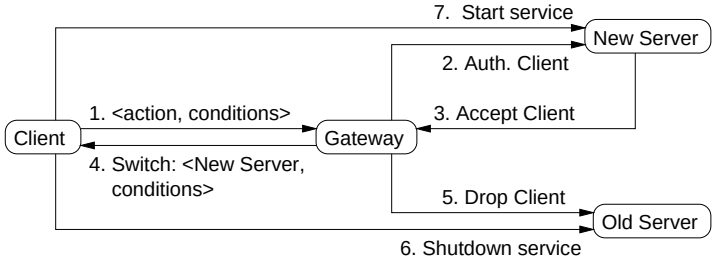


Figure 7.8: End-to-End Negotiation Protocol

1. In the first step, the client negotiator sends the gateway the action it needs to perform and a list of conditions the action must satisfy. Typically, the action requested may include the selection of a new server from the facility. After the requests from the client negotiator are received, the gateway then ranks available servers within the facility using a fuzzy inference engine and a rule base, presented in the previous section.

2. In the second step, on behalf of the client, an authorization request is sent from the gateway to the negotiator of the top ranked server. In the third step, this server immediately send a reply to the gateway with either an acceptance or a denial of service. If a denial is received, the gateway then cycles through steps 2 and 3 in decreasing order from the top ranked server until a server accepts the client.

3. When the gateway returns the selected server in the fourth step, it must also provide a list of conditions the server satisfies so that the client application can adjust itself to the conditions met by the server. For example, the client's action in step 1 could be to switch to a new format and the conditions could include a list of acceptable formats. The gateway would then return not only the new server but also the specific format from the list which the server satisfies.

4. After sending the switch notification message to the client negotiator in the fourth step, the gateway immediately proceed to send a "drop client" request to the original server that serves the client, so that the server may release resources used for serving contents to this client. This consists of the fifth step of the protocol.

5. Upon receiving the notification message from the gateway, the client negotiator immediately executes the sixth step, which shuts down all network sockets and associated resources on the client related to the original server.

6. Finally, the client establishes a new connection with the new server, with the assistance of the negotiators on both sides.

7.6 Summary

In this chapter we presented a the design of the third tier of the *Agilos* architecture, namely, the gateway and negotiators. We take a gateway-centric architecture, which consists of multiple server negotiators and a centralized gateway. This facility provides the user with several points of entry into a single composite service with each server satisfying a different set of QoS conditions. We then proved that this architecture could be easily and effectively implemented to provide a base for application-specific extensions. We will postpone experimental results to Chapter 10. We will also discuss the actual interface to the application-specific extensions in Chapter 8.

Chapter 8

Application Deployment Interface

8.1 Overview

The previous chapters have presented various theoretical analysis and algorithmic designs about the *Agilos* middleware architecture. These chapters may be summarized as follows: The *Agilos* architecture is designed to support application-aware Quality of Service (QoS) adaptations in distributed multimedia applications. Such applications are complex in nature, opening up a wide variety of adaptation choices and configuration possibilities. Different from related work in middleware architectures, the *Agilos* architecture attempts to exert active control of these applications, rather than transparently providing services by internal reconfigurations.

From this chapter, we progress to discuss implementation-related issues of both the *Agilos* architecture and *OmniTrack* application. Implementation-wise, the *Agilos* architecture emphasizes the following two objectives. First, *Generic applicability*. No application-specific knowledge should be incorporated in the basic design of *Agilos* components, so that a wide variety of applications may be deployed under the control of the middleware. Second, *Ease of deployment.* The interaction between the application and *Agilos* should be designed so that required modifications to the application are minimized.

This chapter presents the design of the *Application Deployment Interface* (ADI) that seamlessly supports the two implementation objectives of the *Agilos* architecture. With respect to the application to be deployed on top of *Agilos*, such an interface can be seen as a wide collection of

functional programming interfaces for applications (APIs), deployment strategy and procedures, as well as required deployment tools and utilities. For an easy and rapid deployment, such an interface should be designed to be *simple* and easy to carry out. For generic applicability requirements, such an interface should be designed to integrate only application-neutral elements into the interface specifications. All application-specific components within the *Agilos* architecture should be derived from an *implementation* of such an interface specification. A clearly defined division line is thus drawn between application-neutral and application-specific parts of the middleware.

The rest of the chapter is organized as follows. Section 8.2 gives a step-by-step instruction on the required deployment procedure and strategies, which are part of the Application Deployment Interface. Section 8.3 defines the control interface exported by the application and examples of its IDL specification. Section 8.4 discuss application deployment issues related to gateways in order to facilitate wide-scale reconfigurations in multi-client multi-server mappings. Section 8.6 summaries the chapter.

8.2 Deployment Procedures and Strategies

In order for the *Agilos* middleware architecture to actively exert control of the applications, there are several general steps that the applications should follow. These steps are outlined in details in this section.

8.2.1 Preparation for Component-Oriented Interfaces

The goal of *Agilos* middleware architecture is to be able to deploy the widest range of applications possible under its control and support. Because of the heterogeneous manner that these applications are built, the first step before any actual deployment should be to prepare the application for component-oriented interfaces. The actual procedures that to be followed for such preparation vary from application to application, and can be classified into three categories depending on how the application was originally implemented. We list these categories as follows.

Componentized Applications

Since the start, the *Agilos* architecture was exclusively designed and implemented with the Common Object Request Broker Architecture (CORBA) [8] in mind, though the algorithmic design and theoretical results also apply to other component models, such as the Microsoft Component Object Model [9]. This dictates the following: First, all interactions among internal middleware components are designed to be carried via CORBA. Second, the control actions should be delivered via CORBA to the applications. Third, the observation and monitoring of application-specific QoS parameters are communicated via CORBA to the observers in *Agilos*. For these reasons, if the applications have already been designed to implement any CORBA interfaces for the purpose of remote procedure calls across process boundaries and application components, the first step of preparing for component-oriented interfaces is automatically complete. We refer to these applications *CORBA-aware* applications. The *Agilos* architecture naturally supports CORBA-aware applications well.

Applications that may be CORBAfied

Most of the "legacy" applications, including some mission-critical complex multimedia applications such as the *OmniTrack* application, are not inherently CORBA-aware. However, it may be quite straightforward to CORBAfy some applications by explicitly instrument these applications at the source level, as long as they present some favorable properties. These properties include:

1. No major conflicts with the header files of a specific CORBA implementation[1]. This includes redefinitions of function or constant names, as well as programming language incompatibilities in order to include those files (namely languages except for C, C++ and Java). Most applications conform to this requirement.

2. When a graphical user interface is required in the application, such an interface is message-driven and implemented with a message processing loop. Examples of such message-driven

[1]In our implementation, we use ORBacus 3.1.3 as underlying CORBA support, thus the application should be compatible with all header files and libraries in ORBacus 3.1.3.

GUI programming models are the X event loop or the Windows message loop. This is required to integrate the CORBA event handler, which is normally implemented within `boa -> impl_is_ready(...)`. In the *OmniTrack* application, we are able to CORBAfy the application by adding asynchronous CORBA event handling calls inside the Windows message loop. Similar methods may also be used in other applications with a message loop that handles GUI messages. In details, we consider the following scenario.

Without loss of generality, let us assume that the application to be CORBAfied is a GUI application running under Windows, which is the case for *OmniTrack*. Generally, such an application should include a Windows message loop, as in the following example of an Multiple Document Interface (MDI) Windows application:

```
while (GetMessage( &msg, NULL, 0, 0))
{
    if (!TranslateMDISysAccel (hWndMDIClient, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Such an application can be augmented conveniently by inserting a function call to the CORBA event handler, so that incoming CORBA events could be handled simultaneously.

```
while (GetMessage( &msg, NULL, 0, 0))
{
    if (!TranslateMDISysAccel (hWndMDIClient, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    // query for any incoming CORBA events for 'timeout' ms
    TCorbaInstance -> react (timeout);
}
```

3. The source files for the application are required to be recompiled in order to link the definition of a critical class `TCorba`, which implements actual details that handle CORBA initialization, shutdown and incoming CORBA events. `TCorbaInstance`, in the code

shown previously, is an instance of class *TCorba*. The constructor and `react` functions of this class are defined as follows.

```
// Initializing CORBA for the entire application

TCorba::TCorba()
{
    int dummy = 0;

    // Step 1: create ORB and BOA

    orb = CORBA_ORB_init(dummy, 0);
    boa = orb -> BOA_init(dummy, 0);

    orb -> conc_model(CORBA_ORB::ConcModelReactive);
    boa -> conc_model(CORBA_BOA::ConcModelReactive);

    // Step 2: Locate CORBA reactor (default to SelectReactor)
    //         for reacting to CORBA messages in Windows message loop

    reactor = OBReactor::instance();
    boa -> init_servers();

    // Step 3: Locate the CORBA Property Service

    ifstream in("propserv.ref");
    char str[1000];
    in >> str;

    CORBA_Object_var propservObj = orb -> string_to_object(str);
    assert(!CORBA_is_nil(propservObj));

    PropertySet = CosPropertyService_PropertySetDef::_narrow(propservObj);
    assert(!CORBA_is_nil(PropertySet));

    // CORBA initialization completed.
    return;
}

// reacts to incoming CORBA events in the application
void TCorba::react (int WaitPeriod)
{
    CORBA_Long timeout = (CORBA_Long) WaitPeriod;
    reactor -> dispatchOneEvent (timeout);
}
```

8.2.2 Installing Probes for Application-Specific QoS Parameters

In order to appropriately observe all application-specific QoS parameters, instrumentation code, which we refer to as *probes*, should be plugged into the application. These probes can only be implemented by the application developer, since the *Agilos* middleware architecture does not have any knowledge about specific methods of instrumentation with respect to QoS parameters in the application. Thus, the installation of such probes must be done as part of the deployment procedures for the application.

However, the implementation of application-specific probes may be designed to be not completely ad-hoc. For this purpose, a generic class `TMetrics` is defined as an abstract class and a skeleton implementation of all application parameter probes, and any application-specific implementation may be derived from the `TMetrics` class. Instances of such a derived class can then be statically linked in the application.

The implementation of *TMetrics* class takes advantage of the CORBA Property Service extensively. The CORBA Property Service is designed as place holder for an unlimited series of name/value pairs, representing any CORBA properties. Both CORBAfied applications and *Agilos* middleware components can communicate with the CORBA Property Service via standard CORBA events. The utilization of CORBA Property Service as a bridge completely detaches the middleware from the application. On one hand, the applications, assisted by the *TMetrics* class, are responsible of collecting important application-level parameters and reporting them to the CORBA Property Service at predetermined constant intervals. On the other hand, the middleware components are responsible to poll these observed values from the CORBA Property Service whenever there are needs to do so. A prototype illustration of the TMetrics class definition is as follows.

```
class TMetrics
{
    CORBA_String_var s;
    CORBA_Any any;
    char * MetricName[MaxMetrics];
    long MetricValue[MaxMetrics];
    int CurrentNumMetrics;
```

```
public:

    TMetrics();

    ~TMetrics();

    void Add(char * name);
    void Update(char * name, long value);
    void Commit(void);
    long Retrieve(char * name);
};

extern TMetrics * TMetricsInstance;
```

Whenever the application has observed values for a specific QoS parameter, it propagates the values to the CORBA Property Service. For example, the next code segment communicates the current frame size, tracking duration and number of active trackers to the CORBA Property Service.

```
// Report to the CORBA Property Service
TMetricsInstance -> Update ("TrackingDuration", clock.measure());
TMetricsInstance -> Update ("FrameSize", v->width() * v->height());
TMetricsInstance -> Update ("NumOfActiveTrackers",
           (long) TrackerManager -> activeList.n_trackers());
TMetricsInstance -> Commit ();
```

On the other hand, the observer polls these observed values from the CORBA Property Service with a similar mechanism, as shown in the following implementation as the observer monitors the frame size in the tracking client of *OmniTrack*:

```
// Collects the Frame Size data from the Property Service
try
{
    s = CORBA_string_dup("FrameSize");
    any = TCorbaInstance -> PropertySet -> get_property_value(s);
}
catch (const CosPropertyService_PropertyNotFound&)
{
    assert(false);
}
CORBA_ULong value;
any >>= value;

// 'value' is the observed value for the frame size
...
```

8.3 Application Control Interface

Once the application is prepared for deployment under the control of the *Agilos* middleware, a key procedure for the deployment is to define the *Application Control Interface* within the application, as well as properly register such an interface with the *Agilos* middleware.

8.3.1 Interface Definitions

The *Application Control Interface* is a set of functions that the *Agilos* middleware layer may call in order to tune and reconfigure the applications on their behalf. This set of functions can be divided into two groups:

1. *Parameter-Tuning Interfaces*: The functions exported in this group serve to fine tune individual application-level quantitative parameters. In the *OmniTrack* example, these parameters include the frame rate, image quality, compression ratio, number of active trackers, etc. These functions are used by the quantitative configurator to tune internal QoS parameters in the application.

2. *Reconfiguration Interfaces*: The functions exported in this group are responsible for the activation of reconfiguring actions, which could be the activation of compression for live video or the switching of tracking servers in the *OmniTrack* example. In a complex web server application, the reconfiguring action may be the switching from serving both dynamic and static web pages to serving only static web pages, in order to reduce CPU overloading. The functional configurator uses such interfaces to activate reconfigurations within the application.

As an example, a simplified version of the Application Control Interface for *OmniTrack* may be written as follows, in the form of CORBA IDL:

```
interface OmniTrack_Control
{
    // For parameter-tuning controls
```

```
    void DecreaseTrackedRegion();
    void IncreaseTrackedRegion();
    void AdjustImageSize(in unsigned short width, in unsigned short height);

    // For reconfigurations with respect to tracking clients

    void AddTracker();
    void DropTracker();
    void ReplaceTracker();
    void MJpegCompress();
    void RawVideo();
    void BlacknWhiteImage();

    // For reconfigurations with respect to gateways

    short add_server(in string server);
    short drop_server(in string server);
    short switch_server(in string server, in string format);
};
```

8.3.2 Interface Registration with *Agilos* Middleware

Once the Application Control Interface is defined in the form of CORBA IDL, the *Agilos* middleware layer should be notified so that it is aware of such an interface definition. Such notification is referred to as *interface registration*.

In order to complete the interface registration process, each Application Control Interface should have its unique identifier. Each function within the interface should have its own associated identifier, corresponding to a unique linguistic variable in the fuzzy linguistic rule base. In the above example, the identifier for the Application Control Interface is `OmniTrack_Control`. The associated linguistic variable for the function `AdjustImageSize` is `size`, and for `RawVideo` it is `raw`. Such an interface corresponds to the following linguistic rules:

```
if (cpu is low) and (rate is high) then rateaction:= raw;
if (cpu is low) and (rate is moderate) then rateaction:= size;
if (cpu is low) and (rate is low) then rateaction:= blacknwhite;
if (cpu is moderate) and (rate is high) then rateaction:= raw;
if (cpu is moderate) and (rate is moderate) then rateaction:= size;
if (cpu is moderate) and (rate is low) then rateaction:= blacknwhite;
if (cpu is high) and (rate is low) then rateaction:= compress;
if (cpu is high) and (rate is moderate) then rateaction:= size;
if (cpu is high) and (rate is high) then rateaction:= compress;
```

```
if (cpu is low) and (rate is moderate) then cpuaction:= droptracker;
if (cpu is low) and (rate is high) then cpuaction:= droptracker;
if (cpu is low) and (rate is low) then cpuaction:= droptracker;
if (cpu is moderate) and (rate is high) then cpuaction:= adjustregion;
if (cpu is moderate) and (rate is moderate) then cpuaction:= adjustregion;
if (cpu is moderate) and (rate is low) then cpuaction:= adjustregion;
if (cpu is high) and (rate is low) then cpuaction:= addtracker;
if (cpu is high) and (rate is moderate) then cpuaction:= addtracker;
if (cpu is high) and (rate is high) then cpuaction:= addtracker;
```

The correspondence between the linguistic variables, such as `raw` and `size`, and interface functions, such as `RawVideo`, is listed in a plainly formatted specification file. Such a file is thus read by a source code generator to automatically generate the required source files for the middleware to appropriately access these functions when the corresponding control actions are activated by the configurator. It is the application's responsibility to initiate such a specification file, along with the linguistic rules and IDL definitions for the application control interface. An example of the specification file is listed as follows.

```
id  function              lingvar          type    min     max

1   AddTracker            addtracker       config
2   DropTracker           droptracker      config
3   AdjustImageSize       size             tune    10000   76800
4   DecreaseTrackedRegion adjustregion     tune
5   IncreaseTrackedRegion adjustregion     tune
6   BlacknWhiteImage      blacknwhite      config
7   MJpegCompress         compress         config
8   RawVideo              raw              config
```

8.4 Deployment in the Third Tier

As *Agilos* middleware components, the gateway and negotiators are responsible to facilitate wide-scale reconfigurations with respect to multi-client and multi-server relationships. Such reconfigurations include server switching capabilities according to relevant information such as the server workload, video format or video properties. Before application deployment, we assume that the application only includes an implementation of the basic client-server relationship, and all functional enhancements are implemented in the *Agilos* middleware. In this section, we discuss

the required deployment procedures so that a basic client-server application may be extended to a multi-client multi-server environment.

8.4.1 Gateway

If we examine the interface between the gateway and a client-server based application, we will observe that some of the functionalities are application-specific, while the rest are generic to all applications. We thus define a base class with virtual or pure virtual functions that define all application-neutral functionalities within a gateway. When a new application is to be deployed, an application-specific class is then defined to inherit this base class, with all virtual functions redefined and all application-specific functions added.

Such solutions for *Agilos* are proposed based on a comparison study upon the advantages and drawbacks of various alternative solutions. There are two primary concerns. First, how easy it is for the application to be deployed? It is best if the efforts of such deployment is minimized. For this purpose, the interface should be as generic as possible, and a runtime binding (late binding) mechanism is preferable than a compile-time binding (early binding). Second, the performance overhead should be minimized. Our solution is a balance and tradeoff between the two design objectives. With a generic base class that all application-specific implementations inherit from, we are able to build all internal mechanisms within the gateway so that only virtual functions in the generic base class is called at compile time, while functions in the actual application-specific implementation are called at runtime.

In our current implementation of *Agilos*, the base class that defines a generic interface is as follows.

```
class Gw2app
{
public:
    // Initialize the gateway-application interface
    virtual void initialize() { };

    // Inform the gateway of a new application client
    virtual ClientHost *CreateClientHost(Host *, char *);
```

```
    // Inform the gateway of a new application server
    virtual ServerHost *CreateServerHost(Host *, char *);

    // Process the requests sent from an existing client
    virtual void ProcessRequest(ClientHost *, AssertionList *);

    // Process the requests sent from an existing server
    virtual void ProcessRequest(ServerHost *, AssertionList *);

    // Relocate the client to another server
    virtual void RelocateClient(ClientHost *);

    // Request the best server based on application-specific criteria
    virtual ServerHost *RequestBestServer(ClientHost *, AssertionList *) = 0;

};
```

As an example of how applications may extend such a generic interface, we look at the *OmniTrack* application. In *OmniTrack*, a new class named `Gw2vt` is declared as a derived class inherited from the base class `Gw2app`. In this class, all virtual functions in the base class are redefined to implement functionalities specific to *OmniTrack*:

```
class Gw2vt : public Gw2app
{
public:
    void initialize();
    ClientHost *CreateClientHost(Host *, char *);
    ...
    ServerHost *RequestBestServer(ClientHost *, AssertionList *);

    int GetServerLoad(VTServerHost *, TrackedSubject *);
};
```

We notice that in addition to all virtual functions declared in the base class `Gw2app`, the class `Gw2vt` defines a new function `GetServerLoad`, which is specific to *OmniTrack*. Generally, the application is free to define any new functions in the class. This relaxed interface provides additional flexibilities.

8.4.2 Negotiators

The application deployment interface for the negotiators is designed similarly as that for the gateway. An application-specific base class, `App2neg`, is defined to provide a basic interface to

the application:

```
class App2neg
{
public:
    virtual void ProcessReply(const char *) = 0;
    virtual void ServerErrorOccurred() = 0;
};
```

The application then defines a derived class that provides additional functionalities and implements the basic interface in `App2neg`. In the *OmniTrack* example, such a derived class is named `Vt2neg`, interfacing the tracking client and the negotiator on the client side:

```
class Vt2neg : public App2neg
{
public:
    short RequestFormat(const char *);
    short RequestTurn(const char *);
    short RequestMove(const char *);
    short RequestMoveWithFormat(const char *, const char *);
    void RequestToken();
    void ReleaseToken();
    void SetTrackedSubject(const char *, const char *);
    void ProcessReply(const char *);
    void ServerErrorOccurred();
};
```

Similar to the gateway, once the generic base class and all virtual functions are installed, the implementation of a application-neutral negotiator may simply call these virtual functions in the base class, which will be bound to the application-specific implementations in the derived class at run time.

8.5 An Example of Application Deployment

Having presented all the required steps and interfaces for application deployment in the previous section, we provide a detailed example explaining how these deployment strategies and interfaces are used for deploying a simple video-on-demand application, how application QoS parameters are specified to the middleware components, and how control interfaces may be defined.

8.5.1 Identifying Application QoS Parameters

The initial task in the deployment process is to identify all tunable and observable application QoS parameters, and all reconfiguration possibilities. In a basic video-on-demand (VOD) application, there are at least two tunable parameters: frame rate and image resolution. There are at least one important reconfiguration choice: media format (e.g., with or without audio track, MPEG, Motion JPEG or ASF).

8.5.2 Preparing Component-oriented Interfaces

Following the deployment procedures outlined in Section 8.2, we consider our VOD application as a "legacy" application, and not inherently CORBA-aware. Thus, it needs to be CORBAfied with the following steps:

1. From the *Agilos* source collection, obtain the source files that contain definitions for the class `TCorba`.

2. Create a new instance of class `TCorba` in the main program of our VOD application, and export its controls interface to the *Agilos* middleware, as in:

```
#include "TCorba/TCorba.h"
...
   TCorbaInstance = new TCorba;
   TCorbaInstance -> ExportControlsInterface();
...
```

3. Instrument the main GUI event handling loop of the application and add the CORBA event handling calls, as in the following example in a Windows-specific implementation of our VOD application:

```
while (GetMessage(&msg, NULL, 0, 0))
{
   ...
   TCorbaInstance -> react (timeout);
}
```

8.5.3 Installing Probes for Observable Parameters

Following the procedures outlined in Section 8.2.2, we can monitor internal QoS parameters within the VOD application by following the steps below.

1. From the *Agilos* source collection, obtain the source files that contain definitions for the class `TMetrics`.

2. Create a new instance of class `TMetrics` in the main program of our VOD application, and register the names of all internal parameters to be monitored:

```
#include "TMetrics/TMetrics.h"
   ...
   TMetricsInstance = new TMetrics;
   TMetricsInstance -> Add("FrameRate");
   TMetricsInstance -> Add("Resolution");
   ...
```

3. Instrument the appropriate source files of the VOD application, so that values of internal parameters are reported to the CORBA Property Service via the `TMetrics` interface, as in:

```
TMetricsInstance -> Update ("FrameRate", CurrentFRate);
```

   where `CurrentFRate` is the result of calculating the current frame rate.

8.5.4 Defining the Application Control Interface

As our simple VOD application has two parameters to tune and one reconfiguration choice, the Application Control Interface, written in the Interface Definition Language (IDL) of CORBA, is in the following form:

```
interface VOD_Control
{
  // For parameter-tuning controls

  void TuneResolution(in unsigned long resolution);
  void TuneFrameRate(in unsigned short frate);
```

```
  // For reconfigurations
  void ActivateMPEG();
  void ActivateASF();
  void RemoveAudioTrack();
  void AddAudioTrack();
}
```

8.5.5 Registering the Application Control Interface

As presented in Section 8.3.2, *Agilos* needs to be notified of the specific Application Control Interface that the VOD application has defined. The specification file for such notification may be defined as follows:

```
id  function          lingvar           type         min           max

1  ActivateMPEG       mpeg              config
2  ActivateASF        asf               config
3  RemoveAudioTrack   noaudio           config
4  AddAudioTrack      audio             config
5  TuneResolution   resolution        tune         64000         786432
6  TuneFrameRate      frate             tune          5             30
```

Such a specification file links the actual functions in the control interface with the linguistic values in the rule base. Having defined such a mapping, an appropriate rule base and associated membership functions for the linguistic values should be defined for the quantitative and functional configurators, in the form presented in Section 8.3.2. These steps complete the basic process of deploying a new VOD application under the control of *Agilos*, if third-tier support for multiple servers and clients is not necessary.

8.6 Summary

This chapter presents the interface between a complex distributed application and the *Agilos* middleware architecture. The design objectives of such an interface is the following. First, we focus on the easy deployment of a new application under the control of *Agilos*. Second, the *Agilos* architecture should be generic enough to meet the demands of any existing legacy application

in need of adaptation support. On one hand, we believe that the application-specific rule base is similar with any flexible scripting languages that expresses functionalities, except that the rule base reconfigures the behavior of a generic inference engine. On the other hand, the interface definition method we have proposed is both specific to the application and pluggable into the middleware.

Chapter 9

Implementation in Windows NT

To continue the discussion in Chapter 8 with respect to the application deployment interfaces, in this chapter, we present detailed implementation issues of both the *OmniTrack* application and the *Agilos* Architecture. Section 9.1 gives an overview of implementation choices, Section 9.2 presents issues in the process of implementing *OmniTrack* and deploying the application on top of *Agilos*, and Section 9.3 presents issues in the implementation of *Agilos* architecture itself.

9.1 Overview

The *Agilos* middleware architecture and *OmniTrack* application are both implemented in Windows NT 4.0, using Visual C++ 6.0. We use CORBA as our component object model to facilitate interactions among middleware components and applications. The specific implementation we use is ORBacus 3.1.3 [27] from Object Oriented Concepts, Inc.

The *OmniTrack* application, an omni-directional visual tracking system, is a fully distributed, client-server based multi-threaded application. Since the tracking server needs to serve multiple clients simultaneously, each client connection is created in its own thread within Windows NT. Similarly, the tracking client needs to receive video frames from the network, while executing multiple computationally intensive tracking algorithms at the same time. This demands that the tracking algorithms are executed in a separate NT thread, while implementing close inter-thread synchronization to keep a typical consumer-producer relationship with the thread that receives

video frames from the communication channel. The tracking server can serve either live video captured by a video camera via the Matrox Meteor frame grabber card, or artificially generated animations.

The implementation of the *Agilos* architecture consists of implementation of all middleware components, including the observer, the adaptors, the configurators, the QualProbes, the client and server negotiators, and the gateway. Except for the communication between the negotiators and the gateway, which is through standard network sockets, all other interactions among components are via CORBA as well. The observer monitors the system resource availability via the Performance Data Helper library, which is a dynamic linked library available in Windows NT to assist probing various metrics in the operating system kernel through the Windows registry.

9.2 Implementation of *OmniTrack* Application

9.2.1 Migration from Unix to Windows NT

As a first step of implementing the *OmniTrack* application, we have taken XVision [11], a standalone visual tracking application originally developed on the Unix platform, and migrated the implementation to Windows NT 4.0. The original XVision implements various tracking algorithms, including the SSD (Sum of Squared Difference), line and corner algorithm. The implementation of these algorithms are platform neutral and straightforward to migrate. Therefore, in the migration process, we focus on migrating the graphical user interface and the video capture device drivers to Windows NT, while inheriting all platform-neutral tracking algorithms with only slight modifications.

There are two major challenges in the migrating process. First, for the purpose of live video capturing, the original XVision system uses various types of frame grabber or digitizer devices, and the device driver interfaces are only available on specific Unix platforms (Solaris and SGI IRIX). Most of these hardware devices are proprietary and not available under Windows NT. Second,

all source code within XVision that deals with user interface are X Windows specific, including live video display and mouse interactions. With these challenges, two solutions exist towards a successful migration. First, retain most of the original source code, and use porting layers (e.g. Interix from Softway Systems [28] or Cygwin from Cygnus Solutions [29], among others) to rapidly migrate code to Windows NT with minor modifications. Second, rewrite the source and migrate to native SDKs, in both Win32 SDK for user interface and device driver SDKs for hardware support.

Considering future extensions and the difficulty level of deployment under the control of the *Agilos* middleware architecture, we chose the latter alternative and rewrote relevant source code in order to migrate from X Windows specific code to the native Win32 SDK, as well as from proprietary hardware specific drivers in Unix to a Windows NT based interface SDK available for the *Matrox Meteor* frame digitizer card, our hardware of choice. Because of the relative simplicity of user interface, this approach was proved to be preferable and allowed much more flexibility with respect to future interface extensions or upgrades (for example, from Win32 SDK to DirectDraw SDK to enhance the video performance).

9.2.2 Extension to a Client-Server Based Application

Since the original XVision performs all of its features on the same end system, our next step of implementing the *OmniTrack* application is to extend the application to a client-server based model. With respect to the design choices, we adopted a *Thin Server* approach, where the role of server is to capture live video and to send them through the network to one or multiple clients, while the client performs all CPU-intensive tracking calculations on multiple objects. With respect to implementation, we have encapsulated all network related implementation within the `NetworkDevice` class, which serves as a replacement to the original classes that provide interfaces to the frame grabbing hardware, for example the `ColorMeteor` class. The new `NetworkDevice` class is designed to export identical interfaces to the tracking kernel compared with original hardware interface classes, so that only minor modifications are needed on the client. We used

Windows Socket 2 Interface as a basis for network programming, and utilized datagram (UDP) sockets for video transmission. We show the definition of the class `NetworkDevice` as the following.

```
class NetworkDevice : public ColorVideo, public JTCMonitor
{

    // Window handle
    static HWND hwnd;
    // The video buffer to be displayed
    static long * videoBuffer;
    int info_type;
    int map_color(Color x);
    void set_color(Color x);

protected:

    // Communication thread
    static TComm *comm;

public:

    void ReturnOnError(void);

    long * getVideoBuffer(void);
    virtual void prepareNextFrame();
    void grabNextFrame();

    NetworkDevice ();
    virtual ~NetworkDevice()
    void close();
    ...
}
```

Since the video stream received by the client via the `NetworkDevice` class may have different codec formats, we defined specific C++ classes for each codec format in order to meet specific handling requirements. These classes are derived from the generic `NetworkDevice` class, and thus inherit all basic functionalities related to socket communications. Each derived class implement an important virtual function `prepareNextFrame` defined in `NetworkDevice`, which handles the decoding process of video frames based on specific image formats and quality for each frame. As an example, we have implement the classes `RawDevice` and `MJpegDevice`, that are derived from `NetworkDevice`:

```
class MJpegDevice : public NetworkDevice
{

public:

    virtual void prepareNextFrame();

    MJpegDevice ();
    virtual ~MJpegDevice();
    ...
}
```

9.2.3 Extension to a Multi-threaded Application

We notice that in the definition of the `NetworkDevice` class, there are no member functions that are responsible for interacting with network sockets. Rather, a static instance of the class `TComm` is defined. This is the result of an enhancement to a multi-threaded client implementation, which is implemented with the assistance of the JThreads/C++ package available with ORBacus. There are two main threads of execution in the tracking client, as follows.

1. *Communication thread*. This thread is primarily implemented in the class `TComm`. It is responsible for maintaining a circular ring buffer that holds received video frames from the communication channel. It is also responsible for handling all network sockets and connection setup and teardowns.

2. *Computation thread*. This thread is the main thread of the tracking client application. It is responsible for executing all tracking algorithms in a round robin fashion, as well as all user interactions and graphical user interfaces. During each iteration that the tracking algorithms run, no other events are processed in the same thread. The tracking algorithms are executed at a fixed time interval. This is demonstrated in the following code in the implementation of the window procedure:

```
// Message 'WM_TIMER' is received every fixed interval
case WM_TIMER:
    // Prepares the next frame
    v -> prepareNextFrame();
```

```
            // Actually executes all trackers
            if (TrackerManager -> TrackerGroup -> hasChild())
                TrackerManager -> TrackerGroup -> track();

            // Display to a window
            w.show (v -> image());
        break;
```

The separation of functionalities between the two threads has a major advantage: It allows the tracking algorithms to be executed asynchronously along with the communication tasks, such as receiving video frames. This dramatically increases the tracking frequency, which is no longer tied to the frame rate that the client is able to receive from the network.

Having this advantage, the tradeoff is that the complexities of implementation is significantly increased, due to the strict synchronization requirement imposed by the shared circular ring buffer between the two threads. Obviously, the communication thread serves as a producer to inject new frames into the shared buffer, while the computation thread acts as a consumer, executes tracking algorithms and consumes existing frames in the buffer. Such a producer-consumer relationship requires extra mutexes and conditions so that the two threads are properly synchronized and critical sections are protected against concurrent access. For example, the class `TComm` is defined as follows:

```
class TComm : public JTCThread
{
    // the socket connection with the server
    SOCKET soc;
    bool abort;

    // The monitors
    JTCMonitor IncomingFrame_, ConsumedFrame_, NewSocket_, SwitchSocket_;

    // The receiving buffer
    char * rBuffer;

    int err, iBindType;
    char szBuffer [MaxPathLength];
    bool fShutDownServer;

public:
    JTCMutex CriticalSection;
```

```
    int nrows, ncols;

    TComm(): iBindType(GET_PORT_AND_REMOTE_ADDR);

    ~TComm();

    virtual void run();
    void initialize(void);
    void terminate(void);
    void WaitForConsumption(void);
    void FrameConsumed(void);
    void WaitForIncomingFrames(int timeout);
    void FramesIncoming(void);
    void WaitForNewSocket(void);
    void NewSocket(void);
    bool hasSocket(void);
    void switchSocket(SOCKET, bool);
    void clearSocket();
    SOCKET getActiveSocket();
};

// pointer to TComm, handle is required by JThreads/C++
typedef JTCHandleT<TComm> TCommHandle;
```

As we may notice, four monitors and a mutex are introduced to ensure that the complex behavior of both the communication thread and the computation thread are properly synchronized, especially when the tracking client is in the process of switching from one server to another. These monitors and mutexes significantly increase the complexity of implementation.

The graphical user interface of the tracking client is shown in Figure 9.1.

9.2.4 Deployment with the *Agilos* Architecture

Once the basic client-server based *OmniTrack* application is developed, we attempt to deploy the application under the control of the *Agilos* middleware architecture. Following the steps detailed in Chapter 8, we have CORBAfied our *OmniTrack* implementation by inserting the CORBA event handler in the Windows message handling loop, exactly as shown in Section 8.2.

Similarly, the basic client-server based application is extended to multiple clients and servers by extending the interfaces of negotiators, as illustrated in Chapter 8. The specific criteria that the gateway uses to select the best server include the server workload in terms of CPU load, the angle

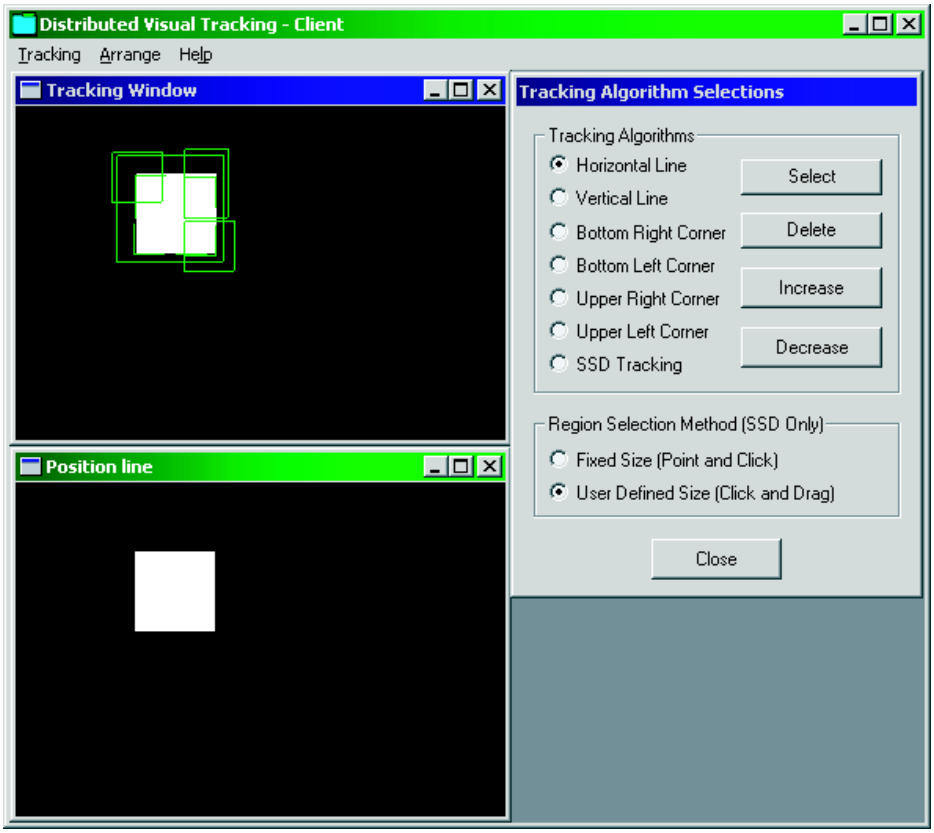


Figure 9.1: The Tracking Client in *OmniTrack*

of view to the objects, as well as the codec format video is served.

Finally, we have implemented an user configurator specific to the *OmniTrack* application. The user configurator is responsible for accepting user preferences with respect to three types of reconfiguration choices. First, the panable camera on the server can be turned to the left or right on user commands. Second, the user may explicitly switch from the current server to the server on the left or right. Third, the user may specify advanced options such as the type of objects so that the information may be used to select the best server available for specific objects. Figure 9.2 shows the graphical user interface of the user configurator.

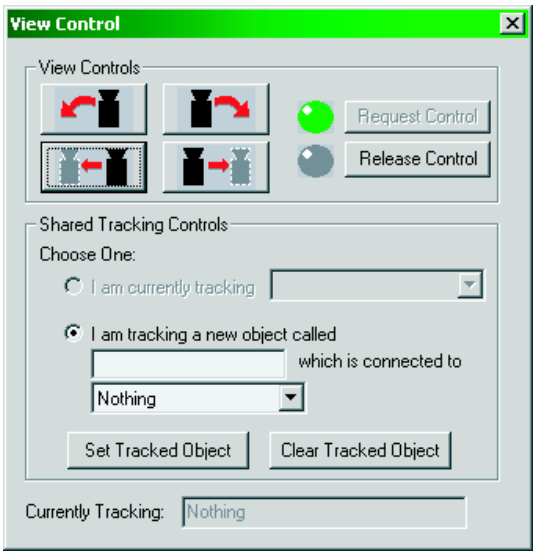


Figure 9.2: The User Configurator of *OmniTrack*

9.3 Implementation of *Agilos* Architecture

9.3.1 Adaptor

We have implemented two types of adaptors in the *Agilos* architecture: the CPU adaptor and the bandwidth adaptor. These adaptors implement the customized control algorithm presented in Chapter 3. The observer, presented in the next section, provides observed values as input to the adaptors. The communication between the adaptors and observers is facilitated by the CORBA Property Service. Furthermore, the output of both adaptors are also propagated to the CORBA Property Service as a set of CORBA properties, which are retrieved by the functional configurator. The mechanisms used for the propagation and retrieval to and from the CORBA Property Service are identical to those presented in Section 8.2 for delivering application QoS parameters from the application to the observer.

9.3.2 Observer

The observer has three distinct responsibilities. First, it retrieves the application-specific QoS parameters from the CORBA Property Service. Second, the observer observes system resource

availability, such as CPU load and throughput from client to the server, via the Performance Data Helper library provided by Windows NT as a service to peek the system parameters in the NT kernel. The optimal estimation algorithm presented in Chapter 4 is also implemented in the observer to estimate the end-to-end network throughput between the client and the server. Third, the observer visualizes all observed values, including system resources and application QoS parameters, in an animated illustration window similar to the Windows NT Task Manager. Figure 9.3 shows the results of such a visualization process of six different observed parameters.

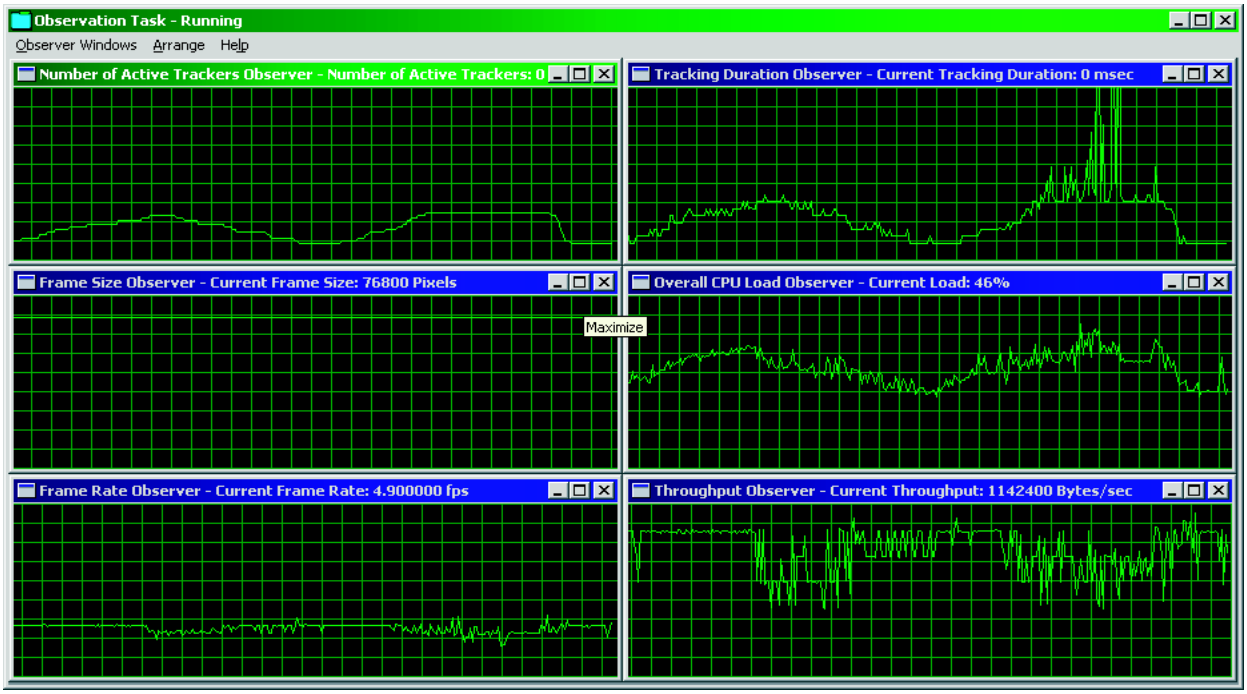


Figure 9.3: The Observer in *Agilos*

9.3.3 Configurator

The functional configurator includes the implementation of a fuzzy inference engine. We have adopted the C-FLIE inference engine implementation [24] as well as its rule specification format for specifying the rule base and membership functions. Since the original C-FLIE engine is implemented in C, we have implemented a set of wrapper functions in C++. The most important functions are defined as follows:

```
// The main entry point of the Inference Engine
void loadInferenceRuleFile(const char *rule_file_name = NULL);
void inferenceEngine(LingValList &LINK, const char * rule_file_name =
NULL);
```

These wrapper functions provide simplified interfaces to the rest of the configurator implementation to activate the inference engine with minimal required parameters. As we notice, the only parameters they need are an instance of the class `LingValList`, as well as a file name for the rule base itself. The class `LingValList` defines an array that encapsulates all linguistic variables used in the rule base.

After the output is generated from the fuzzy inference engine, the configurator needs to convert the generated results to the corresponding control actions in the application. The conversion process is mathematically modeled in Section 5.3.4 as part of the defuzzification process. In order to implement this conversion process, we introduce a configuration file that only defines the membership functions of all linguistic values. Such a file is complimentary to the actual file that defines the rule base. For example, in the *OmniTrack* implementation with the rule base defined as in Section 8.3.2, such a configuration file is as follows:

```
lingvar rateaction
class bandw 0 0 200 500
class compress 400 500 800 1200
class chopped 800 1200 1500 1700
class uncompress 1600 1800 2000 2000
lingvar cpuaction
class droptracker 0 0 30 40
class replacetracker 30 40 50 60
class adjustregion 50 60 70 80
class addtracker 70 80 100 100
end
```

After such a configuration file is defined, the conversion process may be encapsulated in a series of functions, as shown in the following:

```
// Load the configuration file
void loadLinguisticValueDef(const char *def_file_name);

// Retrieves the next string token from the configuration file that
// contains membership functions
void RetrieveToken(const char * token);
```

```
// Parse the configuration file to retrieve all definitions of membership
// functions for linguistic values
void ParseDefinitions(void);

// Convert the output from the inference engine into actual control actions
int ReverseMapping(const char * lingvar, int metric, float & value);
```

9.3.4 QualProbes

The implementation of QualProbes strictly follows the QualProbes services kernel algorithm presented in Figure 6.3. Before the algorithm is executed, the dependency tree of application QoS parameters needs to be explicitly specified in a file by the application developer. As an example, for the dependency tree illustrated in Figure 6.1, such a specification file is formated as follows:

```
# id  |  observe   |   tune   |    left child   |   right child
  1     precision      -1             2               3
  2      velocity      -1             -1              -1
  3     tfrequency     -1             4               5
  4     framerate      1              6               1001
  5     tquantity      -1             7               8
  6     iproperty      -1             9               10
  7     numtrackers    2              1000            -1
  8     proptracker    -1             11              12
  9     ratio          -1             13              14
 10     imagesize      -1             15              16
 11     regionsize     3              1000            -1
 12     ttype          4              1000            -1
 13     codect         5              1000            1001
 14     codecp         6              1000            1001
 15     pixels         7              1001            -1
 16     depth          8              1001            -1
```

The first column of the specification represents the identification number of an application QoS parameter. The second column specifies the name of the parameter, such a name is used for propagation to and retrieval from the CORBA Property Service for current values of the associated QoS parameter. The third column is the index to an array of function pointers that refers to the application control interface. Such an index specifies the particular function to use so that the parameter can be tuned, or reconfiguration choices can be activated in the application. The last two columns specify the parameter ID of the descendants in the dependency tree. Two special IDs

are used for system resources and are defined in related header files: 1000 for the CPU capacity, and 1001 for network throughput.

Once such a specification is established, the algorithm shown in Figure 6.3 can be executed while the application is activated for benchmark runs. In the *OmniTrack* application, we have used an artificially generated animation video, which is streamed from the server to the client. Once such an animation video is utilized, exact coordinates of tracked objects are known, and the tracking precision (with a name of `precision` in the CORBA Property Service) could be measured.

9.3.5 Negotiators and Gateway

The implementation of negotiators and the gateway strictly follow the design outlined in Chapter 7, as well as the application extension interfaces presented in Section 8.4. Client and server side negotiators are implemented to be tightly coupled with the application at compile time, while the gateway is implemented as a separate middleware component. For the purpose of making decisions on which server to switch to, the gateway shares the fuzzy inference engine with the configurator, and uses a different application-specific rule base. For example, one rule in the rule base for the *OmniTrack* application is shown as follows:

```
if (action is request_move) and ((angle_difference is veryclose) and
((format_difference is same) and (server_load is verylight))) then
server_ranking:= high ;
```

Such a rule base is evaluated against the input by the fuzzy inference engine for each server that the gateway manages. The output from the inference engine is the ranking for that particular server with respect to how its capabilities fits the requesting client's needs. The server with the highest ranking is selected by the gateway for the new server to switch to.

The graphical user interface for the gateway is shown in Figure 9.4.

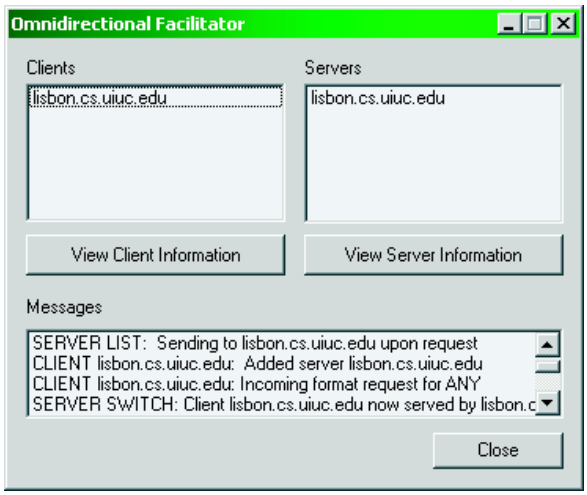


Figure 9.4: The Gateway in *Agilos*

9.4 Summary

This chapter discusses important issues in the process of implementing the *Agilos* middleware architecture and *OmniTrack* application in Windows NT 4.0 with C++. The current implementation of *Agilos* strictly follows our architectural and algorithmic design presented in previous chapters, and this chapter complements previous discussions with implementation choices and details that actually make *Agilos* work effectively with real-world applications, such as *OmniTrack*. The current source tree for the entire implementation (including both *Agilos* and *OmniTrack* contains over 60000 lines of C++ and supplementary code, which brings rich experimental results shown in Chapter 10, and demonstrates our strong commitment to verifying our theoretical work by large-scale implementation-driven experiences.

Chapter 10

Experimental Results

In this chapter, we evaluate the *Agilos* middleware architecture by executing the *OmniTrack* application under the control of *Agilos*. We present a series of scenarios to evaluate respective aspects of *Agilos*, including its effectiveness with respect to meeting the critical performance criterion, as well as the performance overhead *Agilos* introduces to the application, especially with respect to its gateway-centric approach in the third tier.

10.1 Experimental Testbed

We use the *OmniTrack* application, an omni-directional visual tracking system, as an example of the applications deployed under the control of the *Agilos* middleware architecture. The application, along with *Agilos*, is implemented to execute on two dual Pentium Pro 200 Mhz PCs, one Pentium MMX 200Mhz PC, and one Pentium II 400Mhz PC. All PCs runs Windows NT 4.0 Service Pack 5 as their operating system. We use Visual C++ 6.0 Service Pack 3 as the primary development platform for all our C++ source code. We use ORBacus 3.1.3 [27] as our primary choice of CORBA implementation. We have deployed all tracking clients and servers over 10Mbps standard Ethernet.

With respect to CPU load measurements, we perform both application-specific source-level instrumentation to obtain measurements on application QoS parameters such as the tracking duration, as well as system-level probes to measure system resources such as the CPU load. The

application-specific probing results are communicated to the observer by the CORBA property service, while the system-level probes are implemented with the Performance Data Helper Library included in the Platform SDK of Windows NT. In order to simulate the network bandwidth fluctuations, we plug in a throughput simulator to simulate a network through three routers with FIFO packet scheduling and cross traffic. This setup will simulate network fluctuations similar to what occur in the Internet. In addition, in order to measure the tracking precision and repeat experiments, we carry out all our experiments based on animated moving objects served from the tracking server, rather than live video from the camera. Finally, we obtain the tracking precision of one tracker by measuring the distance between the position of the tracker and the actual object it tracks, and we evaluate the overall tracking precision by calculating the average of the precision of all trackers. In *OmniTrack*, the critical performance criterion is to maintain the stability of its overall tracking precision.

10.2 Adaptation Choices in *OmniTrack*

In Section 5.3.2, we gave a brief overview of the tunable application QoS parameters and reconfiguration choices in the *OmniTrack* application. In Section 6.2.1, we again discussed the relationships among application QoS parameters, taking *OmniTrack* as an example. As we have noted, *OmniTrack* is rich in its various options for adaptation. In this section, we present important adaptation choices for *OmniTrack* in details, divided into two categories: parameter-tuning adaptations and reconfigurations.

10.2.1 Parameter-Tuning Adaptations

Parameter-tuning adaptation choices refer to those control actions that tune application tunable QoS parameters for the purpose of adaptation. There are six important parameter-tuning adaptation choices in the current reference implementation of *OmniTrack*, enumerated as follows.

1. *Image size in pixels*. This is the total number of pixels in each frame in the video being streamed from server to client. It may be tuned by (1) chopping the edges of frames; (2) scaling the frames; (3) switching to a server with a different image size. Tuning this parameter affects the bandwidth requirement in *OmniTrack*. In addition, we note that this is a representative of the parameter *content size* in any client-server based applications that feeds contents from the server to the client. For example, a web server may reduce the size of images embedded in the web pages being served.

2. *Image color depth*. The color depth of each frame, which affects the bits allocated for each pixel before any compression. This parameter is also related to bandwidth requirements of the application.

3. *Compression ratio*. If the video is compressed with a specific codec type in *OmniTrack*, this is a relative criterion with regards to the internal parameters within the codec process. If the ratio is high, more CPU resources are used and less network bandwidth is required, and vice versa.

4. *Frame rate*. This is the rate that the server serves video frames to the client, in terms of frames per second. The server is able to vary the rate in order to adapt its bandwidth requirements between the server and the client.

5. *Number of active trackers*. By adding and removing trackers on-the-fly on the tracking client, the CPU requirement of the client is effectively changed.

6. *Size of tracked region*. This is a parameter that characterizes the computational complexity of one tracker. If the size of the tracked region is smaller, less CPU capacity is used, and vice versa.

10.2.2 Reconfigurations

There are three effective reconfiguration choices in the current reference implementation of *OmniTrack*.

1. *Codec type*. The type of codec format used for the video. In our reference implementation, we have implemented Motion JPEG as the alternative codec other than uncompressed video. Similarly, more choices could be provided, such as MPEG, in order to increase the adaptiveness of the application. The codec type may be reconfigured by one of the following alternatives: (1) On-the-fly switching between servers providing different codec types; (2) Changing the codec type on the same server. After reconfiguring *OmniTrack* to use a different codec, both CPU and bandwidth requirements may be affected.

2. *Type of tracker*. The type of each tracker may be one of the following in the reference implementation of *OmniTrack*: SSD tracker, line tracker and corner tracker. Each poses a different load on the CPU. Changing the type of the tracker is accomplished on the tracking client alone, no servers need to be involved.

3. *Server switching*. If the user needs a different angle of view on the tracked object or a different codec type, server switching may be requested from the client side, and the server switching protocol will be accomplished with the assistance of the gateway. The switching of servers is decided by three factors in the current implementation: server workload, angle of view, and codec type being served.

10.3 Experimental Scenarios and Results

The primary purpose of all the experiments conducted with *OmniTrack* is to evaluate the effectiveness and the performance of *Agilos*, our middleware control architecture. Since we wish to study all aspects of *Agilos*, we divide the experiments into ten different scenarios. In this sec-

tion, we present the context and experimental setup of each scenario, followed immediately by the experimental results measured by such a scenario, and the conclusions that we have drawn.

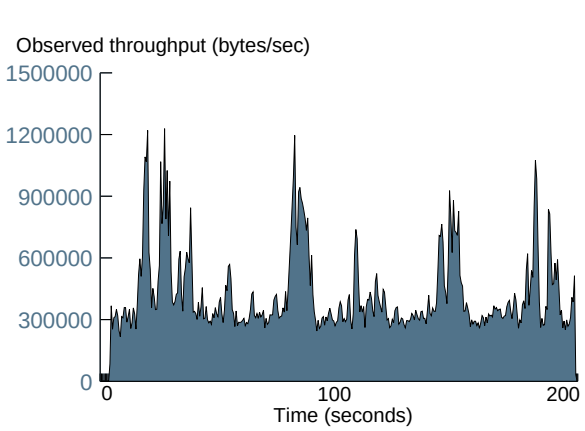
10.3.1 Scenario 1: Testing the First Tier

In this scenario, we test the effects of *Agilos* in a client-server based setup consisting of a single tracking server in *OmniTrack*, which eliminates any influences from the gateway-centric approach in the third tier. We first start with the following basic rule base that focus on only simple parameter tuning parameters, the *image size in pixels*, discussed in the previous section:

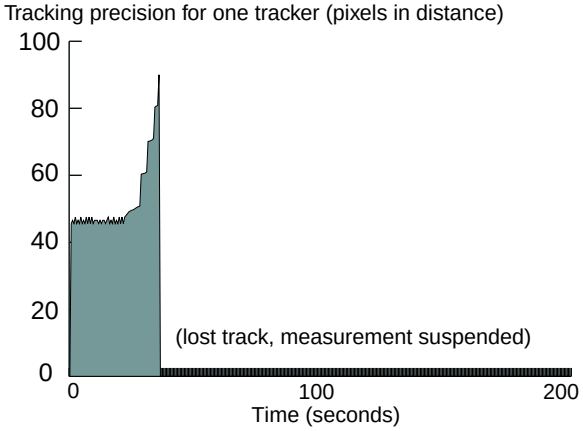
```
if (rate is moderate) then rateaction:= size;
if (rate is low) then rateaction:= size;
if (rate is high) then rateaction:= size;
```

This rule base introduces the effects of "bypassing" the configurator in our tests, so that only the first tier, including the adaptor and the observer, is tested in the a basic client-server setting of *OmniTrack*. Since the output of CPU adaptor is not used in any way in this particular rule base, only the output of the throughput adaptor affects the tunable parameter *image size* in the application. In the adaptors, we have defined α as 0.01, and β as 0.45.

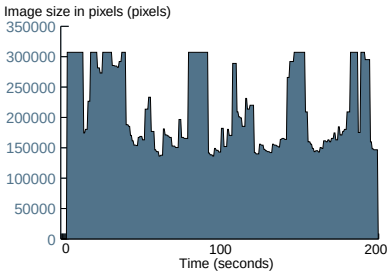
The results we have obtained are shown in Figure 10.1, divided in four parts. Figure 10.1(a) shows the observed variations in network throughput for the video being streamed from the tracking server to the client. Figure 10.1(b) shows the tracking precision for one tracker when the *OmniTrack* application is not supported by *Agilos*. After *Agilos* support is activated, Figure 10.1(c) shows the corresponding reactions generated by the configurator of *Agilos* with the simplified rule base introduced previously. Such results reflect the capabilities of the bandwidth adaptor and observer in the first tier. Figure 10.1(d) shows the resulting tracking precision for one tracker with adaptation activated. The results show that the tracking precision for one tracker improves significantly compared with Figure 10.1(b), and is kept stable for the entire duration of measurements. Figure 10.1(e) shows the collective tracking precision for 30 concurrent active trackers by calculating the average of each tracker's precision. We note that as time proceeds, the tracking precision



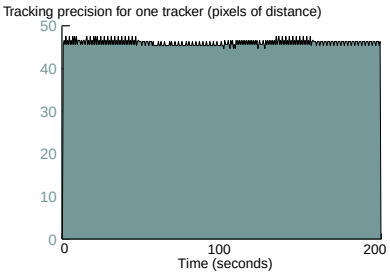
(a) Observed Throughput



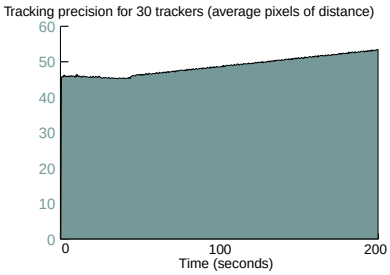
(b) Tracking Precision for One Tracker (without *Agilos*)



(c) Image Size in Pixels After Adaptation



(d) Tracking Precision for One Tracker (with *Agilos*)



(e) Tracking Precision for 30 Trackers (with *Agilos*)

Figure 10.1: Experimental Results for Scenario 1

incrementally increases, illustrated as an upward curve. The tracking precision is measured by the distance between the coordinates of the tracker and the coordinates of the object it tracks, which leads to the conclusion that if the average value increases, a small portion in the collection of trackers drift away from the object and eventually lose track, while the rest of them are stable.

The conclusion we have drawn from the above experimental results is as follows. First, the first tier of *Agilos*, including the adaptor and observer, is effective in maintaining the stability of tracking precision for one tracker, even with the simplified rule base that we have used in the configurator. This demonstrates the validity of our approach with respect to the Task Control Model for the adaptors and the optimal estimation algorithms implemented in the observers. Second, the tracking

precision for 30 concurrent trackers is not properly preserved, mainly because of the reason that more trackers will saturate the CPU load, which is not taken into account in the experimental setup of this particular scenario.

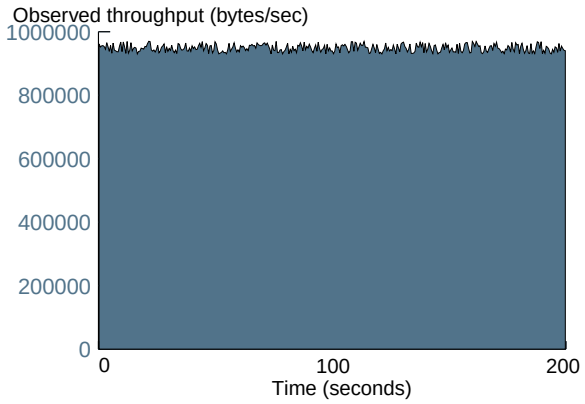
10.3.2 Scenario 2: Testing the Second Tier under Fluctuating CPU Load

In order to experiment with a hybrid combination of application-specific parameter-tuning and reconfiguration choices, which emphasizes the effects of the Configurator, we deploy the configurator in *Agilos* using a full-fledged rule base based on the results of profiling services provided by *QualProbes*. The experiments in Scenario 2 and 3 emphasize testing the effectiveness of the Fuzzy Control Model supported by both CPU and network bandwidth adaptors. Similar to the first scenario, we focus on the basic client-server setup in *OmniTrack* in this scenario, and will proceed to the omni-directional facilities in Scenario 4.

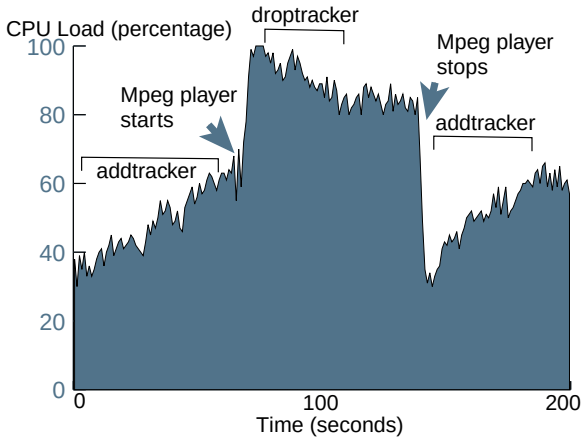
Having experimented with the effects of adaptation under network bandwidth fluctuations in the previous scenario, in Scenario 2, we focus on a particular situation where only the CPU resource availability fluctuates over time. In this situation, only the CPU adaptor will be effective in the actual decision-making process in the configurator. Figure 10.2 shows the results we have obtained. As illustrated in Figure 10.2(c), the adaptation choice we have used to react to CPU load variations is to adjust the number of active trackers by adding and removing them on the fly.

Since we wish to focus only on CPU load variations, the experiments are carried out over a single Ethernet segment with almost constant network bandwidth. In order to introduce variations on CPU load, 70 seconds after the beginning of the experiment, we start the Windows Media Player to play a MPEG-1 video from a local file on the same end system. Such a media player will consume about 50% of CPU if being executed as a standalone application without any other applications concurrently running. In order to study the effects when CPU load decreases dramatically, we stop the media player at 140 seconds into the experiment.

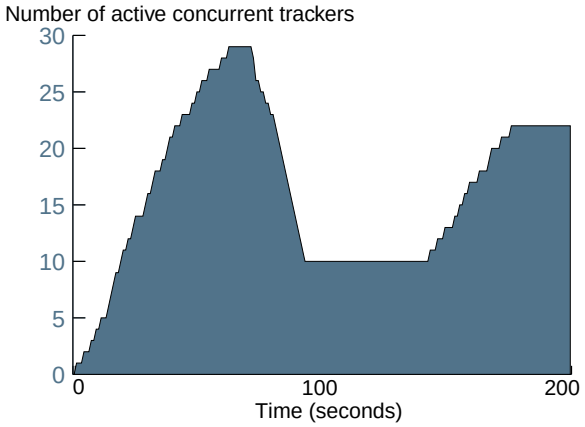
In Figures 10.2(a) and 10.2(b), we show observations with respect to network throughput and end-system CPU load. We may observe that the network throughput remains constant since we



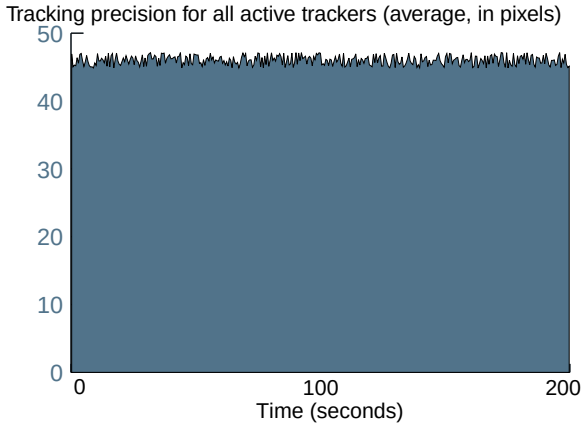
(a) Network Throughput



(b) CPU Load



(c) Number of Active Concurrent Trackers



(d) Average Tracking Precision for All Trackers

Figure 10.2: Experimental Results for Scenario 2

place the server and the client on the same Ethernet segment. From Figure 10.2(b) and 10.2(c), we may notice that during the initial phase of the experiment, new trackers are being added in order to take advantage of idle CPU time, up to a maximum of 29 trackers. However, in the second phase when the Windows Media Player starts the playback process, trackers are being dropped down to a minimum of 10 trackers in order to prevent the CPU from overloading. At 140 seconds into the experiments. when the Windows Media Player stops playing the MPEG-1 file, CPU load dramatically decreases to around 40%, and more trackers are being added to utilize the idle CPU time. The most critical QoS parameter, tracking precision, is measured and illustrated in Figure

10.2(d), where we have shown that the average tracking precision for all active trackers is stable in all phases of the experiments.

To conclude, we note that the tracking precision is kept stable by adapting the number of active trackers based on observed variations in CPU load. In this experiment, such variations are introduced by another CPU-intensive application executing in the same end system.

10.3.3 Scenario 3: Testing the Second Tier under Fluctuating Bandwidth and CPU Load

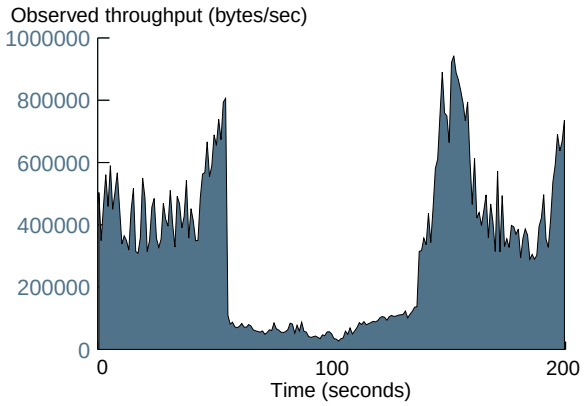
Similarly to Scenario 2, in this scenario, we deploy the configurator in *Agilos* using a full-fledged rule base, and emphasize testing the effectiveness of the Fuzzy Control Model. However, rather than testing under one fluctuating resource types, we examine the adaptation results when both CPU and network bandwidth are fluctuating. In this case, both CPU and network adaptors are actively in effect to support the decision-making process in the configurator.

Figure 10.3 shows the results we have obtained. Table 10.1 shows the control actions generated by the configurator at their respective starting times. The timing of these control actions are also visually embedded in Figure 10.3(b).

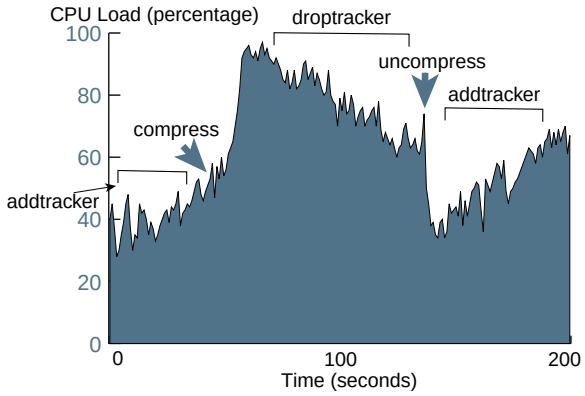
Start Time (sec)	*Control Action from Configurator*
4-40	`addtracker`
56	`compress`
62-120	`droptracker`
139	`uncompress`
145-178	`addtracker`

Table 10.1: Control Actions generated by the Configurator

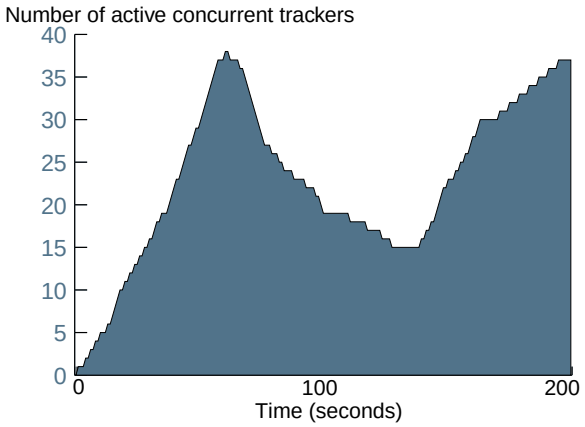
Our analysis on these experimental results is as follows. In Figures 10.3(a) and 10.3(b), we show observations with respect to network throughput and end-system CPU load. These observations drive the first tier adaptors, whose output drives the behavior of functional configurators in the second tier. In the case of this scenario, a hybrid combination of parameter-tuning and re-



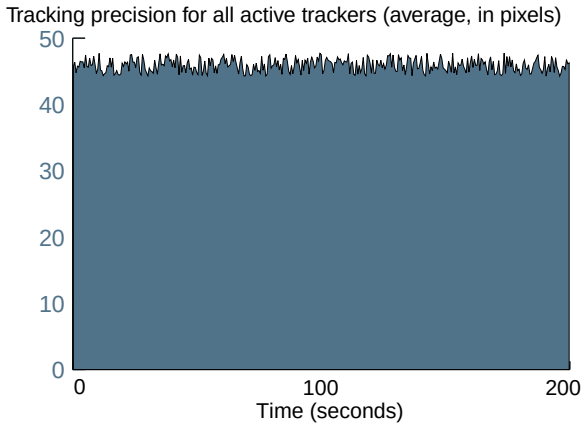
(a) Network Throughput



(b) CPU Load



(c) Number of Active Concurrent Trackers



(d) Average Tracking Precision for All Trackers

Figure 10.3: Experimental Results for Scenario 3

configuration choices is activated by *Agilos*. The specific reconfiguration choices being used are `compress` and `uncompress`, which activates and deactivates the Motion JPEG video codec in both the tracking server and client. The tunable parameter is the number of active trackers. Since the trackers are executed in a round-robin fashion in the computation thread introduced in Section 9.2, the duration of executing all trackers in each iteration is increased when there are more active trackers present. This may impose a heavier CPU load on the end system. Figure 10.3(c) shows the number of active trackers, which is the result of the control actions `droptracker` and `addtracker` generated by the configurator. The reconfigurations `compress` and `uncom-`

`press` significantly reduces the network throughput from the server to the client, and increases the CPU load to near 100%. This leads to another round of parameter tuning adaptations related to `droptracker`. As illustrated in Figure 10.3(d), this combination of parameter-tuning and reconfiguration choices, activated with appropriate timing determined by the configurator, leads to a stable collective tracking precision for all trackers. Such measurement of tracking precision is obtained by calculating the average of the precision of individual trackers.

To conclude the analysis of this scenario, we note that a combination of application-specific parameter-tuning and reconfiguration choices, made by the configurator, is crucial to maintaining the stability of the critical performance criterion. This is manifested by a comparison between Figure 10.3(d) in this scenario and Figure 10.1(e) in Scenario 1. Since Scenario 1 only tests the effects of tuning a single parameter, in this case the *image size*, it does not have the capability of maintaining the tracking precision for all trackers under any resource variations. On the contrary, this scenario shows that the collective tracking precision can be maintained by adding more adaptation choices in the rule base for the configurator, since the application is more versatile and adaptive with a rule base that precisely captures the various adaptation possibilities within the application.

10.3.4 Scenario 4: Testing the Third Tier

After testing the capabilities of the first and second tier in *Agilos*, we proceed to evaluate our gateway-centric design presented in Chapter 7. In this scenario, we have designed three experiments to demonstrate the capabilities of the third tier of *Agilos*, particularly with respect to the gateway's ability to provide each client with a QoS-satisfactory server. We evaluate the gateway's server decision using two criteria: how well the selected server matches the client's reconfiguration request, and while satisfying these requests, how well the gateway performs load balancing based on server's workload.

Each experiment was performed using six clients and three servers. The clients were placed on hosts of varying speed, one client per host. The servers were placed on three hosts, one server per host, in descending order or host processor speed: an MJPEG server on a Pentium II 400 (PII 400),

another MJPEG server on a dual processor Pentium Pro 200 (PPro 200), and an uncompressed server on a Pentium 200 MMX (P 200). The gateway was placed on "PII 400".

For the first experiment, we have manually selected the server for each client to balance best the overall server load. This experiment was performed as a control for evaluating the second and third experiments. The results of this experiment are shown in (a, b, c) of Figure 10.4. The jumps in execution levels indicate when each server begins servicing another client.

For the second experiment, we repeated the first experiment, this time allowing the gateway to select the server for each client. The results of this experiment are shown in (d, e, f) of Figure 10.4. As expected, the gateway satisfied all client requests while performing an equivalent level of load balancing which had been performed manually in the first experiment.

In the second experiment, the relative load on each server was close enough such that a random or round robin placement of the clients would have performed an adequate job of load balancing. To prove that the gateway would respond equally well within a biased service facility configuration, we switched the fastest server, "PII 400", from MJPEG to uncompressed video. This change greatly reduced the processor overhead for this server. For the third experiment, we allowed the gateway to select servers with the new configuration. This time, the gateway placed every client on "PII 400". Even with one server executing, the gateway executing, and all six clients being serviced, the CPU load on this host was still lower than the load on the other servers, each of which executing one server with no connections. Thus, the gateway have successfully demonstrated that it could meet a client's QoS needs while maintaining optimal utilization across the facility.

In addition, we also wish to measure the time necessary to start a minimal omni-directional tracking server in this scenario. The Gateway is started first, immediately followed by the server. The server initialization time is the period between starting time of Gateway and finishing time of server registration at the Gateway. Figure 10.5(a) illustrates the *server initialization time*.

Once the tracking server is established, the client connects to the tracking server and start receiving video. The *client initialization time*, shown in Figure 10.5(b), is the time it takes from starting time of the client till the display of video.

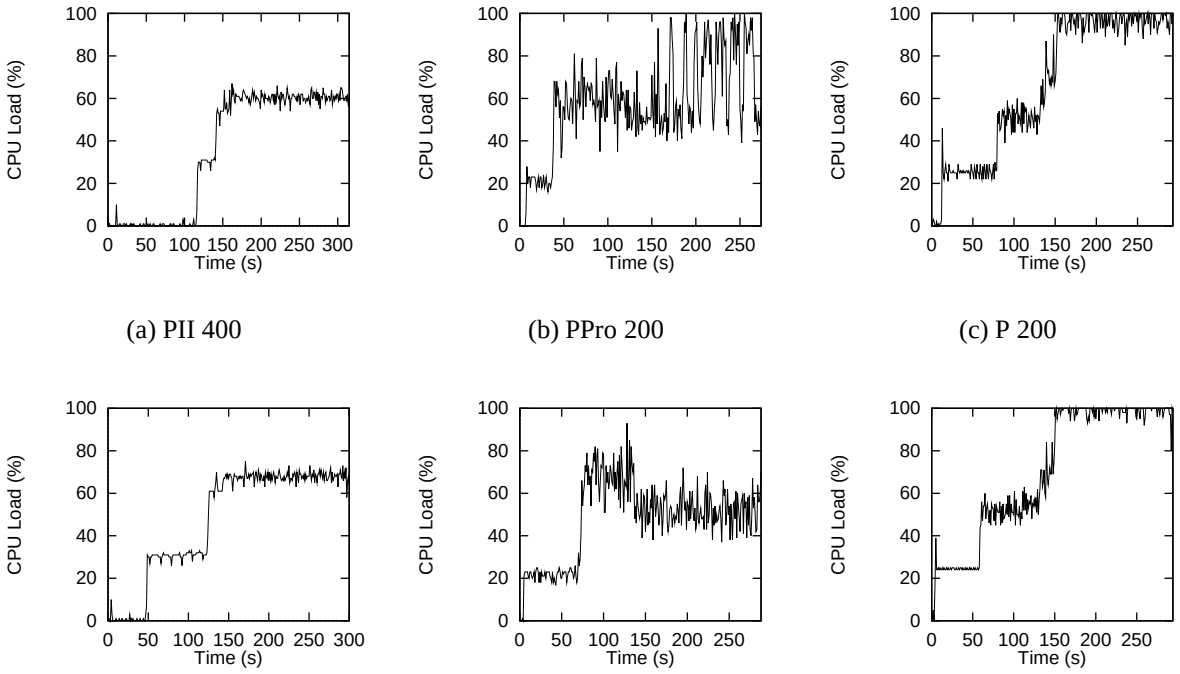
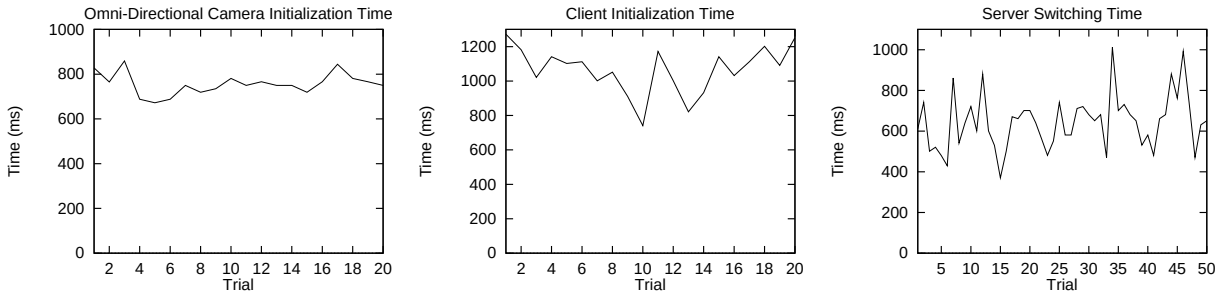


Figure 10.4: Experiments with the Gateway

Figure 10.5(c) shows the *server switching time*, which is the length of duration to execute the server switching protocol illustrated in Figure 7.7. To measure the server switching time, two servers serve live video with different video formats, e.g., uncompressed and MJPEG compressed. The user requests to switch from one server to the other by clicking on the *move* button on the User Configurator. The switching time is the period between the time when a user clicks the button on the User Configurator and the time when the client retrieves the video stream from the new server.

To summarize, we have concluded the following: First, the gateway is able to correctly select a suitable server to meet the client's QoS needs, while maintaining optimal utilization across the facility. Such a selection is the basis of the server switching protocol presented in Figure 7.7, which facilitates all reconfigurations that demands switching among servers on the fly. The gateway made the selections with its own rule base, and with the same fuzzy inference engine being used in the configurators. Second, the gateway is able to initialize the servers and clients within a reasonable time frame, and perform the server switching protocol with some reasonable performance over-



(a) Server Initialization Time (b) Client Initialization Time (c) Server Switching Time

Figure 10.5: The Performance of Gateway Protocols

head. While the server switching protocol is executed, the client may still receive video as usual from the previous server. The network socket to the previous server is only closed when a new socket is established with the new server.

10.4 Conclusions

In this chapter, we have presented our experimental results with respect to running the *Omni-Track* application under the control of the *Agilos* middleware control architecture. We have divided our experiments into three major scenarios, each focusing on a specific tier of the *Agilos* architecture. Our experimental results show that we are able to preserve the tracking precision, the critical performance criterion for *OmniTrack*, in various cases of resource variations. These experiments validate our design of *Agilos* introduced in previous chapters, and demonstrate the effectiveness of its control actions. We have also shown that there are some reasonable performance overhead introduced by the gateway-centric design, however this is the tradeoff for introducing the load balancing and server switching capabilities as a new domain of reconfiguration choices.

Chapter 11

Related Work

It has been widely recognized that many QoS-constrained distributed applications need to be adaptive in heterogeneous environments. Many research problems relevant to our work in the area of QoS adaptations have been studied. We briefly review each of these in the following sections, divided into several areas of focus.

11.1 Communication Protocols

System level adaptive mechanisms in communication and networking protocols have been extensively studied in the past decade. Particularly, the problem of flow control at the packet or cell level has been examined with great interests. *Flow control* refers to the set of techniques that enable a data source to match its transmission rate to the currently available service rate at a receiver and in the network. In this sense, the objective of the application-aware QoS adaptation with respect to network bandwidth resources is largely identical to the goal of the flow control, except that QoS adaptation is performed at the application level, and most mechanisms developed in the past decade related to the flow control are implemented at the datalink, network or transport layers of a protocol stack. We review some of the previous work related to our approach.

Many previous packet or cell level flow control approaches were proposed with the assistance of control theory. Earlier work [30, 31, 14] showed that a control-theoretic way of analyzing flow control problems is both valid and feasible. Notably, Keshav in [14] proposes a packet-pair

flow control algorithm and uses control theory to analyze its stability and performance, under the assumption that a round-robin like scheduling discipline referred to as the *Rate Allocation Server* is deployed in the bottleneck node. In this work it was shown that the flow control algorithm is *stable*, which implies that if a new source becomes active, existing active sources adjust their transmission rates so that after a brief transient period, the system settles down to a new equilibrium. More recent work [32, 33, 34, 15] has continued the pursuit in this domain, mainly focusing on the flow and congestion control issues for ATM switches, particularly under ABR (Available Bit Rate) service. Notably, in [32], the control laws for congestion control, in the case of a single congested node, were derived and stability properties proved. In [15], the stability and sensitivity properties are analyzed in the case of the rate-based flow control for ABR service. In this work, *stability* is described as the ability of the control system to steer the target system back to equilibrium state after a disturbance has occurred. In addition, *sensitivity* is described as the change in performance variables with respect to an infinitestimal change in a parameter variable, while the rest of the parameter variables remains unchanged.

Another noteworthy example of previous work is in the area of fuzzy logic and fuzzy control systems, which are previously studied and applied to solutions for flow control. In [14] fuzzy logic was applied to solve state estimation problems. Pitsillides et al. [24] present a fuzzy control approach used for the purpose of flow control in ATM networks, with linguistic variables being the queue length and the change rate of queue length in each ATM switch. In contrast, our approach focuses on generating control actions to control distributed multimedia applications themselves, with the primary goal being the satisfaction of the critical performance criterion.

Our work focuses on the study of adaptations in the application domain with a focus on the critical performance criterion, as well as adaptations with respect to more than one type of resources. While the mechanisms are certainly different in the application domain, the general approach of utilizing the control theory and the definitions for stability and other transient properties are similar to the previous work. Furthermore, we focus on global fairness properties of the adaptation behavior with respect to observable resources in the same end system, which was not studied in

most of the previous work on packet or cell-level flow control based on control theory.

11.2 Resource Management

Recent research work on resource management mechanisms at the systems level also expressed much interests in studying various kinds of adaptive capabilities. Particularly, in wireless networking and mobile computing research, because of resource scarcity and bursty channel errors in wireless links, QoS adaptations are necessary in many occasions. These adaptation mechanisms are frequently accompanied by new QoS parameters unique to the mobile environment, for example seamless user mobility. For instance, in the work presented in [35, 36, 37], a series of adaptive resource management mechanisms were proposed that apply to the unique characteristics of a mobile environment. These adaptive mechanisms include the division of services into several service classes, predictive advanced resource reservation, and the notion of cost-effective adaptation by associating each adaptation action with a minimal lost of network revenue. Another example is the work of Noble et al. in [6], who investigated an application-aware adaptation scheme, focusing on two characteristics: *fidelity* of data and *agility* of adaptation. The work adopts a different mindset: based on the coordination between the system and individual applications, the system is responsible of monitoring resource changes and notifying the application, while the applications decide how best to adapt when notified. Similarly to the functional separation between the first and second tier in our *Agilos* architecture, this work was also built on the separation principle between adaptation algorithms controlled by the system and application-specific mechanisms addressed by the application. The key idea was to balance and tradeoff between performance and data fidelity. However, a fundamental difference in our work is that we have introduces a second tier in *Agilos*, so that the middleware is well aware of all application-specific adaptation choices through the rule base. This design is in line with the separation principle, but grants more power for the middleware to control the adaptive behavior in the application. In other words, rather than leaving the decision to the applications with respect to which adaptation option to select on resource variations, we

perform such a decision-making process within the *Agilos* middleware itself. The Fuzzy Control Model is applied in order to rigorously model such a process.

Another related category of work studies the problem of dynamic resource allocations, often at the operating systems level [38, 39, 40]. The work in [39] focuses on maximizing the overall *system utility* functions, while keeping QoS received by each application within a feasible range (e.g., above a minimum bound). In [38], a global resource management system was proposed, which relies on middleware services as agents to assist resource management and negotiations. In [40], the work focuses on a multi-machine environment running a single complex application, and the objective is to promptly adjust resource allocation to adapt to changes in application's resource needs, whenever there is a risk of failing to satisfy the application's timing constraints.

The Darwin project [41] from Carnegie Mellon University studies the introduction of value-added services, referred to as *service brokers*, in order to cope with the requirements presented by complex distributed applications that connect many endpoints. The service brokers have application-specific knowledge, and are responsible to identify the computation, storage and communication resources in the network that will satisfy a request from the application, in a way that optimizes a quality or cost metric. The service brokers may be separate entities residing in the network, or they can be components of either an application or a service provider in the end system. These brokers may also be hierarchically concatenated to provide more complex services. The key contribution of the work is that service brokers serve as an application-aware "proxy" between applications and resource management mechanisms, so that resources are optimally allocated globally across multiple network nodes and end points. In contrast, our *Agilos* architecture emphasizes on controlling the applications themselves, so that they optimally adapt to the variations in the environment. Such optimality is measured by the degree of satisfaction of the critical performance criterion in the application.

Rich features in the theory of fuzzy logic were also utilized in the area of dynamic adaptive resource control. As an example, in the AutoPilot [42] project, a fuzzy logic approach is adopted to design actuators that process sensory data observed from high-performance parallel programs,

so that optimal performance can be achieved by adjusting system parameters, such as those in a parallel I/O file system. The actuators and sensors are functionally similar to the adaptors and observers in the first tier of our *Agilos* architecture. However, the objectives and domain of operations are notably different.

In contrast to the above related work, our work distinguishes in its domain, focus and solutions. First, our work on the Task Control Model focuses on the analysis of the actual adaptation dynamics, which is more natural for modeling with a control-theoretic approach, rather than overall system utility factors. Second, rather than focusing on a multi-machine environment running a single complex application, our work focus on an environment with multiple applications competing for a limited amount of shared resources, which we believe is a common scenario easily found in many actual systems. The end systems in our model are loosely, rather than tightly, coupled with heterogeneous networking environment. Third, our work focuses on proposing various schemes for the middleware components to actively control the application, rather than providing resource allocation and management services in the execution environment to meet the application's needs. In other words, we focus on assisting to adapt applications, rather than on resource allocations in the system.

11.3 Middleware Services

Recently, in addition to studies in the networking and resource management levels, many active research efforts are also dedicated to various adaptive functionalities provided by middleware services. For example, [43] proposes real-time extensions to CORBA which enables end-to-end QoS specification and enforcement. [2] proposes various extensions to standard CORBA components and services, in order to support adaptation, delegation and renegotiation services to shield QoS variations. The work applies particularly in the case of remote method invocations to objects over a wide-area network. Similarly, the Da CaPo++ project by Stiller et al. [3] supports a range of multimedia applications by a layer of middleware that integrates various functionalities, e.g., security

and multicasting capabilities, and that automatically configures itself to provide suitable communication protocols and multimedia-oriented services that are adaptable to application needs. The $2K^Q$ project [44] [45] in University of Illinois proposes a resource-aware service configuration model for heterogeneous environments. Similarly, The 2K project [46] proposes a dynamically reconfigurable middleware framework to support resource management of dynamically changing distributed resources within rapidly changing user environments. The work noted in [47] builds on a series of middleware-level agent based services, collectively referred to as the *Dynamic QoS Resource Manager*, that dynamically monitors system and application states and switches *execution levels* within a computationally intensive application. These switching capabilities maximize the user-specified benefits, or promote fairness properties, depending on different algorithms implemented in the middleware.

In contrast to the above mentioned related work, our work is orthogonal to the these approaches, since the middleware control architecture is based on underlying service enabling platforms, which is CORBA in our experimental testbed. In addition, we attempt to provide adaptation support to the applications proactively, rather than integrating adaptation mechanisms in CORBA services so that they are provided transparently to the applications. Furthermore, we attempt to develop mechanisms that are as generic as possible, applicable to applications with various demands and behavior. Finally, we attempt to provide support in the middleware control architecture with respect to multiple resources, notably CPU and network bandwidth.

11.4 Application-Specific Mechanisms

Recent research efforts are also particularly interested in adaptation problems in the application level. For example, the work presented in [5] and [48] uses software feedback mechanisms that enhance system adaptiveness by adjusting video sending rate according to on-the-fly network variations. In [49], Hafid et al. proposed application adaptation at the configuration level, which carries out transparent transition from primary components to alternative components, as well as

at the component level, which redistributes resources in different components so that a QoS trade-off can be made. In [50], a software framework was proposed for network-aware applications to adequately adapt to network variations. [51] and [52] proposed adaptive filtering mechanisms to reshape video streams, performed in either end systems or intermediate nodes in a multipeer distributed environment. In the work of Goktas et al. [53], the time variations along the transmission path of a telerobotics system are modeled as disturbances in the proposed perturbed plant model, in which the mobile robot is the target to be controlled. This is similar to our work that attempts to apply control theory to analyze the adaptation dynamics in a broader range of applications, and in a more rigorous fashion. In [54], a control model is proposed for adaptive QoS specification in an end-to-end scenario. In the work of Bolot et al. [55], rate and error control schemes are proposed at the application level for a video conferencing application in best-effort networks, also utilizing a scheme similar to rate-based feedback control.

Similarly to the above, our approach models applications as a series of tasks, assisted by the feedback loop. However, we differ in the view of the middleware components, which control the adaptation behavior of applications. Furthermore, we propose a separation of control and estimation algorithms: With respect to control, proper choices for adaptation are made on adaptation timing, scale and methods used, which balance between the frequency and adaptation agility of adaptation actions within the application; With respect to estimation, optimal predictions are being made to obtain the best possible estimate of actual states. In addition, the algorithms proposed to determine timing of adaptation in most previous work are heuristic in nature, and the analysis of various adaptive transient properties such as stability, steady-state fairness and agility are not addressed.

Similar to the objectives throughout the architectural design process of *Agilos*, other frameworks and mechanisms have also been brought forth to deal with application-specific adaptation choices and criteria related to the user satisfaction. In the work of Witana et al. [21], a software framework is proposed that facilitates the development of adaptive applications. Much similar to the rule base and configurators in *Agilos*, such a framework allows the installation of application

dependent policies which govern the adaptive behavior of the application. Similar to the critical performance criterion, these policies are defined by the user, and reflect the user's relative satisfaction of QoS. Although with similar goals, our work with *Agilos* takes a more systematic and less ad-hoc approach, which is applicable to the needs of a wider range of applications. As another example, in the work of Abdelzaher et al. [4], a web server QoS provisioning architecture is proposed for the purpose of adapting web contents to provide overload protection of web servers. A load monitoring mechanism is introduced to monitor the server utilization, a content adaptor is used for adapting web contents being served, and a utilization controller, which utilizes the standard PID algorithm, is integrated to control such adaptation in a feedback loop. Similar to our work, the key elements of such an adaptation mechanism are the closed feedback loop and a controller running control algorithms. The difference between this work and *Agilos* is that this work focuses on adapting web server contents, while *Agilos* focus on generic applicability and ease of deployment of various complex applications, rather than specific to the needs of any single application.

11.5 Visual Tracking Systems

We have used the *OmniTrack* application, an omni-directional visual tracking system as a proof-of-concept application in all of our experiments. The tracking algorithms we have used is derived from the previous *XVision* project [11] from Yale University. We have adopted the SSD, edge and line tracking algorithms from *XVision*. There are also a vast quantity of literature related to the tracking problem itself. However, since tracking itself is not the primary focus of this dissertation, most of them are not directly related to this work. A notable exception is the work by Neumann, et al. [56], which presents an architecture for robust detection and tracking of naturally occurring features in unprepared environments. Such an architecture benefits vision-based augmented-reality tracking systems. Significant insights into tracking systems have been obtained thanks to this work, especially with respect to motion estimation equations and tracking error measurement schemes (RMS error and angle error measures). The implementation of tracking precision measurement in

our work is derived from these tracking error measurement schemes.

Chapter 12

Concluding Remarks

12.1 Conclusions

In this dissertation, we presented the *Agilos* middleware control architecture as a viable approach to control the adaptive behavior of applications so that the best possible application-level Quality of Service is achieved under any resource conditions during the lifetime of the applications. *Agilos* is designed as a *three-tier* architecture: In the application-neutral first tier, the adaptors and observers implements a closed control loop based on the Task Control Model. In the application-specific second tier, the configurator takes output produced by the adaptors and makes decisions on control actions based on the Fuzzy Control Model. In addition, QualProbes enhance the effectiveness of the configurator by probing and profiling the relationship among application QoS parameters and resource demands. In the third tier, a gateway-centric design is introduced to facilitate reconfigurations that involve multiple servers and clients. With respect to implementation, the *Agilos* architecture takes advantage of standard service enabling platforms such as CORBA to facilitate communication among middleware components, as well as between *Agilos* and the applications under control, via the application control interface.

The key contribution brought by the *Agilos* architecture is as follows. First, with the Task Control Model implemented in the adaptors, we are now able to reason about and validate such properties as stability, agility, system equilibrium and fairness for application-aware adaptive behavior and timing within distributed systems, which was not possible in previous work. Traditional

analytical models for distributed systems such as queuing theory, process algebra, petri nets and other frameworks allow us to model and analyze properties such as deadlock, starvation or worst case timing upper bounds, but do not allow easy and straightforward reasoning and proofs of adaptive timing properties. Second, with the Fuzzy Control Model and QualProbes we are able to specify application-specific rule bases so that the critical performance criterion for a particular application is met. In addition, the modeling ability of the Fuzzy Control Model is powerful and flexible, in the sense that it is able to capture any nonlinearity necessary for application-specific tuning and reconfiguration actions. Third, with the gateway-centric design introduced by the third tier of *Agilos*, a basic client-server application may be easily migrated to a scenario with multiple clients and servers, thus enabling wide-scale reconfiguration options such as switching among servers on-the-fly.

Our experimental results using an extensive experimental testbed, including the *OmniTrack* application, show that the *Agilos* architecture improves the adaptation awareness and effectiveness of flexible applications with respect to meeting the critical performance criterion, while the adaptive actions are highly flexible and configurable according to the needs of individual applications. They convincingly validate our theoretical analysis and show the feasibility and practicality of using the *Agilos* middleware control architecture for application-aware QoS adaptations.

12.2 Future Work

Within the broad scope of this work, the following research problems are still open and further enhancements to the *Agilos* architecture may assist to address these issues.

- Beyond the *OmniTrack* application, it is desirable to deploy more flexible and complex applications under the control of *Agilos*. These complex applications range from a MPEG-2 video streaming application to a complicated cluster of mission-critical web servers serving dynamic web contents. A careful study of the behavior of *Agilos* after the deployment of new applications will greatly facilitate a more thorough understanding of the interaction between

the middleware and the applications in multiple aspects. In addition, it is also an important step towards an in-depth evaluation of application deployment interfaces (ADI) presented in Chapter 8, with respect to the degree of modifications needed within the new applications for such a deployment process.

- One of the advantages of adopting a Task Control Model in the adaptors is that it may be seamlessly matched to the underlying network-level flow control protocols based on control theory. Even for standard window-based flow control protocols that are heuristic in nature, it is desirable to study the possibilities of dynamically tune the parameters within the adaptors, so that system-level flow control and application-aware adaptations coordinate well with each other. A careful examination of this open problem may help the *vertical integration* of multiple adaptive algorithms in both the middleware and system layers, which prevents the middleware layer to make application control decisions that are in conflict with the underlying system layers such as the network protocol stack.

- The *Agilos* middleware architecture assumes that the end system and networking environments are best effort in nature, and that the applications adapt their behavior to comply to restrictions and variations with regards to resource availability. Such assumptions hold in most of the scenarios for real-world applications. However, current state-of-the-art distributed systems have progressed to the point that some reservation mechanisms and protocols are built in. For example, Microsoft Windows 2000 supports the RSVP protocol in its Windows Sockets implementation, and some leading network routers from Cisco Systems feature built-in support of the RSVP protocol as well. It is advantageous to integrate the adaptation-based approach adopted by *Agilos* and the reservation-based mechanisms provided by emerging operating systems and the network infrastructure. The result of such integration will be able to bring together the level of QoS guarantees provided by reservation-based mechanisms and the best possible adaptive QoS brought by adaptation-based middleware.

- The current design of *Agilos* architecture is *static* in nature. The performance overhead

caused by such an architecture may not be tuned according to the processing capacities of the end systems. It is thus desirable to design and implement a more flexible framework, one that may be reconfigured to be lightweight but with less features. This framework features *meta-adaptation*, in the sense that the *Agilos* middleware itself may be reconfigured. This will provide a new dimension of flexibility with respect to the deployment of applications. For example, a lightweight configuration of *Agilos* may be used on top of operating systems running on handheld devices, such as the PalmOS, where computation and communication resources are so limited that the performance overhead brought by Agilos itself should be trimmed. On the other hand, a full-featured configuration of *Agilos* may be used on mission-critical servers to drive complex applications such as a cluster of web servers serving dynamic contents.

As illustrated, the research areas of adaptation-based middleware systems and application-aware QoS adaptations continue to attract intensive research interests and present exciting open research problems. The past achievements with the *Agilos* middleware architecture have led to an extensive theoretical framework and experimental testbed, which may serve as a milestone for future efforts to tackle these open research challenges.

References

[1] C. Aurrecoechea, A. Campbell, and L. Hauw, "A Survey of QoS Architectures," *ACM/Springer Verlag Multimedia Systems Journal*, vol. 6, no. 3, pp. 138–151, May 1998.

[2] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken, "QuO's Runtime Support for Quality of Service in Distributed Objects," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, England, September 1998.

[3] B. Stiller, C. Class, M. Waldvogel, G. Caronni, and D. Bauer, "A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 9, pp. 1580–1598, September 1999.

[4] T. Abdelzaher and N. Bhatti, "Web Server QoS Management by Adaptive Content Delivery," in *Proceedings of Seventh International Workshop on Quality of Service*, May 1999.

[5] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole, "A Distributed Real-Time MPEG Video Audio Player," *Lecture Notes in Computer Science*, vol. 1018, pp. 151–162, 1995.

[6] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker, "Agile Application-Aware Adaptation for Mobility," in *Proceedings of the 16th ACM Symposium on Operating Systems and Principles*, Saint-Malo, France, October 1997.

[7] M. Satyanarayanan, "Fundamental Challenges in Mobile Computing," in *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, May 1996.

[8] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997.

[9] D. Box, *Essential COM*, Addison-Wesley, Reading, MA, 1997.

[10] D. Hull, A. Shankar, K. Nahrstedt, and J. Liu, "An End-to-End QoS Model and Management Architecture," in *Proceedings of IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, December 1997, pp. 82–89.

[11] G. Hager and K. Toyama, "The XVision System: A General-Purpose Substrate for Portable Real-Time Vision Applications," *Journal of Computer Vision and Image Understanding*, vol. 69, no. 1, pp. 23–37, 1997.

[12] B. Li and K. Nahrstedt, "A Control-based Middleware Framework for Quality of Service Adaptations," *IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms*, vol. 17, no. 9, pp. 1632–1650, September 1999.

[13] B. Li and K. Nahrstedt, "A Control Theoretical Model for Quality of Service Adaptations," in *Proceedings of Sixth IEEE International Workshop on Quality of Service*, Napa Valley, California, May 1998, pp. 145–153.

[14] S. Keshav, "A Control-Theoretic Approach to Flow Control," in *Proceedings of ACM SIGCOMM '91*, September 1991, pp. 3–15.

[15] W. Tsai, Y. Kim, and C-K Toh, "A Stability and Sensitivity Theory for Rate-based Max-Min ABR Flow Control," in *Proceedings of 6th IEEE Singapore International Conference on Networks*, June 1998.

[16] G. Franklin and J. Powell, *Digital Control of Dynamic Systems*, Addison-Wesley, 1981.

[17] B. Li, D. Xu, and K. Nahrstedt, "Optimal State Prediction for Feedback-Based QoS Adaptations," in *Proceedings of Seventh IEEE International Workshop on Quality of Service*, London, UK, June 1999, pp. 37–46.

[18] J. Meditch, *Stochastic Optimal Linear Estimation and Control*, McGraw-Hill, 1969.

[19] R. Stengel, *Optimal Control and Estimation*, Dover Publications, 1994.

[20] B. Li and K. Nahrstedt, "Dynamic Reconfiguration for Complex Multimedia Applications," in *Proceedings of IEEE International Conference on Multimedia Computing and Systems 1999, Vol. 1*, Jun 1999, vol. 1, pp. 165–170.

[21] V. Witana, M. Fry, and M. Antoniades, "A Software Framework for Application Level QoS Management," in *Proceedings of Seventh International Workshop on Quality of Service*, May 1999.

[22] S. Servetto, "Compression and Reliable Transmission of Digital Image and Video Signals," *Ph.D. Thesis, University of Illinois at Urbana-Champaign*, June 1999.

[23] D. Driankov, H. Hellendoorn, and M. Reinfrank, *An Introduction to Fuzzy Control*, Springer-Verlag, 1996.

[24] A. Pitsillides, Y. A. Sekercioglu, and G. Ramamurthy, "Effective Control of Traffic Flow in ATM Networks Using Fuzzy Explicit Rate Marking (FERM)," *IEEE Journal of Selected Areas in Communications*, vol. 15, no. 2, pp. 209–225, February 1997.

[25] B. Li and K. Nahrstedt, "QualProbes: Middleware QoS Profiling Services for Configuring Adaptive Applications," in *ACM Springer-Verlag Lecture Notes in Computer Science, also Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, April 2000, vol. 1795, pp. 256–272.

[26] B. Li, W. Jeon, W. Kalter, K. Nahrstedt, and J.-H. Seo, "Adaptive Middleware Architecture for a Distributed Omnidirectional Visual Tracking System," in *Proceedings of SPIE Multimedia Computing and Networking 2000*, January 2000, pp. 101–112.

[27] Object Oriented Concepts Inc., "ORBacus for C++ and Java," *ftp://ftp.ooc.com/pub/OB/3.1/OB-3.1.1.pdf*, January 1999.

[28] S. Walli, "OpenNT: UNIX Application Portability to Windows NT via an Alternative Environment Subsystem," *Proceedings of the 1st USENIX Windows NT Symposium*, August 1997.

[29] G. Noer, "Cygwin32: A Free Win32 Porting Layer for UNIX Applications," *Proceedings of 2nd USENIX Windows NT Symposium*, August 1998.

[30] R. Jain, *Control-theoretic Formulation of Operating Systems Resource Management Policies*, Garland Publishing Company, 1979.

[31] D.-M. Chiu and R. Jain, "Analysis of Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks," *Computer Networks and ISDN Systems*, vol. 17, pp. 1–14, 1989.

[32] L. Benmohamed and S. Meerkov, "Feedback Control of Congestion in Packet Switching Networks: The Case of a Single Congested Node," *IEEE/ACM Transactions on Networking*, vol. 1, pp. 693–708, December 1993.

[33] S. Mascolo, D. Cavendish, and M. Gerla, "ATM Rate Based Congestion Control Using a Smith Predictor: an EPRCA Implementation," in *Proceedings of IEEE INFOCOM '96*, San Francisco, 1996.

[34] L. Benmohamed and Y.T. Wang, "A Control-Theoretic ABR Explicit Rate Algorithm for ATM Switches with Per-VC Queueing," in *Proceedings of IEEE INFOCOM '98, Session 2B*, 1998.

[35] V. Bharghavan, K.-W. Lee, S. Lu, S. Ha, J. Li, and D. Dwyer, "The TIMELY Adaptive Resource Management Architecture," *IEEE Personal Communications Magazine*, vol. 5, no. 4, August 1998.

[36] S. Lu and V. Bharghavan, "Adaptive Resource Management Algorithms for Indoor Mobile Computing Environments," in *Proceedings of ACM SIGCOMM '96*, August 1996.

[37] S. Lu, K.-W. Lee, and V. Bharghavan, "Adaptive Service in Mobile Computing Environments," in *Proceedings of 5th International Workshop on Quality of Service '97*, May 1997.

[38] J. Huang, Y. Wang, and F. Cao, "On developing distributed middleware services for QoS- and criticality-based resource negotiation and adaptation," *Journal of Real-Time Systems, Special Issue on Operating System and Services*, 1998.

[39] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A Resource Allocation Model for QoS Management," in *Proceedings of 18th IEEE Real-Time Systems Symposium*, December 1997.

[40] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha, "On Adaptive Resource Allocation for Complex Real-Time Applications," in *Proceedings of 18th IEEE Real-Time Systems Symposium*, December 1997.

[41] P. Chandra, A. Fisher, C. Kosak, T. Ng, P. Steenkiste, E. Takahashi, and H. Zhang, "Darwin: Resource Management for Value-Added Customizable Network Service," in *Proceedings of Sixth IEEE International Conference on Network Protocols (ICNP 98)*, October 1998.

[42] R. Ribler, H. Simitci, and D. Reed, "The AutoPilot Performance-Directed Adaptive Control System," *http://www-pablo.cs.uiuc.edu/Publications/publications.htm*, November 1997.

[43] D. Schmidt, D. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications Journal*, vol. 21, no. 4, April 1998.

[44] D. Xu, D. Wichadakul, and K. Nahrstedt, "Resource-Aware Configuration of Ubiquitous Multimedia Service," in *Proceedings of IEEE International Conference on Multimedia and Expo 2000 (ICME 2000)*, July 2000.

[45] K. Nahrstedt, D. Wichadakul, and D. Xu, "Distributed QoS Compilation and Runtime Instantiation," in *Proceedings of IEEE/IFIP International Workshop on Quality of Service 2000 (IWQoS 2000)*, June 2000.

[46] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," in *to appear in IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, April 2000.

[47] S. Brandt, G. Nutt, T. Berk, and J. Mankovich, "A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage," in *Proceedings of 19th IEEE Real-Time Systems Symposium*, December 1998, pp. 307–317.

[48] Z. Chen, S. M. Tan, R. H. Campbell, and Y. Li, "Real Time Video and Audio in the World Wide Web," *World Wide Web Journal*, vol. 1, January 1996.

[49] A. Hafid and G. Bochmann, "Quality of Service Adaptation in Distributed Multimedia Applications," *ACM Springer-Verlag Multimedia Systems Journal*, vol. 6, no. 5, September 1998.

[50] J. Bolliger and T. Gross, "A Framework-Based Approach to the Development of Network-Aware Applications," *IEEE Transactions on Software Engineering, Special Issue on Mobility and Network-Aware Computing*, vol. 24, no. 5, pp. 376–390, May 1998.

[51] A. Campbell and G. Coulson, "QoS Adaptive Transports: Delivering Scalable Media to the Desk Top," *IEEE Network Magazine*, vol. 11, no. 2, pp. 18–27, March 1997.

[52] N. Yeadon, F. Garcia, D. Hutchison, and D. Shepherd, "Filters: QoS Support Mechanisms for Multipeer Communications," *IEEE Journal on Selected Areas in Communications, Special Issue on Distributed Multimedia Systems and Technology*, vol. 14, no. 7, pp. 1245–1262, September 1996.

[53] F. Goktas, J. Smith, and R. Bajcsy, "Telerobotics over Communication Networks," in *Proceedings of 36th IEEE Conference on Decision and Control*, San Diego, California, December 1997, pp. 2393–2399.

[54] J. DeMeer, "On the Specification of End-to-End QoS Control," in *Proceedings of the Fifth International Workshop on Quality of Service*, May 1997, pp. 195–198.

[55] J-C. Bolot and T. Turletti, "Experience with Rate Control Mechanisms for Packet Video in the Internet," *Computer Communication Review*, vol. 28, no. 1, January 1998.

[56] U. Neumann and S. You, "Natural Feature Tracking for Augmented Reality," *IEEE Transactions on Multimedia*, vol. 1, no. 1, pp. 53–64, March 1999.

Vita

Baochun Li was born in Beijing, People's Republic of China in 1971. He received his Bachelor of Engineering degree in Computer Science and Technology from Tsinghua University, Beijing, P. R. China, in July 1995. In August 1995, he was admitted to the Department of Computer Science in the University of Illinois at Urbana-Champaign. In summer 1996, he worked on component-based software engineering as a summer intern in Anderson Consulting. He received his Master of Science degree in Computer Science in May 1997. His Master's thesis is entitled "Adaptive Behavior of Quality of Service in Distributed Multimedia Systems". His research interests include application-aware Quality of Service adaptation, middleware-based platforms, and wireless networking. Since 1997, he has worked on the DARPA-funded EPIQ (End-to-end Quality of Service) project in the MONET (Multimedia Operating Systems and Networking) research group, with a primary focus on middleware adaptation-based QoS solutions, along with experiments on a proof-of-concept distributed visual tracking system.